

Quick answers to common problems

JIRA Development Cookbook

Third Edition

Your one-stop resource for mastering extensions and customizations in JIRA 7+

Jobin Kuruvilla

[PACKT] PUBLISHING enterprise professional expertise distilled

JIRA Development Cookbook

Third Edition

Your one-stop resource for mastering extensions and customizations in JIRA 7+

Jobin Kuruvilla



BIRMINGHAM - MUMBAI

JIRA Development Cookbook

Third Edition

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2016

Production reference: 1230916

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78588-561-7

www.packtpub.com

Credits

Author

Jobin Kuruvilla

Reviewer

Satyendra Gangadhar Narwane

Commissioning Editor

Kunal Parikh

Acquisition Editor

Chaitanya Nair

Content Development Editor

Samantha Gonsalves

Technical Editor

Anushree Arun Tendulkar

Copy Editor

Safis Editing

Project Coordinator

Devanshi Doshi

Proofreader

Safis Editing

Indexer

Rekha Nair

Graphics

Jason Monteiro

Production Coordinator

Aparna Bhagat

About the Author

Jobin Kuruvilla is an Atlassian Consultant with experience in customizing JIRA and writing JIRA plugins for various customers. He is currently working with Go2Group as an Atlassian platinum expert, and is involved in managing Atlassian products for big enterprises as well as small starter license installations. From an IT nerd to a DevOps evangelist, Jobin has performed various roles in his professional journey so far.

Jobin is the author of *JIRA Development Cookbook*, released in 2011, and *JIRA 5.x Development Cookbook*, released in 2013, both published by *Packt Publishing* and well-received in the JIRA community. He also runs a website named *J-Tricks* (`http://www.j-tricks.com`), using which he shares numerous tutorials to help the developer community, who he believes have contributed immensely to his personal development. It is indeed those tutorials that sowed the first seeds of *JIRA Development Cookbook*.

Jobin started his career as a Java/J2EE developer in one of the biggest IT companies in India. After spending his initial years in the SOA world, he got hooked into this amazing product called JIRA, which he came across during the evaluation of a number of third-party tools. Soon, Jobin realized the power of JIRA, and pledged to spread the word. He has been doing it ever since, and he reckons there is a long way to go!

Outside the office, Jobin enjoys sports, especially soccer, and is on the verge of getting addicted to movies and social networking sites.

Acknowledgments

No book is the product of just the author; he just happens to be the one with his name on the cover.

A number of people contributed to the success of this book, and it would take more space than I have to thank each one individually.

First of all, thanks to the Almighty God for helping me to sail through the difficulties in this short life and for making my life as wonderful as it is now.

The next biggest thanks goes to the Content Development Editor, Samantha Gonsalves, and Acquisition Editor, Prachi Bisht, both of whom went through the pain of making me write another book. Also, a big shout-out to Anushree Tendulkar, the Technical Editor, who worked tirelessly to make the book as readable as it is now. And to the entire Packt Publishing team for working so diligently to help bring out another high quality product.

It is amazing to work with talented developers and technical geeks. Thank you to all the intelligent minds, with whom I got a chance to share this amazing journey of programming. Your encouragement and support were/are invaluable to me; you guys rock!

I must also thank the talented Atlassian community who are instrumental in helping each other, sharing solutions, being active in the forums, running user groups, and whatnot. I am just one of the many who have benefited.

Before I wind up, thank you Atlassian for giving us JIRA and a set of other wonderful products. You don't realize how much you are making our lives easier. Go Team!

Last, but not the least, a big thanks to all at Go2group for the support extended in writing this book and believing in my capabilities. We will, together, continue to simplify complexities for our numerous customers.

About the Reviewer

Satyendra Gangadhar Narwane describes himself as someone who is honest, caring, intelligent, hardworking, and ambitious. He has a great sense of humor. He is a post-graduate with masters in computer science from one of the premier Indian university and working as a Sr. Atlassian product expert in Dynamic Network Factory (U.S.A.), he has conducted seminar on behalf of Atlassian for promoting their tools in India, He has developed dozens of custom add-on based on customer needs. He has providing training, administrating & add-on development on Atlassian suite for top companies such as Oracle, Adobe, HCL,Mercedes Benz, MillenniumIT, Filpkart, ShipNet NS, KBC Bank. He is passionate about technology, computers science, traveling, watching movies and enjoy great chats. He has also worked on Jira 7 Development Cookbook.

I would like to express my gratitude to the many people who help me through this book. I would like to thank Packt for giving me a chance to work on this book. Above all I want to thank my mom Sushila, wife Poonam and the rest of my family, who supported and encouraged me in spite of all the time it took me away from them. It was a long and difficult journey for them. I would like to thank Suzanne Coutinho, Chaitanya Nair for helping me in the process of selection and reviewing. Last and not least: I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention."

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

My parents, **Alice** and **Kuruvilla**, who found something in me that I never knew existed. Nothing beats the pain and suffering they have undergone in the process.

"Behind every young child who believes in himself is a parent who believed first."

- Matthew Jacobson

My wife **Anumol**, my best friend for years. I wouldn't be in this place if not for her unconditional love and care.

"Happy is the man who finds a true friend, and far happier is he who finds that true friend in his wife."

- Franz Schubert

My little princesses, Elsa & Anna. Err, I mean **ANNA** & **SARAH**!

"Happiness is getting a hug from your daughter for no reason "

- Anonymous

My sister, **Juby Sara**, the best sister in the world. And to her husband, **Davis**, & son, **Kuttoos**, for keeping her happy. You both are doing a much better job than me ;)

"What's the good of news if you haven't a sister to share it?"

- Jenny DeVries

My friends from **TKM** and **JNV Kottayam** who dared me to dream and then helped me to achieve them. You guys are the best.

"A friend in need...".

Enough said!

This book would not have been possible without your love and understanding.

A big thank you from the bottom of my heart. I have nothing to give back, but my love and prayers.

Table of Contents

Preface 1

Chapter 1: Plugin Development Process 8

Introduction 8

What is a JIRA add-on? 9

The plugin development process 10

Atlassian Marketplace 11

Troubleshooting 11

Setting up the development environment 11

Getting ready 12

How to do it… 12

There's more… 13

Proxy settings for Maven 13

Using local Maven 14

Configuring IDEs to use the SDK 14

Troubleshooting 15

Creating a skeleton plugin 15

Getting ready 15

How to do it… 15

How it works… 16

There's more… 18

One step to your skeleton plugin 18

Creating an Eclipse project 18

Adding plugin modules 19

Getting ready 19

How to do it… 19

How it works… 20

Deploying a JIRA plugin 22

Getting ready 22

How to do it… 23

How it works… 24

There's more… 25

Using a specific version of JIRA 25

Reusing the data in each run 26

Troubleshooting 26

Making changes and redeploying a plugin 26

How to do it… 27

Debugging in Eclipse 27
See also 28
Using FastDev for plugin development 28
Getting ready 28
How to do it… 29
How it works… 31
There's more… 31
Adding ignored files 31
Changing admin credentials 32
See also 32
Testing and debugging 33
Getting ready 33
How to do it… 33
How it works… 34
There's more… 35
Using custom data for integration/functional Tests 35
Testing against different version of JIRA/Tomcat 35
See also 35
Chapter 2: Understanding the Plugin Framework 36
Introduction 36
JIRA architecture 37
Third-party components 37
Webwork 37
Seraph 38
Embedded Crowd 38
PropertySet 38
Active Objects 38
OSWorkflow 38
OfBiz Entity Engine 39
Apache Lucene 39
Atlassian Gadget JavaScript Framework 39
Shared Access Layer 40
Architecture 40
Authentication and user management 41
Property management 42
Presentation 42
Database 42
Workflows 43
Searching 43
Scheduled jobs 43
Plugins 43
Types of plugin modules 44
Reporting 44

Workflows 44
Custom fields 44
Searching 45
Links and tabs 45
Remote invocation 46
Actions and components 46
Other plugin modules 46
What goes into atlassian-plugin.xml? 48
Working with the Plugins1 and Plugins2 versions 50
Development 51
Installation 52
JIRA system plugins 52
Stable and core APIs 55
Modifying Atlassian bundled plugins 56
How to do it… 56
How it works… 57
See also 57
Converting plugins from V1 to V2 57
Getting ready 58
How to do it… 58
How it works… 59
See also 60
Adding resources into plugins 60
Getting ready 60
How to do it… 61
Adding web resources into plugins 62
How to do it… 63
How it works… 64
There's more… 64
Web resource contexts 65
Turning off batch mode 65
Building JIRA from source 66
Getting ready 66
How to do it… 67
How it works… 67
There's more… 68
Making a single class patch 68
See also 68
Adding new webwork actions to JIRA 68
Getting ready 69
How to do it… 70
How it works… 75

There's more… 77
Adding new commands to the action 77
See also 78
Form token handling in webwork actions 78
Getting ready 78
How to do it… 78
How it works… 79
There's more… 80
Providing a token in HTML links 80
Getting the token programmatically 81
Opting out of token checking in remote calls 81
See also 81
Capturing plugin installation/uninstallation events 81
Getting ready 82
How to do it… 83
How it works… 86
See also 87
Chapter 3: Working with Custom Fields 88
Introduction 88
Writing a simple custom field 89
Getting ready 90
How to do it… 90
How it works… 94
There's more… 94
See also 95
Custom field searchers 95
Getting ready 97
How to do it… 97
How it works… 100
There's more… 101
Dealing with custom fields on an issue 101
Getting ready 101
How to do it… 102
How it works… 104
See also 105
Programming custom field options 105
Getting ready 105
How to do it… 105
See also 107

Overriding the validation of custom fields 107
Getting ready 108
How to do it… 108
See also 108
Customizing the change log value 109
Getting ready 109
How to do it… 109
How it works… 110
Migrating from one custom field type to another 112
How to do it… 112
How it works… 113
There's more… 114
Changing the type of a custom field 114
See also 114
Making custom fields sortable 115
Getting ready 115
How to do it… 115
How it works… 116
There's more… 116
See also 116
Displaying custom fields on subtask columns 116
How to do it… 117
How it works… 117
User and date fields 118
How to do it… 118
How it works… 119
See also 120
Adding custom fields to notification e-mails 120
Getting ready 120
How to do it… 121
How it works… 122
Adding help text for a custom field 123
Getting ready 123
How to do it… 123
How it works… 124
Removing the “none” option from a select field 124
How to do it… 125
There's more… 126
Reloading velocity changes without restart (auto reloading) 127
See also 127

Making the custom field project importable 127
How to do it… 128
See also 128
Changing the size of a text area custom field 128
How to do it… 129
See also 130
Chapter 4: Programming Workflows 131
Introduction 131
Writing a workflow condition 133
Getting ready 134
How to do it… 135
How it works… 139
See also 140
Writing a workflow validator 140
Getting ready 141
How to do it… 142
How it works… 146
See also 147
Writing a workflow post function 147
Getting ready 149
How to do it… 149
How it works… 154
See also 155
Editing an active workflow 155
How to do it… 155
How it works… 156
There's more… 156
Modifying workflows in a JIRA database 156
Permissions based on workflow status 157
How to do it… 157
How it works… 159
There's more… 159
Making an issue editable/non-editable using workflow properties 159
See also 160
Including/excluding resolutions for specific transitions 160
How to do it… 160
How it works… 161
See also 162
Adding workflow triggers 162

Getting ready 162
How to do it… 162
How it works… 164
There's more… 165
User mapping from development tools to JIRA 165
See also 165
Internationalization in workflow statuses 166
How to do it… 166
How it works… 168
See also 168
Obtaining available workflow actions programmatically 168
How to do it… 169
How it works… 169
There's more… 170
Getting the action ID's given name 170
Programmatically progressing on workflows 170
How to do it… 171
How it works… 172
Obtaining workflow history from the database 172
Getting ready 172
How to do it… 173
How it works… 174
See also 175
Reordering workflow actions in JIRA 175
How to do it… 176
How it works… 176
Creating common transitions in workflows 177
How to do it… 178
How it works… 179
Creating global transitions in workflows 180
How to do it… 181
How it works… 183
Chapter 5: Gadgets and Reporting in JIRA 184
Introduction 184
Writing a JIRA report 185
Getting ready 186
How to do it… 186
How it works… 192
See also 194

Reports in Excel format 194
Getting ready 194
How to do it… 194
How it works… 196
See also 197
Data validation in JIRA reports 197
Getting ready 198
How to do it… 198
How it works… 199
See also 199
Restricting access to reports 200
Getting ready 200
How to do it… 200
How it works… 200
See also 201
Object configurable parameters for reports 201
How to do it… 202
How it works… 207
See also 209
Writing JIRA gadgets 210
Getting ready 210
How to do it… 211
How it works… 215
There's more… 215
Invoking REST services from gadgets 217
Getting ready 217
How to do it… 217
How it works… 221
See also 222
Configuring user preferences in gadgets 222
Getting ready… 222
How to do it… 222
How it works… 227
There's more… 228
See also 229
Accessing gadgets outside of JIRA 230
Getting ready… 230
How to do it… 230
How it works… 234

See also 234
Chapter 6: The Power of JIRA Searching 235
Introduction 235
Writing a JQL function 237
Getting ready 237
How to do it… 237
How it works… 242
See also 244
Sanitizing JQL functions 244
Getting ready 244
How to do it… 244
How it works… 245
See also 246
Adding a search request view 246
Getting ready 246
How to do it… 247
How it works… 251
There's more… 252
Using Single Issue Views to render search views 252
See also 257
Smart querying using quick search 258
How to do it… 258
There's more… 260
Searching in plugins 261
How to do it… 261
There's more… 263
See also 264
Parsing a JQL query in plugins 264
How to do it… 264
How it works… 265
See also 265
Linking directly to search queries 265
How to do it… 266
How it works… 267
There's more… 267
Index and de-index issues programmatically 268
How to do it… 268
See also 269
Searching on issue entity properties 269

Getting ready 270
How to do it… 270
How it works… 272
There's more… 273
See also 274
Managing filters programmatically 274
How to do it… 275
Creating a filter 275
Updating a filter 276
Deleting a filter 276
Retrieving filters 276
Sharing a filter 277
See also 277
Subscribing to a filter 277
How to do it… 278
How it works… 281
There's more… 281
See also 281
Chapter 7: Programming Issues 282
Introduction 282
Creating an issue from a plugin 283
How to do it… 283
How it works… 285
There's more… 285
Creating the issue using IssueManager 286
See also 286
Creating subtasks on an issue 286
How to do it… 287
See also 288
Updating an issue 288
How to do it… 288
Deleting an issue 289
How to do it… 289
Adding new issue operations 290
Getting ready 290
How to do it… 290
How it works… 293
There's more… 293
See also 294
Conditions on issue operations 294

Getting ready… 294
How to do it… 294
How it works… 296
Working with attachments 296
Getting ready… 297
How to do it… 297
Creating an attachment 297
Reading attachments on an issue 298
Deleting an attachment 298
There's more… 299
Time tracking and worklog management 299
Getting ready… 300
How to do it… 300
Auto adjusting the remaining estimate 300
Logging work and retaining the remaining estimate 301
Logging work with a new remaining estimate 302
Logging work and adjusting the remaining estimate by a value 303
How it works… 303
There's more 304
Updating worklogs 305
Deleting worklogs 306
Auto Adjusting remaining estimate 306
Deleting a worklog and retaining the remaining estimate 306
Deleting a worklog with a new remaining estimate 306
Deleting a worklog and adjusting the remaining estimate 307
Working with comments on issues 307
How to do it… 307
Creating comments on issues 307
Creating comments on an issue and restricting it to a project role or group 308
Updating comments 309
Deleting comments 310
Programming change logs 310
How to do it… 310
How it works… 312
Programming issue links 312
Getting ready… 313
How to do it… 313
There's more… 314
Deleting Issue Links 314
Retrieving Issue Links on an issue 314
JavaScript tricks on issue fields 315
How to do it… 316
How it works… 318

Creating issues and comments from e-mail 319
How to do it… 320
How it works… 322
Chapter 8: Customizing the UI 323
Introduction 323
Changing the basic look and feel 324
How to do it… 325
Adding new web sections in the UI 326
How to do it… 326
How it works… 328
See also 329
Adding new web items in the UI 329
How to do it… 330
How it works… 331
See also 331
Use of decorators and other metadata tags 331
Getting ready 332
How to do it… 334
How it works… 336
See also 336
Adding conditions for web fragments 337
How to do it… 337
How it works… 339
Creating new velocity context for web fragments 340
How to do it… 340
How it works… 341
Adding a new drop-down menu on the top navigation bar 342
How to do it… 342
How it works… 344
Dynamic creation of web items 344
Getting ready 344
How to do it… 345
How it works… 347
Adding new tabs in the View Issue screen 348
Getting ready 348
How to do it… 348
How it works… 351
There's more… 351
Loading issue tab panel asynchronously 351

Adding new tabs in the Browse Project screen 352
Getting ready 353
How to do it… 353
How it works… 355
Adding new links in the Project-centric view 355
Getting ready 355
How to do it… 356
How it works… 356
See also 357
Adding new panels in the project-centric view 357
Getting ready 357
How to do it… 357
How it works… 360
Adding sub-navigation in Project-centric view 361
Getting ready 361
How to do it… 362
How it works… 367
Adding issue link renderers 368
Getting ready 369
How to do it… 370
How it works… 378
See also 379
Displaying dynamic notifications/warnings on issues 379
Getting ready 380
How to do it… 380
There's more… 382
Re-ordering Issue Operations in the View Issue page 383
How to do it… 384
How it works… 385
See also 385
Re-ordering fields in the View Issue page 385
How to do it… 387
See also 388
Chapter 9: Remote Access to JIRA 389
Introduction 389
Writing Java client for REST API 390
Getting ready 390
How to do it… 391
Working with issues 393

Getting ready 393
How to do it… 393
Creating Issues 393
Updating issues 396
Browsing issues 396
Working with attachments 397
Getting ready 397
How to do it… 397
Using input stream and a new filename 397
Using the AttachmentInput object 398
Using file and a new filename 399
Browsing attachments 399
Remote time tracking 400
Getting ready… 400
How to do it… 400
Working with comments 402
Getting ready 402
How to do it… 402
Remote user and group management 403
How to do it… 403
Creating a User 403
Updating a User 405
Adding a User to an application 406
Removing a User from an application 407
Deleting a User 407
Creating a Group 408
Adding a user to a Group 409
Getting users in a Group 410
Removing a user from a Group 412
Deleting a Group 412
Progressing an issue in workflow 413
Getting ready 413
How to do it… 414
Searching issues 415
Getting ready 415
How to do it… 415
Managing versions 416
Getting ready 416
How to do it… 417
Managing components 418
Getting ready 418
How to do it… 418

Remote administration methods 419
How to do it… 420
Creating a Permission Scheme 420
Creating a Project 421
Retrieving the project roles 423
Add actors to a project role 424
How it works… 424
Exposing services and data entities as REST APIs 427
Getting ready 427
How to do it… 427
How it works… 432
Using the REST API browser 433
How to do it… 433
Working with JIRA Webhooks 436
How to do it… 436
How it works… 439
There's more… 440
Chapter 10: Dealing with the JIRA Database 442
Introduction 442
Extending the JIRA database with a custom schema 443
How to do it… 444
How works… 447
Accessing database entities from plugins 448
How to do it… 449
Reading from a database 449
Writing a new record 451
Updating a record 451
Persisting plugin information in the JIRA database 452
How to do it… 452
How it works… 454
Using Active Objects to store data 455
Getting ready… 456
How to do it… 456
How it works… 458
Accessing the JIRA configuration properties 459
How to do it… 460
Getting a database connection for JDBC calls 461
How to do it… 461
Migrating a custom field from one type to another 462
How to do it… 462

Retrieving issue information from a database 463
How to do it… 465
There's more… 467
Retrieving custom field details from a database 467
How to do it… 469
Retrieving permissions on issues from a database 470
How to do it… 472
Retrieving workflow details from a database 473
How to do it… 475
Updating the issue status in a database 475
Getting ready 476
How to do it… 476
Retrieving users and groups from a database 477
How to do it… 478
Dealing with change history in a database 480
How to do it… 480
Chapter 11: Useful Recipes 483
Introduction 484
Writing a service in JIRA 484
Getting ready 484
How to do it… 484
How it works… 486
See also 487
Adding configurable parameters to a service 487
How to do it… 487
How it works… 488
See also 489
Writing scheduled tasks in JIRA 490
How to do it… 490
How it works… 493
See also 494
Writing listeners in JIRA 494
Getting ready 494
How to do it… 494
How it works… 497
See also 497
Customizing e-mail content 497
How to do it… 498
How it works… 499

Redirecting to a different page in webwork actions 501
How to do it… 501
Adding custom behavior for user details 502
Getting ready 502
How to do it… 502
How it works… 506
Deploying a servlet in JIRA 507
Getting ready 507
How to do it… 507
How it works… 509
Adding shared parameters to Servlet Context 510
Getting ready 510
How to do it… 510
How it works… 511
Writing a Servlet Context Listener 511
Getting ready 511
How to do it… 512
How it works… 512
Using filters to intercept queries in JIRA 513
Getting ready 513
How to do it… 513
How it works… 516
Adding and importing components in JIRA 516
Getting ready 517
How to do it… 517
Exposing components to other plugins 518
Importing public components 520
How it works… 521
Adding new module types to JIRA 522
Getting ready 522
How to do it… 522
Creating modules using the new module type 526
Using the new modules 527
How it works… 528
Enabling access logs in JIRA 528
How to do it… 529
How it works… 530
Enabling SQL logging in JIRA 531
How to do it… 531
How it works… 532

Internationalization in webwork plugins 532
How to do it… 532
How it works… 534
Sharing common libraries across v2 plugins 535
Getting ready 536
How to do it… 536
Operations using direct HTML links 538
How to do it… 539
Implementing Marketplace licensing in plugins 542
Getting ready 542
How to do it… 542
How it works… 547
Index 551

Preface

This book is your one-stop resource for mastering JIRA extension and customization. You will learn how to create your own JIRA plugins; customize the look and feel of your JIRA UI; work with workflows, issues, custom fields; and much more.

The book starts with recipes on simplifying the plugin development process, followed by a complete chapter dedicated to the plugin framework to master plugins in JIRA.

Then, we will move on to writing custom field plugins to create new field types or custom searchers. After that, we will learn how to program and customize workflows to transform JIRA into a user-friendly system. We will follow this by looking at customizing the various searching aspects of JIRA, such as JQL, searching in plugins, and managing filters.

The book then steers towards programming issues, that is, creating/editing/deleting issues, creating new issue operations, managing the various other operations available on issues using the JIRA APIs, and so on. In the subsequent chapters, you will learn how to customize JIRA by adding new tabs, menus, and web items; communicate with JIRA using the REST interface; and work with the JIRA database.

The book ends with a chapter on useful and general JIRA recipes.

What this book covers

`Chapter 1`, *Plugin Development Process* , covers the fundamentals of the JIRA plugin development process. It covers in detail the setting up of a development environment, creating a plugin, deploying it, and testing it.

`Chapter 2` , *Understanding Plugin Framework* , covers in detail the JIRA architecture and looks at the various plugin points. It also looks at how to build JIRA from source and extend or override existing JIRA functionalities.

`Chapter 3` , *Working with Custom Fields* , looks at programmatically creating custom fields in JIRA, writing custom field searchers, and various other useful recipes related to custom fields.

`Chapter 4` , *Programming Workflows* , looks at the various ways of programming JIRA workflows. It includes writing new conditions, validators, post functions, and so on and contains related recipes that are useful in extending workflows.

`Chapter 5`, *Gadgets and Reporting in JIRA*, covers the reporting capabilities of JIRA. It looks at writing reports and dashboard gadgets, among others, in detail.

`Chapter 6`, *The Power of JIRA Searching*, covers the searching capabilities of JIRA and how it can be extended using the JIRA APIs.

`Chapter 7`, *Programming Issues*, looks at the various APIs and methods used for managing issues programmatically. It covers the CRUD operations, working with attachments, programming change logs and issue links, and time tracking among others.

`Chapter 8`, *Customizing the UI*, looks at the various ways of extending and modifying the JIRA user interface.

`Chapter 9`, *Remote Access to JIRA*, looks at the REST APIs that enable remote communication with JIRA and the ways of extending them.

`Chapter 10`, *Dealing with the Database*, looks at the database architecture of JIRA and covers the major tables in detail. It also covers the different ways of extending storage and accessing or modifying data using plugins.

`Chapter 11`, *Useful Recipes*, covers a selected list of useful recipes, which do not belong in the preceding categories but are powerful enough to get your attention. Read away!

What you need for this book

This book focuses on JIRA development. You need the following software as a bare minimum:

- JIRA 7.x+
- JAVA 1.8
- Maven 3.x
- Atlassian Plugin SDK 6.x+
- An IDE of your choice. The examples in the book use Eclipse.
- A database supported by JIRA. The examples in the book use H2DB, which comes with standalone JIRA.

Some of the recipes are too simple to use the fully fledged plugin-development process, and you will see this highlighted as you read through the book!

Who this book is for

JIRA Development Cookbook, Third Edition is intended for administrators that will be customizing, supporting and maintaining JIRA for their organizations.

You will need to be familiar and have a good understanding of JIRA's core concepts. For some recipes, basic understanding in HTML, CSS, JavaScript, and basic programming knowledge will also be helpful.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Definitely not! JIRA itself provides a lot of customization options through its user interface, and in more demanding cases, using property files like `jira-config.properties`."

A block of code is set as follows:

```
<settings>
    …
    <proxies>
        <proxy>
            <active>true</active>
            <protocol>http</protocol>
            <host>proxy.demo.com</host>
            <port>8080</port>
            <username>demouser</username>
            <password>demopassword</password>
           <nonProxyHosts>localhost|*.demosite.com</nonProxyHosts>
        </proxy>
    </proxies>
    …
</settings>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can just import the project using the option **File** | **Import** | **Existing Maven Projects**, and select the relevant project"



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors` .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/JIRA-Development-Cookbook-Third-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Plugin Development Process

In this chapter, we will cover the following topics:

- Setting up the development environment
- Creating a skeleton plugin
- Adding plugin modules
- Deploying a JIRA plugin
- Making changes and redeploying a plugin
- Using FastDev for plugin development
- Testing and debugging

Introduction

Atlassian JIRA, as we all know, is primarily an issue tracking and project management system. Since version 7.0, JIRA also comes in different flavors, namely **JIRA Core**, **JIRA Software**, and **JIRA Service Desk**, each packaged to cater to the needs of its various user categories. JIRA Core focuses on business teams, JIRA software on software teams and JIRA service desk on IT and service teams.

What many people do not know, though, is the power of its numerous customization capabilities, using which we can turn it into a different system altogether, much more powerful than these prepackaged flavors! These extra capabilities can take JIRA to the next level, in addition to its core issue tracking and project tracking capabilities for which JIRA, arguably, is the best player in the market.

So what are these customizations? How can we convert the JIRA we know into a product we want? Or maybe just add extra functionalities that are specific to our organization?

The answer to these questions probably can be summarized in a single word, add-ons, also referred to as plugins. JIRA has given the power to its users to write add-ons and customize the functionality in a way they find suitable.

But is that the only way? Definitely not! JIRA itself provides a lot of customization options through its user interface, and in more demanding cases, using property files such as `jira-config.properties`. In some cases, you will also find yourself modifying some of the JIRA core files to tweak functionality or to work around a problem. We will see more of that in the chapters to come, but the best entry point to JIRA customizations is **add-ons**. And that is where we start our cookbook, before we move on to the in-depth details.

What is a JIRA add-on?

So, what is a JIRA add-on? JIRA itself is a web application written in Java. But that doesn't mean you need to know Java to write an add-on, though in most cases you will need to. You might also end up writing a simple descriptor file to add a few links here and there. If that makes the non-Java developer in you happy, watch out for the different plugin modules JIRA supports.

There are two frameworks for writing JIRA add-ons: **Atlassian Connect** and the **Plugins2** framework.

Atlassian Connect add-ons are essentially web applications that operate remotely over HTTP. But they run only on Atlassian Cloud and are well documented at `https://developer.atlassian.com/static/connect/docs/latest/guides/introduction.html`, hence they are outside the scope of this book.

A **Plugins2** plugin is a JAR file that has a mandatory plugin **descriptor** and some optional Java classes and velocity templates. The velocity templates are used to render the HTML pages associated with your plugin, but in some cases, you might also want to introduce JSPs to make use of some pre-existing templates in JIRA. JSPs, as opposed to velocity templates, cannot be embedded in the plugin, but instead they should be dropped into the appropriate folders in the JIRA web application. Hence using velocity templates is recommended over JSPs. You can find more details on writing velocity templates at `http://velocity.apache.org/engine/1.7/user-guide.html#velocity-template-language-vtl-an-introduction`.

The plugin descriptor, the only mandatory part of a plugin, is an XML file which must be named `atlassian-plugin.xml`. This file is located at the root of the plugin. The `atlassian-plugin.xml` file defines the various modules in a plugin. The different types of available plugin modules include reports, custom field types, and so on, and these are discussed in detail in the next chapter.

The plugin development process

The process of developing a JIRA plugin can be of varying complexity, depending on the functionality we are trying to achieve. The plugin development process essentially is a four-step process:

1. Developing the plugin.
2. Deploying it into local JIRA.
3. Testing the plugin functionality.
4. Making changes and redeploying the plugin if required.

Each of these is explained in detail through the various recipes in this book.

JIRA, on start-up, identifies all the plugins that are deployed in the current installation. You can deploy multiple plugins, but there are some things you need to keep an eye on.

The `atlassian-plugin.xml` file has a **plugin key**, which should be unique across all the plugins. It is much similar to a Java package. Each module in the plugin also has a key that is unique within the plugin. The plugin key combined with the module key, separated by a colon, forms the complete key of a plugin module.

The following is a sample `atlassian-plugin.xml` without any plugin modules in it:

```
<!-- the unique plugin key -->
<atlassian-plugin key="com.jtricks.demo" name="Demo Plugin"
  plugins-version="2">
    <!-- Plugin Info -->
    <plugin-info>
        <description>This is a Demo Description</description>
        <version>1.0</version>
        <!-- optional  vendor details -->
        <vendor name="J-Tricks" url="http://www.j-tricks.com"/>
    </plugin-info>
    . . . 1 or more plugin modules . . .
</atlassian-plugin>
```

The plugin, as you can see, has details such as description, version, vendor-details, and so on, in addition to the key and name. When a plugin is loaded, all the unique modules in it are also loaded.

Suppose you have a report module in your plugin; it will look as follows:

```
<report key="demo-report" name="My Demo Report" ....>
...
</report>
```

The plugin key in the preceding case will be `com.jtricks.demo` and the module key will be `com.jtricks.demo:demo-report`.

Hang on; before you start writing your little plugin for a much wanted feature, have a look at the **Atlassian Marketplace** to see if someone else has already done the dirty work for you!

Atlassian Marketplace

Atlassian Marketplace is a one-stop shop where you can find the entire list of commercial and open source plugins people around the world have written. See `https://marketplace.atlassian.com/plugins/app/jira` for more details.

Troubleshooting

A common scenario that people encounter while deploying the plugin is when the plugin fails to load even though everything looks fine. Make sure your plugin's **key** is unique and is not duplicated in one of yours or another third-party's plugin!

The same applies to individual plugin modules.

Setting up the development environment

Now that we know what a plugin is, let's aim at writing one! The first step in writing a JIRA plugin is to set up your environment, if you haven't done that already. In this recipe, we will see how to set up a local environment.

To make plugin development easier, Atlassian provides the **Atlassian plugin software development kit** (**SDK**). It comes along with Maven and a preconfigured `settings.xml` to make things easier.

The Atlassian Plugin SDK can be used to develop plugins for other Atlassian products, including **Confluence**, **Crowd**, and so on, but we are concentrating on JIRA.

Getting ready

The following are the prerequisites for running the Atlassian Plugin SDK:



At the time of writing this recipe, the latest version of the Atlassian Plugin SDK is 6.1.0.

- The default port for the SDK, `2990`, should be available. This is important because different ports are reserved for different Atlassian products.
- Install JDK. Java version 1.8.X is required for Atlassian Plugin SDK 6.1.0. Please verify the compatible Java version for your SDK version.
- Make sure the `JAVA_HOME` is set properly and the command `java -version` outputs the correct Java version details.

How to do it…

1. Once we have Java installed and the port ready, we can download the latest version of Atlassian plugin SDK from `https://developer.atlassian.com/docs/getting-started/set-up-the-atlassian-plugin-sdk-and-build-a-project`.
2. Unzip the version into a directory of your choice or follow the instructions on the page, depending up on the operating system. Let's call this directory `SDK_HOME` going forward.
3. Add the SDK's `bin` directory into the environment `PATH` variable. If you are using the installer, this step is automatically done.
4. Create a new environment variable, `M2_HOME`, pointing to the `apache-maven` directory in your SDK Home. SDK version 4.x+ handles this step automatically.
5. Install the IDE of your choice. Atlassian recommends Eclipse, IntelliJ IDEA, or NetBeans, as they all support Maven.
6. Ready, set, go…

There's more…

With the preceding steps executed properly, we have a development environment for JIRA plugins. You can verify the installation of the SDK by running the following command:

```
atlas-version
```

This command displays the version and runtime information of the installed SDK.

The next step is to create a skeleton plugin, import it into your IDE, and start writing some code! Creating the skeleton plugin, deploying it, and so on, is explained in detail in the following recipes.

If you face issues while downloading the dependencies using Maven, read on.

Proxy settings for Maven

If you are behind a firewall, make sure you configure proxy in the Maven `settings.xml` file. Proxy can be configured as follows:

```
<settings>
    ...
    <proxies>
        <proxy>
            <active>true</active>
            <protocol>http</protocol>
            <host>proxy.demo.com</host>
            <port>8080</port>
            <username>demouser</username>
            <password>demopassword</password>
            <nonProxyHosts>localhost|*.demosite.com</nonProxyHosts>
        </proxy>
    </proxies>
    ...
</settings>
```

Find more about that and other aspects of Maven at
`http://maven.apache.org/index.html`.

Using local Maven

If you are a developer, in many cases you will have Maven already installed in your local machine. In that case, point the `M2_HOME` directory to your local Maven and update the respective `settings.xml` with the repository details in the default `settings.xml` that ships with the Atlassian Plugin SDK.

Or you can simply add the following to the existing `settings.xml`:

```
<pluginRepository>
    <id>atlassian-plugin-sdk</id>
    <url>file://${env.ATLAS_HOME}/repository</url>
    <releases>
        <enabled>true</enabled>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</pluginRepository>
```

Configuring IDEs to use the SDK

If you are using IntelliJ IDEA, it is an easy job because IDEA integrates Maven out of the box. Just load the project by selecting the `pom.xml`! See `https://developer.atlassian.com/docs/developer-tools/working-in-an-ide/configure-idea-to-use-the-sdk` for details.

If you are using Eclipse, make sure you have M2Eclipse installed. This is because Eclipse integrates Maven through the Sonatype M2Eclipse plugin. You can find more details on configuring this at `https://developer.atlassian.com/docs/getting-started/set-up-the-atlassian-plugin-sdk-and-build-a-project/set-up-the-eclipse-ide-for-linux` or `https://developer.atlassian.com/docs/getting-started/set-up-the-atlassian-plugin-sdk-and-build-a-project/set-up-the-eclipse-ide-for-windows`, depending on the OS.

For NetBeans, see `https://developer.atlassian.com/docs/developer-tools/working-in-an-ide/configure-netbeans-to-use-the-sdk`.

Troubleshooting

If you see Maven download errors such as `Could not resolve artifact`, make sure you verify the following:

- Entry in Maven `settings.xml` is correct, that is, it points to the correct repositories.
- Proxy configuration is done if required.
- Antivirus in the local machine is disabled and/or firewall restrictions removed if none of the above works! Seriously, it makes a difference.

Creating a skeleton plugin

In this recipe we will look at creating a skeleton plugin. We will use the Atlassian Plugin SDK to create the skeleton.

Getting ready

Make sure you have the Atlassian Plugin SDK installed.

How to do it…

1. Open a command window and go to the folder where you want to create the plugin.
2. Type `atlas-create-jira-plugin` and press *Enter*.
3. Enter the groupID when prompted. groupID would normally be coming from your organization name and mostly resembles the Java package. Of course, you can enter a different package name as we move forward if you want to keep it separate. groupID will be used to identify your plugin along with artifactID.

   For example: `com.jtricks.demo`

4. Enter the artifactId. The identifier for this artifact. Do not use spaces here.

   For example: `demoplugin`

5. The default version is 1.0-SNAPSHOT. Enter a new version if you want to change it or press *Enter* to keep the default.

   For example: `1.0`

6. Press *Enter* if the package value is same as the groupID. If not, enter the new value here and press *Enter.*

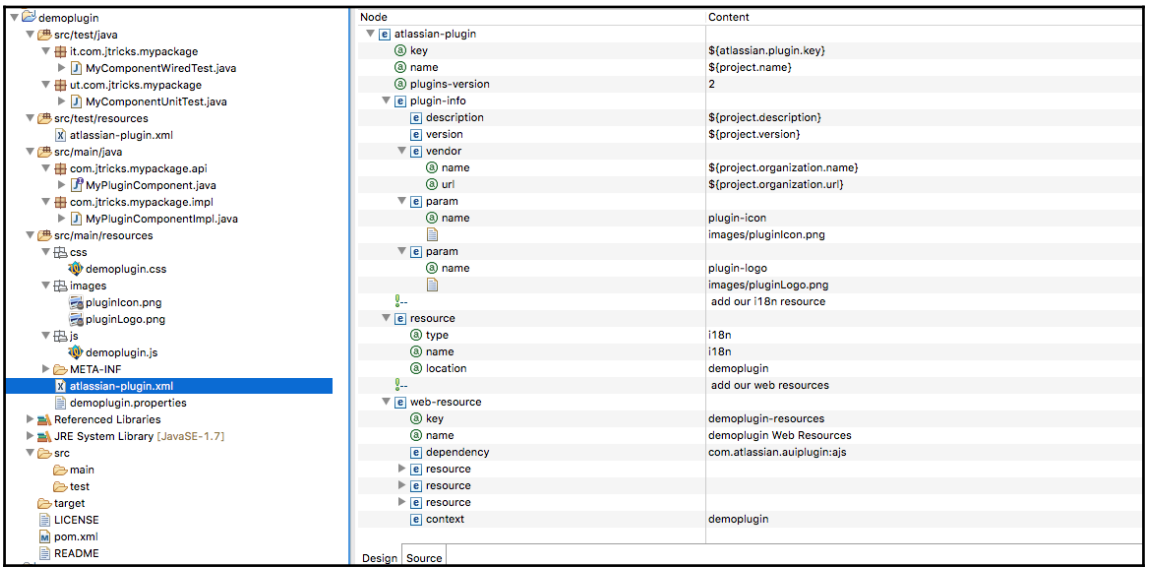
   For example: `com.jtricks.mypackage`

7. Confirm the selection when prompted. If you want to change any of the entered values, type *N* and press *Enter.*
8. Wait for the `BUILD SUCCESSFUL` message.

How it works…

A **skeleton plugin** is nothing but a set of directories and subdirectories along with a `pom.xml` (Maven project object model) file and some sample Java and XML files in the appropriate folders.

Here is a snapshot of how the project will look in Eclipse. It also shows the design view of the default `atlassian-plugin.xml`:



As you can see, there is a `pom.xml` at the root level and a `src` folder. A sample `LICENSE` file and a `README` file are also created for you at the root level.

Under the `src` folder, you will find two folders, `main` and `test`, with an identical folder structure. All your main Java code goes under the `main` folder. Any JUnit tests you write will go into the same location under the `test` folder. There is an additional folder, `it`, under the `test` folder where all the integration tests will go.

You will find the plugin descriptor, that is, `atlassian-plugin.xml`, under `src/main/resources` with sample values already populated in it. The values in the preceding screenshot are populated from the `pom.xml`. In our case, the plugin key will be populated as `com.jtricks.demo:demoplugin` when the plugin is built.

You will also notice that the skeleton plugin has some sample resources, which includes a CSS file, a JavaScript file, a plugin icon image, and a plugin logo image. The CSS and JS files are registered as resources under a `web-resource` plugin module. The `images` folder is also registered as a resource under the same module. The skeleton plugin also has a `demoplugin.properties` file, which is registered as an i18n `resource`. These files are placeholders and we can use them to add the respective functionality to the plugin.

So, that is our plugin skeleton. All that is pending is some useful Java code and proper module types in the `atlassian-plugin.xml`!



Remember, the first Maven run is going to take some time as it downloads all the dependencies in to your local repository. A coffee break might not be enough! If you have a choice, plan your meals.

There's more…

Sometimes, for the geeks, it is much easier to run a single command to create a project without bothering about the step-by-step creation. In this section, we will quickly see how to do it. We will also have a look at how to create an Eclipse project if you opt out of installing **m2eclipse**.

One step to your skeleton plugin

You can ignore the interactive mode by passing parameters such as groupID, artifactId, and so on, as arguments to the `atlas-create-jira-plugin` command:

```
atlas-create-jira-plugin -g my_groupID -a my_artifactId -v my_version -p
my_package   --non-interactive
```

For the example values we saw previously, the single line command will be as follows:

```
atlas-create-jira-plugin -g com.jtricks.demo  -a demoplugin -v 1.0 -p
com.jtricks.mypackage --non-interactive
```

You can pick and choose the parameters and provide the rest in an interactive mode as well!

Creating an Eclipse project

If you are not using m2eclipse, just run the following command from the folder where you have the `pom.xml` file:

```
atlas-mvn eclipse:eclipse
```

This will generate the plugin project for Eclipse, and you can then import this project into the IDE.

Execute `atlas-mvn eclipse:clean eclipse:eclipse` if you want to clean the old project and create again.

With IDEA or m2eclipse, just opening a file will do. That is, you can just import the project using the option **File** | **Import** | **Existing Maven Projects**, and select the relevant project.

Adding plugin modules

The plugin architecture is built around **plugin modules**, and JIRA exposes a number of plugin module types, each with a specific purpose. A plugin can have any number of plugin modules, either of the same type or of different types, as long as they all have a unique key. Throughout this book, we will see how we can use different plugin module types to solve different requirements.

In this recipe we will look at adding plugin modules to an existing plugin project.

Getting ready

Make sure the plugin project already exists or create a new skeleton project as explained, in the previous recipe.

How to do it…

1. Open a command window and go to the plugin project folder, where `pom.xml` resides.

   Type `atlas-create-jira-plugin-module` and press *Enter*. This will show all the available plugin modules as a numbered list, as shown here:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building demoplugin 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-jira-plugin:6.1.2:create-plugin-module (default-cli) @ demoplugin ---
Choose Plugin Module:
1:  Component Import
2:  Component
3:  Component Tab Panel
4:  Custom Field
5:  Custom Field Searcher
6:  Downloadable Plugin Resource
7:  Gadget Plugin Module
8:  Issue Tab Panel
9:  Keyboard Shortcut
10: JQL Function
11: Licensing API Support
12: Module Type
13: Project Tab Panel
14: REST Plugin Module
15: RPC Endpoint Plugin
16: Report
17: Search Request View
18: Servlet Context Listener
19: Servlet Context Parameter
20: Servlet Filter
21: Servlet
22: Template Context Item
23: User Format
24: Version Tab Panel
25: Web Item
26: Web Panel
27: Web Panel Renderer
28: Web Resource
29: Web Resource Transformer
30: Web Section
31: Webwork Plugin
32: Workflow Condition
33: Workflow Post Function
34: Workflow Validator
Choose a number (1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/21/22/23/24/25/26/27/28/29/30/31/32/33/34):
```

2. Select the number against the module that you are planning to add. For example, type `25` and press *Enter* if you want to add a simple `web-item` module to the plugin.
3. Follow the instructions to provide the details required for the selected module. Some of the options may have default values. Some modules might also have an advanced setup. Type `Y` and press *Enter* when prompted, if you want to go to `Advanced Setup`. If not, type `N` and press *Enter*.
4. Once the module is completed, type `Y` or `N` and press *Enter* when prompted to add another plugin module, depending on whether you want to add another module or not.
5. Repeat the steps for every module you want to add.
6. Wait for the `BUILD SUCCESSFUL` message when no more modules are to be added.

How it works…

Similar to the skeleton plugin creation, a set of directories and subdirectories are created during this process, along with a number of Java files or velocity templates required for the selected plugin module.

It also adds the plugin module definition in the `atlassian-plugin.xml` based on our inputs in step 4. A sample plugin descriptor, after adding the web-item module, looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<atlassian-plugin key="${atlassian.plugin.key}" name="${project.name}"
plugins-version="2">
    <plugin-info>
        <description>${project.description}</description>
        <version>${project.version}</version>
        <vendor name="${project.organization.name}"
          url="${project.organization.url}"/>
        <param name="plugin-icon">images/pluginIcon.png</param>
        <param name="plugin-logo">images/pluginLogo.png</param>
    </plugin-info>
    <!-- add our i18n resource -->
    <resource type="i18n" name="i18n" location="demoplugin"/>
    <!-- add our web resources -->
    <web-resource key="demoplugin-resources"
     name="demoplugin Web Resources">
        <dependency>com.atlassian.auiplugin:ajs</dependency>
        <resource type="download" name="demoplugin.css" location="/css
          /demoplugin.css"/>
        <resource type="download" name="demoplugin.js" location="/js
          /demoplugin.js"/>
        <resource type="download" name="images/" location="/images"/>
        <context>demoplugin</context>
    </web-resource>
    <web-item name="My Web Item" i18n-name-key="my-web-item.name"
      key="my-web-item" section="system.user.options/personal"
      weight="1000">
        <description key="my-web-item.description">The My Web Item Plugin
        </description>
        <label key="my-web-item.label"></label>
        <link linkId="my-web-item-link">http://www.j-tricks.com</link>
    </web-item>
</atlassian-plugin>
```

As you can see, a `web-item` module is added. We will see more about the `web-item` module, and other modules mentioned here, in the upcoming chapters.

You can also see a `resource` module and a `web-resource` module, which are added automatically the first time a plugin module is created.

The `resource` module defines the `i18n` resource file for the plugin, and more `key-value` pairs will be added in to this file when more modules are added. The file has the name `{plugin-artifact-name}.properties` and is created under the `src/main/resources{plugin-group-folder}` folder. In our example, the `demo.properties` file is created under the `src/main/resources/com/jtricks/demo` folder.

A sample property file is as follows:

```
#put any key/value pairs here
my.plugin.name=MyPlugin
my-web-item.label=My Web Item
my-web-item.name=My Web Item
my-web-item.description=The My Web Item Plugin
```

The `web-resource` module defines the plugin web resources, which includes a JS file, a CSS file, and an `images` folder that has the default plugin icon and logo images. We will learn more about these modules later in this book.

Deploying a JIRA plugin

In this recipe, we will see how to deploy a plugin into JIRA. We will see both the automated deployment using the Atlassian Plugin SDK and manual deployment.

Getting ready

Make sure you have the development environment set up as we discussed earlier. Also, the skeleton plugin should now have the plugin logic implemented in it.

How to do it…

Installing a JIRA plugin using the Atlassian Plugin SDK is a cakewalk. Here is how it is done:

1. Open a command window and go to your plugin's root folder, that is, the folder where your `pom.xml` resides.
2. Type `atlas-run` and press *Enter*. It is possible to pass more options as arguments to this command for which the details can be found at `https://developer.atlassian.com/docs/developer-tools/working-with-the-sdk/command-reference/atlas-run`.
3. You will see a lot of things happening as Maven downloads all the dependent libraries in to your local repo. As usual, it is going to take lot of time when you run it for the first time.
4. If you are on Windows, and if you see a security alert popping up, click **Unblock** to allow incoming network connections.
5. When the installation is completed, you will see messages similar to the following:

```
ars
[INFO] [talledLocalContainer] INFO: At least one JAR was scanned for TLDs yet contained no TLDs.
 Enable debug logging for this logger for a complete list of JARs that were scanned but no TLDs
were found in them. Skipping unneeded JARs during scanning can improve startup time and JSP comp
ilation time.
[INFO] [talledLocalContainer] May 31, 2016 9:53:27 PM org.apache.catalina.startup.HostConfig dep
loyDirectory
[INFO] [talledLocalContainer] INFO: Deployment of web application directory /Users/jobinkk/Book/
Drafts/Chapter 1/Code/demoplugin/target/container/tomcat8x/cargo-jira-home/webapps/manager has f
inished in 231 ms
[INFO] [talledLocalContainer] May 31, 2016 9:53:27 PM org.apache.coyote.AbstractProtocol start
[INFO] [talledLocalContainer] INFO: Starting ProtocolHandler ["http-nio-2990"]
[INFO] [talledLocalContainer] May 31, 2016 9:53:27 PM org.apache.coyote.AbstractProtocol start
[INFO] [talledLocalContainer] INFO: Starting ProtocolHandler ["ajp-nio-8009"]
[INFO] [talledLocalContainer] May 31, 2016 9:53:27 PM org.apache.catalina.startup.Catalina start
[INFO] [talledLocalContainer] INFO: Server startup in 70455 ms
[INFO] [talledLocalContainer] Tomcat 8.x started on port [2990]
[INFO] jira started successfully in 252s at http://Jobins-MBP.home:2990/jira
[INFO] Type Ctrl-D to shutdown gracefully
[INFO] Type Ctrl-C to exit
[WARNING] [talledLocalContainer] log4j:WARN No appenders could be found for logger (com.amazonaw
s.jmx.spi.SdkMBeanRegistry).
[WARNING] [talledLocalContainer] log4j:WARN Please initialize the log4j system properly.
[WARNING] [talledLocalContainer] log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noc
onfig for more info.
```

6. Open `http://localhost:2990/jira` in your browser.
7. Log in using the username `admin` and password `admin`.

8. Test your plugin! You can always go to **Administration | Add-ons | Manage add-ons** menu to confirm that the plugin is deployed properly.

If you already have a local JIRA installed or if you want to manually install your plugin due to some reason, all you need to do is to package the plugin JAR and install it via **UPM** (**Universal Plugin Manager**) as described at `https://confluence.atlassian.com/display/UPM/Installing+add-ons#Installingadd-ons-Installingbyfileupload`. Or, you can copy it across to `JIRA_Home/plugins/installed-plugins` directory and restart JIRA.

You can package the plugin using the following command:

```
atlas-mvn clean package
```

Use `atlas-mvn clean install` if you also want to install the package plugin into your local repo.

How it works…

There is only one single command that does the whole thing: `atlas-run`. When you execute this command, it does the following:

- Builds your plugin `.jar` file. It also builds the `.obr` file, which is the OSGi Bundle Repository file; that is essentially a `.jar` file containing our plugin and all dependent plugins, if we have any. More details on .obr files can be read at `https://developer.atlassian.com/docs/faq/advanced-plugin-development-faq/bundling-extra-dependencies-in-an-obr`.
- Downloads the latest/specified version of JIRA to your local machine if it is the first time you are running the command.
- Creates a virtual JIRA installation under your plugin `/target` folder.
- Copies the `.jar` file in to the `/target/jira/home/plugins/installed-plugins` directory.
- Starts JIRA in the Tomcat container.

Now, if you look at your `target` folder, you will see a lot of new folders that were created for the virtual JIRA installation! The main two folders are the `container` folder that has the Tomcat container set up and the `jira` folder that has the JIRA WAR along with the JIRA home setup.

You will find the database (HSQLDB), indexes, backups, and attachments under `/target/jira/home`, and you will see your `jira-webapp` at `/target/container/tomcat8x/cargo-jira-home/webapps/jira`.

If you have any JSPs that need to be put under the webapp, you will have to copy it to the appropriate folder under the aforementioned path.

There's more…

It is also possible to use a specific version of JIRA or to reuse the data that we have used for testing.

Using a specific version of JIRA

As mentioned earlier, `atlas-run` deploys the latest version of JIRA. But what if you want to deploy the plugin into an earlier version of JIRA and test it?

There are two ways to do it:

- Mention the JIRA version as an argument to `atlas-run`; make sure you run `atlas-clean` if you already have the latest version deployed:

  a. Run `atlas-clean` (if required)

  b. Run `atlas-run -v 5.0` or `atlas-run -version 5.0` if you are developing for JIRA version 5.0. Replace the version number with a version of your choice.

- Permanently change the JIRA version in your plugin `pom.xml`:

  a. Go to your `pom.xml`

  b. Modify the `jira.version` property value to the desired version.

  c. Modify the `jira.data.version` to a matching version.

  This is how it will look for JIRA 5.0:

```
<properties>
  <jira.version>5.0</jira.version>
  <jira.data.version>5.0</jira.data.version>
</properties>
```

Reusing the data in each run

Suppose you added some data on to virtual JIRA; how do you retain it when you clean start-up JIRA next time?

This is where a new SDK command comes to our rescue.

After `atlas-run` is finished, that is, after you pressed *Ctrl* + *C*, execute the following command:

```
atlas-create-home-zip
```

This will generate a file named `generated-test-resources.zip` under the `target/jira` folder. Copy this file to the `/src/test/resources` folder or any other known location. Now modify the `pom.xml` to add the following entry under configurations in the `maven-jira-plugin`:

```
<productDataPath>${basedir}/src/test/resources/generated-test-
resources.zip</productDataPath>
```

Modify the path accordingly. This will reuse the data the next time you run `atlas-run` after an `atlas-clean`.

Troubleshooting

Missing JAR file exception? Make sure the `local-repository` attribute in the `settings.xml` points to the embedded Maven repository that comes with the SDK. If the problem still persists, manually download the missing `.jar` files and use `atlas-mvn install` to install them in to the local repository.

Watch out for the proxy settings or antivirus settings that can potentially block the download in some cases!

BeanCreationException? Make sure your plugin is version 2. Check `atlassian-plugin.xml` to see if the `plugins-version="2"` entry is there or not. If not, add the entry, as shown here:

```
<atlassian-plugin key="${atlassian.plugin.key}" name="${project.name}"
 plugins-version="2">
```

Run `atlas-clean` followed by `atlas-run`.

Making changes and redeploying a plugin

Now that we have deployed the test plugin, it is time to add some proper logic, redeploy the plugin, and test it. Making the changes and redeploying a plugin is pretty easy. In this recipe, we will quickly look at how to do this.

How to do it…

You can make changes to the plugin and re-deploy it while the JIRA application is still running. Here is how we do it:

1. Keep the JIRA application running in the window where we ran `atlas-run`.
2. Open a new command window and go to the root plugin folder where your `pom.xml` resides.
3. Run `atlas-cli`.
4. Wait for the message – `Waiting for commands...`.
5. Run `pi`. `pi` stands for plugin install and this will compile your changes, package the plugin JAR, and install it into the `installed-plugins` folder.

As of JIRA 4.4, all the modules are reloadable and hence can be redeployed using this technique.

Debugging in Eclipse

It is also possible to run the plugin in debug mode and point to your IDE's remote debugger to it.

Following are the steps to do it in Eclipse:

1. Use `atlas-debug` instead of `atlas-run`.
2. Once the virtual JIRA is up and running with tour plugin deployed in it, go to **Run | Debug Configurations** in Eclipse.
3. Create a new **Remote Java Application**.
4. Give a name, keep the defaults, and give the port number as `5005`. This is the default debug port on which the virtual JIRA runs. In case you would like to use a different port, it is possible to change the debug port by passing `--jvm-debug-port` argument to `atlas-debug`.
5. Happy debugging!

See also

- *Setting up the development environment*
- *Creating a skeleton plugin*

Using FastDev for plugin development

We have seen how to use `atlas-cli` to reload a plugin without having to restart `atlas-run`. It is a pretty good way to save time, but Atlassian have walked the extra mile to develop a plugin named **FastDev**, which can be used to reload plugin changes, during development, to all Atlassian applications, including JIRA. And that from the browser itself!

Getting ready

Create a plugin and use `atlas-run` to run the plugin as in the aforementioned recipe. Let us assume we are doing it on the plugin we created in the previous recipe, the one with the sample web-item.

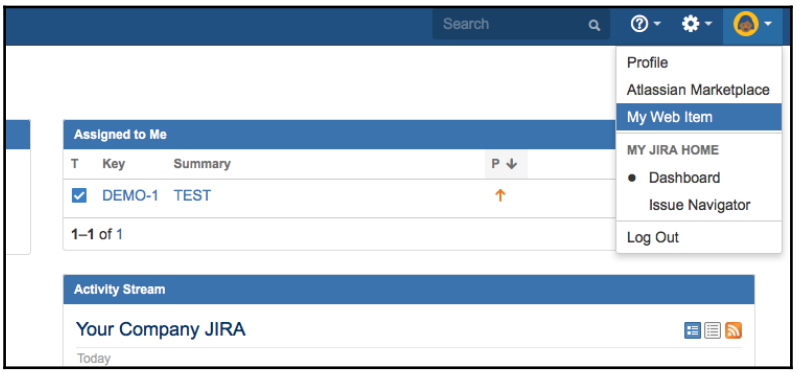
Make sure **FastDev** is enabled in the `maven-jira-plugin` configuration in the `pom.xml`, as shown here:

```
<plugin>
    <groupId>com.atlassian.maven.plugins</groupId>
    <artifactId>maven-jira-plugin</artifactId>
    <version>${amps.version}</version>
    <extensions>true</extensions>
    <configuration>
        <productVersion>${jira.version}</productVersion>
        <productDataVersion>${jira.version}</productDataVersion>
        <enableQuickReload>true</enableQuickReload>
        <enableFastdev>true</enableFastdev>
    </configuration>
</plugin>
```



Enabling FastDev is an important step, as it might be disabled by default, depending on the version of Atlassian SDK.

If we run that sample web-item plugin, using `atlas-run`, we can access JIRA at port 2990 as mentioned before and the web-item will look as highlighted here:

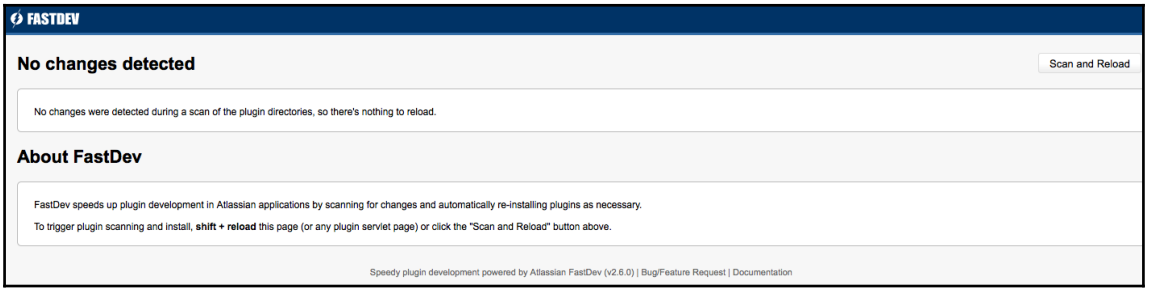


As you can see, the **My Web Item** link appears along with **Profile** link, under the **Personal** section. What if you wanted the link at the **jira-help** section, along with **Online Help**, About JIRA, and so on?

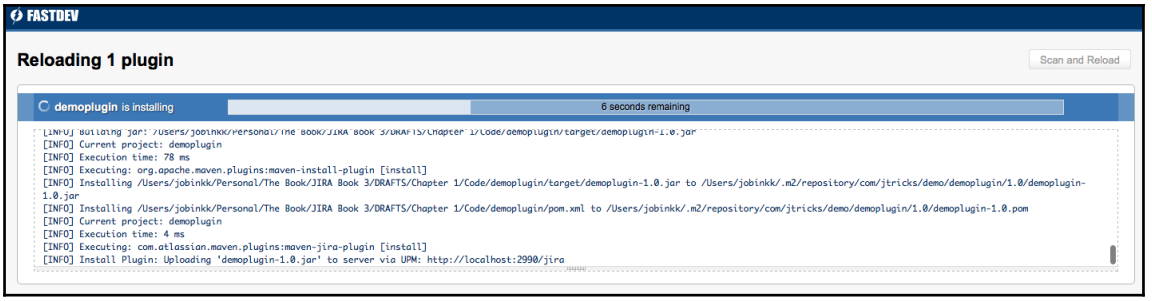
How to do it…

The following are the simple steps to make the change to the plugin and reload it using **FastDev**:

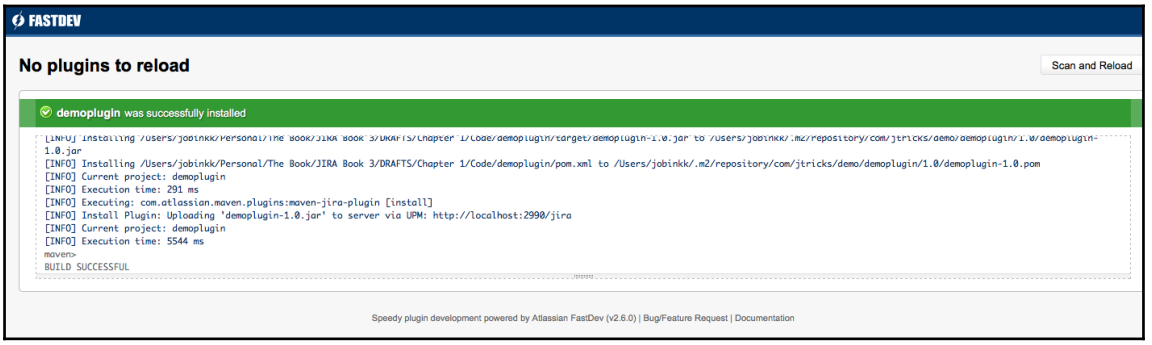
1. Access the FastDev servlet on the browser in the following path: `http://localhost:2990/jira/plugins/servlet/fastdev`. You will find the servlet as shown here:



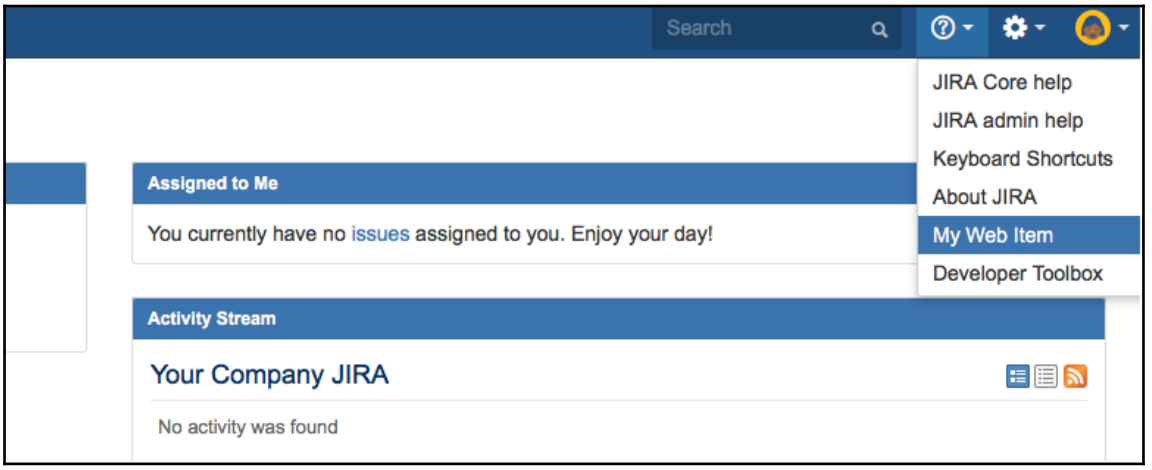
2. Make the necessary changes to your plugin. In this example, the change is pretty small. All we do is modify the `atlassian-plugin.xml` to change the **section** in the **web-item** module from `section="system.user.options/personal"` to `section="system.user.options/jira-help"`.
3. Click on the **Scan and Reload** button on the servlet (see the preceding image). On clicking the button, **FastDev** will reload the plugin. You can track the progress and see the logs on the browser itself, as shown here:



4. Once the plugin is successfully reloaded, you will see the following screen with a success message:



5. Reload the JIRA page to see if the change is effective. In our example, the new menu item moved under the **Help** section, as shown here:



Using **FastDev** is very effective while building a plugin from scratch and testing it, as the pieces gradually fall in to the right places.

How it works…

When the **Scan and Reload** button is pressed, **FastDev** looks for files that have changed since the last time plugin was installed. If it detects any changes, it starts a Maven process that re-installs the plugin.

More information on **FastDev** can be found at `https://developer.atlassian.com/docs/developer-tools/automatic-plugin-reinstallation-with-fastdev`.

There's more…

There is more to **FastDev** than the default configurations. They can be set in your plugin's `pom.xml` file by adding the required property to the `systemPropertyVariables` node, which in turn goes under the plugin `configuration` node.

You will find the details in the preceding Atlassian documentation, but the most useful ones are mentioned in the following section.

Adding ignored files

While looking for changes, it ignores certain files like `.js` or `.css` files that don't need a re-install of the plugin. Following is the full list of files/directories that are ignored:

Type	Property name	Default(s)
Directory	`fastdev.no.reload.directories`	`.svn`
Extension	`fastdev.no.reload.extensions`	`.js`, `.vm`, `.vmd`, `.fm`, `.ftl`, `.html`, `.png`, `.jpeg`, `.gif`, `.css`
File	`fastdev.no.reload.files`	

If you want to ignore additional files or directories, you can add them using the preceding properties, as shown here:

```
<systemPropertyVariables>
    ...
    <fastdev.no.reload.directories>images</fastdev.no.reload.directories>
    <fastdev.no.reload.extensions>classpath</fastdev.no.reload.extensions>
<fastdev.no.reload.files>${basedir}/src/main/resources/LCIENSE.txt</fastdev
.no.reload.files>
</systemPropertyVariables>
```

Changing admin credentials

FastDev uses the default `admin/admin` credential to reinstall the plugin. But if the username or password is different, use the `fastdev.install.username` and `fastdev.install.password` property, as shown here:

```
<systemPropertyVariables>
    ...
    <fastdev.install.username>myusername</fastdev.install.username>
    <fastdev.install.password>mypassword</fastdev.install.password>
</systemPropertyVariables>
```

See also

- Setting up the development environment
- Creating a skeleton plugin
- Making changes and redeploying a plugin

Testing and debugging

In the world of **Test Driven Development** (**TDD**), writing tests is part and parcel of the development process. I don't want to bore you with why testing is important!

Let us just say that all the advantages of TDD hold true for JIRA plugin development as well. And if you are wondering what exactly TDD is, start at `http://en.wikipedia.org/wiki/Test-driven_development`.

In this recipe, we will see the various commands for running unit tests and integration tests in JIRA plugins.

Getting ready

Make sure you have the plugin development environment set up and that you have created the skeleton plugin.

You might have noticed that there are two sample test files, one each for unit tests and integration tests, created under the `src/test/java/ut/` and `src/test/java/it/` folders.

You can build on top of these files to create a suite of test cases and use the SDK commands to run them.

How to do it…

The first step is, of course, to write some tests. I recommend the use of some powerful testing frameworks such as **JUnit** in collaboration with mocking frameworks such as **PowerMock** or **Mockito**. Make sure you have the valid dependencies added on to your `pom.xml`. The Atlassian SDK adds **JUnit** and **Mockito** by default under the dependencies, but you can change them if you are planning to use other frameworks.

Let us now make the huge assumption that you have written a few tests!

The following is the command to run your unit tests from the command line:

```
atlas-unit-test
```

The normal Maven command `atlas-mvn clean test` also does the same thing.

If you are running the integration tests, the command to use is as follows:

```
atlas-integration-test
```

The Maven command is as follows:

```
atlas-mvn clean integration-test
```

Once we are on to the stage of running tests, we will see it failing at times. Then comes the need for debugging. Check out the `*.txt` and `*.xml` files created under `target/surefire-reports/`, which has all the required information on the various tests that are executed.

Now, if you want to skip the tests at the various stages, use `-skip-tests`.

For example, `atlas-unit-test --skip-tests` will skip the unit tests.

You can also use the Maven options directly to skip the unit/integrations tests, or both together:

- `-Dmaven.test.skip=true`: This skips both unit and integration tests
- `-Dmaven.test.unit.skip=true`: This skips unit tests
- `-Dmaven.test.it.skip=true`: This skips integration tests

How it works…

The `atlas-unit-test` command merely runs the related Maven command `atlas-mvn clean test` in the backend to execute the various unit tests. It also generates the outputs into the `surefire-reports` directory for reference or debugging.

The `atlas-integration-test` does a bit more. It runs the integration tests in a virtual JIRA environment. It will start up a new JIRA instance running inside a Tomcat container, set up the instance with some default data, including a temporary license that lasts for three hours, and execute your tests!

How does JIRA differentiate between the unit tests and integration tests? This is where the folder structure plays an important role. Anything under the `src/test/java/it/` folder will be treated as integration tests and everything under `src/test/java/ut/` folder will be treated as unit tests.

There's more…

There is more to it.

Using custom data for integration/functional Tests

While `atlas-integration-test` makes our life easier by setting up a JIRA instance with some default data in it, we might need some custom data as well to successfully run a few functional tests.

We can do this in a few simple steps:

1. Export the data from a preconfigured JIRA instance into XML.
2. Put it under the `src/test/xml/` directory.
3. Provide this path as the value for `jira.xml.data.location` property in the `localtest.properties`, under `src/test/resources`.

The XML resource will then be imported to the JIRA before the tests are executed.

Testing against different version of JIRA/Tomcat

Just like the `atlas-run` command, you can use the `-v` option to test your plugin against a different version of JIRA. As before, make sure you do an `atlas-clean` before running the tests if you had tested it against another version before.

You can also use the `-c` option to test it against a different version of the Tomcat container.

For example, `atlas-clean && atlas-integration-test -v 3.0.1 -c tomcat5x` will test your plugin against JIRA version 3.0.1 using Tomcat container 5.

See also

- Setting up the development environment
- Deploying a plugin

2
Understanding the Plugin Framework

In this chapter, we will see more details on the JIRA architecture and the plugin framework. We will also see the following recipes:

- Modifying Atlassian bundled plugins
- Converting plugins from V1 to V2
- Adding resources into plugins
- Adding web resources to the plugin
- Building JIRA from source
- Adding new webwork actions to JIRA
- Form token handling in webwork actions
- Capturing plugin installation/uninstallation events

Introduction

As we saw in the previous chapter, the JIRA plugin development process is probably an easier task than we expected it to be. With the help of Atlassian plugin SDK, developers can spend more time worrying about the plugin logic than on the troublesome deployment activities. And yes, after all, it is the plugin logic that is going to make an impact!

This chapter details how the various components fit into JIRA's architecture and how JIRA exposes the various pluggable points. We will also see an overview of JIRA's system plugins to find out how JIRA uses the plugin architecture to its own benefit, followed by some useful recipes.

JIRA architecture

We will quickly see how the various components within JIRA fit in to form the JIRA we know. It is best described in a diagram, and Atlassian has a neat one, along with a detailed explanation at `https://developer.atlassian.com/jiradev/jira-platform/jira-architecture/jira-technical-overview`. We will redraw the diagram a little bit to explain it in a brief but useful way.

Before we jump into the JIRA architecture, let us take a look at the third-party components used inside JIRA. A deep dive into these components is not necessary, but an understanding of these components will surely help us to understand the JIRA architecture better.

Third-party components

Before we dig deeper into the JIRA architecture, it is probably helpful to understand a few key components and familiarize yourself with them. JIRA's major third-party dependencies are outlined next.



It is not mandatory to know all about these frameworks, but it will be very helpful during plugin development if you have an understanding of these.

After the brief introduction of these components, we will see how they work together inside JIRA in the next section.

Webwork

Webwork is nothing but a Java web application development framework. It is a MVC framework similar to **Struts**. The following is a quick overview of webwork as you find it in the **OpenSymphony** documentation:

> *"It is built specifically with developer productivity and code simplicity in mind, providing robust support for building reusable UI templates, such as form controls, UI themes, internationalization, dynamic form parameter mapping to JavaBeans, robust client- and server-side validation, and much more."*

Read more about JIRA webwork actions at
`https://developer.atlassian.com/jiradev/jira-platform/jira-architecture/jira-technical-overview/jira-webwork-actions`.



Note that JIRA uses webwork1 and not webwork2. In this book, all instances of webwork refer to the webwork1 version. JIRA itself refers to the technology as webwork, but you will notice that the files, plugin modules, and so on, use Webwork1 in them just to emphasize the version.

Seraph

Seraph is Atlassian's open source web authentication framework. It provides a simple, extensible authentication system that JIRA uses for all authentication purposes.

Read more about seraph at `http://docs.atlassian.com/atlassian-seraph/latest/`.

Embedded Crowd

JIRA uses **Crowd**, Atlassian's identity management and **single sign-on** (**SSO**) tool, for user management. JIRA embeds a subset of Crowd's core modules, which provides the user management capabilities.

More information on Crowd can be found at
`http://www.atlassian.com/software/crowd/overview`.

PropertySet

PropertySet is again another open source framework from OpenSymphony that helps you to store a set of properties against any “entity” with a unique ID. The properties will be key/value pairs and can only be associated with a single entity at a time.

Active Objects

Active Objects is a new **ORM** (**object relational mapping**) layer into Atlassian products. It is implemented as a plugin and provides data storage that can be used by plugins to persist their private data. It enables easier, faster, and scalable data access compared to PropertySet.

More details on Active Objects can be found at
`https://developer.atlassian.com/display/AO/Active+Objects`. We will also see more details about AO in `Chapter 10`, *Dealing with the JIRA Database*.

OSWorkflow

OSWorkflow is yet another open source framework from the OpenSymphony group. It is an extremely flexible workflow implementation that is capable of driving complex conditions, validators, post functions, and so on, along with many other features.

OfBiz Entity Engine

OfBiz stands for "Open for Business" and the **OfBiz entity engine** is a set of tools and patterns used to model and manage entity-specific data.

As per the definition from the standard entity-relation (**ER**) modeling concepts of **Relational Database Management Systems** (**RDBMS**):

> *An Entity is a piece of data defined by a set of fields and a set of relations to other entities.*

Read more about Ofbiz entity engine at
`https://cwiki.apache.org/confluence/display/OFBIZ/Entity+Engine+Guide`.

Apache Lucene

Apache Lucene is a text search engine library. Following is a simple definition of Apache Lucene that you can find in its documentation:

> *"Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform."*

More about Lucene and its potential can be found at
`http://lucene.apache.org/core/index.html`.

Atlassian Gadget JavaScript Framework

JIRA has a powerful gadget framework. Atlassian has gone OpenSocial with its gadgets and in order to help developers in creating gadgets, Atlassian has introduced a **Gadget JavaScript Framework** that encapsulates a lot of common requirements and functionalities used within gadgets.

More about this framework can be read at
`https://developer.atlassian.com/display/GADGETS/Using+the+Atlassian+Gadgets+JavaScript+Framework`.

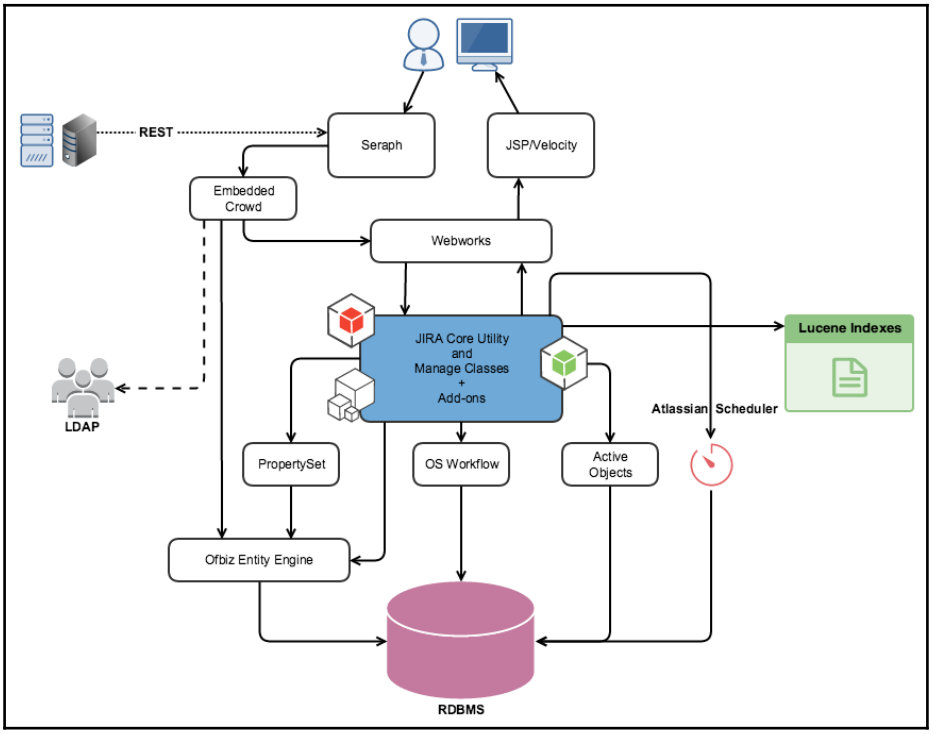
Shared Access Layer

Atlassian uses **Shared Access Layer**, also called **SAL**, to expose APIs that fulfill common functionalities across all Atlassian applications. It is most useful for cross-application development, but you can also use the SAL APIs for common services such as job scheduling, i18n lookup, and so on.

Read more about SAL at `https://developer.atlassian.com/docs/atlassian-platform-common-components/shared-access-layer`.

Architecture

It is best to learn the intricacies of system architecture with the help of a diagram. For the benefit of a brief but meaningful explanation on the JIRA Architecture, let us have a quick look at the following diagram:



JIRA is a web application built using the MVC Architecture. It is fully written in JAVA and is deployed as a WAR file into a JAVA servlet container such as Tomcat.

The majority of the JIRA core functionality revolves around the **JIRA Utility and Manager Classes**, which thus become the heart of JIRA. But it also interacts with a lot of third-party components, which we saw earlier, to deliver powerful functionalities such as workflows, permissions, user management, searching, and so on.

As with any other web application, let us start with the incoming requests. Users usually interact with JIRA using web browsers. But it is also possible to interact with JIRA using its REST APIs.

Authentication and user management

The user authentication, whichever way the request comes, is done in JIRA using Seraph, Atlassian's open source web authentication framework. Seraph is implemented as a servlet filter and it intercepts each and every incoming request and associates it with a specific user. It supports various authentication mechanisms, such as HTTP basic authentication, form-based authentication, and so on, and even looks up already stored credentials in a user session when implemented with SSO.

However, Seraph doesn't do any user management itself. It delegates this to the Embedded Crowd framework. One additional thing that Seraph does in JIRA is to intercept URLs starting with `/admin/` and allow users only if they have the "Global Admin" permission.

Coming back to authentication and other user management functions, Crowd is a SSO and identity management system from Atlassian, embedded now in JIRA. Plugin developers can now use CrowdService to manage users and groups, for which more information can be found at `http://docs.atlassian.com/crowd/current/com/atlassian/crowd/embedded/api/CrowdService.html`.

Following are the core Crowd functionalities:

- User, group, and membership management – Stores all details in JIRA DB
- Authentication – Includes Password matching
- Allows connection to external user directories such as LDAP, AD, Crowd, and so on
- Synchronizes external user data to local database for faster access

Property management

JIRA lets you add key/value pairs as properties on any available *entity* like user, group, project, issue, and so on. It uses OpenSymphony's PropertySet to do this. Three major cases where PropertySet is used internally in JIRA are as follows:

- To store user preferences such as e-mail, full name, and so on
- To store application properties
- To store chosen preferences of Portlets/Gadgets on user dashboards

We can also use the PropertySet in our plugins to store custom data as key/value pairs.

In earlier versions of JIRA, PropertySet was the only technology used to store plugin information and other data related to plugins. But JIRA now supports a technology called**ActiveObjects**, which can be used to store plugin data. ActiveObjects is explained in detail in `Chapter 10`, *Dealing with JIRA Database*.

Presentation

The presentation layer in JIRA is built using **JSPs** and **Velocity** templates. Another experimental component available in JIRA5+, for presentation layer, is **soy** templates.

The web requests, coming on to JIRA, are processed by OpenSymphony's Webwork1 framework. The requests are handled by webwork actions, which internally use the JIRA service layer. The service classes expose the core Utility and Manager classes that perform the tasks behind the scenes!

Database

JIRA talks to its database using the Ofbiz entity engine module. Its database schema is defined in the `entitymodel.xml` and `entitygroup.xml` files residing at `WEB-INF/classes/entitydefs` folder. The entity configuration details go into `entityengine.xml` under `WEB-INF/classes`, whereas the DB connectivity details are stored in `dbconfig.xml` residing in the JIRA home directory.

JIRA supports a wide variety of database products for which more details can be found at `https://confluence.atlassian.com/jira/connecting-jira-to-a-database-185729615.html`.

Workflows

Workflows are one of the most important features in JIRA. It provides us with a highly configurable workflow engine, which uses OpenSymphony's **OSWorkflow** behind the scenes. It lets us customize the workflows by adding new *steps* and *transitions*, and for each transition we can add *conditions*, *validators*, and *post functions*. We can also write plugins to add more of these, in addition to the ones that ship with vanilla JIRA. We will see all that in detail in the coming chapters.

Searching

JIRA uses Apache **Lucene** to perform indexing in JIRA. Whenever an issue is changed in JIRA, it performs a partial re-indexing to update the related indexes. JIRA also lets us do a full re-index at any time, manually, from the Administration screen.

Searching in JIRA is done using these indexes, which are stored in the local drive. We can even store search queries as filters whose results get updated as the indexes change.

Scheduled jobs

JIRA uses the **Atlassian Scheduler**, which is part of **SAL**, to schedule jobs within JIRA. The jobs, including the subscriptions to the filters and the custom ones we add, are scheduled and executed by the PluginScheduler component, which is exposed by SAL.

We will see more details on how we can write a new scheduled job using **PluginScheduler** in `Chapter 11`, *Useful Recipes*.

Plugins

Last, but not least, **plugins**—also referred to as **add-ons**—fit into the JIRA architecture to provide extra functionalities or to alter some of the existing ones. The plugins mostly use the same JIRA core utility classes and manager classes as webwork actions do, but in some cases also add/contribute to the list.

There are different pluggable points in JIRA, which we will see in detail in this chapter.

This, I hope, gives you a brief introduction to the JIRA architecture and the major components used in it. We will see most of these in detail and how to customize them by writing plugins in the coming chapters. Off you go!

Types of plugin modules

JIRA supports a large number of plugin modules. All these modules are various extension points, using which we can not only add new functionalities in to JIRA, but also extend some of the existing functionalities.

This section briefly describes what each plugin module is designed to do. More information on each of these modules is covered in the upcoming chapters.

Let us group the different modules based on functionality instead of seeing them all together.

Reporting

Following are the various plugin module types that can be used for reporting in JIRA:

Module type	Description
gadget	Adds new gadgets into the user's dashboard. These gadgets can also be accessed from other applications.
report	Adds new reports into JIRA. These reports are accessed from the project home page.

Workflows

Following are the various plugin module types that can be used to extend workflows in JIRA:

Module type	Description
workflow-condition	Adds new workflow conditions to the JIRA workflow. Conditions can be used to limit the workflow actions to users, based on predefined criteria.
workflow-validator	Adds new workflow validations to the JIRA workflow. Validations can be used to prevent certain workflow actions when the criteria are not met.
workflow-function	Adds new workflow post functions to the JIRA workflow. These can be used to perform custom actions after a workflow action is executed

Custom fields

Custom fields are essential for JIRA issues, and the following plugin module type adds more custom field types to JIRA:

Module type	Description
customfield-type	Adds new custom field types to JIRA. We can customize the look-and-feel of the fields in addition to custom logic. See also customfield-searcher.

Searching

Following are the various plugin module types that can be used for searching in JIRA. These module types vary, from adding custom field searchers to exposing new JQL functions and adding more index values inside JIRA:

Module type	Description
customfield-searcher	Adds new field searchers on to JIRA. The searcher needs to be mapped with the relevant custom fields.
jql-function	Adds new JQL functions to be used with JIRA's advanced searching.
search-request-view	Adds a new view in the issue navigator. They can be used to show the search results in different ways.
index-document-configuration	Allows add-ons to add key/value stores to JIRA entities, such as issues or projects. These values are indexed by JIRA and can be queried via a REST API or through JQL

Links and tabs

Following are the various plugin module types that can be used to add more content inside JIRA pages. They include links that can be placed in various sections to tabs that can display more content on selected pages:

Module type	Description
web-section	Adds new sections in application menus. Each section can contain one or more links under it.
web-item	Adds new links that will appear at a defined section. The section here can be the new ones we added or the existing JIRA web sections.

project-tab panel	Adds new tabs to the **Browse Project** screen. We can define what has to appear in the tab.
component-tab panel	Adds new tabs to the **Browse Component** screen. As stated in the **Browse Project** screen, we can define what to appear in the tab.
version-tab panel	Adds new tabs to the **Browse Version** screen. Same as the preceding module type.
issue-tab panel	Adds new tabs to the **View Issue** screen. Similar to other tabs, here also we can define what appears in the tab.
web-panel	Defines panels or sections that can be inserted into an HTML page.
issue-link-renderer	Allows you to add new custom renders for issue links to JIRA. Useful for **Remote Issue** links.

Remote invocation

JIRA exposes a lot of its functionality, for remote invocation, via REST API. The following plugin modules type allows us to expose the missing and/or custom functionality in JIRA via REST API, on top of the existing methods:

Module type	Description
rest	Creates new REST APIs for JIRA to expose more services and data entities.

Actions and components

Following are the various plugin module types that can be used to add new webwork actions in JIRA and to add new components to JIRA's component system:

Module type	Description
webwork	Adds new webwork actions along with views into JIRA, which can add new functionality or override existing ones.
component	Adds components to JIRA's component system. These are then available for use in other plugins and can be injected into them.
component-import	Imports components shared by other plugins.

Other plugin modules

Following are the various plugin module types that do not belong to any of the aforementioned categories, but are useful in their own way:

Module type	Description
resource	Adds downloadable resources into the plugins. A resource is a non-JAVA file such as JavaScript, CSS, image files, and so on.
web-resource	Similar to the above, adds downloadable resources into the plugins, but these are added to the top of the page with the cache-related headers set to never expire. We can also specify the resources to be used only in specific contexts. Multiple resource modules will appear under a web-resource module.
Web-resource-transfomer	Manipulates static web resources before they are batched and delivered to the browser.
Servlet	Deploys a JAVA servlet onto JIRA.
Servlet-context-listener	Deploys a JAVA Servlet Context Listener.
Servlet-context-param	Sets parameters in the Servlet context shared by the plugin's servlets, filters, and listeners.
Servlet-filter	Deploys a JAVA servlet filter onto JIRA. The order and position in the application's filter chain can be specified.
User-format	Adds custom behaviors for user details. Used to enhance the user profile.
Keyboard-shortcut	Available only from 4.1.x. Defines new keyboard shortcuts for JIRA. You can also override the existing shortcuts from JIRA 4.2.x!
ao	Allows the use of Active Objects service to persist plugin data.
Message-handler	Adds a custom message handler in JIRA. It can be used to receive e-mail messages and process them to do any sort of operations in JIRA.
routing	URL Routing Plugin Module is used to map any URL to one of our own choosing. This can be used to add pretty URLs in JIRA, to add more meaning to the URLs used in the plugin.
sitemesh	SiteMesh Decoration Plugin Module is used to decorate URLs that by default will not be decorated by JIRA. This is typically the case when we have our own pretty URLs in place. See routing module.

module-type	Dynamically adds new plugin module types to the plugin framework. The new module can be used by other plugins.

What goes into atlassian-plugin.xml?

Let's look deeper into the plugin descriptor named `atlassian-plugin.xml`.

Following is how the plugin descriptor will look when the skeleton plugin is created:

```
<atlassian-plugin key="${atlassian.plugin.key}" name="${project.name}"
plugins-version="2">
  <plugin-info>
    <description>${project.description}</description>
    <version>${project.version}</version>
    <vendor name="${project.organization.name}"
      url="${project.organization.url}"/>
    <param name="plugin-icon">images/pluginIcon.png</param>
    <param name="plugin-logo">images/pluginLogo.png</param>
  </plugin-info>
</atlassian-plugin>
```

We need to add more details into it depending on the type of plugin we are going to develop. The plugin descriptor can be divided into three parts:

- **atlassian-plugin element**: This forms the root of the descriptor. The following attributes populate the `atlassian-plugin` element:

  `key`: This is probably the most important attribute. It should be a unique key across the JIRA instance, and will be used to refer the different modules in the plugin, just like we use the packages in a Java application.
  If you see `${project.groupId}.${project.artifactId}` or `${atlassian.plugin.key}` as the plugin key, it picks up the values from your `pom.xml` file. When the plugin is built, the key will be **`YOUR_GROUP_ID.YOUR_ARTIFACT_ID`**.

  `name`: Give an appropriate name for your plugin. This will appear in the **Add-ons** menu under **Administration**.

  `plugins-version`: This is different from the version attribute. `plugins-version` defines whether the plugin is version 1 or 2. If this element is absent, the plugin will be treated as a v1 plugin. More details on plugin versions 1 and 2 can be found in the next section.

`state`: This is an optional element to define the plugin as disabled, by default. Add `state="disabled"` under the `atlassian-plugin` element to disable the plugin on installation.

- **plugin-info**: This section contains information about a plugin. It not only provides information that is displayed to administrators, but also, optionally, provides bundle instructions to the OSGI network:

  `description`: A simple description about your plugin.

  `version`: The actual version of your plugin, which will be displayed under the plugin menu along with the name and description. This version will be checked against the version in Marketplace to show available updates!

  `application-version`: Here you can define the minimum and maximum version of the JIRA application that is supported by your plugin. `<application-version min="7.0.0" max="7.0.5"/>` will be supported from `7.0.0` to `7.0.5`. But remember, this is only for information's sake. The plugin might still work fine in JIRA 7.1!

  `vendor`: Here you can provide details about the plugin vendor. It supports two attributes: `name` and `url`, which can be populated with the organization's name and URL, respectively. Similar to plugin key, you can populate this from the `pom.xml` file, as you would have noticed in the skeleton descriptor.

  `param`: This element can be used to define name/value attributes for the plugin. You can pass as many attributes as you want. For example, `<param name="configure.url">/secure/JTricksConfigAction.jspa</param>` defines the configuration URL for our demo plugin.

  `bundle-instructions`: Here we define the OSGI bundle instructions, which will be used by the **Maven Bundle plugin** while generating the OSGI bundle. More about this can be read under **aQute bnd** tool: `http://www.aqute.biz/Code/Bnd`. Following are the two elements in a snapshot:

Export-Package: This element defines the package in this plugin that can be exposed to other plugins. All other packages will remain private.

Import-Package: This element defines the packages that are outside this plugin but that are exported in other plugins.

- **Plugin modules**: This is the place where the various plugin modules, which we saw briefly in the earlier section, will appear.

Hopefully, you now have your plugin descriptor ready with all the necessary attributes!

Working with the Plugins1 and Plugins2 versions

In this section, let us quickly see how Plugins1 version is different from Plugins2 version.

Before we go on to the details, it is essential to understand the importance of both the versions. Prior to 4.x, JIRA used to support only Plugins1 version. So why do we need Plugins2 version?

The key motive behind version 2 plugins is to keep the plugins as a bundle, isolated from the other plugins and the JIRA core classes. It makes use of the OSGI platform (`http://www.osgi.org/`) to achieve this. While it keeps the plugins isolated, it also gives you a way to define dependencies between plugins, leaving it to the plugin developer's convenience. It even lets you import or export selected packages within the plugin, giving increased flexibility.

The fact that the version2 plugins are deployed as OSGI bundles also means that the plugins are dynamic in nature. They may be installed, started, updated, stopped, and uninstalled at any time during the running of the framework.

It is the developer's choice to go for the Plugins1 version or the Plugins2 version, depending on the nature of the plugin.

Development

Let us see the key differences between Plugins1 version and Plugins2 version at various stages of plugin development:

	Plugins1	**Plugins2**
Version	**No plugins-version element in** `atlassian-plugin.xml`**.**	**Include the** `plugins-version` **element in the** `atlassian-plugin.xml` **as follows:**`<atlassian-plugin key="${project.groupId}.${project.artifactId}" name="${project.artifactId}" plugins-version="2">`
External dependencies	Include the dependent libraries with the provided scope in your `pom.xml` file if the jars are added into `WEB-INF/lib`, or compile scope if the JARs should be embedded into the plugin.	Dependent libraries must be included in the plugin, as the plugin cannot make use of resources under `WEB-INF/lib`. This can be done in two ways: • Provide the scope in the `pom.xml` file at compile. In this case, the jars will be picked up by the Plugin SDK and added into the `META-INF/lib` folder of the plugin. • Manually add the dependent jar files into the `META-INF/lib` directory inside the plugin. You can also make your plugin dependent on other bundles. See *Managing Complex Dependencies*, in this table.
Dependency injection	Done by **Pico Container** in JIRA. All registered components can be injected directly.	Done by the plugin framework. Not all JIRA's core components are available for injection in the constructor. Use the `component-import` module to access some of the dependencies that are not directly accessible within the plugin framework. Use it also to import public components declared in other plugins.
Declaring new components	Use the component module to register new components. Once done, it is available to all the plugins.	Use the component module to register components. To make it available to other plugins, set the `public` attribute to `true`. It is `false` by default, making it available only to the plugin in which it is declared.
Managing complex dependencies	All the classes in version1 plugins are available to all other v1 plugins and JIRA core classes.	Version2 plugins allow us to optionally import/export selected packages using `bundle-instructions` in the plugin descriptor, or alternatively, by the `Import-Package/Export-Package` options while building the bundle. The **Bundle Dependency System** thus allows you to define complex dependencies between plugins, eliminating the class path conflicts and forced upgrade of plugins.

Installation

Now, let us see the key differences between Plugins1 version and Plugins2 version during plugin installation:

Plugins1	Plugins2
Plugin must be on the application classpath. Therefore, deploy it under the `WEB-INF/lib` folder.	Plugin must *not* be on the application classpath. It is loaded using the plugin framework. Hence the plugin is deployed under `${jira-home}/plugins/installed-plugins/` folder.

JIRA system plugins

In this section, we will see a brief overview of the JIRA System plugins.

A lot of JIRA's functionality is written in the form of plugins. It not only showcases what we can achieve using plugins, but also helps us, as developers, to understand how the various pieces fit together.

If it is the `atlassian-plugin.xml` file that describes the plugin functionalities, JIRA maintains the equivalent descriptions in the `*.xml` files placed under `WEB-INF/classes` folder. You will also find the related classes in the exploded folders under `WEB-INF/classes`.

Let us have a quick look at the various system plugin XMLs that can be found in `WEB-INF/classes` and the functionality they support:

System plugin XML	Functionality
`system-attachment-processor-plugin.xml`	**Defines all of the default attachment processors in JIRA. Handles the ZIP and JAR extensions.**
`system-comment-field-renderer.xml`	Handles the rendering of issue comments.
`system-contentlinkresolvers-plugin.xml`	System content link resolvers resolve parsed content links into link objects: • Attachment link resolver • Anchor link resolver • JIRA issue link resolver • User profile link resolver

System plugin XML	Functionality
`system-customfieldtypes-plugin.xml`	JIRA system custom fields are all the out-of-the-box custom fields in JIRA and the searcher associations. Examples: • Text field • Text area • User picker • Select
`system-footer-plugin.xml`	This plugin renders the content of the footer in JIRA.
`system-global-permissions.xml`	Defines all of the default **`global`** permissions in JIRA: `SYSTEM_ADMIN`, `ADMINISTER`, `USE`, `USER_PICKER`, `CREATE_SHARED_OBJECTS`, `MANAGE_GROUP_FILTER_SUBSCRIPTIONS`, and `BULK_CHANGE`.
`system-helppaths-plugin.xml`	Provides the URLs that JIRA uses for help documentation.
`system-issueoperations-plugin.xml`	System issue operations renders the issue operations using web-items grouped using web-sections. Examples: • Edit issue • Assign issue • Log work
`system-issuetabpanels-plugin.xml`	System issue tab panels renders the various tabs on the view issue page: • All tab panel • Comment tab panel • Work log tab panel • Change history tab panel
`system-issueviews-plugin.xml`	Renders the single issue view and the various search request views: • Single issue views : XML, Word, printable • Search views : XML, RSS, RSS (comments), printable, MS Word, full content, MS Excel (all fields), MS Excel (current fields), charts
`system-jql-function-plugin.xml`	Built-in JQL functions.

System plugin XML	Functionality
`system-keyboard-shortcuts-plugin.xml`	Built-in keyboard shortcuts.
`system-macros-plugin.xml`	JIRA's base system macros.
`system-project-permissions.xml`	Defines all of the default project permissions in JIRA: `ADMINISTER_PROJECTS`, `BROWSE_PROJECTS` and so on.
`system-projectroleactors-plugin.xml`	System project role actors built-in project role actors (`User Role Actor` and `Group Role Actor`) and the associated webwork actions.
`system-rendercomponentfactories-plugin.xml`	Renderer component factories plugin instantiates renderer components using the plugin system. Examples are macro renderer, link renderer, URL renderer, and so on.
`system-renderers-plugin.xml`	Built-in system renderers are as follows: • Wiki style renderer • Default text renderer
`system-reports-plugin.xml`	Built-in system reports.
`system-top-navigation-plugin.xml`	Renders the content of the top navigation bar in JIRA. Has a collection of web-items and web-sections.
`system-user-format-plugin.xml`	Renders a user in JIRA differently at different places.
`system-user-profile-panels.xml`	Renders the panels on the user profile page.
`system-webpanels-plugin.xml`	Defines web panels under the `jira-banner` location.
`system-webresources-plugin.xml`	System web resources includes static resources like JavaScript files, style sheets, and so on.
`system-workflow-plugin.xml`	System workflow conditions, functions, and validators.

In addition to using these files as a starting point for JIRA plugin development, we might sometimes end up modifying these files to override the way JIRA works. Having said that, it is not recommended to modify the core files, unless it is a business-critical change, as it will complicate JIRA upgrades. Care must be taken to upgrade all such changes when your JIRA instance is upgraded.

That concludes the lengthy introduction to the JIRA Architecture. Let us quickly move on to the recipes in this chapter. Time to code!

Stable and core APIs

Before moving on to the excitement of coding, it makes sense to mention the JIRA **stable** and **core** APIs. If you have coded or used plugins in JIRA releases prior to 5.x, you might already know that those plugins had a different version for most JIRA releases, even for the minor releases. Whenever a new JIRA version was released, developers had to frantically modify their plugins to make them compatible with the new JIRA version. And the reason behind the chaos was pretty simple: JIRA API has been evolving at a great pace!

Then came JIRA 5.0, and Atlassian introduced the stable and core APIs. And with great interest, the development community read the blog written by Rich Manalang: `http://blogs.atlassian.com/2012/03/stable-apis-yes-we-have-them/`.

The earlier `atlassian-jira` dependency, which was used in JIRA plugins prior to JIRA5, is now replaced by two different dependencies: `jira-api` and `jira-core`. `jira-api`, also known as the stable API, is a set of classes or interfaces which will not be changed without prior notice. Atlassian will maintain binary compatibility for these classes and interfaces.

`jira-core`, on the other hand, contains internal JIRA classes, and they may change without prior notice. Plugin developers are free to have dependency on these core classes, but will have to bear the risk of creating new plugin versions if those classes change across versions, even minor ones.

To use stable APIs in your plugin, the following dependency needs to be added in to the `pom.xml`:

```
<dependency>
     <groupId>com.atlassian.jira</groupId>
     <artifactId>jira-api</artifactId>
     <version>${atlassian.product.version}</version>
     <scope>provided</scope>
</dependency>
```

For core APIs, the dependency will be as follows:

```
<dependency>
     <groupId>com.atlassian.jira</groupId>
     <artifactId>jira-core</artifactId>
     <version>${atlassian.product.version}</version>
     <scope>provided</scope>
</dependency>
```

The default `pom.xml` created by **Atlassian Plugin SDK** has both the dependencies in it, but the `jira-core` dependency is commented out. Developers can uncomment the same, at the risk of breaking the code in upcoming JIRA versions, to make use of the core classes.

Before jumping into the first recipe in this chapter, take a look at the JIRA API policy at `https://developer.atlassian.com/jiradev/jira-apis/java-api-policy-for-jira`.

Modifying Atlassian bundled plugins

As we discussed earlier, more and more standard functionalities are pushed into bundled plugins as opposed to the JIRA core product. No better way to showcase the plugin architecture, it must be said!

But that does make the life of high-end users, who want to modify those functionalities, a bit difficult.

Let me take a "once used to be simple" scenario!

> *"I want to display description before issue details in the* **View Issue** *page"*

This used to be pretty easy, because all you needed to do was to modify the relevant JSP file and that was it!

But now, the View Issue screen rendering is done by the `jira-view-issue-plugin`, and it is an Atlassian system plugin. Although the actual work is simple, we only need to modify the `atlassian-plugin.xml` making those changes effective is not as simple as editing the `jira-view-issue-plugin-xxx.jar` file from the `JIRA_Home/plugins/.bundled-plugins` folder.

Let us see why!

How to do it…

The reason is pretty simple. All the system plugin JAR files under `JIRA_Home/plugins/.bundled-plugins` are overwritten by the original JAR files from `JIRA_Install_Directory/atlassian-jira/WEB-INF/classes/atlassian-bundled-plugins` folder whenever JIRA is restarted.

So, how do we override it? There is only one solution: modify the JAR files under the `atlassian-bundled-plugins` folder directly! Following are the steps, and these steps are applicable to modifying any Atlassian bundled plugins:

1. Put your customized system plugin JAR in place of the existing JAR file under the `JIRA_Install_Directory/atlassian-jira/WEB-INF/classes/atlassian-bundled-plugins` folder. You can either build the customized JAR file from its source, or just extract the JAR, modify the files, and create it back.
2. Restart JIRA.

And that's it!

Step 1 can be done easily if you have a utility like 7-Zip in Windows. Or you can use your favorite operating system's archiving capabilities to do the same thing.

And for the example we considered, all we need to do is to modify the `atlassian-plugin.xml` file inside the `jira-view-issue-plugin-xxx.jar` file. The different modules on the view issue page such as details module, description module, and so on, are web-panel plugin modules, and their order can be modified with the `weight` attribute.

By default, details module has `weight 100` and description module has weight `200`. Change the order and it is as simple as that.

How it works…

As mentioned earlier, JIRA overwrites the `JIRA_Home/plugins/.bundled-plugins` folder with the contents from the `JIRA_Install_Directory/atlassian-jira/WEB-INF/classes/atlassian-bundled-plugins` folder during startup. Now that we have the modified JAR file, it will overwrite what is there in the `.bundled-plugins` folder.

Care must be taken to migrate these changes over when JIRA is upgraded next!

See also

- The *Setting up the development environment* recipe in `Chapter 1`, *Plugin Development Process*

Converting plugins from V1 to V2

If you are upgrading a plugin from JIRA 3.13.x or earlier versions, one of the important differences is the introduction of v2 plugins. While designing the upgrade to the new JIRA version, it makes perfect sense to sometimes migrate the plugins from v1 to v2, although it is not a mandatory step. In this recipe, we will see how to convert a version1 plugin to a version2 plugin.

Getting ready

There are a couple of questions we need to ask before the plugin is converted:

- **Are all the packages used by the plugin available to OSGi plugins?**
  This is very important because JIRA doesn't expose all the packages to OSGi plugins. The list of packages exported and made available to the plugins2 framework can be found in the `com.atlassian.jira.plugin.DefaultPackageScannerConfiguration` class.
- **Are all the components used by the plugin available to OSGi plugins?**
  Similar to the previous question, we need to make sure the components are also exposed to the OSGi plugins.Unfortunately, there is no definite list provided by Atlassian to check this. To check if the components are available, use dependency injection inside the plugin. The plugin will fail in the start-up if the component is not available.

How to do it…

The actual conversion process of v1 plugins to v2 is easier than you think if the packages and the components that you have used in the plugin are available to the OSGi plugins. Here are the steps for conversion:

1. Add the `plugins-version="2"` attribute in `atlassian-plugin.xml`. This is probably the only mandatory step in the conversion process. You will be amazed to see that many of the plugins will work as is! Once added, the plugin descriptor looks like the following:

```
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
  name="Demo Plugin" plugins-version="2">
</atlassian-plugin>
```

2. Modify the source code, if required. This includes migration to the new API if you are moving to a new JIRA version with API changes, working out the changes if some of the packages/components not exported to OSGi are used in the v1 plugin, and so on.
3. Customize the package imports and exports by defining them in the bundle manifest. You can do this by using the bundle instructions we saw while explaining the `atlassian-plugin.xml` earlier in this chapter, or simply by adding the appropriate entries into the manifest file in your JAR.
   This is an optional step, which you need to do only if you want to import packages from another plugin/bundle or you want to export some of your packages to make them available to other plugins.
4. Expose your custom plugin components to other plugins using the component module. You must set the `public` attribute to `true` in the component registered in your `atlassian-plugin.xml` file. That is, `public="true"`.
5. You must import the components specifically if you want to use the components declared publicly in other plugins. Use the `component-import` module to do this:

```
<component-import key="democomponent"
  interface="com.jtricks.DemoComponent" />
```

6. You can also optionally add advanced spring configurations by adding **Spring Dynamic Modules** (**SpringDM**) configuration files (of the format `*.xml`) under the `META-INF/spring/` directory. The SpringDM loader will then load these files. The details are outside the scope of this book.

How it works…

The v2 plugin JAR file created with the Atlassian descriptor containing the required modules goes through the following journey:

1. The plugin is loaded at JIRA start-up and JIRA identifies the new JAR.
2. `DirectoryLoader` checks whether the new plugin is version2 or version1.
3. If version2, it checks for the OSGI manifest entries, which you can enter in the `MANIFEST.MF` file. If found, the plugin is installed as an OSGI bundle and started.
4. If the OSGI manifest entries are not present, JIRA uses the BND tool (`http://www.aqute.biz/Code/Bnd`) to generate the manifest entries and insert them into the `MANIFEST.MF` file.

5. It then checks for the presence of an explicit `atlassian-plugin-spring.xml`. If the file is present, the plugin is then deployed as an OSGI bundle, as in step 2.
6. If the `atlassian-plugin-spring.xml` file is absent, it then scans the `atlassian-plugin.xml` file and converts the registered components and others into OSGI references or OSGI services and creates an `atlassian-plugin-spring.xml` file.
7. Once the `atlassian-plugin-spring.xml` file is created, the plugin is deployed as an OSGI bundle and installed into the **PluginManager**.

JIRA thus gives us the flexibility to define our own custom OSGI manifest entries and references, or lets JIRA do the dirty work by defining them appropriately in the plugin descriptor.

See also

- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*

Adding resources into plugins

It is often required to add static resources such as JavaScript files, CSS files, and so on, in our plugins. To enable JIRA to serve these additional static files, they should be defined as downloadable resources.

Getting ready

A resource can be of different types. It is normally defined as a non-Java file that the plugin requires to operate.

Examples of resources that you will come across during JIRA plugin development include, but are not restricted to, the following:

- Velocity (`*.vm`) files required to render a view
- JavaScript files
- CSS files
- Property files for localization

How to do it…

To include a resource, add the resource module to the `atlassian-plugin.xml` file. The resource module can be added as part of the entire plugin or can be included within another module, restricting it just for that module.

The following are the attributes and elements available for the resource module, and their uses:

Name	Description
name	Name of the resource. This is used by the plugin or module to locate a resource. You can even define a directory as a resource by adding a trailing `/`.
namePattern	Pattern to use when loading a directory resource.
type	Type of the resource. Examples: • **download** for resources such as CSS, JavaScript, Images, and so on • **velocity** for velocity files
location	Location of the resource within the plugin jar. The full path to the file without a leading slash is required. When using `namePattern` or pointing to directory resource, a trailing `/` is required.
property (key/value)	Used to add properties as key/value pairs to the resource. Added as a child tag to resources. Example: `<property key="content-type" value="text/css"/>`
param (name/value)	Used to add name/value pairs. Added as a child tag to resources. Example: `<param name="content-type" value="image/gif"/>`

All you have to do is to add the `resource` tag to the `atlassian-plugin.xml` file, either at the plugin level or at a module level. The resource will then be available for use.

The resource definition for an image will look as follows:

```
<resource type="download" name="myimage.gif" location="includes/images/
   myimage.gif">
<param name="content-type" value="image/gif"/>
</resource>
```

A CSS file might look as follows:

```
<resource type="download" name="demostyle.css" location="com/jtricks/
  demostyle.css"/>
```

Once the resource is defined in the plugin descriptor, you can use it anywhere in the plugin. Following is how you refer to the resource.

Let us consider that you have a directory referenced as follows:

```
<resource type="download" name="images/"location="includes/images/"/>
```

A `demoimage.gif` file can be a reference in your velocity template, as follows:

```
$requestContext.baseUrl/download/resources/${your_plugin_key}:${module_key}
  /images/ demoimage.gif
```

A sample piece of code used in your plugin looks as follows:

```
<img id="demo-image" src="$requestContext.baseUrl/download/resources
  /com.jtricks.demo:demomodule/images/ demoimage.gif"/>
```

Here, `com.jtricks.demo` is the `plugin_key` and `demomodule` is the `module_key`.

More details and available values for the param element can be found at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/downloadable-plugin-resources`.

Adding web resources into plugins

The web resources plugin module, like the resource module we just saw, allows defining downloadable resources. The difference is that the web resources are added at the top of the page in the header with the cache-related headers set to never expire.

An additional advantage of using the web resources module is that we can specify the resources to be included in specific contexts within the application.

How to do it…

The `root` element for the web resource plugin module is `web-resource`. It supports the following attributes:

Name	Description
`key`	The only mandatory attribute. This should be unique within the plugin.
`state`	Indicates whether the plugin module should be `disabled` by default or not. Default value is `"enabled"`. `state="disabled"` will keep this module disabled at startup.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the web resource.
`system`	Indicates whether the plugin module is a system plugin module. This property is available only to non-OSGI plugins. Default value is `false`.

The following are the key elements supported:

Name	Description
`description`	Description of the module. `i18n` keys supported.
`resource`	All the resources to be added as web resources. See *Adding resources into plugins*.
`dependency`	Used to define dependency on the other web-resource modules. The dependency should be defined as pluginKey:web-resourceModuleKey. Example: `<dependency>com.jtricks.demoplugin:demoResource</dependency>`
`context`	Define the `context` where the web resource is available.
`condition`	Adds conditions to define whether the resource should be loaded or not.
`transformation`	Makes a particular transformer available to this web resource. Transformers are used to alter static web resources before they are batched and delivered to the browser.

We can define the `web-resource` module by populating the attributes and elements appropriately.

An example would look as follows:

```
<web-resource key="demoplugin-resources" name="demoplugin Web Resources">
    <dependency>com.atlassian.auiplugin:ajs</dependency>
    <resource type="download" name="demoplugin.css" location="/css
      /demoplugin.css"/>
    <resource type="download" name="demoplugin.js"
     location="/js/demoplugin.js"/>
    <resource type="download" name="images/" location="/images"/>
    <context>demoplugin</context>
</web-resource>
```

How it works…

When a web resource is defined, it is available for you in the plugin just like your downloadable plugin resources. As mentioned earlier, these resources will be added to the top of the page in the header section.

In your action class or servlet, you can access these resources with the help of WebResourceManager**.** Inject the manager class into your constructor and you can then use it to define the resource as follows:

```
webResourceManager.requireResource("com.jtricks.demoplugin:demoplugin-
resources");
```

The argument should be `pluginKey:web-resourceModuleKey`.



By default, all the resources under the web-resource module are served in batch mode, that is, in a single request. This reduces the number of HTTP requests from the web browser.

There's more…

Before we wind up this recipe, it is probably a good idea to identify the available contexts for web resources and also to see how we can turn off the batch mode while loading resources.

Web resource contexts

Following are the available web resource contexts for all Atlassian products:

- `atl.general`: Everywhere except administration screens
- `atl.admin`: Administration screens
- `atl.userprofile`: User profile screens
- `atl.popup`: Browser pop-up windows

JIRA supports a lot more contexts and the full list can be found at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/web-resource-plugin-module`.

You can have multiple contexts added, like this:

```
<web-resource key="demoresources" name="Demo Resources">
  <resource type="download" name="demoscript.js" location="includes
    /js/demoscript.js" />
  <context>atl.general</context>
  <context>atl.admin</context>
</web-resource>
```

You can even define a custom context here. For example:

```
<context>com.jtricks.democontext</context>
```

And then load these resources into the pages by adding the following in your page:

```
$webResourceManager.requireResourcesForContext("com.jtricks.democontext")
```

Resources added solely under a custom context will be loaded only on pages where that context is imported as shown previously. This is very helpful when you want to define resources like JavaScripts and make sure they are available only on the pages used by the plugin.

Turning off batch mode

As mentioned earlier, the resources, defined under a web resource module, are loaded in a single batch to reduce the number of HTTP requests from the browser. But if you want to switch off the batch mode for some reason, it can be achieved in two ways:

1. You can switch of batch mode `system-wide` by adding a property: `plugin.webresource.batching.off=true into jira-config.properties`.

This can have an impact on client-side performance, as batching will be switched off on all pages and for all resources.

2. It can be turned off on individual resources by adding a param element as follows:

```
<resource type="download" name="demoscript.js"
  location="includes/js/demoscript.js">
  <param name="batch" value="false"/>
</resource>
```

Building JIRA from source

One of the best things about JIRA, if you have a valid license, is that you get to see the source code. To see it, modify it, break it… err, modify it, because you have the license to do it!



Please note that Atlassian does not support building JIRA from source and it should be done only when there is a business-critical requirement that cannot be fulfilled by a third-party or custom add-on.

Getting ready

Following are some of the pre-requisites prior to building JIRA from source:

- A valid JIRA license to get access to the source code
- An environment with JDK 1.7 or higher



Apache Maven is required to build the JIRA Source, but the JIRA source code (7.x+) now includes the Maven distribution as well. If you would like to use a local Maven version, make sure you update the build scripts accordingly.
If you are building an earlier version of JIRA, make sure you have Maven installed.

How to do it…

Following are the steps to create the JIRA WAR from the source code:

1. Configure Maven if you are using a local Maven version or if you are building a version prior to JIRA 7.
2. Download the JIRA source code from `https://my.atlassian.com/download/source/jira`.
3. Extract the JIRA source archive to a local directory, which we will call `JIRA_DIR`, and navigate to it.
4. Run build.bat if on Windows, or build.sh if on Linux or Mac.
5. If the build fails with a missing dependency, download the dependency from a public Maven repository and install it into the local Maven repo. For example, **jta 1.0.1** is a dependency required by the build. Make sure it is installed into the Maven repository using the following:

```
mvn install:install-file -DgroupId=jta -DartifactId=jta
-Dversion=1.0.1 -Dpackaging=jar -Dfile=jta-1_0_1B-classes.jar
```



Make sure the dependencies are installed into the appropriate repository. For example, the embedded Maven creates its own repository under the local repo folder under `JIRA_DIR`. The missing dependencies should be added in to this repo and not in to the default repo under `~/.m2/repository` folder.

6. Repeat step 5 for every missing dependency, until the build succeeds.
7. Confirm that the WAR file is created properly under `JIRA_DEV/jira-project/jira-distribution/jira-webapp-dist/target` subdirectory.

How it works…

As you have seen, the process is pretty straightforward and Maven, the magician, does the actual build.

JIRA makes the setup a lot easier by shipping the Maven distribution, including a pre-configured `settings.xml` file, along with the source code.

Once the WAR file is created, deploy it into **Apache Tomcat 7.x**, and enjoy the power of JIRA!

There's more…

What if you want to make only a single class patch? Is it worth building the whole war if it is a small change in one of the source files? And then risk losing all the advantages of the standalone JIRA installation?

Making a single class patch

Making a single class patch is easier than you think. Following are the instructions:

1. Configure Maven and download the JIRA source code as mentioned earlier.
2. Extract the JIRA source archive to a local directory, which we will call `JIRA_DIR` and navigate to the jira-project subdirectory.
3. Create an Eclipse project or IDEA project by running one of the following:

```
mvn eclipse:eclipse
```

   Or

```
mvn idea:idea
```

4. Open the project in the IDE.
5. Make the changes in the appropriate file and compile it. The compiled class file will be available in the target/classes directory of the respective Maven module.
6. Deploy the class file to the appropriate location in the JIRA installation directory under `atlassian-jira/WEB-INF/classes` folder. For example, the compiled `com.atlassian.mail.server.managers.OFBizMailServerManager` class file will be available in the `JIRA_DIR/jira-project/jira-components/jira-core/target/classes/com/atlassian/mail/server/managers` folder. You can deploy it under the `JIRA_INSTALL/atlassian-jira/WEB-INF/classes/com/atlassian/mail/server/managers` folder.

See also

- The *Setting up the development environment* recipe in `Chapter 1`, *Plugin Development Process*

Adding new webwork actions to JIRA

Most of the time, plugin developers will find themselves writing new actions in JIRA to introduce new functionality. Usually, these actions are invoked from new web-item links configured at different places in the UI. It could also be from customized JSPs or other parts of the JIRA framework.

New actions can be added to JIRA with the help of the **webwork** plugin module.

Getting ready

Before we start, it probably makes sense to have a look at the webwork plugin module. Following are the key attributes supported:

Name	Description
`key`	A unique **key** within the plugin. It will be used as the identifier for the plugin.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the webwork action.
`roles-required`	Used to determine which roles can access this action. The permissions are the short names found in the `com.atlassian.jira.security.Permissions` class, and will only work for global-based permissions.

The following are the key elements supported:

Name	Description
`description`	Description of the webwork module.
`actions`	This is where we specify the webwork1 actions. A webwork module must contain at least one action element. It can have any number of actions.

For each webwork1 action, we should have the following attributes populated:

Name	Description
`name`	Fully qualified name of the action class. The class must extend `com.atlassian.jira.action.JiraActionSupport`.

`alias`	An alias name for the action class. JIRA will use this name to invoke the action.
`roles-required`	Used to determine which roles can access this action. The permissions are the short names found in `com.atlassian.jira.security.Permissions` class, and will only work for global based permissions.



Note that the `roles-required` attribute can be used in the parent `webwork` element, child `action` elements, or both!

The following element is supported for the `webwork1` action:

Name	Description
`view`	Delegates the user to the appropriate **view**, based on the output of the action. This element has an attribute: name that maps to the return value of the action class. The element's value is the path to the renderable view, be it using a velocity template or jsp file.

Now that you have seen the attributes and elements supported, we can have a look at a sample webwork module before proceeding to create one:

```
<webwork1 key="j-tricks-demo-action" name="JTricks Demo Action"
  i18n-name-key="j-tricks-demo-action.name">
 <description key="j-tricks-demo-action.description">
   Demo Action to showcase webwork plugin module</description>
 <actions>
  <action name="com.jtricks.jira.webwork.DemoAction" alias="DemoAction">
   <view name="success">
      /templates/j-tricks-demo-action/demoaction/joy.vm
   </view>
   <view name="input">
      /templates/j-tricks-demo-action/demoaction/input.vm
   </view>
   <view name="error">
      /templates/j-tricks-demo-action/demoaction/tears.vm
   </view>
  </action>
 </actions>
</webwork1>
```

How to do it…

Let us now aim at creating a sample webwork action. For the example, we can create an action that takes a user input, prints it out in the console, and displays it on the output page after modifying the input. It doesn't have any business logic, but the purpose is to demonstrate how the webwork plugin module works!

Following are the steps:

1. Add the new webwork action module into your `atlassian-plugin.xml`. To make it easier, you can copy the aforementioned snippet in to your plugin descriptor. You can also create it from scratch using the `atlas-create-jira-plugin-module` command in Atlassian Plugin SDK, as we saw in `Chapter 1`, *Plugin Development Process*.
2. Create the action class `DemoAction` under the package `com.jtricks.jira.webwork`. The class must extend `com.atlassian.jira.action.JiraActionSupport`. It is even easier to extend `JiraWebActionSupport`, as it already implements a lot of the required methods.
3. Identify the parameters that you need to receive from the user. Create private variables for them with the name exactly the same as that of the related HTML tag.Assuming that we are going to use a text field with the name `userName`, we need to define a string variable of the same name in our action class:

```
    private String userName;
```

4. Create setter methods for the variables that are used to get values from the input view, that is, for the variables that are submitted in the form.
   In our example, we retrieve the userName from the input view and process it in the action class. So, we need to create a setter method for that, which will look like this:

```
public void setUserName(String userName) {
 this.userName = userName;
}
```

5. Identify the parameter that needs to be printed in the output page. We can use the same `userName` variable but let us use a different variable to differentiate it from the input. Let us call it `modifiedName`:

```
private String modifiedName;
```

6. Create getter methods for the parameters to be populated in the output view. Velocity or JSPs will invoke these getter methods to retrieve the value from the Action class. For our example, we need a getter method for modifiedName, which looks as follows:

```
public String getModifiedName() {
  return modifiedName;
}
```

7. Override the methods of interest. This is where the actual logic will fit in. It is entirely up to the plugin developer to determine which methods need to be overridden. It totally depends on the logic of the plugin. The three main methods of interest are the following. But you can completely omit these methods and write your own commands and related methods, more of which we will see at the end of the recipe:

- **doValidation:** This is the method where the input validation happens. Plugin developers can override this method and add their own bits of custom validations.

- **doExecute**: This is where the action execution happens. When the input form is submitted, the doExecute method is called if there are no validation errors. All the business logic is done here and the appropriate `view` name is returned, based on the execution result.In our example, we use this method to modify the input string:

```
if (TextUtils.stringSet(userName)) {
  this.modifiedName = "Hi, "+userName;
  return SUCCESS;
 } else
 {
  return ERROR;
 }
```

- **doDefault**: This method is invoked when the "default" command is used. In our example, DemoAction!default.jspa will invoke the doDefault method.In our example, we use this method to redirect the user to the input page:

```
return "input";
```

8. Create the Velocity templates for the various views.

9. The 'input' view, in our example, uses the template: `/templates/j-tricks-demo-action/demoaction/input.vm`. Add the full HTML code as follows:

```
<html>
  <head>
    <title>J-Tricks Configuration</title>
    <meta name="decorator" content="atl.general" />
  </head>
  <body>
   <div class="aui-page-panel">
    <div class="aui-page-panel-inner">
      <section class="aui-page-panel-content">
        <h2>My Input Form</h2><br>
        <form class="aui" method="POST"
          action="$requestContext.baseUrl/secureDemoAction.jspa">
           <div class="form-body">
            <div class="field-group">
              <label for="userName">Name
               <span class="aui-icon icon-required">(required)
               </span>
              </label>
              <input class="text medium-field" type="text"
              id="userName" name="userName"
              placeholder="Enter a name">
             <div class="description">
               Please provide a name.
               Leave it blank to see error page
             </div>
            </div>
           </div>
           <div class="buttons-container form-footer">
             <button class="aui-button aui-button-primary">
               Submit
            </button>
           </div>
        </form>
      </section>
    </div>
   </div>
  </body>
</html>
```

Notice that the text field has the same name that we defined as a variable in our action class: `userName`.



The various `div` and `span` elements, with the appropriate CSS classes, are from the **Atlassian User Interface** (**AUI**) guidelines. More details on form elements can be read at `https://docs.atlassian.com/aui/latest/docs/forms.html`, and page layout at `https://docs.atlassian.com/aui/latest/docs/page-content-layout.html`.

Note that `$requestContext.baseUrl` gets you the Base URL of JIRA and will be useful if your JIRA instance has a context path. The `$requestContext` variable is populated in to the context by the JIRAActionSupport, which we have overridden.

10. Create the success view to print `modifiedName` in `/templates/j-tricks-demo-action/demoaction/joy.vm`:

```
<html>
 <head>
  <title>J-Tricks Configuration</title>
  <meta name="decorator" content="atl.general" />
 </head>
 <body>
  <div class="aui-page-panel">
   <div class="aui-page-panel-inner">
    <section class="aui-page-panel-content">
     <h2>Output</h2><br>
     <font color="green">$modifiedName</font>
    </section>
   </div>
  </div>
 </body>
</html>
```

11. Create the error view in `/templates/j-tricks-demo-action/demoaction/tears.vm`:

```
<html>
 <head>
  <title>J-Tricks Configuration</title>
   <meta name="decorator" content="atl.general" />
 </head>
 <body>
  <div class="aui-page-panel">
```

```
    <div class="aui-page-panel-inner">
     <section class="aui-page-panel-content">
      <h2>Oh No, Error!</h2><br> Please specify a user name!
     </section>
    </div>
   </div>
  </body>
 </html>
```

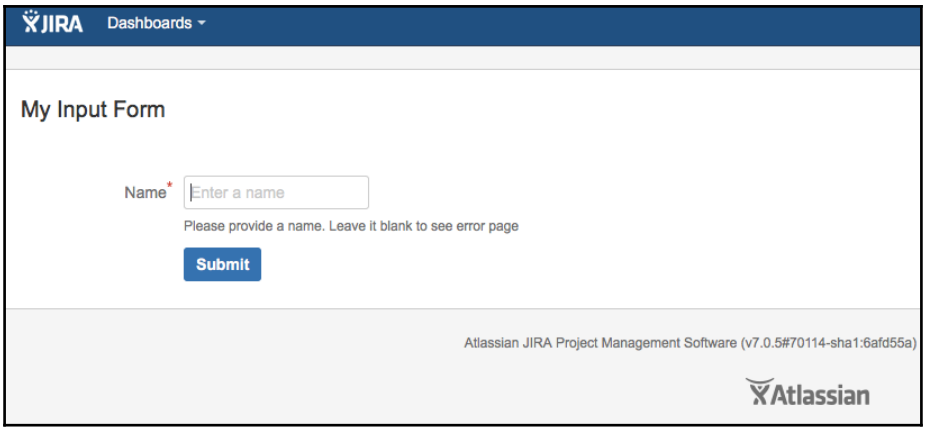
12. Package the plugin and deploy it.
13. Point your browser to `${jira_base_url}/secure/DemoAction!default.jspa`.
14. Enter some name and submit the form to see the result. Also, test the error page by submitting the form without a name.

How it works…

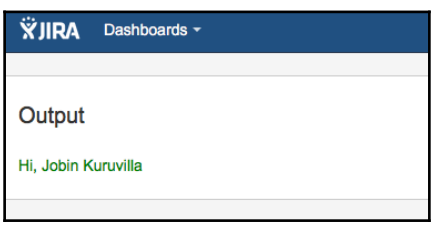
It is probably worth utilizing this section to see how the flow works in our example. Let us see it happening as a step-by-step process:

- When `${jira_base_url}/secure/DemoAction!default.jspa` is invoked, the plugin framework looks for the action `DemoAction` registered in the `atlassian-plugin.xml` file and identifies the command and view associated with it.
- Here, the default command is invoked and so the `doDefault()` method in the action class is executed.
- `doDefault()` method returns the view name as input.

The input view is resolved as `input.vm`, which presents the form to the user:

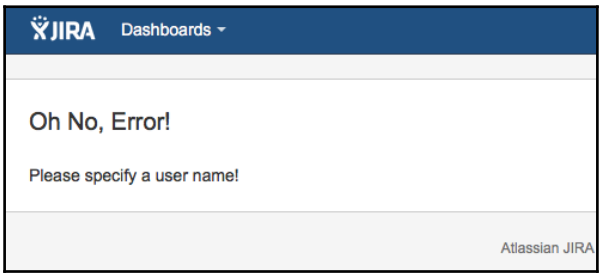


- On the form, webwork populates the `userName` value in the action class using the setter method.
- In the execution flow, first the `doValidation()` method is invoked. If no error is there, which is the case in our example, it invokes the `doExecute()` method. If there is any error in `doValidation()`, the execution stops and the user is sent back to the input (current) view. You can print the error messages appropriately on the input view, if there are any. See webwork1 documentation for details.
- In `doExecute()`, the input string, `userName`, is modified and assigned to `modifiedName`. Then success is returned. If there are any errors (for example, the `userName` is empty), the user is sent to the error view instead of success.
- The success view is resolved as `joy.vm`, where the `modifiedName` is printed, as shown here:



- `$modifiedName` will invoke the `getModifiedName()` method to print the modified name.

- If error is returned, the view is resolved as `error.vm` and the appropriate error message is shown:



Like this, we can write complex actions in JIRA that can be used to customize a lot of aspects of JIRA.

There's more…

It is also possible to add custom commands to the webwork actions, in addition to the `doExecute()` and `doDefault()` methods. This enables the developer to invoke the action using user-friendly commands, say, `ExampleAction!hello.jspa`.

Adding new commands to the action

You can write new methods like `doCommand()` in your actions class and just invoke them by `ExampleAction!command.jspa`.

For example, write a method, `doHello()`, in your action class, and you can invoke it by calling `ExampleAction!hello.jspa`.

You can also define different user-friendly aliases for these new commands and can have separate views defined for them. In this case, you will have to modify the action tag to include a `command` tag in addition to the views.

The following is a short example of how to do this. The `atlassian-plugin.xml` file should be modified to include the new command under the action:

```
<action name="com.jtricks.DemoAction" alias="DemoAction">
    <view name="input">/templates/input.vm</view>
    <view name=" success ">/templates/joy.vm</view>
    <view name="error">/templates/tears.vm</view>
    <command name="hello" alias="DemoHello">
```

```
            <view name="success">/templates/hello.vm</view>
            <view name="error">/templates/tears.vm</view>
        </command>
    </action>
```

Here, you can invoke the method by calling `DemoHello.jspa`, and returning `success` will take the user to `/templates/hello.vm` instead of `/templates/joy.vm`.

See also

- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Form token handling in webwork actions

JIRA uses a token authentication mechanism, for webwork actions, to add an extra level of security against XSRF (`Cross-site request forgery`). JIRA's core product and bundles plugins already use this in its code but it is an opt-in mechanism for custom add-ons.

In this recipe, we will see how we can modify the action plugin we wrote in the previous recipe to include this extra layer of security.

Getting ready

Develop the webwork plugin, as explained in the previous recipe.

How to do it…

Adding form token handling is pretty easy. Following are the two steps to include it in our action plugin:

1. Modify the action class to include `RequiresXsrfCheck` annotation in the methods that are executed in the action. In our class, the method is `doExecute()`. The modified method definition will look like the following:

```
@Override @RequiresXsrfCheck protected String doExecute()
throws Exception
{
   //Business Logic
```

```
    }
```

As you can see, we have added the `@com.atlassian.jira.security.xsrf.RequiresXsrfCheck` annotation in the method definition.

2. Modify the view template to include the XSRF token in the input form. In our plugin, the input form is rendered by `/templates/j-tricks-demo-action/demoaction/input.vm`. In a velocity template file, the XSRF token can be added as follows:

```
<input type="hidden" name="atl_token" value="$atl_token" />
```

The modified form element will look like the following:

```
<form class="aui" method="POST"
 action="$requestContext.baseUrl/secure/DemoAction.jspa">
 <div class="form-body">
    <div class="field-group">
     ..userName element..
    </div>
    <input type="hidden" name="atl_token" value="$atl_token" />
 </div>
 ..buttons-container..
</form>
```

If the view is rendered using a `.jsp` file, the token can be added as follows:

```
  <webwork:component name="'atl_token'" value="/xsrfToken"
template="hidden.jsp"/>
```

And that's it! Our action class is now protected against requests that don't supply a valid XSRF token.

How it works…

Once the `RequiresXsrfCheck` annotation is added in the action method, the plugin framework looks for a valid **XSRF** token in the action request, before processing the request. If the request is missing a valid token, an error is displayed to the user, as shown here:



If there is a valid token in the request, it is processed as usual.

There's more…

What if you want to send the token from other forms, actions, and so on?

Providing a token in HTML links

It is also easy to supply an XSRF token directly in an HTML link. You can do it from a velocity template, as follows:

```
/secure/DemoAction.jspa?userName=Jobin&atl_token=${atl_token}
```

You can also do it from a `.jsp` file, as follows:

```
/secure/DemoAction.jspa?userName=Jobin&atl_token=<webwork:property
value="/xsrfToken"/>
```

Getting the token programmatically

If you want to get hold of the token in a Java class, it can be done as follows:

```
import com.atlassian.jira.security.xsrf.XsrfTokenGenerator;

XsrfTokenGenerator xsrfTokenGenerator =
  ComponentAccessor.getComponent(XsrfTokenGenerator.class);
String token = xsrfTokenGenerator.generateToken(request);
```

This token can be sent as part of the request to methods protected by `RequiresXsrfCheck` annotation.

Opting out of token checking in remote calls

Scripts accessing JIRA remotely may not have an existing HTTP session or may have trouble getting hold of a valid token, as they don't have access to `XsrfTokenGenerator`. They can opt out of the token checking by providing the following HTTP header in the request:

```
X-Atlassian-Token: no-check
```

See also

- The *Adding new webwork actions to JIRA* recipe of this chapter

Capturing plugin installation/uninstallation events

We have seen plugins, which are great in terms of functionality but come with a big list of configuration steps. Much like a treadmill—great asset but very hard to assemble!

Is there a way we can handle these configurations—like creating custom fields, adding options, creating listeners, services and so on—automatically, when the plugin is installed? The answer is *Yes*.

Getting ready

Atlassian Spring Scanner is a set of libraries that make plugins faster to load and easier to develop. In this recipe, we will use Spring Scanner to write a listener class that also makes use of some **Spring** interfaces for lifecycle management.

Following are the steps to set up the environment:

1. Create a skeleton plugin, without any modules in it.
2. Add the following dependency in the `pom.xml` for the **Spring** interfaces we will use in this recipe:

```
<dependency>

   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>2.5.6</version>
   <scope>provided</scope>

</dependency>
```

3. Add the dependencies required for the Atlassian Spring Scanner. <!– This is for the annotations you need to indicate components and OSGI services. –>:

```
<dependency>
 <groupId>com.atlassian.plugin</groupId>
 <artifactId>atlassian-spring-scanner-annotation</artifactId>
 <version>${atlassian.spring.scanner.version}</version>
 <scope>compile</scope>
</dependency>
<!-- This is the runtime code for interpreting the build time
 index files -->

<dependency>
 <groupId>com.atlassian.plugin</groupId>
 <artifactId>atlassian-spring-scanner-runtime</artifactId>
 <version>${atlassian.spring.scanner.version}</version>
 <scope>runtime</scope>
</dependency>
```



The latest version of plugin SDK (6.1.0 at the time of writing this) automatically adds this in the `pom.xml`.

4. Add the **spring scanner maven plugin**. This should also be available, by default, if you are using the latest SDK. If not, add the following:

```
<plugin>
 <groupId>com.atlassian.plugin</groupId>
 <artifactId>atlassian-spring-scanner-maven-plugin</artifactId>
 <version>${atlassian.spring.scanner.version}</version>
 <executions>
  <execution>
   <goals>
     <goal>atlassian-spring-scanner</goal>
   </goals>
   <!-- process-classes seems to be skipped if you are using
    scala so perhaps use prepare-package -->
   <phase>process-classes</phase>
  </execution>
 </executions>
 <configuration>
   <includeExclude>
    +com.atlassian.jira.plugins.issue.create.*
   </includeExclude>
   <scannedDependencies>
     <dependency>
      <groupId>com.atlassian.plugin</groupId>
      <artifactId>
        atlassian-spring-scanner-external-jar
      </artifactId>
     </dependency>
   </scannedDependencies>
   <verbose>false</verbose>
 </configuration>
</plugin>
```

This plugin will scan the code and build the index files needed at runtime.

With that, we are ready to write some code.

How to do it…

It is simple. Really!

Let us look at creating a custom field automatically during a plugin installation and deleting it during uninstallation. The same logic applies for enabling/disabling of the plugin.

Following are the steps to write a listener plugin that does this:

1. Write the plugin listener class. Let us call it `com.jtricks.PluginListener`.
2. Implement the `InitializingBean` and `DisposableBean` interfaces. This will force us to implement the `afterPropertiesSet()` and `destroy()` methods.
3. Add the scanner annotations to the code, in the appropriate places. Following are the annotations we will use in this recipe:

- **@ExportAsService**: This exposes the component as an OSGI service
- **@Named or @Component**: This tells Spring that this is a singleton component
- **@ComponentImport**: This imports other OSGI components
- **@Inject or @Autowired**: This denotes the constructor that does the dependency injection

Add business logic to the `afterPropertiesSet()` and `destroy()` methods.

After writing the code, our simple listener will look like this:

```
@ExportAsService ({PluginListener.class})
@Named ("pluginListener")
public class PluginListener implements InitializingBean,
 DisposableBean {

   @ComponentImport
   private final CustomFieldManager customFieldManager;
   @ComponentImport
   private final FieldScreenManager fieldScreenManager;

   @Inject
   public PluginListener(CustomFieldManager customFieldManager,
     FieldScreenManager fieldScreenManager) {
      this.customFieldManager = customFieldManager;
      this.fieldScreenManager = fieldScreenManager;
   }

   @Override
   public void destroy() throws Exception {
      //Handle plugin disabling or un-installation here
   }

   @Override
   public void afterPropertiesSet() throws Exception {
      //Handle plugin enabling or installation here
   }
}
```

We can now add the business logic in `afterPropertiesSet()` and `destroy()` methods to create and delete the custom field:

```
@Override
public void afterPropertiesSet() throws Exception {
    //Create a list of issue types for which the
    custom field needs to be available
    List<IssueType> issueTypes = new ArrayList<IssueType>();
    issueTypes.add(null);

       //Create a list of project contexts for which the
        custom field needs to be available
   List<JiraContextNode> contexts =
    new ArrayList<JiraContextNode>();
   contexts.add(GlobalIssueContext.getInstance());

   //Add custom field
   CustomField cField =
    this.customFieldManager.createCustomField(
      TEST_TEXT_CF, "A Sample Text Field",

      this.customFieldManager.getCustomFieldType(
      "com.atlassian.jira.plugin.system
      .customfieldtypes:textfield"),

      this.customFieldManager.getCustomFieldSearcher(
     "com.atlassian.jira.plugin.system
     .customfieldtypes:textsearcher"),

    contexts, issueTypes);

   // Add field to default Screen
   FieldScreen defaultScreen = fieldScreenManager.getFieldScreen
    (FieldScreen.DEFAULT_SCREEN_ID);
   if (!defaultScreen.containsField(cField.getId())) {
      FieldScreenTab firstTab = defaultScreen.getTab(0);
      firstTab.addFieldScreenLayoutItem(cField.getId());
   }
}
@Override
public void destroy() throws Exception {
   //Get the already installed custom field by name
   CustomField cField =
   this.customFieldManager
   .getCustomFieldObjectByName(TEST_TEXT_CF);
   //Remove if not null
   if (cField != null) {
      this.customFieldManager.removeCustomField(cField);
```

```
        }
    }
```

How it works…

Atlassian Spring Scanner reads through the code, identifies the aforementioned annotations and converts them into a set of index files, which are placed in the `META-INF` directory of the plugin JAR.

These indexes are used by the Atlassian plugin system, at runtime, to invoke the services and components. Using these annotations also improves performance because they load significantly faster than traditional V2 plugins that require runtime transformation!

Since our listener is registered as a component, it will become a Spring bean when loaded and hence we can use the Spring interfaces `InitializingBean` and `DisposableBean` to capture plugin lifecycle events.

Whenever the component (`PluginListener` in our example) is registered, that is, during the enabling or installation of the plugin, the `afterPropertiesSet()` method from `InitializingBean` is invoked. Similarly, when the component is unregistered, during plugin disabling or uninstallation, the `destroy()` method from `DisposableBean` is invoked.

So, how does it help during plugin configurations?

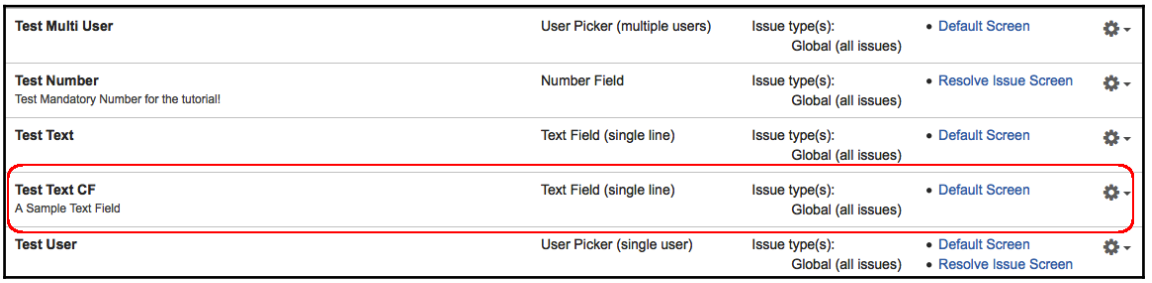
Suppose we have a plugin that needs a Text custom field for its functionality, with a predefined name. Users will have to configure this manually when they install the plugin and remove it when they uninstall it. This is also prone to manual errors. Why not do these configurations in the plugin itself?

In the preceding code, we created a text custom field named **Test Text CF** during the plugin enabling/installation and associated it with the default screen. We also added the logic to remove this field when the plugin is disabled or un-installed.



It is not always a wise idea to delete a custom field during uninstallation as it will also delete all the values associated with it. Deleting the values is not an irreversible action and can cause unexpected issues. For example, somebody might accidently disable a plugin and it will result in loss of data. The example is just to show the power of such a simple code.

As soon as the plugin is installed, a custom field is automatically created, as shown in the following screenshot:



Test Multi User	User Picker (multiple users)	Issue type(s): Global (all issues)	• Default Screen
Test Number Test Mandatory Number for the tutorial!	Number Field	Issue type(s): Global (all issues)	• Resolve Issue Screen
Test Text	Text Field (single line)	Issue type(s): Global (all issues)	• Default Screen
Test Text CF A Sample Text Field	Text Field (single line)	Issue type(s): Global (all issues)	• Default Screen
Test User	User Picker (single user)	Issue type(s): Global (all issues)	• Default Screen • Resolve Issue Screen

Test the various scenarios by uninstalling, installing, disabling, and enabling the plugin to see the custom field disappearing and reappearing!

See also

- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

3
Working with Custom Fields

In this chapter, we will cover:

- Writing a simple custom field
- Custom field searchers
- Dealing with custom fields on an issue
- Programming custom field options
- Overriding validation of custom fields
- Customizing change log value
- Migrating from one custom field type to another
- Making custom fields sortable
- Displaying custom fields on subtask columns on parent issue
- User and date fields
- Adding custom fields to notification mails
- Adding help text for a custom field
- Removing the “none” option from a select field
- Making the custom field project importable
- Changing the size of a text area custom field

Introduction

For an issue tracking application, the more details you can provide about an issue, the better. JIRA helps by giving us some standard issue fields that are most likely to be used while creating an issue. But what if we need to capture additional information such as the name of the reporter's dad, or something else that is worth capturing, perhaps the SLA or the estimated costs? For this, we can make use of custom fields.

With JIRA comes a group of predefined custom field types. It includes types like `Number Field`, `User Picker`, and so on, which are most likely to be used by JIRA users. But as you become a power user of JIRA, you might come across the need for a customized field type. That is where people start writing custom field plugins-to create new field types or custom searchers.

We will use this chapter to learn more about custom fields.

Writing a simple custom field

In this recipe, we will see how to write a new custom field type. Once created, we can create a number of custom fields of this type on our JIRA instance that can then be used to capture information on the issues.

New custom field types are created with the help of the `customfield-type` module. The following are the key attributes and elements supported:

Attributes:

Name	Description
key	This should be unique within the plugin.
class	This must implement the `com.atlassian.jira.issue.customfields.CustomFieldType` interface.
i18n-name-key	The localization key for the human-readable name of the plugin module.
name	Human-readable name of the web resource.

Elements:

Name	Description
description	Description of the custom field type.
resource type=”velocity”	Velocity templates for the custom field views.
Valid-searcher	From JIRA 5.2, you can define searcher in the custom-field definition itself. Most useful when we have a new custom field type that wants to make use of JIRA core searchers. It has two attributes: `package`—the key of the Atlassian plugin where the custom field searcher resides, and `key`—the module key for the custom field searcher.

Getting ready

Before we start, create a skeleton plugin. Next, create an eclipse project using the skeleton plugin, and we are good to go!

How to do it…

In this recipe, let us look at an example custom field type to ease understanding. Let us consider the creation of a read only custom field that stores the name of the user who last edited the issue. It is simple in functionality and enough to explain the basic concepts.

The following are the major steps to do:

1. Modify the `atlassian-plugin.xml` file to include the `customfield-type` module. Make sure the appropriate class name and views are added.
   For our example, the modified `atlassian-plugin.xml` will look as follows:

```
<customfield-type name="Read Only User CF"
   i18n-name-key="read-only-user-cf.name"
   key="read-only-user-cf"
   class="com.jtricks.jira.customfields.ReadOnlyUserCF">
   <description key="read-only-user-cf.description">
     Read only custom field to store the name of
     the last edited user
   </description>
   <resource name="view" type="velocity"
    location="templates/com/jtricks/view-readonly-user.vm"/>
   <resource name="column-view" type="velocity"
```

```
     location="templates/com/jtricks/view-readonly-user.vm"/>
    <resource name="xml" type="velocity"
     location="templates/com/jtricks/view-readonly-user.vm"/>
    <resource name="edit" type="velocity"
     location="templates/com/jtricks/edit-readonly-user.vm"/>
</customfield-type>
```

2. Make sure the `key` is unique inside the plugin.
3. Implement the class. As mentioned in the introduction, the class must implement the `com.atlassian.jira.issue.customfields.CustomFieldType`. Interface. If you do this, make sure you implement all the methods in the interface.



An easier way is to override some of the existing custom field implementations, which are similar to the type you are developing. In such cases, you will only need to override certain methods or maybe just modify the velocity templates!
The details on existing implementations can be found at the Javadocs for the `CustomFieldType` interface. `NumberCFType`, `DateCFType`, `UserCFType`, and so on, are some useful examples.

In our example, the class is `com.jtricks.jira.customfields.ReadOnlyUserCF`. Now, our field type is nothing but a text field in essence, and so it makes sense to override the already existing `GenericTextCFType`.

Following is how the class will look:

```
@Scanned
public class ReadOnlyUserCF extends GenericTextCFType {

 private final JiraAuthenticationContext
  jiraAuthenticationContext;
 protected ReadOnlyUserCF(
   @ComponentImport
     CustomFieldValuePersister customFieldValuePersister,
   @ComponentImport
     GenericConfigManager genericConfigManager,
   @ComponentImport
     TextFieldCharacterLengthValidator
       textFieldCharacterLengthValidator,
  @ComponentImport
     JiraAuthenticationContext jiraAuthenticationContext) {
       super(customFieldValuePersister, genericConfigManager,
       textFieldCharacterLengthValidator,
```

```
            jiraAuthenticationContext);
        this.jiraAuthenticationContext = jiraAuthenticationContext;
    }
}
```

As you can see, the class extends the `GenericTextCFType` class. We perform constructor injection to call the super class constructor. All you need to do is add the required components as arguments in the public constructor of the class and Spring will inject an instance of those components at runtime. Here, in addition to injecting the arguments required by super class, we have also assigned the `JiraAuthenticationContext` argument to a local variable, named `jiraAuthenticationContext`, since we need this for our implementation.



Note that we have used the Atlassian Spring Scanner annotations, details of which can be found at `https://bitbucket.org/atlassian/atlassian-spring-scanner`. In our class, `@Scanned` is used to indicate that the class needs build-time attention but it is not a component itself. `@ComponentImport` imports the OSGi components.

4. Implement/override the methods of interest. As mentioned earlier, implement all the required methods, if you are implementing the interface directly.
   In our case, we are extending the `GenericTextCFType` class and so we need to override only the required methods. The only method that we override here is the `getVelocityParameters` method where we populate the velocity params with additional values. In this case, we will add the current user's name to the velocity params and will later use this in the velocity context to generate the views. The same method is used in creating the different views in different scenarios, that is, create, edit, and so on.
   Following is the code snippet:

```
@Override
public Map<String, Object> getVelocityParameters(
  final Issue issue, final CustomField field,
  final FieldLayoutItem fieldLayoutItem) {
     final Map<String, Object> params =
     super.getVelocityParameters(issue, field, fieldLayoutItem);
     params.put("currentUser",
     jiraAuthenticationContext.getLoggedInUser().getName());
  return params;
}
```

5. Create the templates defined in the `atlassian-plugin.xml` file. The templates should be written in the way you want the fields to appear in different scenarios. If you take a closer look, we have defined four velocity resources, but using only two velocity template files, as the `view-readonly-user.vm` is shared across "view", "column-view", and "xml" resources.



Each of the resource names is used in different contexts. "view" is used while displaying the custom field on the issue details page. "column-view" is used while displaying the custom field in the issue navigator's list view. "xml" is used while exporting the issue as an XML from the issue navigator.

6. In this example, we only need to show the `readonly` field in all the three mentioned cases, and so the template will look as follows:

```
$!value
```

7. This code uses velocity syntax, the details of which can be found at `http://velocity.apache.org/engine/devel/developer-guide.html`. All we are doing here is to display the existing value of the field, on the current issue.
8. The edit template should be a read-only text field with `id` as the custom field's ID, as JIRA uses this to store values back into the database when the issue is edited. The template looks as follows:

```
#disable_html_escaping()
#customControlHeader ($action $customField.id
$customField.name $fieldLayoutItem.required
$displayParameters $auiparams)
    <input class="text" id="$customField.id"
     name="$customField.id" type="text"
     value="$!currentUser" readonly="readonly"/>
#customControlFooter ($action $customField.id
$fieldLayoutItem.fieldDescription
$displayParameters $auiparams)
```

9. This is very similar to the edit template used by JIRA's text field. Here we use the field `currentUser`, which we added into the velocity context in step 4. The value of the text field is set as `$!currentUser`. The ID of the field is `$customField.id` and the `readonly` attribute is present to make it read-only. Also notice the control header and footer, which provide the usual look and feel.
10. Package the plugin and deploy it!

Remember, more complex logic and beautification can go into the class and velocity templates. As they say, the sky is the limit!



When you are doing constructor injection, **Atlassian Spring Scanner** annotations need to be used if you are using the spring scanner plugin in the pom.xml. More details of this can be found at `https://bitbucket.org/atlassian/atlassian-spring-scanner`, and another example of this can be found in the *Capturing plugin installation/un-installation events* recipe in `Chapter 2`, *Understanding the Plugin Framework*. If the spring scanner plugin is not used, constructor injection will work without the annotations, but the component import for certain components might need a `component-import` definition in the `atlassian-plugin.xml`.

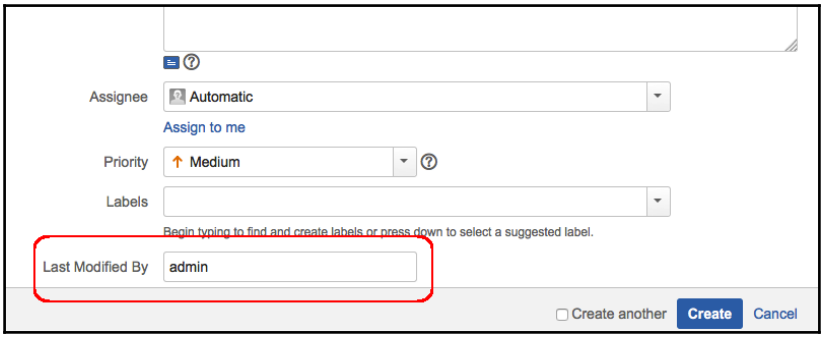
How it works…

Once the plugin is installed, it is available under **Administration** | **Issues** | **FIELDS** | **Custom Fields**.

Create a new custom field of the type we just created, "Read Only User CF", and map it into the appropriate issue types and projects. Also add the fields to the appropriate screens. Once done, the field will be available on the issue at the appropriate places.

More details on adding a custom field can be found at `https://confluence.atlassian.com/jira/adding-a-custom-field-185729521.html`.

In our example, whenever an issue is edited, the name of the user who is editing the issue is stored as a read only value in the custom field, as shown in the following screenshot:



There's more…

You might have noticed that we added only one parameter in the velocity context, that is, `currentUser`, but we have used `$!value` in the view template. Where does this variable `value` come from?

JIRA already populates the custom field velocity contexts with some existing variables, in addition to the new ones we add. `value` is just one among them, and the full list can be found at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/custom-field-plugin-module/custom-field-velocity-context-unwrapped`.

You may notice that `authcontext` is already available in the velocity context and so we could have implemented this example by getting the current user in the velocity template itself, instead of injecting the `JiraAuthenticationContext` in the constructor of the class and getting the `currentUser` variable from it in the class. But we have done that just for the purpose of explaining the example.

See also

- The *Creating a skeleton plugin* recioe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Custom field searchers

Writing the custom field type is one thing, but making it available to one of JIRA's most powerful functionalities, Search, is another! When you create the custom field, you can associate the searcher to be used along with it.

In most cases, you wouldn't need a custom searcher. Instead, you can use the built-in custom field searchers in JIRA itself. The list includes, but is not restricted to, Text Field Searcher, Date Searcher, Number Searcher, User Searcher, and so on.

The first step, of course, is to determine what kind of Searcher your new field needs. For example, a text field can easily be searched with Text Searcher. A number field can be searched with Number Searcher or Number Range Searcher. You might even want to extend one of these searchers to add some extra functionality, like some special conditions or hacks you want to introduce!

Here is how JIRA has defined the Text Searcher for its system custom fields:

```
<customfield-searcher key="textsearcher" name="Free Text Searcher"
 i18n-name-key="admin.customfield.searcher.textsearcher.name"
 class="com.atlassian.jira.issue.customfields.searchers.TextSearcher">
  <description key="admin.customfield.searcher.textsearcher.desc">
    Search for values using a free text search.
  </description>
  <resource type="velocity" name="search"
   location="templates/plugins/fields/edit-searcher
   /search-basictext.vm"/>
  <resource type="velocity" name="view" location="templates/plugins
   /fields/view-searcher/view-searcher-basictext.vm"/>
  <valid-customfield-type
   package="com.atlassian.jira.plugin.system.customfieldtypes"
   key="textfield"/>
  <valid-customfield-type
   package="com.atlassian.jira.plugin.system.customfieldtypes"
   key="textarea"/>
  <valid-customfield-type
   package="com.atlassian.jira.plugin.system.customfieldtypes"
   key="readonlyfield"/>
</customfield-searcher>
```

As you can see, it makes use of the `customfield-searcher` module. The custom fields that should be searchable using this `Free Text Searcher` should be added under the `valid-customfield-type` tag.

An alternative way of doing this is to use the `valid-searcher` tag in the `customfield-type` module, which we will see in our example.

First, let us take a look at the `customfield-searcher` module. Following are the key attributes and elements supported by this module:

Attributes:

Name	Description
`key`	This should be unique within the plugin
`class`	Must implement the `com.atlassian.jira.issue.customfields.CustomFieldSearcher` interface
`i18n-name-key`	The localization key for the human-readable name of the plugin module
`name`	Human-readable name of the web resource

Elements:

Name	Description
`description`	Description of the custom field searcher module
`resource type="velocity"`	Velocity templates for the custom field searcher views
`valid-customfield-type`	Defines the custom field types this searcher can apply to. It has two attributes: `package`–the key of the atlassian plugin where the custom field resides, and `key`–the module key for the custom field type.

Let us see in detail how to define a searcher for the custom field we wrote in the previous recipe.

Getting ready

Make sure you have created the `Read Only User` custom field (`com.jtricks.jira.customfields.ReadOnlyUserCF`) using the previous recipe.

How to do it…

As usual, we will do it as a step-by-step procedure:

1. Add the `customfield-searcher` module into the `atlassian-plugin.xml` file. In our example, the field is a read-only text field that holds the username, and so it makes sense to use the existing `TextSearcher` instead of writing a new Searcher class. The module will look as follows:

```
<customfield-searcher key="readonly-user-searcher"
 name="Read Only User Searcher"
 class="com.atlassian.jira.issue.customfields
 .searchers.TextSearcher">
  <description key="admin.customfield.searcher.textsearcher.desc">
    Search for Read Only User using a free text search
  </description>
  <resource type="velocity" name="search"
   location="templates/plugins/fields/edit-searcher
   /search-basictext.vm"/>
  <resource type="velocity" name="view"
   location="templates/plugins/fields/view-searcher
   /view-searcher-basictext.vm"/>
  <valid-customfield-type package="com.jtricks"
```

```
        key="read-only-user-cf"/>
    </customfield-searcher>
```

Here we use `com.atlassian.jira.issue.customfields.searchers.TextSearcher` that implements the `com.atlassian.jira.issue.customfields.CustomFieldSearcher` interface. If we need to write custom searchers, the appropriate class should appear here.

We also need to define the velocity templates for `edit` and `view` scenarios.

2. Implement the custom field searcher class. In this case, we can skip this step, as we are going with the already implemented class `TextSearcher`.
   Even if we are implementing a custom searcher, it might be wise to extend an already existing searcher class and override only the methods of interest to avoid implementing everything from scratch. Having said that, it is entirely up to the developer to give a brand new implementation.
   The only mandatory thing to note is that the searcher class must implement the `com.atlassian.jira.issue.customfields.CustomFieldSearcher` interface.
3. Write the velocity templates. For a custom field searcher, there are two views, `Edit` and `view`, both of which will appear on the issue navigator.
   The `edit` template is used when the filters are created/edited. The `view` template is used when the filter is viewed or the search results are viewed in the issue navigator.
   In our example, we have used the in-built JIRA templates, but it is perfectly fine to give a custom implementation of these templates.
4. Make sure the `valid-customfield-type` tags are correctly entered.
   There is a basic, but very common, error you might make here. The `package` attribute here refers to the **atlassian-plugin** key where the custom field resides and not the Java package where the Searcher class resides! Just to make it clear, the atlassian-plugin key is the key in the first line of your `atlassian-plugin.xml`, which is populated from the `pom.xml` as `com.jtricks` in our case:

```
<atlassian-plugin key="${atlassian.plugin.key}"
  name="${project.name}" plugins-version="2">
```

This package (plugin key), along with the custom field key (`read-only-user-cf` in this case), will point to the right custom field. This would also mean that you could have the same `read-only-user-cf` in another plugin with a different plugin key!

5. Package the plugin and deploy it.

But what if we do not have any reason to implement a new searcher class? Is there a way that is easier than what we did in the preceding steps?

In the preceding case, we have used an existing JIRA Searcher class and the same thing can be done in a single step, using `valid-searcher` tag, as shown here:

- Identify the searcher to use and add it using `valid-searcher` tag under the `customfield-type` module. In our case, we will select `textsearcher`. The updated `atlassian-plugin.xml` definition will look like the following.

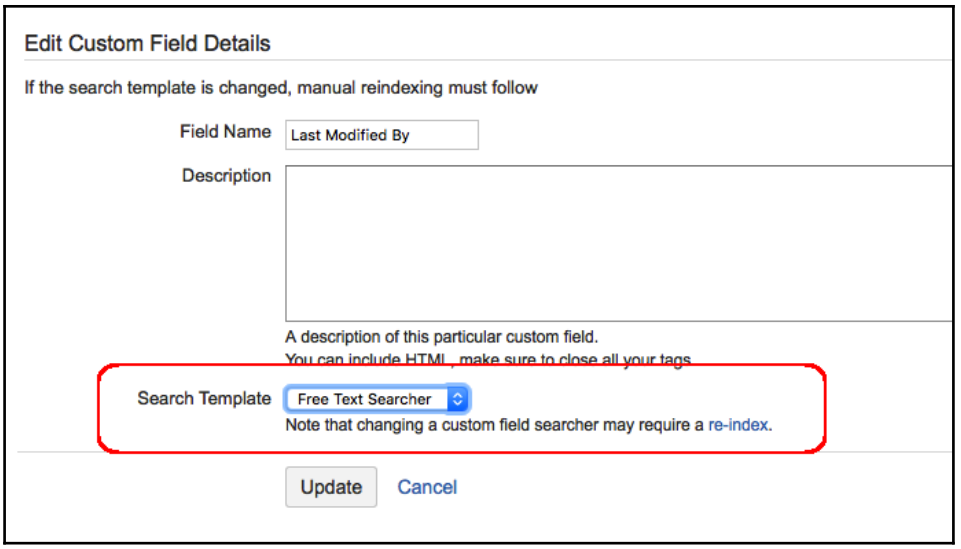
```
<customfield-type name="Read Only User CF"
 i18n-name-key="read-only-user-cf.name"
 key="read-only-user-cf"
 class="com.jtricks.jira.customfields.ReadOnlyUserCF">
  <description key="read-only-user-cf.description">
    Read only custom field to store the name of the
    last edited user</description>
  <resource name="view" type="velocity"
    location="templates/com/jtricks/view-readonly-user.vm"/>
  <resource name="column-view" type="velocity"
    location="templates/com/jtricks/view-readonly-user.vm"/>
  <resource name="xml" type="velocity"
    location="templates/com/jtricks/view-readonly-user.vm"/>
  <resource name="edit" type="velocity"
    location="templates/com/jtricks/edit-readonly-user.vm"/>
  <valid-searcher
    package="com.atlassian.jira.plugin.system.customfieldtypes"
    key="textsearcher"/>
</customfield-type>
```

Notice the new `valid-searcher` tag and how it makes it a lot simpler!

But if you want to extend a searcher or write your own searcher, please follow the steps we discussed originally, that is, using the `customfield-searcher` module.

How it works…

Once the custom field type is associated with a searcher using the `customfield-searcher` module or
`valid-searcher` element, you will see it appear in the searcher drop-down when a custom field of that type is edited:



Once the searcher is changed, a re-indexing must be done for the changes to be effective.

It is possible to have more than one custom field using the same searcher. This can be achieved by defining more than one custom field using the `valid-customfield-type` element in the `customfield-searcher` module, or by using the same searcher in `valid-searcher` element by more than one `customfield-type` definition.

Similarly, a single custom field type can have more than one searcher. This can be achieved by defining the same custom field type under more than one `customfield-searcher` module, or by using more than one `valid-searcher` element in the `customfield-type` module.

Once the searcher is defined against the custom field, you can see it appearing in the issue navigator *when the correct context is selected*. The last part is extremely important because the field will be available to search only when the context chosen is correct. For example, if field X is available only on bugs, it won't appear on the issue navigator when the issue type selected has both bugs and new features. Refresh the search menu after the correct context is selected to see your field. This is applicable only for *simple* searching.

There's more…

With the introduction of v2 plugins, courtesy of OSGI bundles, referring to the built-in JIRA searcher classes directly in the `atlassian-plugin.xml` file will fail sometimes because it can't resolve all the dependencies (the notorious unsatisfied dependency errors!). This is because some of the classes are not available for dependency injection in the version 2 plugins, as they were in version 1 plugins.

But there is an easy hack to do it. Just create a dummy custom Searcher class with the constructor that does the dependency injection for you:

```
public class MySearcher extends SomeJiraSearcher {
   public MySearcher(PluginComponent ioc) {
       super(ioc, ComponentManager.getInstanceOfType(anotherType));
   }
}
```

If that doesn't work either, add the field to the `system-customfield-types.xml` file under `WEB-INF/classes` along with the JIRA system custom fields, that is, one more `valid-customfield-type` entry into the relevant `customfield-searcher` element. If you do this, remember to apply this workaround when JIRA is upgraded!

Dealing with custom fields on an issue

In this recipe, we will see how to work with custom fields on an issue. It covers reading a custom field value from an issue and then updating the custom field value on the issue, with and without notifications.

Getting ready

Identify the places where the custom fields needs to be manipulated, be it on a listener, workflow element, or somewhere else in our plugins.

How to do it…

We will see how to access the value of a custom field and modify the value as we go along.

The following are the steps to read the custom field value from an `Issue` object:

1. Create an instance of the `CustomFieldManager` class. This is the manager class that does most of the operations on custom fields. There are two ways to retrieve a manager class:

   a. Inject the manager class in the constructor of your plugin class implementation.

   b. Retrieve the `CustomFieldManager` directly from the `ComponentAccessor` class. It can be done as follows:

   ```
   CustomFieldManager customFieldManager =
    ComponentAccessor.getCustomFieldManager();
   ```

2. Retrieve the `customField` object using the `customfield` name or the ID:

   ```
   CustomField customField =
     customFieldManager.getCustomFieldObject(new Long(10000));
   ```

   The following can also be used:

   ```
   CustomField customField =
     customFieldManager.getCustomFieldObjectByName(demoFieldName);
   ```

   Note that both ways have advantages and disadvantages. Custom field `name` is most likely to change, as an administrator can change the name in the user interface. If you use the `getCustomFieldObjectByName` method in the code, you will have to update the code every time the name is changed. `id`, on the other hand, will not change that often.



   ID can, however, change when you move the fields across environments, or if you manually create the field in different instances. If you use `getCustomFieldObject` method in a plugin and the plugin is deployed in different JIRA instances, each environment might have a different custom field ID, which can result in undesired problems.

   It is always better to make the `id` or `name` configurable in a plugin and use the configured value to make sure the plugin is generic enough to withstand a `name` or `id` change.

3. Once the custom field object is available, its value for an issue can be retrieved as follows:

```
Object value = issue.getCustomFieldValue(customField);
```

4. Cast the value object to the appropriate class. For example, `String` for a text field, `Option` for a Select field, `List<Option>` for a Multi Select field, `Double` for a number field, and so on.

If you want to update the custom field values back on to the issue, continue with the following steps.

1. Create a modified value object with the old value and the new value:

```
ModifiedValue modifiedValue =
  new ModifiedValue(value, newValueObject);
```

Obtain the `FieldLayoutItem` associated with the custom field for this issue:

```
FieldLayoutManager fieldLayoutManager =
 ComponentAccessor.getFieldLayoutManager();
FieldLayoutItem fieldLayoutItem =
 fieldLayoutManager.getFieldLayout(issue)
  .getFieldLayoutItem(customField);
```

2. Update the custom field value for the issue using the `fieldLayoutItem`, `modfiedValue` and a default change holder:

```
customField.updateValue(fieldLayoutItem, issue,
  modifiedValue, new DefaultIssueChangeHolder());
```

The advantage of doing this, or the disadvantage, depending on your perception, is that the custom field value change will not send the update event or log the change history.

If you want to trigger the update event and capture the change, please complete the following steps:

1. Initialize `IssueService` and create an `IssueInputParameters` object:

```
IssueService issueService = ComponentAccessor.getIssueService();
IssueInputParameters issueInputParameters =
  issueService.newIssueInputParameters();
```

2. Add the new custom field value into the input parameters map:

```
issueInputParameters.addCustomFieldValue(customField.getIdAsLong(),
```

```
 "New Value with event");
```

3. Update the issue using `IssueService`:

```
ApplicationUser loggedInUser =
 ComponentAccessor.getJiraAuthenticationContext().getLoggedInUser();
UpdateValidationResult validationResult =
 issueService.validateUpdate(loggedInUser, issue.getId(),
 issueInputParameters);
if (validationResult.isValid()){
  IssueResult result = issueService.update(loggedInUser,
    validationResult);
  if (result.isValid()){
    Systen.out.println("New value Set!");
  }
}
```



Note that the value passed to the custom field via `IssueInputParameters` map must be in the format the field expects.

More about `IssueService` will be explained in `Chapter 7`, *Programming Issues*. Also see `http://docs.atlassian.com/jira/latest/com/atlassian/jira/bc/issue/IssueService.html`.

This will throw an `issue updated` event and all the handlers will be able to pick up the event, as expected.

How it works…

The following are the things taking place in the backend when the custom field value is changed using one of the aforementioned methods:

- The value is updated in the database
- Indexes are updated to hold the new values

When the `IssueService` is used, the following also takes place:

- All the screen validations are done
- An *Issue Updated* event is fired, which in turn fires notifications and listeners
- A change record is created and the change history is updated with the latest changes

See also

- The *Working with issues* recipe in `Chapter 9`, *Remote Access to JIRA*

Programming custom field options

We have seen how to create a custom field type, search for it, and read/update its value from/on an issue. But one important aspect of multi-valued custom fields, and one that we haven't seen yet, is custom field options.

On a multi-valued custom field, the administrator can configure the allowed set of values, also called `options`. Once the options are configured, users can only select values within that set of options, and validation is done to ensure that this is the case.

So, how do we programmatically read those options, or add a new option to the custom field so that it can be later set on an issue? Let us have a look at that in this recipe.

Getting ready

Create a multivalued custom field, say `X`, in your JIRA instance. Add a few options to the field `X`.

How to do it…

To deal with custom field options, Atlassian has written a manager class named `OptionsManager`.

Here are the steps to get the options configured for a custom field:

1. Get an instance of the `OptionsManager` class. Similar to any other manager class, this can be done in two ways:

    a. Inject the manager class in the constructor

    b. Directly get an instance from the `ComponentAccessor` class as:

```
optionsManager = ComponentAccessor.getOptionsManager();
```

2. Retrieve the field configuration schemes for the custom field.

   There could be more than one field configuration scheme for a custom field, each with its own set of projects, issue types, and so on, defined in different contexts:

```
FieldConfigSchemeManager fieldConfigSchemeManager =
  ComponentAccessor.getComponent(FieldConfigSchemeManager.class);
List<FieldConfigScheme> schemes =
  fieldConfigSchemeManager.getConfigSchemesForField(customField);
```

   Identify the field configuration scheme of interest to us, by checking projects, issue types, or some other criteria.

3. Retrieve the field configuration from the scheme identified previously:

```
FieldConfig config = fieldConfigScheme.getOneAndOnlyConfig();
```

   If you have the issue object, you can replace step 2 and step 3 with a single step:

```
FieldConfig config = customField.getRelevantConfig(issue);
```

4. Once the field configuration is available, we can use it to retrieve the options on the custom field for that field configuration.

   The options could be different for different contexts, and that is the reason why we retrieve the `config` first and use it to get the options:

```
Options options = optionsManager.getOptions(config);
List<Option>existingOptions = options.getRootOptions();
```

   Take a look at the following options:

- `option.getValue()` will give the name of the option while iterating on the preceding list
- `option.getChildOptions()` will retrieve the child options in the case of a cascading select or any other multilevel select

If you need to add new options to the list, it is again `OptionsManager` that comes to the rescue. We do it as follows:

1. Create the new option:

```
Option option = optionsManager.createOption(config, null,
 sequence, value);
```

   The first parameter is `config`, which we saw earlier. The second parameter is `parent option ID`, used in case of a multilevel custom field, such as cascading select. It will be `null` for single-level custom fields. The third parameter is `sequence`, which determines the order in which the options will appear. The fourth parameter is the actual `value` to be added as an option.

2. Add the new `option` to the list of options and update:

```
existingOptions.add(option);
optionsManager.updateOptions(existingOptions);
```

3. Deleting options is also possible like this, but we shouldn't forget to handle existing issues with those option values. `OptionsManager` exposes a lot of other useful methods to handle custom field options, which can be found in the Java docs.

See also

- The *Writing a simple custom field* recipe in this chapter

Overriding the validation of custom fields

We have seen how to write a custom field and set its options programmatically. We also discussed how the value set on a multivalued custom field is validated against its set of preconfigured options. If the value doesn't belong to it, the validation fails and the issue can't be created or updated.

But what if we have a scenario where we need to suppress this validation? What if we need to add values to an issue that don't come from its preconfigured options? Normally, you would add this to the options programmatically, as we've seen before, but what if we don't want to do this for some reason? This is just one example of when you can suppress the validation in your custom field.

Getting ready

Create your custom field, as we saw in the first recipe of this chapter.

How to do it…

All you need to do here is to suppress the validation happening in the original parent custom field if you are extending an existing custom field type such as `MultiSelectCFType`. The following is the method you should override:

```
@Override
public void validateFromParams(CustomFieldParams arg0, ErrorCollection
arg1, FieldConfig arg2) {
// Suppress any validation here
}
```

You can override this method to add any additional validation as well; it's not just for suppressing validation!

If you are writing a custom field type from scratch, you will be implementing the `CustomFieldTyp` interface. You will then need to implement the preceding method and then you can do the same thing.

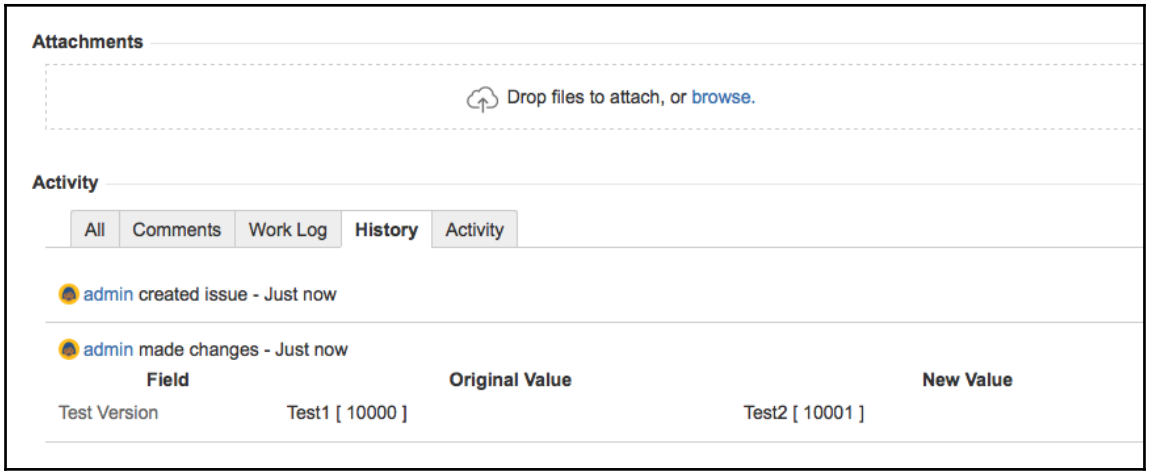
If you are interested and have access to the JIRA source code, go and have a look at how the validation is done in some of the existing custom field types!

See also

- The *Writing a simple custom field* recipe in this chapter

Customizing the change log value

One scenario we might come across when writing certain custom field types is to manipulate the way we display the change log. For a normal `Version Picker` custom field, the change log is displayed as follows:



Here, **Test Version** is the field name. The first value you see, `Test1 [10000]`, is the old value, and the second value, `Test2 [10001]`, is the new value.

Getting ready

Write your custom field type, as described in the first recipe of this chapter.

How to do it…

As you saw in the preceding screen, the change log value for both the old value and the new value are displayed in the following format:

```
change log string [change log id]
```

Both the string value and ID value are stored in the `ChangeItem` table. But before storing the value in the database, this value is generated from the individual custom fields. That is where we need to intercept to manipulate the way the change log is written.

There are two methods, one for `changelog string` and another for `changelog id`, that need to be modified. Here are the method definitions in the interface:

```
public String getChangelogValue(CustomField field, T value);
public String getChangelogString(CustomField field, T value);
```

All you need to do is implement these methods, or override them if you are extending an existing custom field type, to put in your custom implementation.

If you don't want the string to appear in the change history, just return `null` in the `getChangelogString` method. Note that if `null` is returned in the `getChangelogValue` method, the change log will not be created!

Let us consider a simple example, where the change history string is truncated if the string has more than 100 characters. In this case, the `getChangelogValue` returns an empty string and `getChangelogString` returns the truncated string. The overridden methods are as shown here:

```
@Override
public String getChangelogValue(CustomField field, T value) {
  return "";
}

@Override
public String getChangelogString(CustomField field, T value) {
  String val = (String) value;
    if (val != null && val.length() > 100){
      val = val.substring(0, 100) + "....";
    }
    return val;
}
```

How it works…

Whenever a value is changed for a custom field, it updates the value in the `CustomFieldValue` table. In addition, it also stores the changes on the issue by making a change log entry.

For every set of changes happening on an issue at a single update, a record is created under the `ChangeGroup` table. It stores the name of the user who made the change (author), the time when the change was created, and the ID (issue).

For every change group, there will be one or more change items stored in the `ChangeItem` table. It is in this table that the old and new values for fields are stored. For both the old and new value, there are two columns in the table—one for the string representation and another for the ID. The following is the entity definition for the `ChangeItem` table:

```
<!-- entity to represent a single change to an issue. Always part of a
change group -->
<entity entity-name="ChangeItem" table-name="changeitem" package-name="">
   <field name="id" type="numeric"/>
   <field name="group" col-name="groupid" type="numeric"/>
   <!-- whether this is a built in field ('jira') or a
    custom field ('custom') - basically used to avoid
    naming scope problems -->
   <!-- also used for keeping record of the bug_id of a bug
    from Bugzilla Import-->
   <!-- and for keeping record of ids in issue move-->
   <field name="fieldtype" type="long-varchar"/>
   <field name="field" type="long-varchar"/>
   <field name="oldvalue" type="extremely-long"/>
   <!-- a string representation of the new value
    (i.e. "Documentation" instead of "4" for a component
     which might be deleted) -->
   <field name="oldstring" type="extremely-long"/>
   <field name="newvalue" type="extremely-long"/>
   <!-- a string representation of the new value -->
   <field name="newstring" type="extremely-long"/>
   <prim-key field="id"/>
   <relation type="one"  rel-entity-name="ChangeGroup">
        <key-map field-name="group" rel-field-name="id"/>
   </relation>
   <index name="chgitem_chggrp">
        <index-field name="group"/>
   </index>
   <index name="chgitem_field">
        <index-field name="field"/>
   </index>
</entity>
```

The field's `oldvalue` and `newvalue` are populated using the method `getChangelogValue`. Similarly, the field's `oldstring` and `newstring` are populated using `getChangelogString`.

These fields are the ones used while displaying the change history.

Migrating from one custom field type to another

Have you been using JIRA for more than a year, or are you a power user of JIRA? That is, have you performed huge customizations, created numerous plugins, used lot of use cases, and so on? Then it is very likely that you have come across this scenario. You want to move the values from an old custom field to a new field.

JIRA doesn't have a standard way of doing this, but you can achieve this to an extent by modifying the JIRA database. However even with SQL, there are some restrictions for doing this.

The first and most important thing to check is that both the fields are compatible. You can't move the values from a text field to a number field without extra checks and validations. If there is a value such as `1234a` stored in one of the issues, it can't be stored as a number field as it is not a valid number. Another example is converting a multiline text field to a single-line text field. It is not possible, as the large value in a multiline text field will not fit into the 256 characters of a single-line text field. Similar logic applies to all the field types.

Let us see the migration of compatible types and discuss a few other scenarios in this recipe.

How to do it…

Let us assume you have two text fields, `Field A` and `Field B`. We need to migrate the values on every issue from `Field A` to `Field B`. Here are the steps that should be executed:

1. Shut down the JIRA instance.
2. Take a backup of the database. We can revert to this backup if anything goes wrong.
3. Connect to your database.
4. Execute the following SQL query:

```
Update customfieldvalue set customfield = (select id
 from customfield where cfname='Field B')
 where customfield =
(select id from customfield where cfname='Field A')
```

The query assumes that the custom field names are unique. If you have more than one custom field with the same name, use the IDs instead.

5. Commit the changes.
6. Disconnect from the database.
7. Start JIRA.
8. Re-index JIRA by going to **Administration** | **System** | **Advanced** | **Indexing**.

That should do it! Verify your changes both on the issue and in the filters.



All the SQL statements and database references are based on **Oracle 10g**. Please modify it to suit your database.

How it works…

All that we did here was change the custom field ID in the `customfieldvalue` table. The other steps are standard steps for executing any SQL in JIRA.

Remember, if you have two custom fields with the same `name`, make sure you use the correct `id` instead of finding it using the `name` in SQL.

Now, this will work fine if both the fields are of the same type. But what if you want to move the values from one type to another? This may not always be possible, because some of the values in the `customfieldvalue` table may not be compatible with other custom field types.

Let us consider migrating a normal text field to a text area custom field. The value in the text area custom field is stored as a `CLOB` in the `textvalue` column in the database. But the value in a normal text field is stored as `VARCHAR 2(255)` in the `stringvalue` column. So, when you convert, you need to update the custom field ID, read the `VARACHAR2(255)` value from the `stringvalue` column, and store it in the `textvalue` column as a `CLOB`. You also need to set the no-longer-used `stringvalue` to `null` in order to free space in the database.

In this example, if you are trying the reverse order, that is, migrating from text area to text field, you should take into consideration the length of the text, and remove the extra text, as the text field can hold only up to 256 characters.

You can find the data type for the various custom fields by looking at the `getDatabaseType` method. For a `TextField`, the method looks as follows:

```
Protected PersistenceFieldType getDatabaseType(){
    return PersistenceFieldType.TYPE_LIMITED_TEXT;
}
```

Other available field types are `TYPE_UNLIMITED_TEXT` (for example, text area), `TYPE_DATE` (date custom field), and `TYPE_DECIMAL` (for example, `number field`).

There's more…

Sometimes we just need to change the type of a custom field instead of creating a new one and then migrating the values across. Let us quickly see how to do it.

Changing the type of a custom field

In this case, the table that needs to be updated is the `CustomField` table. All we need to do is to update the `customfieldtypekey`. Just set the new custom field type key, which will be `{YOUR_ATLASSIAN_PLUGIN_KEY}:{MODULE_KEY}`.

For a text field, the key is `com.atlassian.jira.plugin.system.customfieldtypes:textfield`.

For incompatible types, we need to consider all the aforementioned cases and update the `CustomFieldValue` table accordingly.

See also

- The *Retrieving custom field details from database* recipe in `Chapter 10`, *Dealing with the JIRA Database*

Making custom fields sortable

We have seen the creation of new custom fields, writing new searchers for them, and so on. Another important feature with the fields, be it custom fields or the standard JIRA fields, is to use them for sorting. But simply writing a new custom field type won't enable sorting on that field.

In this recipe, we will see how to enable sorting on custom fields.

Getting ready

Create the new custom field type that we need to enable searching for.

How to do it…

This is easy to do. There are only two simple steps that you need to do to make sure the custom field is a sortable field:

1. Implement the `SortableCustomField` interface. A new custom field type will look like the following:

```
public class DemoCFType extends AbstractCustomFieldType
  implements SortableCustomField {
```

   If you are extending an existing custom field type such as `TextCFType`, it already implements the interface.

2. Implement the `compare` method. The following is an example:

```
public int compare(Object customFieldObjectValue1,
 Object customFieldObjectValue2, FieldConfig fieldConfig) {
  return new DemoComparator().compare(customFieldObjectValue1,
   customFieldObjectValue2);
}
```

   `DemoComparator` here is a custom comparator that we can write to implement the sorting logic.
   Just invoke `SortableCustomField.compare()` if a custom comparator is not needed.



This is actually a fallback method. If the custom field has a searcher associated with it, the sorting is done with the help of the `getSorter` method of `SortableCustomFieldSearcher`. The method returns a `LuceneFieldSorter`, which will take care of sorting. `NumericFieldStatisticsMapper` is a good implementation of `LuceneFieldSorter` if you want to get some guidance on getting started on a custom searcher.
If the custom field has no searcher associated with it, the `compare` method will be used, as explained in the recipe.

How it works…

Once the custom field implements the `SortableCustomField` interface, we can click on its header on the issue navigator, and see it getting sorted based on the logic we implemented.

There's more…

`BestNameComparator`, `FullNameComparator`, `LocaleComparator`, `GenericValueComparator`, and so on, are some reusable comparators that ship with JIRA. There is no definite list, but you will find quite a lot of them in the JIRA source, if you have access.

See also

- The *Writing a simple custom field* recipe in this chapter
- The *Making the custom field project importable* recipe in this chapter

Displaying custom fields on subtask columns

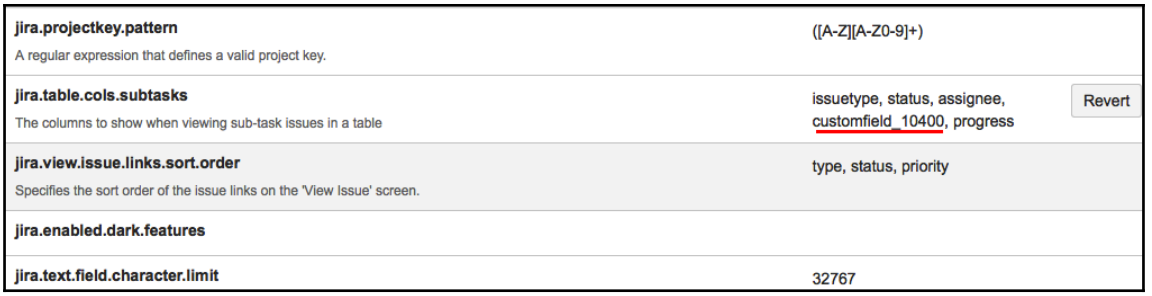
This is one of the easiest things that you can do! But it adds a lot of value at times. We are talking about adding extra columns for subtasks on the parent issue page.

We know how to add fields to subtasks in JIRA, don't we? Let us see how we can make those fields appear in the parent issue page. We will use a URL custom field as an example in this recipe.

How to do it…

Adding new columns to the subtasks view can be done easily by modifying the advanced property named `jira.table.cols.subtasks`. This property is available under **Administration** | **System** |**General Configuration** | **Advanced Settings**.

For example, if you want to add a custom field to the existing list of columns, you can do that by including the custom field ID, as shown here:



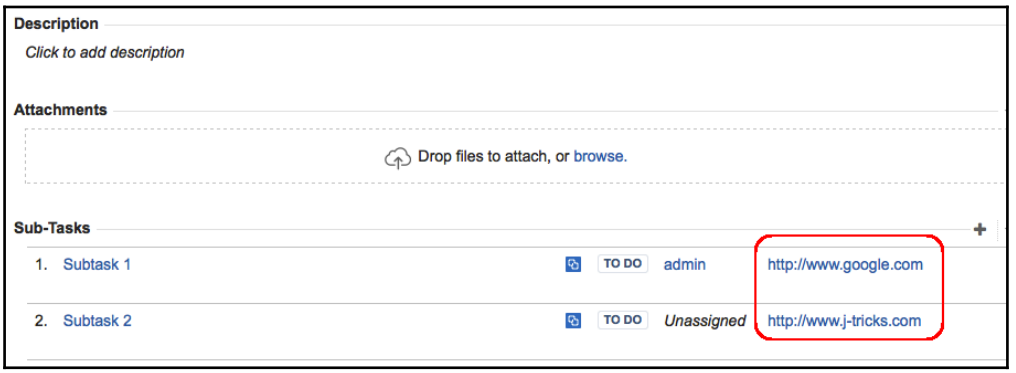
The updated property value will be `issuetype, status, assignee, customfield_10400, progress`.

Similarly, you can add the `id` of other standard or custom fields, if you want to see them in the subtask list. But care must be taken not to add too many fields or add big fields such as description, as that will clutter the parent issue page.

How it works…

JIRA renders the subtask columns on the View Issue page by looking at the aforementioned property. While adding the standard subtask fields is useful, adding custom fields can be extremely helpful sometimes.

In our example, we have added `customfield_10400`, where `10400` is the numeric ID for the custom field. It stores the URL associated with the task, as shown:



Looks useful, doesn't it?

User and date fields

JIRA boasts of a simple, clutterless UI. You will find this theme in the View Issue page, and one of the standout things is the way issue fields are arranged on the page. The fields are grouped based on their type and you will see separate sections for the user and date fields, as shown in the next screenshot:



But how do we define the custom fields that we develop as a date field or a user field?

How to do it…

When you write your new date fields or user fields, all you need to do to make it appear in the correct sections is to implement the right interface!

For a user field, the new custom field type class should implement the following interface:

```
com.atlassian.jira.issue.fields.UserField
```

For a date field, implement the following:

```
com.atlassian.jira.issue.fields.DateField
```

If you are extending the existing date fields or user fields, they already implement the interface, and hence they will appear automatically in there!

What if you do not want your field in the special date/user sections? Simply ignore these interfaces. The fields will appear just like normal custom fields and will then appear in the order specified under field configurations.

How it works…

This is quite simple. JIRA looks out for classes implementing the `UserField`/`DateField` interfaces and displays them in the respective sections. On the standard custom field section, it doesn't show these fields.

Ever wondered where this check is done in the JIRA source code? The view is rendered in the `jira-view-issue-plugin-xxx.jar`, but the actual check is done in the `util` class: `com.atlassian.jira.issue.fields.util.FieldPredicates`.

Here is the sample code for validating `Date` fields inside the `FieldPredicates` class:

```
private static final Predicate<Field> PREDICATE_DATE_FIELD = new
CustomFieldTypePredicate(DateField.class);
private static final Predicate<Field> PREDICATE_DATE_CUSTOM_FIELD =
Predicates.allOf(PREDICATE_CUSTOM_FIELD, PREDICATE_DATE_FIELD);
/**
* Return a predicate that will return true if the input field is a date
field.
*
* @return a predicate that will return true if the input field is a date
field.
*/
public static Predicate<Field> isDateField() {
    return FieldPredicates.PREDICATE_DATE_FIELD;
```

```
}
/**
* Return a predicate that will return true if the input field is a date
custom field.
*
* @return a predicate that will return true if the input field is a date
custom field.
*/
public static Predicate<Field> isCustomDateField() {
    return PREDICATE_DATE_CUSTOM_FIELD;
}
```

You will find similar code for other custom field types, including User custom fields.

See also

- The *Writing a simple custom field* recipe in this chapter

Adding custom fields to notification e-mails

One of the main features of JIRA is its capability to send notifications to selected people on selected events! It is often a requirement for JIRA users to customize these notifications, mainly to add more content in the form of custom fields.

If you understand velocity templates, adding custom fields to notification e-mails is a cakewalk, as we will see in this recipe.

Getting ready

You should know the custom field `id` that you need to add into the template. The `id` can be found in the URL that you see when you hover over the **Edit** operation on the custom field in the administration page.

How to do it…

Let us have a look at adding a custom field, `X`, into a notification e-mail when an issue is updated. The following are the steps:

1. Identify the template that needs to be updated. For each event in JIRA, you can find the template associated with it in the `email-template-id-mappings.xml`, residing under the `WEB-INF/classes` folder.
   In this case, the event is `Issue Updated`, and the matching template is `issueupdated.vm`.
   Once the template is identified, the files are present under `WEB-INF/classes/templates/email/text/` and `WEB-INF/classes/templates/email/html/`.
2. Modify the template to include the custom field name and value wherever required.
   The name of the custom field can be retrieved as follows:

```
$customFieldManager.getCustomFieldObject(
 "customfield_10010").getName()
```

   The actual value can be retrieved as follows:

```
$issue.getCustomFieldValue(
 $customFieldManager.getCustomFieldObject("customfield_10010")))
```

   In both cases, `10010` is the numeric ID of the `customfield` that we discussed before.

If there are lots of custom fields that need to be added to the templates, it probably makes sense to add a macro for it. If you look at the `WEB-INF/classes/templates/email` folder, there is a `macros.vm` file in which all the common macros are created. You can add a custom macro to that file with the required HTML tags. An example, in JIRA 7.0, is as follows:

```
#macro (printcf $cfId)
 #if
 ($issue.getCustomFieldValue($customFieldManager
  .getCustomFieldObject($cfId)))
    #set ($cfVal =
        $issue.getCustomFieldValue($customFieldManager
        .getCustomFieldObject($cfId)))
    <tr>
       <th>
         $customFieldManager.getCustomFieldObject
```

```
                ($cfId).getName():
            </th>
            <td>$cfVal</td>
        </tr>
    #end
#end
```

You can add custom fields in to the appropriate templates just by calling this macro:

```
#printcf("customfield_10010")
```

Make sure the appropriate tags are used in different JIRA versions. You can find the CSS classes and so on from the existing templates. The preceding macro just has the table rows and columns, so make sure the `printcf` command is included inside an existing `table` tag.

For example, in JIRA 7.0, you can add the preceding macro in the `changelog.vm` file under the `WEB-INF/classes/templates/email/html/includes/fields` folder. It should go inside the `table` tags, as shown here:

```
<table class="keyvalue-table">
  ....
  #printcf("customfield_11900")
</table>
```

How it works…

The e-mail notifications are rendered using velocity templates. JIRA already has a lot of objects in the velocity context, including the `customFieldManager` and `issue` objects that we have just used.

The full list of objects available in the velocity context for e-mail templates can be found in the Atlassian documentation at `https://developer.atlassian.com/jiradev/jira-platform/jira-architecture/jira-templates-and-jsps/velocity-context-for-email-templates`.

In this case, we used the `customFieldManager` object to retrieve information about the custom field, and then we used the `issue` object to retrieve its value from the issue. You are free to use any other objects or add your own HTML tags to the appropriate velocity templates to dispatch the desired notification e-mails.

Adding help text for a custom field

As our JIRA instance grows, demanding more and more information from the users through the custom fields, it becomes the norm to let the users know what we expect from them. Apart from a bunch of tutorials that we can prepare for them, it makes sense to give them some help right there on the screen, next to the field.

Let us see the various options on how to do it.

Getting ready

Make sure you have the custom field, for which the help needs to be displayed and configured properly.

How to do it…

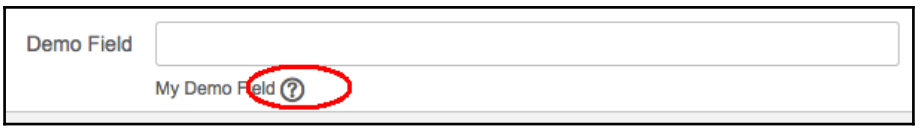
You can edit the custom field under **Administration** | **Issues** | **Fields** | **Custom Fields** to modify the description of your custom field. If you have multiple field configurations, make sure you edit the custom field description under the field configuration used by your project and issue type.

But a small description might not be enough in many cases. Following are some of the ways in which you can provide detailed help for your users.

This is just common sense. Just link to an external documentation about the field. We can do this easily by adding a few hyperlinks in the description of the custom field. And to make it even better, we can reuse some of the JIRA styles to make sure the help link appears consistent across the system, as shown here:

```
My Demo Field <a class="help-lnk" data-helplink="local"
href="http://www.j-tricks.com">
    <span class="aui-icon aui-icon-small aui-iconfont-help"></span> </a>
```

The help link will appear on the field description, as shown here:

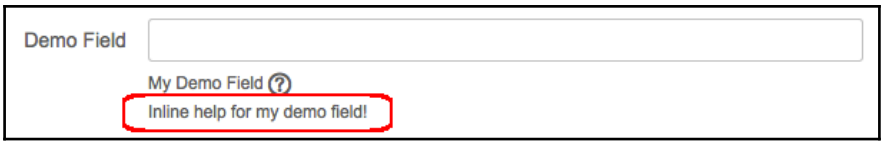


Provide inline help. This is suitable if the help isn't big enough to be put in documentation, but at the same time, you don't want them to appear along with the description of the field! In this case, we go for a little JavaScript trick, where we hide the help text under an HTML `DIV` and toggle the visibility as the user clicks on the help image.

Put the following under the field description after modifying the relevant text. Here, `My Demo Field` is the actual field description and `Inline help for my demo field!` is the extra help we added:

```
My Demo Field <span class="aui-icon aui-icon-small aui-iconfont-help"
 onclick=" AJS.$('#mdfFieldHelp').toggle();" ></span>
  <div id="mdfFieldHelp" style="display:none">
     Inline help for my demo field!
  </div>
```

On clicking on the help image, the inline help appears as shown here:



Short and sweet, right?

How it works…

JIRA, thankfully, allows HTML rendering on its description field. We have just used the HTML capabilities to provide some help for the field. It gives us lot of options and the aforementioned ones are just pointers on how to exploit it.

Removing the “none” option from a select field

If you are a JIRA plugin developer, you must have come across this feature request before. Some people just don't like the `none` option in the select fields for various reasons. One reason, obviously, is to force the users to select a valid value.

How to do it…

There are two ways to hide the `none` option on the client side, using a JavaScript hack, or on the server side, by modifying the velocity templates that renders the select field. Let us see both, in different contexts.

First, we will see how to hide the `none` option on a given custom field with the ID `10000`. All we have to do here is to add the following JavaScript snippet in the description of the field. As mentioned in the previous recipe, make sure the description is edited in the appropriate field configuration, if applicable:

```
<script type='text/javascript'>
  JIRA.bind(JIRA.Events.NEW_CONTENT_ADDED, function (e, context) {
    hideNone();
  });
  AJS.$(document).ready(function () {
    hideNone();
  });
  function hideNone(){
    AJS.$("#customfield_10000 option[value='-1']").remove();
  }
</script>
```



We need to make sure the snippet is executed on the `NEW_CONTENT_ADDED` event as well, to handle inline edits.

Once this snippet is added, the `none` option will be hidden when the custom field with ID `10000` is edited.

Now, let us see how we can modify the velocity templates to do the same thing. `Select Field` is a system custom field that uses velocity templates to render the view and edit screens. In order to remove the `none` option, we need to modify the edit template.

Note that editing the template will remove the `none` option from all custom fields of type `select`. If you want to remove it from only certain select custom fields, don't forget to add extra checks in the velocity templates to do the same.

For any system custom field, you can find the associated classes and their velocity templates from the `system-customfieldtypes-plugin.xml` file, residing under the `WEB-INF/classes` folder.

In our case, we can find the following snippet related to `select-field`:

```
<customfield-type key="select" name="Select List (single choice)"
 i18n-name-key="admin.customfield.type.select.name"
 class="com.atlassian.jira.issue.customfields.impl.SelectCFType">
    <description key="admin.customfield.type.select.desc">
     A single select list with a configurable list of options.
    </description>

    <resource type="download" name="customfieldpreview.png"
      location="/images/customfieldpreview/select.png">
        <param name="source" value="webContextStatic"/>
    </resource>

    <resource type="velocity" name="view"
      location="templates/plugins/fields/view/view-select.vm"/>
    <resource type="velocity" name="edit"
      location="templates/plugins/fields/edit/edit-select.vm"/>
    <resource type="velocity" name="xml"
      location="templates/plugins/fields/xml/xml-select.vm"/>

    <category>STANDARD</category>
</customfield-type>
```

As evident from the preceding snippet, the edit template for the select field is `templates/plugins/fields/edit/edit-select.vm`. That is the file we need to modify.

All we need to do now is to navigate to the file and remove the following lines:

```
#if (!$fieldLayoutItem || $fieldLayoutItem.required == false)
  <option value="-1">$i18n.getText("common.words.none")</option>
#else
  #if ( !$configs.default )
    <option value="">$i18n.getText("common.words.none")</option>
  #end
#end
```

The remaining code in the template *must not* be deleted. Restart JIRA to make the change effective.



The same approach can be used to remove the `none` option from other fields, such as `radio buttons`, `multi select`, `cascading select`, and so on. The actual code to remove will differ, but the approach is the same.

There's more…

There's more to it…

Reloading velocity changes without restart (auto reloading)

You can configure JIRA to reload the changes to velocity templates without a restart. To do this, you need to make two changes to the `velocity.properties` file under `WEB-INF/classes`:

1. Set the `class.resource.loader.cache` property to `false`. It is true by default.
2. Uncomment the `velocimacro.library.autoreload=true` property. This can be done by removing the `#` at the beginning of the line.

Restart JIRA and the changes to the velocity templates will be reloaded without another restart!

See also

- The *Changing the size of a text area custom field* recipe in this chapter

Making the custom field project importable

Importing projects is the default functionality in JIRA. And while importing projects, JIRA lets you copy all the issue data across, but only if it is asked to do so!

Let us see how we can make the custom field's project importable, or in simple words, inform JIRA that our fields are okay to be imported!

How to do it…

To tag our custom field as project-importable, we need to implement the following interface:

```
com.atlassian.jira.imports.project.customfield.ProjectImportableCustomField
```

You will have to then implement the following method:

```
ProjectCustomFieldImporter getProjectImporter();
```

There are existing implementations for the `ProjectCustomFieldImporter` class, such as the `SelectCustomFieldImporter` class, which we can reuse. It is in this class that we check whether the value getting imported is a valid value or not.

For example, in the case of a `select` field, we need to make sure that the value being imported is a valid option configured in the custom field on the target system. It is entirely up to the users to implement the various rules at this stage.

See the Javadocs at `http://docs.atlassian.com/jira/latest/com/atlassian/jira/imports/project/customfield/ProjectCustomFieldImporter.html` for more details on doing custom ProjectCustomFieldImporter implementations.

See also

- The *Making custom fields sortable* recipe in this chapter

Changing the size of a text area custom field

As we have discussed before, JIRA ships with some predefined custom field types. One such commonly used type is the `Text Area` field.

The `Text Area` field has a predefined width and height, which is not customizable. It is often a requirement of JIRA users to increase the size of the field, either globally, or for a particular custom field.

How to do it…

Just like any other custom fields, the `Text Area` field is also rendered using velocity templates. From the `system-customfieldtypes-plugin.xml` file, we can find out that the location of the edit template is `templates/plugins/fields/edit/edit-textarea.vm`:

```
<customfield-type key="textarea" name="Text Field (multi-line)"
.........................................
    <resource type="velocity" name="edit" location="templates/plugins/
    fields/edit/edit-textarea.vm"/>
.................................
</customfield-type>
```

If we need to increase the size, we need to modify the template to increase the `rows` or `cols` property, as per the requirements.

If we need to increase the width (number of columns) to 50 and height (number of rows) to 8, the `cols` and `rows` properties need to be updated to `50` and `8`, respectively. The template will then look like the following code:

```
#disable_html_escaping()
#customControlHeader ($action $customField.id $customField.name
$fieldLayoutItem.required $displayParameters $auiparams)
.........................................
    $!rendererParams.put("rows", "8")
    $!rendererParams.put("cols", "50")
.........................................
#customControlFooter ($action $customField.id
$fieldLayoutItem.fieldDescription $displayParameters $auiparams)
```

If this needs to be done only for a selected `customfield`, just add a condition at the beginning of the template to handle the custom field separately. The template will then look like the following lines of code:

```
#disable_html_escaping()
#customControlHeader ($action $customField.id $customField.name
$fieldLayoutItem.required $displayParameters $auiparams)
#if ($!customField.id=="customfield_10010")
  ## Modify rows and cols only for this custom field
  $!rendererParams.put("rows", "8")
  $!rendererParams.put("cols", "50")
  ........................................
#elseif ($!customField.isRenderable() && $rendererDescriptor)
 // reminder of the original snippet here
.............................................................
#customControlFooter ($action $customField.id
$fieldLayoutItem.fieldDescription $displayParameters $auiparams)
```

Hopefully, that gives you an idea about increasing the size of the `Text Area` custom field.

As usual, JIRA should be restarted to make this change effective, unless *velocity autoloading* is enabled, as we discussed in the previous recipe.

See also

The *Removing the "none" option from a select field* recipe in this chapter

4
Programming Workflows

In this chapter, we will cover the following topics:

- Writing a workflow condition
- Writing a workflow validator
- Writing a workflow post function
- Editing an active workflow
- Permissions based on workflow status
- Including/excluding resolutions for specific transitions
- Adding workflow triggers
- Internationalization in workflow transitions
- Obtaining available workflow actions programmatically
- Programmatically progressing on workflows
- Obtaining workflow history from a database
- Re-ordering workflow actions in JIRA
- Creating common transitions in workflows
- Creating global transitions in workflows

Introduction

Workflows are a standout feature in JIRA that helps users to transform JIRA into a user-friendly system. It helps users to define a lifecycle for the issues, the purpose for which they are using JIRA, and so on. As the Atlassian documentation says at `https://confluence.atlassian.com/jira/configuring-workflow-185729632.html`:

> *A JIRA workflow is the set of steps and transitions an issue goes through during its lifecycle. Workflows typically represent business processes.*

JIRA uses OpenSymphony's **OSWorkflow,** which is highly configurable and, more importantly, pluggable, to cater for the various requirements. JIRA uses four different plugin modules to add extra functionalities into its workflow, which we will see in detail throughout this chapter.

To make things easier, JIRA ships with default workflows for each use case. We can't modify the default workflows, but we can copy them into a new workflow and amend it to suit our needs. Before we go into the development aspect of a workflow, it probably makes sense to understand the various components of a workflow.

The two most important components of a JIRA workflow are **step** and **transition**. At any point of time, an **issue** will be in a step. Each step in the workflow is linked to a workflow status (`https://confluence.atlassian.com/adminjiraserver070/defining-status-field-values-749382903.html`) and it is this status that you will see on the issue at every stage. A transition, on the other hand, is a link between two steps. It allows the user to move an issue from one step to another (which essentially moves the issue from one status to another). In this chapter, the terms “transition” and “workflow action” will be used interchangeably.

Here are a few key points to remember or understand about a workflow:

- An issue can exist in only one step at any point in time.
- A status can be mapped to only one step in the workflow.
- A transition is always one way. So if you need to go back to the previous step, you need a different transition.
- A transition can optionally specify a screen to be presented to the user with the right fields on it.

OSWorkflow, and hence JIRA, provides us with the option of adding various elements into a workflow transition which can be summarized as follows:

- **Conditions**: A set of conditions that need to be satisfied before the user can actually see the workflow action (transition) on the issue
- **Validators**: A set of validators that can be used to validate the user input before moving to the destination step
- **Post Functions**: A set of actions that will be performed after the issue is successfully moved to the destination step
- **Triggers**: Events that occur in a connected development tool, such as Bitbucket Server, that trigger the transition in JIRA

These four elements give us the flexibility of handling the various use cases when an issue is moved from one status to another. Every workflow action uses these elements and might also have a workflow screen. A transition executes in the following sequence:

- **Conditions**: Workflow action is visible only if the condition is satisfied
- **Transition screens**: Workflow screen is presented to the user, if configured
- **Validators**: User inputs are validated
- **Post functions**: Post functions are executed on successful validation and status change



Note that when triggers are configured on a workflow transition, it ignores any conditions or validators configured on that transition. Post functions will be executed as always.

JIRA ships with a few built-in triggers, conditions, validators, and post functions. There are plugins out there, which also provide a wide variety of useful workflow elements. And if you still can't find the one you are looking for, JIRA lets us write them as plugins. We will see how to do this in the various recipes in this chapter.

Hopefully, that will give you a fair idea about the various workflow elements. A lot more on JIRA workflows can be found in the JIRA documentation at `https://confluence.atlassian.com/jira/configuring-workflow-185729632.html`.

Writing a workflow condition

What are workflow conditions? They determine whether a workflow action is available or not. Considering the importance of a workflow in business processes and how there is a need to restrict the actions, either to a set of people (groups, roles, and so on) or based on some criteria (for example, the field is not empty), writing workflow conditions is almost inevitable.

Workflow conditions are created with the help of the `workflow-condition` module. The following are the key attributes and elements supported. Visit `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/workflow-plugin-modules#WorkflowPluginModules-Conditions` for more details:

Attributes:

Name	Description
`key`	This should be unique within the plugin.
`class`	Class to provide contexts for rendered velocity templates. It must implement the `com.atlassian.jira.plugin.workflow.WorkflowPluginConditionFactory` interface.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the workflow condition.

Elements:

Name	Description
`description`	**Description of the workflow condition.**
`condition-class`	Class to determine whether the user can see the workflow transition. It must implement `com.opensymphony.workflow.Condition`. It is recommended to extend the `com.atlassian.jira.workflow.condition.AbstractJiraCondition` class.
`resource type="velocity"`	Velocity templates for the workflow condition views.

Getting ready

As usual, create a skeleton plugin. Then create an eclipse project using the skeleton plugin and we are good to go!

How to do it…

In this recipe, let's assume we are going to develop a workflow condition that limits a transition only to the users belonging to a specific project role. The following are the steps to write our condition:

1. Define the inputs needed to configure the workflow condition.
   We need to implement the `WorkflowPluginFactory` interface, which mainly exists to provide velocity parameters to the templates. It will be used to extract the input parameters that are used in defining the condition.
   To make it clear, the inputs here are not the inputs used while performing the workflow action, but the inputs used in defining the condition.The condition factory class, `RoleConditionFactory` in this case, extends the `AbstractWorkflowPluginFactory`, which implements the `WorkflowPluginFactory` interface. There are three abstract methods that we should implement, namely, `getVelocityParamsForInput`, `getVelocityParamsForEdit`, and `getVelocityParamsForView`. All of them, as the name suggests, are used for populating the velocity parameters for the different scenarios.

   In our example, we need to limit the workflow action to a certain project role, and so we need to select the project role while defining the condition. The three methods will be implemented as follows:

```
private static final String ROLE =
  "role"; private static final String ROLES = "roles";
    ...
@Override
protected void getVelocityParamsForEdit
(Map<String, Object> velocityParams,
 AbstractDescriptor descriptor) {
    velocityParams.put(ROLE, getRole(descriptor));
    velocityParams.put(ROLES, getProjectRoles());
}
@Override
protected void getVelocityParamsForInput
(Map<String, Object> velocityParams){
    velocityParams.put(ROLES, getProjectRoles());
}
@Override
protected void getVelocityParamsForView(Map<String,
Object> velocityParams, AbstractDescriptor descriptor) {
   velocityParams.put(ROLE, getRole(descriptor));
}
```

Let's look at the methods in detail:

a. `getVelocityParamsForInput`: This method defines the velocity parameters for the input scenario, that is, when the user initially configures the workflow. In our example, we need to display all the project roles, so that the user can select one to define the condition. The method `getProjectRoles` merely returns all the project roles and the collection of roles is then put into the velocity parameters with the key `ROLES`.

b. `getVelocityParamsForView`: This method defines the velocity parameters for the view scenario, that is, how the user sees the condition after it is configured. In our example, we have defined a role, and so we should display it to the user after retrieving it from the workflow descriptor. If you have noticed, the descriptor, which is an instance of `AbstractDescriptor`, is available as an argument in the method. All we need is to extract the role from the descriptor, which can be done as follows:

```
private ProjectRole getRole(AbstractDescriptor descriptor) {
  if (!(descriptor instanceof ConditionDescriptor)){
    throw new IllegalArgumentException
    ("Descriptor must be a ConditionDescriptor.");
  }
  ConditionDescriptor functionDescriptor =
    (ConditionDescriptor) descriptor;
    String role = (String) functionDescriptor.getArgs().get(ROLE);
    if (role != null && role.trim().length() > 0)
    return getProjectRole(role);
     else
    return null;
}
```

Just check if the descriptor is a condition descriptor or not, and then extract the role as shown in the preceding snippet.

c. `getVelocityParamsForEdit`: This method defines the velocity parameters for the edit scenario, that is, when the user modifies the existing condition. Here we need both the options and the selected value. So, we put both the project roles collection and the selected role on to the velocity parameters.

2. The second step is to define the velocity templates for each of the three aforementioned scenarios: **input**, **view**, and **edit**. We can use the same template here for input and edit with a simple check to keep the old role selected for the edit scenario. Let us look at the templates:

a. `role-condition-input.vm`: Displays all project roles and highlights the already-selected one in the edit mode. In the input mode, the same template is reused, but the selected role will be null and so a null check is done:

```
<tr>
  <td class="fieldLabelArea">Project Role: </td>
  <td nowrap>
    <select name="role" id="role">
     #foreach ($field in $roles)
      <option value="${field.id}"
        #if ($role && (${field.id}==${role.id})) SELECTED
        #end
      >$field.name</option>
     #end
    </select>
    <br><font size="1">
      Select the role in which the user should be present!
    </font>
  </td>
</tr>
```

b. `role-condition.vm`: Displays the selected role:

```
#if ($role)
  User should have ${role.name} Role!
#else
  Role Not Defined
#end
```

3. The third step is to write the actual condition. The condition class should extend the `AbstractJiraCondition` class. Here, we need to implement the `passesCondition` method. In our case, we retrieve the project from the issue, check if the user has the appropriate project role, and return true if the user does:

```
public boolean passesCondition(Map transientVars,
 Map args, PropertySet ps) {
   Issue issue = getIssue(transientVars);
   ApplicationUser user = getCallerUser(transientVars, args);
   Project project = issue.getProjectObject();
   String role = (String)args.get(ROLE);
   Long roleId = new Long(role);
   return projectRoleManager.isUserInProjectRole(user,
   projectRoleManager.getProjectRole(roleId), project);
}
```

The issue on which the condition is checked can be retrieved using the `getIssue` method implemented in the `AbstractJiraCondition` class. Similarly, the user can be retrieved using the `getCallerUser` method. In the preceding method, `projectRoleManager` is injected in the constructor, as we have seen before.



Make sure you are using the appropriate scanner annotations for constructor injection, if the **Atlassian Spring Scanner** is defined in the pom.xml. See `https://bitbucket.org/atlassian/atlassian-spring-scanner` for more details.

4. We can see that the `ROLE` key is used to retrieve the project role ID from the `args` parameter in the `passesCondition` method. In order for the `ROLE` key to be available in the `args` map, we need to override the `getDescriptorParams` method in the condition factory class, `RoleConditionFactory` in this case. The `getDescriptorParams` method returns a map of sanitized parameters, which will be passed into workflow plugin instances from the values in an array form submitted by velocity, given a set of `name:value` parameters from the plugin configuration page (that is, the `input-parameters` velocity template). In our case, the method is overridden as follows:

```
public Map<String, String> getDescriptorParams(Map<String,
Object> conditionParams) {
    if (conditionParams !=
    null && conditionParams.containsKey(ROLE)) {
       return MapBuilder.build(ROLE,
       extractSingleParam(conditionParams, ROLE));
    }
    // Create a 'hard coded' parameter
   return MapBuilder.emptyMap();
}
```

The method here builds a map of the `key:value` pair, where key is `ROLE` and the value is the role value entered in the input configuration page. The `extractSingleParam` method is implemented in the `AbstractWorkflowPluginFactory` class. The `extractMultipleParams` method can be used if there is more than one parameter to be extracted!

5. All that is left now is to populate the `atlassian-plugin.xml` file with the aforementioned components. We will use the `workflow-condition` module and it looks like the following block of code:

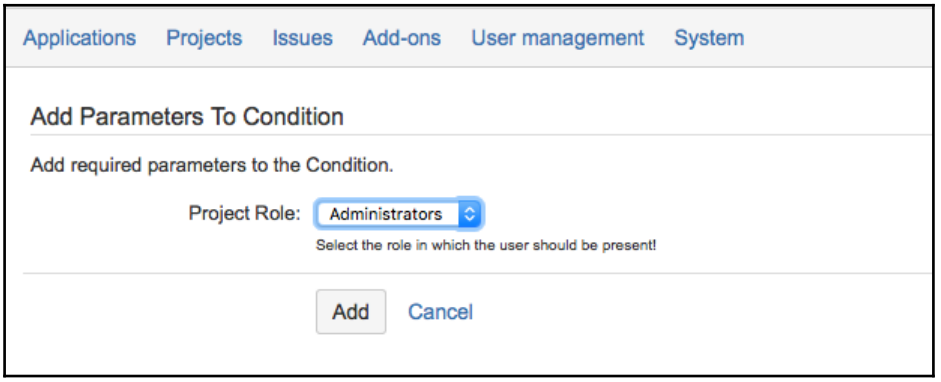
```
<workflow-condition key="role-condition"
 name="Role Based Condition"
```

```
    i18n-name-key="role-condition.name"
    class="com.jtricks.jira.workflow.RoleConditionFactory">

      <description key="role-condition.description">
         Role Based Workflow Condition
      </description>
      <condition-class>
        com.jtricks.jira.workflow.RoleCondition
      </condition-class>

       <resource type="velocity" name="view"
       location="templates/conditions/role-condition.vm"/>

       <resource type="velocity" name="input-parameters"
       location="templates/conditions
       /role-condition-input.vm"/>

       <resource type="velocity" name="edit-parameters"
       location="templates/conditions
       /role-condition-input.vm"/>

  </workflow-condition>
```

6. Package the plugin and deploy it!

How it works…

After the plugin is deployed, we need to modify the workflow to include the condition. The following screenshot is how the condition looks when it is added initially. This, as you now know, is rendered using the input template:



After the condition is added (that is, after selecting the **Administrators** role), the view is rendered using the view template and looks as shown in the following screenshot:



If you try to edit it, the screen will be rendered using the same input template and the **Administrators** role, or whichever role was selected earlier, will be preselected.

After the workflow is configured, when the user goes to an issue, they will be presented with the transition only if they are a member of the project role where the issue belongs. It is while viewing the issue that the `passesCondition` method in the `condition` class is executed.

See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Writing a workflow validator

Workflow validators are specific validators that check whether some predefined constraints are satisfied or not while progressing on a workflow. The constraints are configured in the workflow and the user will get an error if some of them are not satisfied. A typical example would be to check if a particular field is present or not before the issue is moved to a different status.

Workflow validators are created with the help of the `workflow-validator` module. The following are the key attributes and elements supported:

Attributes:

Name	Description
`key`	This should be unique within the plugin.
`class`	Class to provide contexts for rendered velocity templates. It must implement the `com.atlassian.jira.plugin.workflow.WorkflowPluginValidatorFactory` interface.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the workflow validator.

Elements:

Name	Description
`description`	Description of the workflow validator.
`validator-class`	Class which does the validation. It must implement `com.opensymphony.workflow.Validator`.
`resource type="velocity"`	Velocity templates for the workflow validator views.

Visit `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/workflow-plugin-modules#WorkflowPluginModules-Validators` for more details.

Getting ready

As usual, create a skeleton plugin. Then create an eclipse project using the skeleton plugin and we are good to go!

How to do it…

Let us consider writing a validator that checks whether a particular custom field has a value entered on the issue or not! We can do this by using the following steps:

1. Define the inputs needed to configure the workflow validator. We need to implement the `WorkflowPluginValidatorFactory` interface, which mainly exists to provide velocity parameters to the templates. It will be used to extract the input parameters that are used in defining the validator.
   To make it clear, the inputs here are not the inputs used while performing the workflow action, but the inputs used in defining the validator. The validator factory class, `FieldValidatorFactory` in this case extends the `AbstractWorkflowPluginFactory` interface and implements the `WorkflowPluginValidatorFactory` interface.

   Just like conditions, there are three abstract methods that we should implement. They are `getVelocityParamsForInput`, `getVelocityParamsForEdit`, and `getVelocityParamsForView`. All of them, as the names suggest, are used for populating the velocity parameters in different scenarios.

   In our example, we have a single input field, which is the name of a custom field. The three methods will be implemented as follows:

```
@Override
protected void getVelocityParamsForEdit(Map<String,
Object> velocityParams, AbstractDescriptor descriptor) {
    velocityParams.put(FIELD_NAME, getFieldName(descriptor));
    velocityParams.put(FIELDS, getCFFields());
}
@Override
protected void getVelocityParamsForInput(Map<String,
Object> velocityParams){
    velocityParams.put(FIELDS, getCFFields());
}
@Override
protected void getVelocityParamsForView(Map<String, Object>
velocityParams, AbstractDescriptor descriptor) {
    velocityParams.put(FIELD_NAME, getFieldName(descriptor));
}
```

You may have noticed that the methods look quite similar to the ones in a workflow condition, except for the business logic! Let us look at the methods in detail:

a. `getVelocityParamsForInput`: This method defines the velocity parameters for input scenario, that is, when the user initially configures the workflow. In our example, we need to display all the custom fields so that the user can select one to use in the validator. The method `getCFFields` returns all the custom fields and the collection of fields are then put into the velocity parameters with the key `FIELDS`.

b. `getVelocityParamsForView`: This method defines the velocity parameters for the view scenario, that is, how the user sees the validator after it is configured. In our example, we have defined a field, and so we should display it to the user after retrieving it from the workflow descriptor. You may have noticed that the descriptor, which is an instance of `AbstractDescriptor`, is available as an argument in the method. All we need is to extract the field name from the descriptor, which can be done as follows:

```
private String getFieldName(AbstractDescriptor descriptor) {
    if (!(descriptor instanceof ValidatorDescriptor)) {
        throw new IllegalArgumentException("Descriptor
        must be a ConditionDescriptor.");
    }
    ValidatorDescriptor validatorDescriptor =
    (ValidatorDescriptor) descriptor;
    String field = (String)
      validatorDescriptor.getArgs().get(FIELD_NAME);
    if (field != null && field.trim().length() > 0)
        return field;
    else
        return NOT_DEFINED;
}
```

Just check if the descriptor is a validator descriptor or not and then extract the field as shown in the preceding snippet.

c. `getVelocityParamsForEdit`: This method defines the velocity parameters for the edit scenario, that is, when the user modifies the existing validator. Here, we need both the options and the selected value. So, we put both the custom fields collection and the field name onto the velocity parameters.

2. The second step is to define the velocity templates for each of the three aforementioned scenarios, namely, input, view, and edit. We can use the same template here for input and edit, with a simple check to keep the old field selected for the edit scenario. Let us look at the template:

   a. `field-validator-input.vm`: Displays all custom fields and highlights the already selected one in edit mode. In input mode, the field variable will be null, and so nothing is preselected:

```
<tr>
  <td class="fieldLabelArea">Custom Fields: </td>
  <td nowrap>
    <select name="field" id="field">
      #foreach ($cf in $fields)
        <option value="$cf.name"
          #if ($cf.name.equals($field)) SELECTED
          #end
        >$cf.name</option>
      #end
    </select>
    <br>
    <font size="1">Select the Custom Field to be
    validated for NULL</font>
  </td>
</tr>
```

   b. `view-fieldValidator.vm`: Displays the selected field:

```
#if
 ($field)Field '$field' is Required!
#end
```

3. The third step is to write the actual validator. The validator class should implement the `Validator` interface. All we need here is to implement the `validate` method. In our example, we retrieve the custom field value from the issue and throw an `InvalidInputException` if the value is `null` (empty):

```
public void validate(Map transientVars, Map args, PropertySet ps)
 throws InvalidInputException {
  Issue issue = (Issue) transientVars.get("issue");
  String field = (String) args.get(FIELD_NAME);
  CustomField customField =
  customFieldManager.getCustomFieldObjectByName(field);
  if (customField!=null){
    //Check if the custom field value is NULL
    if (issue.getCustomFieldValue(customField) == null){
```

```
            throw new InvalidInputException("The field:"+field+"
            is required!");
            }
        }
    }
```

`transientVars Map` contains the variables that are given as inputs to the workflow `initialize` or `doAction` methods. For example, the issue on which the validation is performed can be retrieved from the `transientVars` map as shown earlier. The `args` map contains properties we have set, for example, the field name in our case. `customFieldManager` is injected in the constructor as usual.



Make sure you are using the appropriate scanner annotations for constructor injection, if the **Atlassian Spring Scanner** is defined in the `pom.xml`. See `https://bitbucket.org/atlassian/atlassian-spring-scanner`for more details.

4. All that is left now is to populate the `atlassian-plugin.xml` file with these components. We will use the `workflow-validator` module, and it looks like the following block of code:

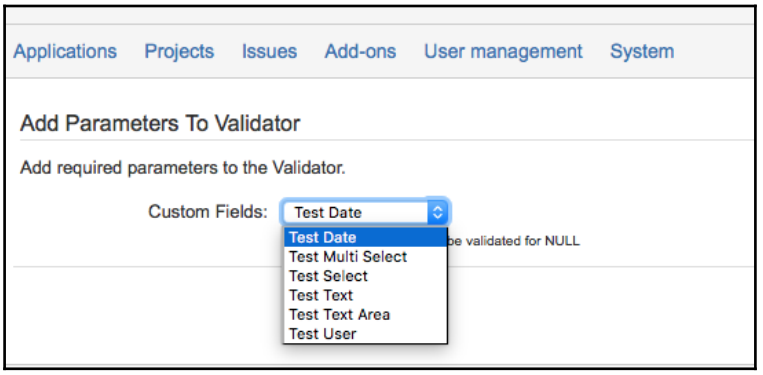
```
<workflow-validator key="field-validator"
name="Field Validator"
i18n-name-key="field-validator.name"
class="com.jtricks.jira.workflow.FieldValidatorFactory">
    <description key="field-validator.description">
            Field Not Empty Workflow Validator
    </description>

     <validator-class>
        com.jtricks.jira.workflow.FieldValidator
     </validator-class>
     <resource type="velocity" name="view"
     location="templates/validators/field-validator.vm"/>
     <resource type="velocity" name="input-parameters"
     location="templates/validators/field-validator-input.vm"/>

      <resource type="velocity" name="edit-parameters"
      location="templates/validators/field-validator-input.vm"/>
</workflow-validator>
```

5. Package the plugin and deploy it!

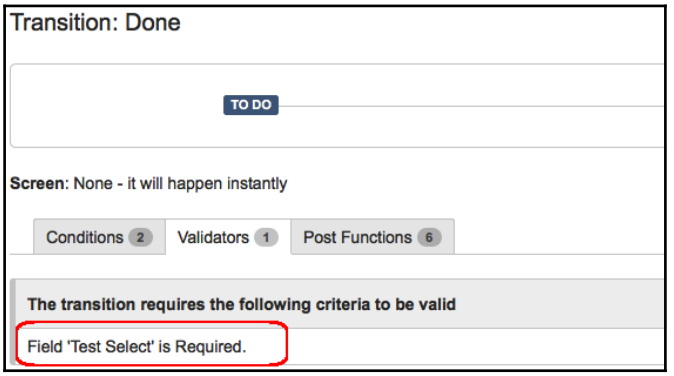
Note that we have stored the custom field `name` instead of the `ID` in the workflow, unlike what we did in the workflow condition. However, it is safe to use the ID because administrators can rename the fields, which would then need changes in the workflows.

How it works…

After the plugin is deployed, we need to modify the workflow to include the validator. The following screenshot is how the validator looks when it is added initially. This, as you now know, is rendered using the input template:



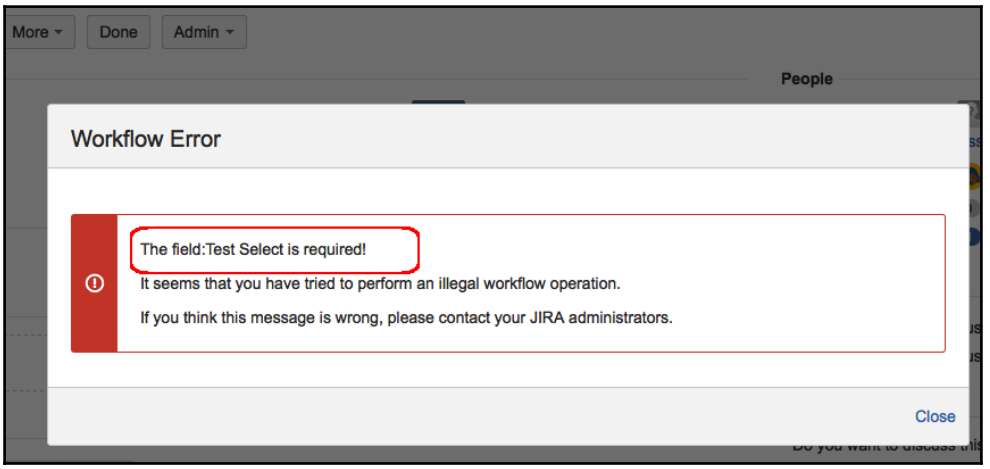
After the validator is added (after selecting the **Test Select** field), it is rendered using the view template and looks as follows:



If you try to edit the validator, the screen will be rendered using the same input template and will have the appropriate field, **Test Select** in this case, preselected.

After the workflow is configured, when the user goes to an issue and tries to progress it, the validator will check if the **Test Select** field has a value or not. It is at this point that the `validate` method in the `FieldValidator` class is executed.

If the value is missing, you will see an error, as shown in the following screenshot:



See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Writing a workflow post function

Let us now look at workflow post functions. Workflow post functions are very effective and heavily used. They allow you to do a lot of things when you progress on the workflow on an issue. A lot of customizations and workarounds take this route!

Workflow post functions are created with the help of the `workflow-function` module. The following are the key attributes and elements supported:

Attributes:

Name	Description
`key`	This should be unique within the plugin.
`Class`	Class to provide contexts for rendered velocity templates. It must implement the `com.atlassian.jira.plugin.workflow.WorkflowNoInputPluginFactory` interface if the function doesn't need input, or `com.atlassian.jira.plugin.workflow.WorkflowPluginFunctionFactory` if it needs input.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the workflow function.

Elements

Name	Description
`description`	Description of the workflow function.
`function-class`	Class which does the validation. It must implement `com.opensymphony.workflow.FunctionProvider`. It is recommended to extend `com.atlassian.jira.workflow.function.issue.AbstractJiraFunctionProvider`, as it already implements many useful methods.
`resource type="velocity"`	Velocity templates for the workflow function views.

There are three other elements that can be used with a post function. They are explained as follows:

- `orderable` (`true`/`false`): This specifies if this function can be re-ordered within the list of functions associated with a transition. The position within the list determines when the function actually executes.
- `unique` (`true`/`false`): This specifies if this function is unique. Determines if it is possible to add multiple instances of this post function on a single transition.
- `deletable` (`true`/`false`): This specifies if this function can be removed from a transition.

See `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/workflow-plugin-modules#WorkflowPluginModules-Functions` for more details, including properties `addable`, `weight`, and `default`.

Getting ready

As usual, create a skeleton plugin. Then create an eclipse project using the skeleton plugin and we are good to go! Also, create a `Test User` custom field of type `User Picker`, as we will be using it in our recipe.

How to do it…

Say we have a user custom field and we want to set the current user or a specified user name on to the custom field when a particular transition happens. A typical use case for this will be to store the name of the user who last resolved an issue. The following are the steps to write a generic post function that sets the current username or a username provided by the user on a user custom field:

1. Define the inputs needed to configure the workflow post function. As opposed to workflow conditions and validators, there are two interfaces available for a workflow post function factory class. If there are no inputs needed to configure the function, the factory class must implement `WorkflowNoInputPluginFactory`. An example will be to set the current user's name as the custom field value instead of the user configured name. If inputs are needed to configure the post function, the factory class must implement `WorkflowPluginFunctionFactory`. In our example, we take the username as the input.
   Both the interfaces mainly exist to provide velocity parameters to the templates. They will be used to extract the input parameters that are used in defining the functions. To make it clear, the inputs here are not the inputs used while performing the workflow action, but the inputs used in defining the post function.
   The function factory class, `SetUserCFFunctionFactory` in this case, extends the `AbstractWorkflowPluginFactory` and implements the `WorkflowPluginFunctionFactory` interface. Just like conditions, there are three abstract methods that we should implement, namely, `getVelocityParamsForInput`, `getVelocityParamsForEdit`, and `getVelocityParamsForView`. All of them, as the names suggest, are used for populating the velocity parameters for the different scenarios:

```
@Override
protected void getVelocityParamsForEdit(Map<String,
Object> velocityParams, AbstractDescriptor descriptor) {
   velocityParams.put(USER_NAME, getUserName(descriptor));
}
```

```
@Override
protected void getVelocityParamsForInput(Map<String,
Object> velocityParams){
    velocityParams.put(USER_NAME, CURRENT_USER);
}
@Override
protected void getVelocityParamsForView(Map<String,
Object> velocityParams, AbstractDescriptor descriptor) {
    velocityParams.put(USER_NAME, getUserName(descriptor));
}
```

You may have noticed that the methods look very similar to the ones in workflow conditions or validators, except for the business logic! Let us look at the methods in detail:

- `getVelocityParamsForInput` : This method defines the velocity parameters for the input scenario, that is, when the user initially configures the workflow. In our example, we need to use a text field that captures the username to be added on the issue.
- `getVelocityParamsForView`: This method defines the velocity parameters for the view scenario, that is, how the user sees the post function after it is configured. In our example, we have defined a field, and so we should display it to the user after retrieving it from the workflow descriptor. You may have noticed that the descriptor, which is an instance of `AbstractDescriptor`, is available as an argument in the method. All we need is to extract the username from the descriptor, which can be done as follows:

```
private String getUserName(AbstractDescriptor descriptor) {
  if (!(descriptor instanceof FunctionDescriptor)) {
    throw new IllegalArgumentException("Descriptor must
    be a FunctionDescriptor.");
  }
  FunctionDescriptor functionDescriptor = (FunctionDescriptor)
  descriptor;
  String user = (String) functionDescriptor.getArgs()
  .get(USER_NAME);

  if (user != null && user.trim().length() > 0)
    return user;
  else
    return CURRENT_USER;
  }
```



Just check if the descriptor is a validator descriptor or not, and then extract the field as shown in the preceding snippet.

- `getVelocityParamsForEdit`: This method defines the velocity parameters for the edit scenario, that is, when the user modifies the existing validator. Here we need both the options and the selected value. So, we put both the custom fields' collection and the field name on to the velocity parameters.

2. The second step is to define the velocity templates for each of the three scenarios: input, view, and edit. We can use the same template here for input and edit with a simple check to keep the old field selected for the edit scenario. Let us look at the templates:

   a. `set-user-cf-function-input.vm`: This displays a text field, with the current value if applicable, in the edit mode:

```
<tr>
  <td class="fieldLabelArea">UserName: </td>
  <td nowrap> <input type="text" name="user" value="$!user"/>
   <br>
   <font size="1">Enter the userName to be set on the
    Test User CustomField</font>
  </td>
</tr>
```

   b. `set-user-cf-function.vm`: This displays the selected value:

```
 #if
    ($user)
     The <b>Test User</b> CF will be set with value :
    <b>$user</b>
#end
```

3. The third step is to write the actual function. The function class must extend the `AbstractJiraFunctionProvider` interface. All we need here is to implement the `execute` method. In our example, we retrieve the username from the issue and set it on the `Test User` custom field:

```
public void execute(Map transientVars, Map args,
PropertySet ps) throws WorkflowException {
    MutableIssue issue = getIssue(transientVars);
    ApplicationUser user = null;
    if (args.get("user") != null) {
```

```
            String userName = (String) args.get("user");
           if (userName.equals("Current User")) {
            // Set the current user here!
              user = authContext.getLoggedInUser();
           } else {
              user = userManager.getUserByName(userName);
           }
       } else {
       // Set the current user here!
       user = authContext.getLoggedInUser();
     }
     // Now set the user value to the custom field
     CustomField userField =
     customFieldManager.getCustomFieldObjectByName("Test User");
     if (userField != null) {
        setUserValue(issue, user, userField);
      }
}
```

Like a validator, the issue on which the post function is executed can be retrieved using the `transientVars` map. The user can be retrieved from the `args` map.

Here, the `setUserValue` method simply sets the username on the passed custom field, as shown in the following block of code:

```
private void setUserValue(MutableIssue issue,
ApplicationUser user, CustomField userField) {
    issue.setCustomFieldValue(userField, user);
    Map modifiedFields = issue.getModifiedFields();
    FieldLayoutItem fieldLayoutItem =
     ComponentAccessor.getFieldLayoutManager()
     .getFieldLayout(issue).getFieldLayoutItem(userField);

    DefaultIssueChangeHolder issueChangeHolder = new
     DefaultIssueChangeHolder();

    final ModifiedValue modifiedValue = (ModifiedValue)
     modifiedFields.get(userField.getId());

    userField.updateValue(fieldLayoutItem, issue, modifiedValue,
     issueChangeHolder);
}
```

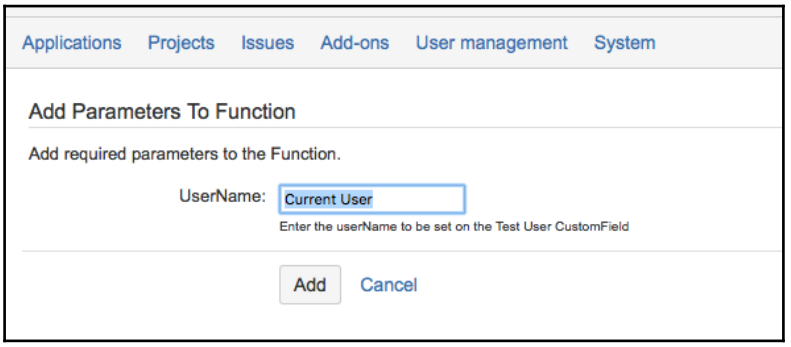
4. All that is left now is to populate the `atlassian-plugin.xml` file with these components. We will use the `workflow-condition` module and it looks like the following block of code:

```
<workflow-function key="set-user-cf-function"
 name="Set User CF Function"
 i18n-name-key="set-user-cf-function.name"
 class="com.jtricks.jira.workflow.SetUserCFFunctionFactory">

   <description key="set-user-cf-function.description">
        Set Defined User or Current User
   </description>

   <function-class>
        com.jtricks.jira.workflow.SetUserCFFunction
   </function-class>

  <orderable>true</orderable>
  <unique>true</unique>
  <deletable>true</deletable>

  <resource type="velocity" name="view"
   location="templates/postfunctions/set-user-cf-function.vm"/>

  <resource type="velocity" name="input-parameters"
   location="templates/postfunctions
   /set-user-cf-function-input.vm"/>

  <resource type="velocity" name="edit-parameters"
   location="templates/postfunctions
   /set-user-cf-function-input.vm"/>

</workflow-function>
```

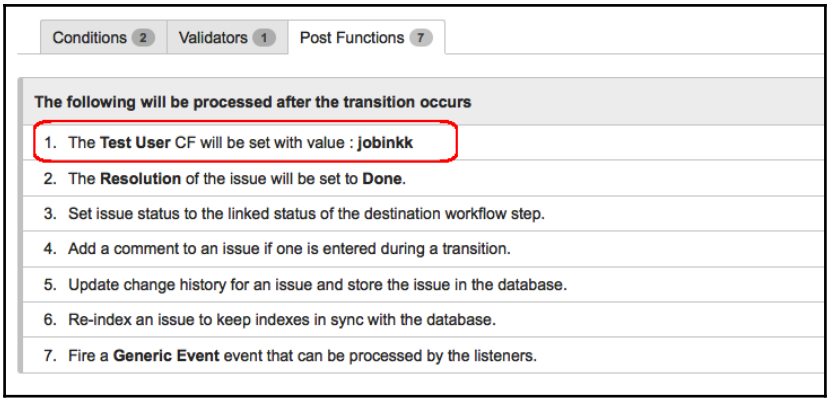
5. Package the plugin and deploy it!

How it works…

After the plugin is deployed, we need to modify the workflow to include the function. Clicking on our post function takes us to the configuration page, shown below. This, as you now know, is rendered using the input template:



After the function is added (after entering in the **UserName** field), it looks as follows:



If you try to edit the post function, the screen will be rendered using the same input template and the **UserName** field will be populated with the value we entered earlier. After the workflow is configured, when the user executes the workflow action, the **Test User** custom field is set with the configured value, **jobinkk** in this case.

See also

- The *Creating a skeleton plugin recipe* in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin recipe* in `Chapter 1`, *Plugin Development Process*

Editing an active workflow

We have seen how the workflow plays an important role in configuring our JIRA and how we can write plugins to add more workflow conditions, validators, and post functions. Once these plugins are added, we need to modify the workflow to include the newly created components at the appropriate transitions.

Modifying an inactive workflow or creating a new workflow is pretty easy. You can add the conditions/validators/post functions when you create the transition or just click on the transition to modify them. But to edit an active workflow, there are a few more steps involved which we will see in this recipe.

A workflow is active when it is being used in an active workflow scheme that is tied to a project. You can check whether a workflow is active by navigating to **Administration** | **Issues** | **Workflows**.

How to do it…

The following are the steps to edit an active workflow:

1. Login as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Click on the **Edit** link on the workflow you want to edit. The link can be found under the **Operations** column.
4. Click on the *step* or *transition* that you want to modify. In the **Diagram** mode, click on the *step* or *transition* and select **Edit** in the popup window.
5. Make the changes. The changes won't be effective until the workflow is published.
6. After all the changes are made, click on the **Publish Draft** link at the top of the page if you are still viewing the modified workflow.
7. Make a copy of the old workflow when prompted, if you need a backup, and click on **Publish**.

How it works…

After making changes to the draft and clicking on **Publish**, the new workflow will be active. However, there are some limitations to this procedure, which are detailed as follows:

- You can't delete an existing workflow step
- You can't edit the status associated with an existing step
- If an existing step has no outgoing transitions, such as on the final step, you can't add any new outgoing transitions
- You can't change the step IDs for any existing steps

If you want to overcome these limitations, you need to copy the workflow, modify the copy, and make it active by migrating the projects on to the new workflow.

After the new workflow is active, any transitions on the issue will be based on the new workflow.

There's more…

If you want to modify an active workflow, thus overcoming some of the aforementioned limitations, but don't want to go through the pain of migrating all the projects involved, you might want to look at modifying it directly in the JIRA database.



You should be extremely careful about the changes while doing this. For example, if there are issues in a status that is removed in the modified workflow, those issues will be stuck at the removed status. The same can happen for the removed steps.

Also, this workaround is not recommended unless you are an experienced DBA who knows how to handle large XML data in the database version used by your JIRA instance.

Modifying workflows in a JIRA database

The following are the steps to modify the workflows in the database:

1. Export the workflow that needs to be modified into XML. You can do it using the **Download as XML** button while viewing a workflow.
2. Modify the XML to include your changes (or alternatively, make changes in a copy of the JIRA workflow and export that as XML).

3. Stop the JIRA instance.
4. Connect to your JIRA database.
5. Take a backup of the existing database. We can revert to this backup if anything goes wrong.
6. Update the `JIRAWORKFLOWS` table to modify the `descriptor` column with the new XML file for the appropriate workflow. When the workflow XML is huge, it might be useful to rely on database-specific methods to update the table. For example, we can use Oracle XML database utilities (`http://download.oracle.com/docs/cd/B12037_01/appdev.101/b10790/xdb01int.htm`), if JIRA is connected to the Oracle database.
7. Commit the changes and disconnect from the database.
8. Start the JIRA instance.
9. Re-index JIRA.

Permissions based on workflow status

If you have administered JIRA, you probably already know about the global permissions and project permissions. And you have seen more about workflow conditions in the previous recipes. But how about restricting issue permissions based on the workflow status?

JIRA gives us an option to restrict issue operations (such as edit, comment, and so on) depending on the current status, using workflow properties, as detailed in this recipe. Workflow properties are nothing but key/value pairs defined on a workflow step or a transition.

How to do it…

The following are the steps to control issue permissions, using workflow properties:

1. Log in as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Create a draft of the workflow, if it is active. Navigate to the *step*, which needs to be modified.

4. Click on the **View Properties** link.
5. Enter the permission property into the **Property Key** field. The property is of the form – `jira.permission.[subtasks.]{permission}.{type}[.suffix]` where:

    a. `subtasks`: This is optional. If included, the permission is applied on the issue's subtasks. If not, the permission is applied on the actual issue.

    b. `permission`: This is a short name specified in the `Permissions` (`http://docs.atlassian.com/software/jira/docs/api/latest/com/atlassian/jira/security/Permissions.html`) class.The following are the permitted values, as of JIRA 7.0: `admin`, `use`, `sysadmin`, `project`, `browse`, `create`, `edit`, `update` (same as edit), `scheduleissue`, `assign`, `assignable`, `attach`, `resolv`, `close`, `transition`, `comment`, `delete`, `work`, `worklogdeleteall`, `worklogdeleteown`, `worklogeditall`, `worklogeditown`, `link`, `sharefilters`, `groupsubscriptions`, `move`, `setsecurity`, `pickusers`, `viewversioncontrol`, `modifyreporter`, `viewvotersandwatchers`, `managewatcherlist`, `bulkchange`, `commenteditall`, `commenteditown`, `commentdeleteall`, `commentdeleteown`, `attachdeleteall`, `attachdeleteown`, and `viewworkflowreadonly`.

    c. `type`: This is a type of permission granted/denied. The values can be `group`, `user`, `assignee`, `reporter`, `lead`, `userCF`, `projectrole`.

    d. `suffix`: This is an optional suffix to make the property unique when you have the same type added more than once! `jira.permission.edit.group.1`, `jira.permission.edit.group.2`, and so on. This is because of the OSWorkflow restriction that the property value should be unique.

6. Enter the appropriate value in the **Property Value** field. If the type is group, enter a group. If it is a user, enter a username, and so on. It might be useful to give a few examples here:

- `jira.permission.comment.group=some-group`
- `jira.permission.comment.denied=some_random_value`
- `jira.permission.edit.group.1=group-one`
- `jira.permission.edit.group.2=group-two`
- `jira.permission.modifyreporter.user=username`

- `jira.permission.delete.projectrole=10000`
- `jira.permission.subtasks.delete.projectrole=10000`

7. Go back and publish the workflow, if it was active. If not, associate the workflow with the appropriate schemes.

How it works…

When a particular permission property is tied to a workflow status, JIRA looks at it and enforces it. It is to be noted that workflow permissions can only restrict permissions set in the permission scheme, not grant permissions.

For example, if you have the edit permission restricted to `jira-administrators` in the permission scheme, adding `jira.permission.edit.group=jira-users` wouldn't grant the permission to `jira-users`.

But instead, if you had both of these groups with the edit permission, only `jira-users` will be allowed to edit, as defined in the workflow permission.

There's more…

There are lots of use cases for workflow properties, but let us quickly take a look at one of them.

Making an issue editable/non-editable using workflow properties

We know that the edit permission on an issue is controlled through the **Edit Issue Permission**. This is used within the permissions schemes tied to a project and it blocks/allows editing of the issue, irrespective of which status it is in! But many a time, the need arises to block an issue being edited at a specific status. An example would be to prevent editing on a closed issue or to block users' logging time on a particular status.

We can make an issue editable or non-editable using the `jira.issue.editable` workflow property. The following is the step-by-step procedure:

1. Log in as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.

3. Create a draft of the workflow if it is active. Navigate to the *step*, which needs to be modified.
4. Click on the **View Properties** link.
5. Enter `jira.issue.editable` into the **Property Key** field.
6. Enter `false` in the **Property Value** field, if you want to prevent editing on the issue after this transition is performed. Use `true` as the value, if you want to make it editable.
7. Go back and publish the workflow if it was active. If not, associate the workflow with the appropriate schemes.

Note that the property is added on a workflow *step* and not a *transition*.

See also

- The *Editing an active workflow* recipe in this chapter

Including/excluding resolutions for specific transitions

If you haven't noticed already, resolutions in JIRA are global. If you have a resolution **Resolved**, it appears whenever the resolution field is added on a transition screen. This might not make sense in some cases. For example, it doesn't make sense to add the resolution **Resolved** when you are rejecting an issue.

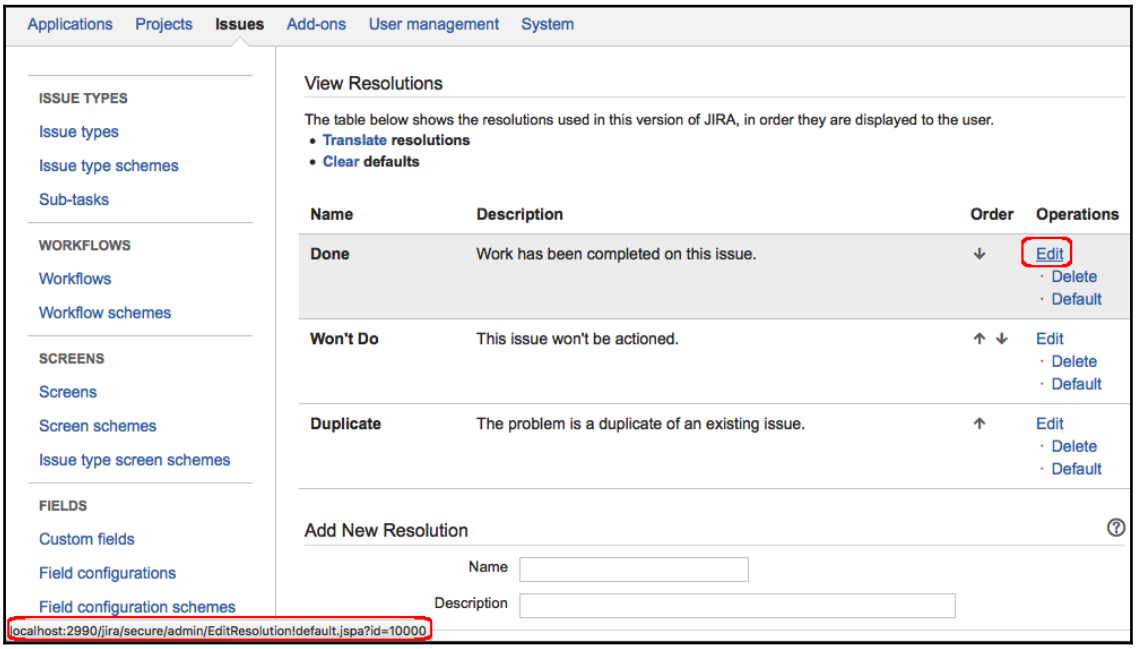
Let us see how we can pick and choose resolutions based on workflow transitions.

How to do it…

We can include/exclude specific resolutions on workflow transitions using the `jira.field.resolution.include` and `jira.field.resolution.exclude` properties. The following is the step-by-step procedure:

1. Log in as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Create a draft of the workflow, if it is active. Navigate to the transition, which needs to be modified.

4. Click on the **View properties of this transition** link.
5. Enter `jira.field.resolution.include` or `jira.field.resolution.exclude` into the **Property Key** field, depending on whether you want to include or exclude a specific resolution.
6. Enter the comma-separated list of resolution IDs that you want to include/exclude, under the **Property Value** field. The resolution ID can be obtained by navigating to **Administration** | **Issues** | **Resolutions**, and hovering over the **Edit** link:



7. You can also find the resolution ID by querying the resolutions table in the database.
8. Click on **Add**.
9. Go back and publish the workflow if it was active. If not, associate the workflow with the appropriate schemes.

Note that the property is added on a workflow *transition* and not a *step*.

How it works…

When the `jira.field.resolution.exclude` property is added, all the resolutions whose IDs are entered as comma-separated values under the `Property Value` field are excluded from the screen during that transition.

On the other hand, when `jira.field.resolution.include` is added, only the resolutions whose IDs are entered as comma-separated values under the `Property Value` field are shown on the screen.

See also

- The *Editing an active workflow* recipe in this chapter

Adding workflow triggers

Workflow triggers, as explained in the beginning of this chapter, are configured to trigger state transitions in JIRA based on events occurring in a connected development tool such as Bitbucket server, FishEye/Crucible, and so on.

Getting ready

Make sure JIRA is connected to a supported development tool. As of JIRA 7.0, the following are the tools supported:

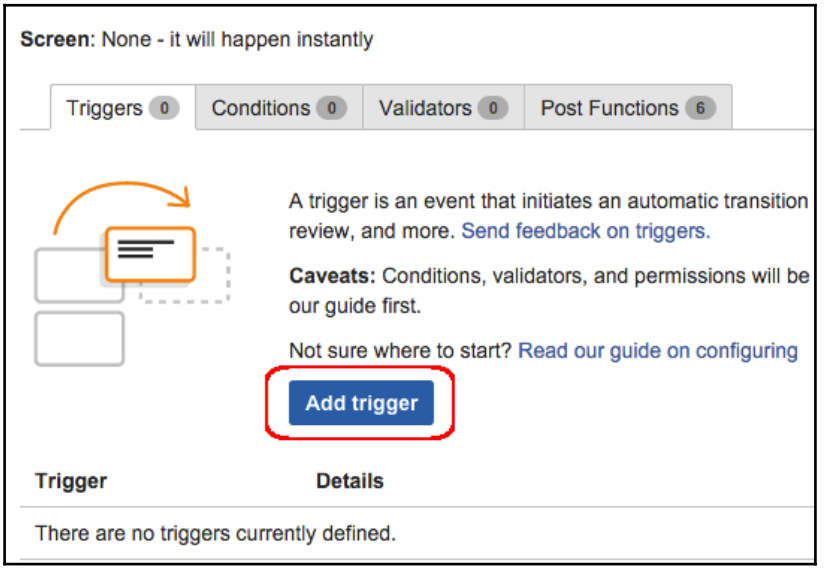
- Bitbucket Server (Stash 3.2.0 or later)
- FishEye/Crucible 3.5.2 (or later)
- GitHub Enterprise 11.10.290 (or later)
- Bitbucket Cloud
- GitHub

More details on integrating JIRA with development tools can be found at `https://confluence.atlassian.com/adminjiraserver070/integrating-with-development-tools-776637096.html`.

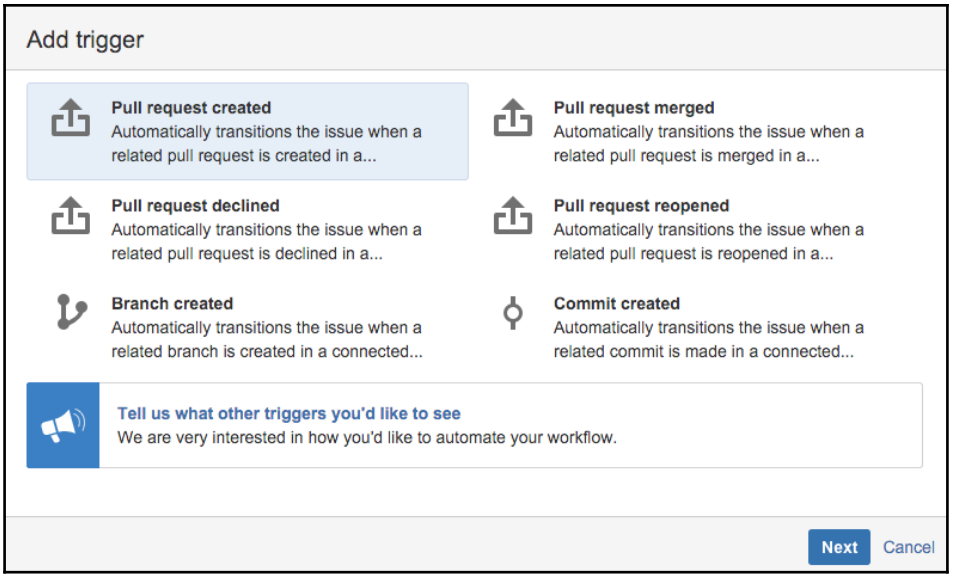
How to do it…

Setting up workflow triggers is very easy. The following is the step-by-step procedure:

1. Log in as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Create a draft of the workflow, if it is active. Navigate to the transition, which needs to be modified.
4. Click on **Triggers**.
5. Click on **Add trigger**, shown in the following screenshot:



6. Select from one of the available triggers and proceed to configure the trigger:



How it works…

Depending on the connected application, event triggers are handled separately. For example, events from Bitbucket Cloud and GitHub are processed via the DVCS Connector Plugin. It processes events via two mechanisms:

- **Webhook-triggered synchronization**: Bitbucket cloud or GitHub posts data to JIRA using DVCS connector when an event occurs. JIRA will process the data and trigger the appropriate workflow transition(s).
- **Scheduled synchronization**: In the event of connection failures, the event is stored in the apps and JIRA will do a scheduled synchronization every 60 minutes. This acts as a backup to webhook-triggered synchronization.

Bitbucket server and FishEye/Crucible behave differently. They use application links and are solely responsible for sending the events as soon as they occur. The events might be lost in the event of a connection failure.

Also, there are event limits set on all development tools, and any events sent after the limit are discarded to avoid overloading on the JIRA side.



Note that when triggers are configured on a workflow transition, it ignores any conditions or validators configured on that transition. Post functions will be executed as always.

More details on all of this can be read at `https://confluence.atlassian.com/adminjiracloud/configuring-workflow-triggers-776636696.html`.

There's more…

If there are workflow post functions in the transitions using triggers, and those post functions require a valid JIRA user, make sure the development tool user is mapped to a valid user in JIRA.

User mapping from development tools to JIRA

Different development tools use different e-mail addresses and usernames to identify a mapped user in JIRA. The details of identifying the developer tool e-mail address and username, for different events, is explained at `https://confluence.atlassian.com/adminjiracloud/configuring-workflow-triggers-776636696.html#Configuringworkflowtriggers-usermapping`.

Once the username and e-mail address are identified in the development tool, it is mapped to a JIRA user using the following logic:

1. The matching is always done first with the e-mail address and then username.
2. If there is only a single JIRA user with the same e-mail address, the issue is transitioned as that JIRA user.
3. If there are no JIRA users with matching e-mail addresses, the issue is transitioned as an anonymous user.
4. If there are multiple JIRA users with the same e-mail address, a matching username is searched for among those users. If there is a JIRA user with the matching username, the issue is transitioned as that JIRA user. If there are no users with the matching username, the issue is transitioned as an anonymous user.

See also

- The *Editing an active workflow* recipe in this chapter

Internationalization in workflow statuses

If people around the world, speaking different languages, use your JIRA instance, it is likely that you use internationalization to convert JIRA into their own language. But things like the workflow action name, button name, and so on are configured in the workflow and not as i18n properties. And therefore, they are limited to a single language.

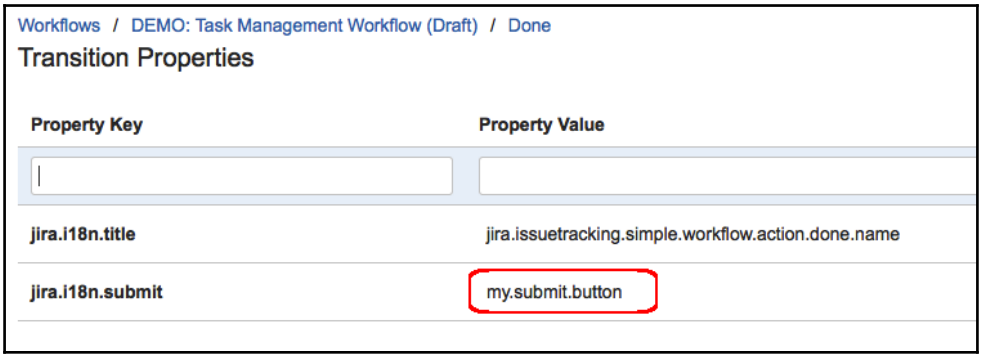
This is where workflow properties come to our rescue, again!

How to do it…

We can modify the workflow action submit button name or the action name using the properties `jira.i18n.submit` or `jira.i18n.title`, respectively. The following is the step-by-step procedure:

1. Identify the i18n file to modify. For the default language, you can find it in the `atlassian-jira/WEB-INF/classes/com/atlassian/jira/web/action/JiraWebActionSupport.properties` file. For other languages, the `i18n` file is embedded in the `atlassian-jira/WEB-INF/atlassian-bundled-plugins/jira-core-language-pack-_<language code>_<country code>-<jira version>-<jar version>.jar` file.
2. Edit the `JiraWebActionSupport.properties` file or the appropriate file inside the identified i18n jar file at `\com\atlassian\jira\web\action\JiraWebActionSupport_<language code>_<country code>.properties`. You can use a utility, such as 7Zip, to edit the file inside the jar. Alternatively, you can extract the jar, modify the file, and archive it again!
3. Add your i18n property and its value: `my.submit.button=My Submit Button in English`.

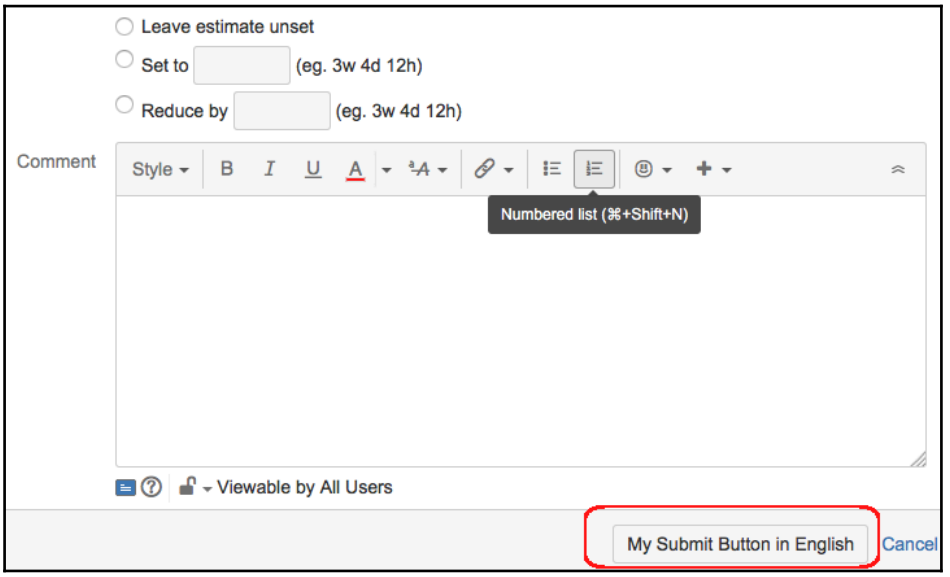
4. Update the file and restart JIRA to pick up the new property.
5. Log in as a JIRA administrator.
6. Navigate to **Administration** | **Issues** | **Workflows**.
7. Create a draft of the workflow, if it is active. Navigate to the transition, which needs to be modified.
8. Click on the **View Properties** link.
9. Enter `jira.i18n.submit` or `jira.i18n.title` into the **Property Key** field, depending on whether you want to modify the submit button name or the action name. Let us consider the example of the submit button:
10. Enter the i18n key that we used in the property file, under the **Property Value** field. In our example, the key is `my.submit.button`. The modified properties will be as shown here:



11. Click on **Add**.
12. Go back and publish the workflow, if it was active. If not, associate the workflow with the appropriate schemes.

How it works…

Once the workflow is published, JIRA will populate the submit button name from the i18n property file the next time the transition happens. In our example, the transition screen will look like the following screenshot:



As you can see, the button name is changed to **My Submit Button in English**. All you need to do now is modify the other language jars to include the proper translations!

See also

- The *Editing an active workflow* recipe in this chapter

Obtaining available workflow actions programmatically

Often in our programs, we may come across the need to retrieve the current workflow actions available on the issue. Let us have a look at how to do this using the JIRA API.

How to do it…

Follow these steps:

1. Retrieve the JIRA workflow object associated with the issue:

```
JiraWorkflow workFlow =
ComponentAccessor.getWorkflowManager().getWorkflow(issue);
```

   Here, the issue is the current issue, which is an instance of the `com.atlassian.jira.issue.Issue` class.

2. Get the issue status and use it to retrieve the current workflow step linked to the issue:

```
Status status = issue.getStatusObject();
com.opensymphony.workflow.loader.StepDescriptor currentStep =
 workFlow.getLinkedStep(status);
```

3. Retrieve the set of available actions from the current step:

```
List<ActionDescriptor> actions = currentStep.getActions();
```

   Here, `actions` is a list of `com.opensymphony.workflow.loader.ActionDescriptor`.

4. Iterate on the `ActionDescriptors` and get the details for each action, depending on the requirement! The name of the available action can be printed as follows:

```
for (ActionDescriptor action : actions) {
    System.out.println("Action: "+action.getName())
}
```

How it works…

WorkflowManager is used to perform a lot of operations related to workflows, such as creating/updating/deleting a workflow, copying it, creating a draft, and so on. Here, we used it to retrieve the workflow object, based on the issue selected. Please check the API (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/workflow/WorkflowManager.html`) for a full list of available operations using `WorkflowManager`.

Once we retrieve the JIRA workflow, we get the current step using the status. As you have seen before in this chapter, a workflow *status* is linked to one, and only one workflow *step*. Once we get the *step*, we can get a load of information from it, including the available actions from that *step*.

Jolly good?

There's more…

Even though the recipe shows how to retrieve the action name, it is the ID that is used while programmatically progressing an issue through its workflow.

Getting the action ID's given name

Once the action name is available, you can easily retrieve the action ID by iterating on the list of actions, as shown in the following lines of code:

```
private int getActionIdForTransition(List<ActionDescriptor> actions, String
actionName) {
  for (ActionDescriptor action : actions) {
    if (action.getName().equals(actionName)) {
      return action.getId();
    }
  }
  return -1; // Handle invalid action
}
```

Programmatically progressing on workflows

Another operation that we perform normally on workflows is to programmatically transition the issues through its workflow statuses. Let us have a look at how to do this using the JIRA API.

How to do it…

Transitioning issues is done using the `IssueService` (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/bc/issue/IssueService.html`). Here is how you do it:

1. Get the `IssueService` object, either by injecting it in the constructor or as follows:

```
IssueService issueService = ComponentAccessor.getIssueService();
```

2. Find out the action ID for the action to be executed. You can either get it by looking at the workflows (that is, the number within brackets alongside the transition name) if you know it is not going to change, or retrieve it by using the action name (refer to the previous recipe).
3. Populate the `IssueInputParameters` if you want to modify anything on the issue, such as assignee, reporter, resolution, and so on! It represents an issue builder and is used to provide parameters that can be used to update the issue during the transition:

```
IssueInputParameters issueInputParameters = new
 IssueInputParametersImpl();
issueInputParameters.setAssigneeId("someotherguy");
issueInputParameters.setResolutionId("10000");
```

   A full list of supported fields can be found at `http://docs.atlassian.com/jira/latest/com/atlassian/jira/issue/IssueInputParameters.html`.

4. Validate the transition:

```
TransitionValidationResult transitionValidationResult =
 issueService.validateTransition(user, 12345L, 10000L,
 issueInputParameters);
```

   Let's look at the parameters used in the preceding code snippet:

   a. `user`: This is the current user, or the user who will be performing the transition

   b. `12345L`: This is the issue ID

c. `10000L`: This is the action ID

d. `issueInputParameters`: The parameters we populated in the previous step

5. If `transitionValidationResult` is valid, invoke the transition operation. Handle it if it is not valid. Make sure you use the same user:

```
if (transitionValidationResult.isValid()){
    IssueResult transitionResult = issueService.transition(user,
    transitionValidationResult);
    if (!transitionResult.isValid()){
    // Do something
   }
}
```

We need to do a final check on the result as well to see if it is valid!

6. That will transition the issue to the appropriate state.

How it works…

Once the action ID is correct and the parameters are validated properly, `IssueService` will do the background work of transitioning the issues.

Obtaining workflow history from the database

JIRA captures changes on an issue in its "change history". It is pretty easy to find them by going to the change history tab on the view issue page.

But often, we would like to find out specific details about the various workflow statuses that an issue has gone through in its lifecycle. Going through the change history and identifying the status changes is a painful task when there are tens of hundreds of changes on an issue. People normally write plugins to get around this, or go directly to the database.

Even when it is achieved using plugins, the background logic is to look at the tables in the database. In this recipe, we will look at the tables involved and write the SQL query to extract workflow changes for a given issue.

Getting ready

Make sure you have an SQL client installed and configured that will help you to connect to the JIRA database.

How to do it…

Follow these steps:

1. Connect to the JIRA database.
2. Find out the `id` of the issue for which you want to extract the workflow changes. If you don't have the ID on hand, you can get it from the database using the issue key as follows:

   ```
   select id from jiraissue where pkey = "JIRA-123"
   ```

   Where `JIRA-123` is the issue key.

3. Extract all the change groups created for the issue. Every set of changes made on an issue during a single operation (for example, edit, workflow transition, and so on) is grouped into a single `changegroup` by JIRA. It is on the `changegroup` record that JIRA stores the associated `issueid` and the `created` date (the date when the change was made):

   ```
   select id from changegroup where issueid = '10010'
   ```

   Where `10010` is the `issueid`, the ID we extracted in the previous step. While extracting the change groups, we can even mention the created date if you want to see only changes on a specific date! Use the author field to restrict this to changes made by a user.

4. Extract `status` changes for the group/groups selected:

   ```
   select oldstring, newstring from changeitem
   where fieldtype = "jira" and field = "status"
   and groupid in ( 10000, 10010 )
   ```

   Here, the groupids `10000`, `10010`, and so on, are IDs extracted in the previous step. Here, `oldstring` is the original value on the issue and `newstring` is the updated value. Include `oldvalue` and `newvalue`, if you want to get the status IDs as well.

You can write it in a single query, as shown next, or modify it to include more details. But hopefully this gives you a starting point:

```
select oldstring, newstring from changeitem
where fieldtype = "jira" and field = "status"
and groupid in ( select id from changegroup where issueid = '10010');
```

Another example of how to extract the details, along with the created date, is to use inner join as follows:

```
select ci.oldstring, ci.newstring, cg.created
from changeitem ci inner join changegroup cg on ci.groupid = cg.id
where ci.fieldtype = "jira" and ci.field = "status"
and cg.issueid = '10010';
```

Over to you DBAs now!

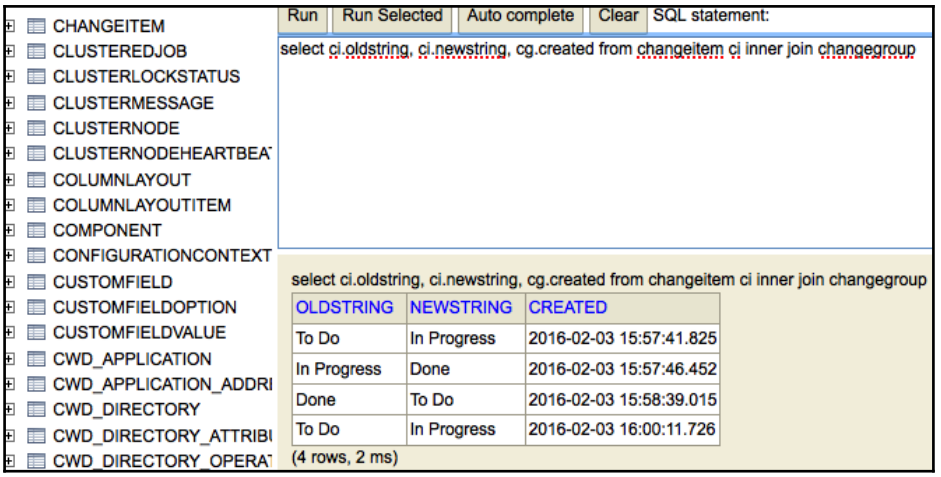
How it works…

As mentioned, the changes at any single operation on an issue are stored as a `changegroup` record in the JIRA database. The main three columns, `issueid`, `author`, and `created`, are all parts of this table.

The actual changes are stored in the `changeitem` table with its foreign key `groupid` pointing to the `changegroup` record.

In our case, we are looking specifically at the workflow statuses, and so, we query for records that have the `fieldtype` value of `jira` and `field` of `status`.

A sample output of the query (that uses an inner join) is as follows:



See also

- The *Retrieving workflow details from a table* recipe in `Chapter 10`, *Dealing with the JIRA Database*

Reordering workflow actions in JIRA

On a JIRA workflow, the available actions that appear on the **View Issue** page are normally ordered in the sequence those transitions were created in. This works fine most of the time, but, in some cases, we will want to change the order in which it appears on the issue screen!

To achieve this logical ordering of workflow actions on the **View Issue** page, JIRA provides us with a workflow property named `opsbar-sequence`. Let us see how we modify the ordering using this property instead of tampering with the workflow.

How to do it…

Follow these steps:

1. Log in as a JIRA Administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Create a draft of the workflow, if it is active. Navigate to the transition, which needs to be modified.
4. Click on the **View Properties** of the transition.
5. Enter `opsbar-sequence` into the **Property Key** field.
6. Enter the `sequence` value under the **Property Value** field. This value should be relative to the values entered in the other transitions.
7. Click on **Add**.
8. Go back and publish the workflow, if it was active. If not, associate the workflow with the appropriate schemes.

Note that the property is added on a workflow *transition* and not a *step*. If the property is set only on a few transitions, they will appear before the transitions that don't have the property set on them.

How it works…

Let us consider the following example where the **Reject** workflow action appears after **Done**:



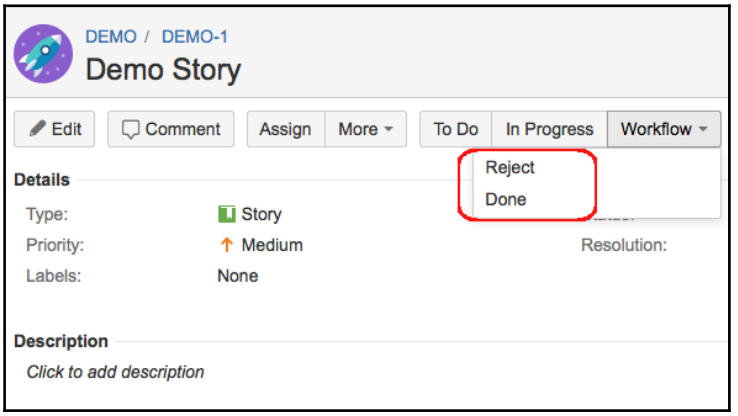
What if you want to put **Reject** before **Done**?

As there are four operations here, we can order them as shown in the following table with the sequence values against them:

Workflow action	Sequence
To Do	10
In Progress	20
Reject	30
Done	40

Note that the sequence numbers can even be 1, 2, 3, and 4. There are no restrictions on how and where the numbers should start. It is advised to keep 10, 20, and so on, because it will be easier to insert new transitions in between, if required in future.

After we modify the workflow using the property and the aforementioned sequence numbers, as we saw in the previous section, the actions are ordered as follows:



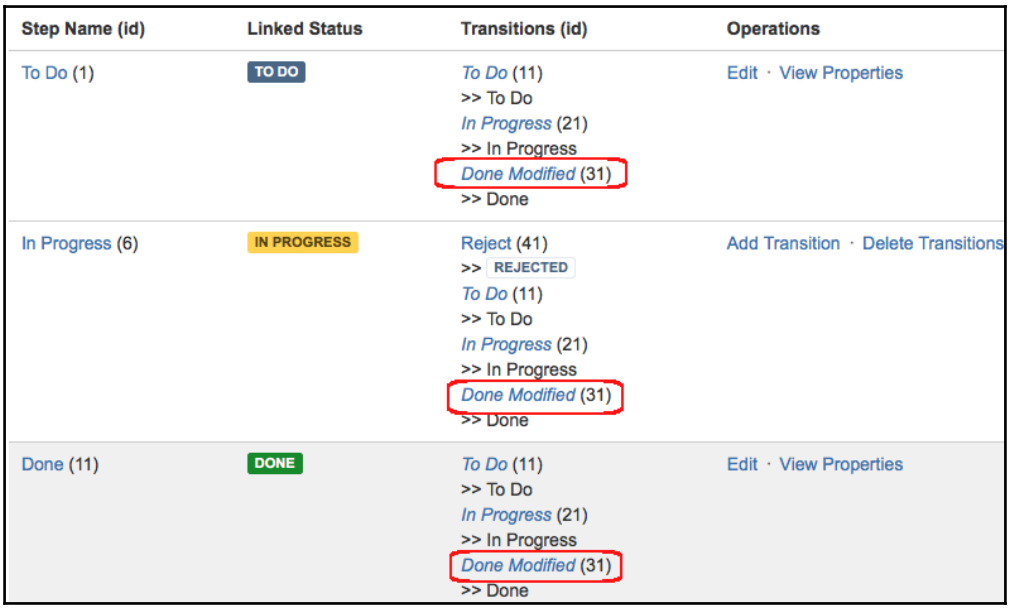
As you can see, **Reject** now appears before **Done**. Please note that the order of the workflow actions are changed only in the **View Issue** page and not in the **View Workflow Steps** page, where you modify the workflow steps.

Creating common transitions in workflows

Configuring workflows can be a painful thing, especially when there are similar transitions used in 10 different places and they get changed every now and then. The change might be the simplest thing possible, such as editing just the name of the transition, but we still end up modifying it in 10 places.

This is where **OSWorkflow**'s common actions come to our rescue. These are transitions that can be shared across the various statuses. The most important thing to remember here is that the target status of the common transitions will be the same, and they share the same workflow elements like triggers, conditions, validators, post functions, and even transition properties.

The default workflows in JIRA come with some common transitions. For example, the **Software Simplified Workflow** comes with common transitions like **In Progress** and **Done**. If you need to modify one of these transitions, you need to modify it only once and the change will be made everywhere. If we modify the **Done** transition name to **Done Modified**, it changes everywhere, as shown below:



In this recipe, let us look at adding a new common transition.

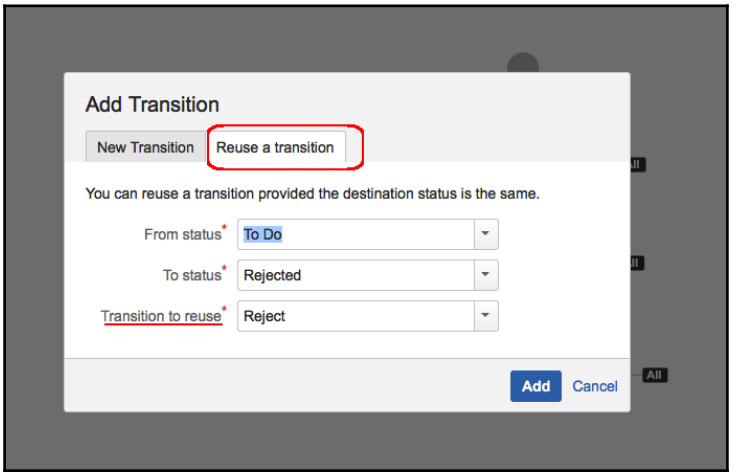
How to do it…

The first step in adding a common transition is to create a workflow transition, with all the required elements in it. Let us assume that we have added a new **Rejected** state in our workflow and created a transition from **In Progress** to **Rejected** state, named **Reject**.

If we need to add the same transition, that is, to **Rejected** state from another source status like **To Do**, we can do this by creating a common transition.

The following is the step-by-step procedure:

1. Create the common transition, **Reject**, as explained previously.
2. Create a transition from the new source status, **To Do** in this case, using the workflow **Diagram** view.
3. In the **Add Transition** pop up window, select the **Reuse a transition** tab.
4. Select the **From Status**, **To Do**, and the **To Status**, **Rejected**. Depending on our selection, the **Transition to reuse** box will be populated with available transitions that we can reuse, as shown here:



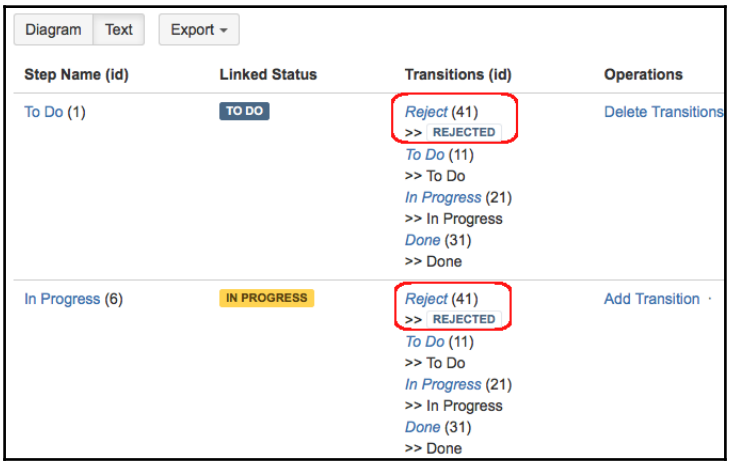
5. Click **Add**.

With that, the common transition is created! You can use the same transition again by following steps 2-4.

How it works…

JIRA workflows fundamentally use OpenSymphony's OSWorkflow, as we saw in `Chapter 2`, *Understanding Plugin Framework*. OSWorkflow gives us the flexibility to add common actions by modifying the workflow XML. JIRA provides the icing on the cake by providing a powerful HTML editor that can be used to create these common transitions very easily.

The following screenshot is how the updated workflow looks in the **Text** view:



As you can see, the new transition has the same ID, `41`, from both **To Do** and **In Progress** states. We can modify any attribute on this transition, and it will be reflected in all the places where the transition is used.

Creating global transitions in workflows

We have seen how to configure common transitions in the previous recipe. Global transitions in workflows are a similar useful feature. A global transition is a transition in which the destination step has all other steps in the workflow as incoming steps. That is, this transition will act as a transition from all steps to the destination step chosen in this transition, and you need to only modify in a single place if there is any change.

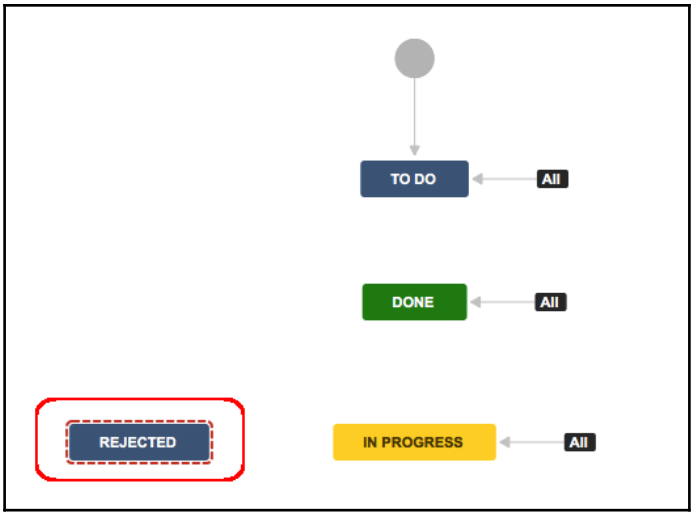
Creating a global transition is fairly easy using the workflow designer, and it is only supported in the diagram mode.

How to do it…

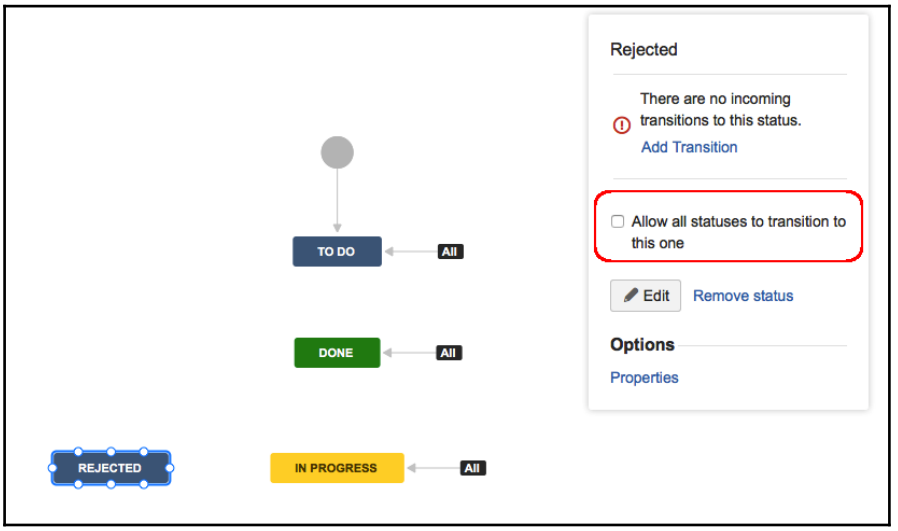
Let us take the example of **Reject** that we saw in the earlier recipe. What if we want to reject an issue from every single state in the workflow? In that case, it doesn't make sense to create common transitions as we will have to repeat those steps for every source status in the workflow.

Instead, we can create a single global transition to the Rejected status, as explained here:

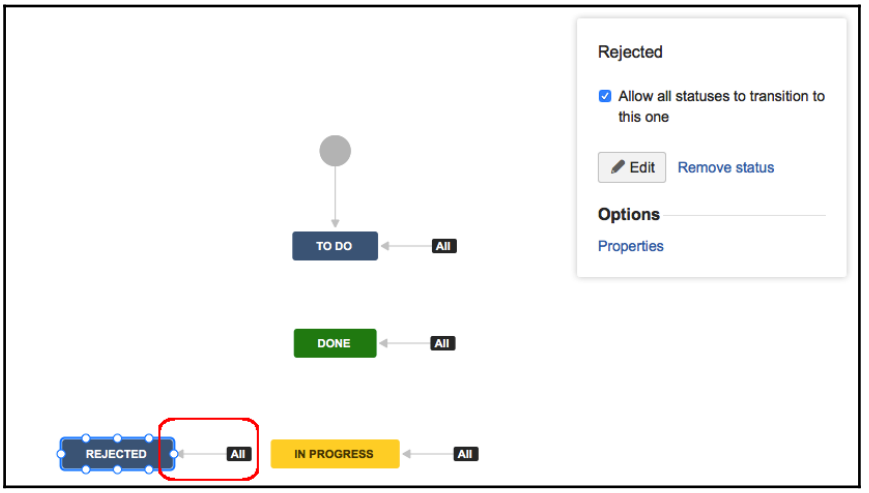
1. Go to the workflow editor in the **Diagram** view.
2. Add the new status in the workflow, **Rejected**, in this case. Once done, there are no transitions to this state, as shown here:



3. Click on the new status and check the box that says **Allow all statuses to transition to this one**, as shown here:



4. Once done, JIRA adds a global transition to the new status from all other statuses, as shown here:



5. You may now click on the transition and modify its properties or add workflow elements such as triggers, conditions, validators, or post functions.

How it works…

The following screenshot shows what the updated workflow looks like in the **Text** view:



As you might have noticed, all the transitions in our example are global transitions, including the one we just added!

5
Gadgets and Reporting in JIRA

In this chapter, we will cover the following topics:

- Writing a JIRA report
- Reports in Excel format
- Data validation in JIRA reports
- Restricting access to reports
- Object configurable parameters for reports
- Writing JIRA gadgets
- Invoking REST services from gadgets
- Configuring user preferences in gadgets
- Accessing gadgets outside of JIRA

Introduction

Reporting support in an application like JIRA is inevitable! With so much data spanning different projects and issues, and lots of project planning done on it, we need more and more reports with customized data according to our needs.

There are two different kinds of reporting available in JIRA:

- **Gadgets** that can be added into a user's dashboard – gadgets are mini applications built using HTML and JavaScript that can run on any OpenSocial gadget container. They communicate with JIRA using REST APIs and retrieve the required information before rendering the display for the user appropriately. As the JIRA dashboard is now an OpenSocial gadget container, we can even add third-party gadgets onto it, provided they meet the gadget specifications. Similarly, JIRA gadgets can be added onto other containers like iGoogle, Gmail, and so on, but other gadget containers do not support all features of JIRA gadgets.
- **Normal JIRA reports** – JIRA also provides an option to create reports that show statistics for particular people, projects, versions, or other fields within issues. These reports can be invoked from a number of places in the user interface, and can be used to generate simple tabular reports, charts, and so on, and can then be exported to Excel if supported.

JIRA provides a number of built-in reports, the details of which can be found at `https://confluence.atlassian.com/jira/generating-reports-185729499.html`.

In addition to the **gadgets** and **reports** that JIRA provides, there are a lot of them available in the Atlassian marketplace. But still, we might end up writing some that are customized specifically for our organization and that is where JIRA's plugin architecture helps us by providing two plugin modules – one for reports and another for gadgets.

In this chapter, we will look at writing JIRA reports and gadgets in more detail.

Writing a JIRA report

As we just mentioned, a JIRA report can display statistical information based on all elements within JIRA – for example, issues, projects, users, issue types, and so on. They can have HTML results and, optionally, Excel results.

To add new reports in JIRA, you can use the **report plugin module**. The following are the key attributes and elements supported:

Attributes:

Name	Description
`key`	This should be unique within the plugin.
`class`	Class to provide contexts for rendered velocity templates. Must implement the `com.atlassian.jira.plugin.report.Report` interface. Recommended to extend the `com.atlassian.jira.plugin.report.impl.AbstractReport` class.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	Human-readable name of the report. Appears in the plugins page. Default is the plugin key.

Elements:

Name	Description
`description`	Description of the report.
`label`	User-visible name of the report.
`resource type="velocity"`	Velocity templates for the report views.
`resource type="18n"`	Javaproperties file for the i18n localization.
`properties`	Reports configurable parameters that used to accept user inputs.

Getting ready

Create a skeleton plugin using the Atlassian plugin SDK.

How to do it…

Let us consider creating a very simple report with little business logic in it. The example we choose here is to display the **key** and **summary** of all the *issues* in a selected*project*. The only input for the report will be the *project name*, which can be selected from a drop-down list.

The following is the step-by-step procedure to create this report:

1. Add the report plugin module in the plugin descriptor.
   In this first step, we will look at populating the entire plugin module in the `atlassian-plugin.xml` file:

   a. Include the report module:

```
<report name="All Issues Report"
 i18n-name-key="all-issues-report.name"
 key="all-issues-report"
 class="com.jtricks.jira.reports.AllIssuesReport">
    <description key="all-issues-report.description">
      This report shows details of all issues a specific project
    </description>
    <label key="all-issues-report.label"></label>
</report>
```

   As usual, the plugin module should have a unique **key**. The other most important attribute here is the **class**. `AllIssuesReport`, in this case, is the class that populates the context for the velocity templates used in the report display. It holds the business logic to retrieve the report results based on the criteria entered by the user.

   b. Include the `i18n` property resource that can be used for internationalization within the report. You can skip this part if you do not prefer a separate i18n resource. In that case, the key/value properties will be added at the i18n resource defined at the plugin level:

```
<!-- this is a .properties file containing the i18n keys for
this report -->
<resource type="i18n" name="i18n"
location=" com.jtricks.jira.reports.AllIssuesReport" />
```

   Here, the `AllIssuesReport.properties` file will be present in the `com.jtricks.jira.reports` package under the resources folder in your plugin. All the keys that you used should be present in the properties file with the appropriate values.

   c. Include the velocity template resources within the report module:

```
<resource name="view" type="velocity"
 location="templates/allissues/allissues-report.vm"/>
```

Here, we defined the velocity template that will be used to render the HTML view for the report.

d. Define the user-driven properties:

```
<!-- the properties of this report which the user must select
before running it -->
<properties>
   <property>
    <key>projectId</key>
    <name>report.allissues.project.name</name>
    <description>
      report.allissues.project.description
    </description>
    <type>select</type>
    <values class=
     "com.jtricks.jira.reports.ProjectValuesGenerator"/>
   </property>
</properties>
```

This is a list of properties that will be rendered appropriately on the report's configuration page. In our example, we need to select a project from a select list before generating the report. For this, we have defined a project property here for which the type is `select`. JIRA will automatically render this as a select list by taking the key/value pair from the `ProjectValuesGenerator` class. We will see more details on the types supported in the coming recipes.

Now we have the plugin descriptor filled in with the details required for the report plugin module. The entire module now looks as follows:

```
<report name="All Issues Report"
 i18n-name-key="all-issues-report.name"
 key="all-issues-report"
 class="com.jtricks.jira.reports.AllIssuesReport">
  <description key="all-issues-report.description">
    This report shows details of all issues a specific project
  </description>
  <resource name="view" type="velocity"
   location="templates/allissues/allissues-report.vm"/>
  <resource name="excel" type="velocity"
   location="templates/allissues/allissues-report-excel.vm"/>
  <label key="all-issues-report.label"></label>
  <!-- the properties of this report which the user must select
   before running it -->
   <properties>
     <property>
      <key>projectId</key>
```

```
      <name>report.allissues.project.name</name>
      <description>
       report.allissues.project.description</description>
      <type>select</type>
      <values class=
        "com.jtricks.jira.reports.ProjectValuesGenerator"/>
     </property>
   </properties>
</report>
```

2. Populate the `i18n` resource properties file. As mentioned, this could be the i18n file defined for the report module or this could be the one at plugin level. Populate the appropriate one with the key/value properties:

```
all-issues-report.label=All Issues Report
all-issues-report.name=All Issues Report
all-issues-report.description=This report shows details of all
 issues a specific project
report.allissues.project.name=Project
report.allissues.project.description=Select the project
```

3. Create the **value generator** class. This is the class that is used to generate the values to be used for rendering the user properties on the report input page. In our example, we have used the `ProjectValuesGenerator` class.
   The class that generates the values should implement the `ValuesGenerator` interface. It should then implement the `getValues()` method to return a key/value map. The value will be used for display, and the key will be returned as the property value, which will be used in the report class.
   In the `ProjectValuesGenerator` class, we use the project ID and the name as the key/value pair:

```
public class ProjectValuesGenerator implements
ValuesGenerator<String> {
  public Map<String, String> getValues(Map userParams) {
    Map<String, String> projectMap = new HashMap<String,
    String>();
      List<Project> allProjects =
        ComponentAccessor.getProjectManager().getProjectObjects();
      for (Project project : allProjects) {
        projectMap.put(project.getId().toString(),
        project.getName());
      }
      return projectMap;
  }
}
```

4. Create the report class. This is where the actual business logic lies.

   The report class, `AllIssuesReport` in this case, should extend the `AbstractReport` class. It can just implement the `Report` interface, but `AbstractReport` has some already implemented methods, and hence is recommended.

   The only mandatory method we need to implement here is the `generateReportHtml` method. We need to populate a map here that can be used to render the velocity views. In our example, we populate the map with variable issues, which is a list of issue objects in the selected project.

   The selected project can be retrieved using the key value entered in the property in the `atlassian-plugin.xml` file:

   ```
   final String projectid = (String) reqParams.get("projectId");
   final Long pid = new Long(projectid);
   ```

   We can now use this `pid` to retrieve the list of issues using the method `getIssuesFromProject`:

   ```
   List<Issue> getIssuesFromProject(Long pid) throws SearchException {
     JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
     builder.where().project(pid);
     Query query = builder.buildQuery();
     SearchResults results =
       this.searchService.search(this.authContext.getLoggedInUser(),
       query, PagerFilter.getUnlimitedFilter());
     return results.getIssues();
   }
   ```

   Now all we need to do here is populate the map with this and return the rendered view as follows:

   ```
   final Map<String, Object> velocityParams = new HashMap<String,
     Object>();
   velocityParams.put("issues", getIssuesFromProject(pid));
   return descriptor.getHtml("view", velocityParams);
   ```

   You can populate any useful variable like this, and it can then be used in the velocity templates to render the view.

   The class now looks as follows:

   ```
   @Scanned
   public class AllIssuesReport extends AbstractReport {
   ```

```
    private final JiraAuthenticationContext authContext;
    private final SearchService searchService;
    private final ProjectManager projectManager;
    public AllIssuesReport(@ComponentImport JiraAuthenticationContext
      authContext, @ComponentImport SearchService searchService,
      @ComponentImport ProjectManager projectManager) {
          super();
          this.authContext = authContext;
          this.searchService = searchService;
          this.projectManager = projectManager;
      }
      public String generateReportHtml(ProjectActionSupport action,
      Map reqParams) throws Exception {
        final Map<String, Object> velocityParams =
          getVelocityParams(action, reqParams);
        return descriptor.getHtml("view", velocityParams);
      }
      private Map<String, Object>
      getVelocityParams(ProjectActionSupport action,
      Map reqParams) throws SearchException {
          final String projectid = (String)
            reqParams.get("projectId");
          final Long pid = new Long(projectid);
          final Map<String, Object> velocityParams =
            new HashMap<String, Object>();
            velocityParams.put("report", this);
            velocityParams.put("action", action);
            velocityParams.put("issues", getIssuesFromProject(pid));
            return velocityParams;
      }
      List<Issue> getIssuesFromProject(Long pid)
      throws SearchException {
        JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
        builder.where().project(pid);
        Query query = builder.buildQuery();
        SearchResults results =
        this.searchService.search(this.authContext.getLoggedInUser(),
        query, PagerFilter.getUnlimitedFilter());
        return results.getIssues();
      }
}
```



Note the use of Atlassian Spring Scanner annotations for constructor injection. You can read more about it at `https://bitbucket.org/atlassian/atlassian-spring-scanner`.

5. Create the velocity template. In our case, we are using `templates/allissues/allissues-report.vm`. We will use the issues variable we populated in the report class, iterate on it, and display the issue key and summary:

```
#disable_html_escaping()
<table class="aui">
  <thead>
    <tr>
      <th id="key">Key</th>
      <th id="summary">Summary</th>
    </tr>
  </thead>
  <tbody>
    #foreach ($issue in $issues)
    <tr>
      <td headers="key">
        <a href="/browse/$issue.key">$issue.key</a>
      </td>
      <td headers="summary">$issue.summary</td>
    </tr>
    #end
  </tbody>
</table>
```

6. With that, our report is ready. Package the plugin and deploy it.

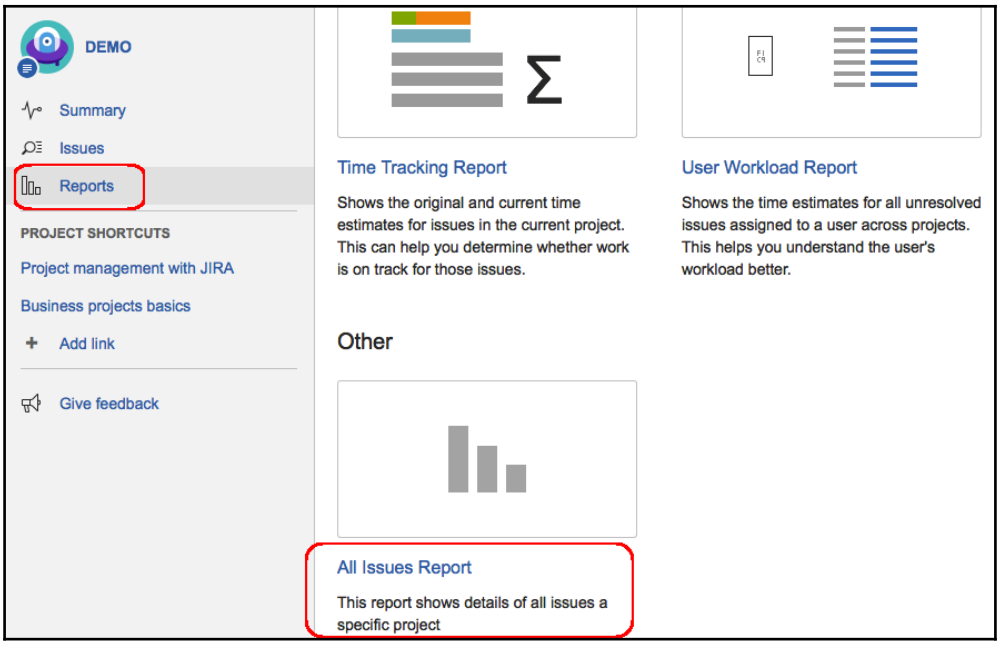
We will see more on creating Excel reports, validation within reports, and so on in the coming recipes.

How it works…

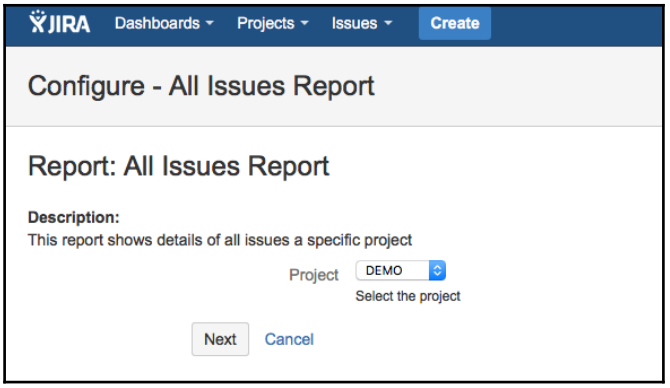
The whole logic of how it works can be outlined as follows.

1. The input view of reports is generated by the object configurable properties, a set of predefined properties used to populate input parameters in JIRA. In our example, we used the `select` property. We will see more of this in detail later in this chapter.
2. The report class gets these properties, uses them to retrieve the details required in the report, and populates the velocity context with the details.
3. Velocity templates use these details to render the report.

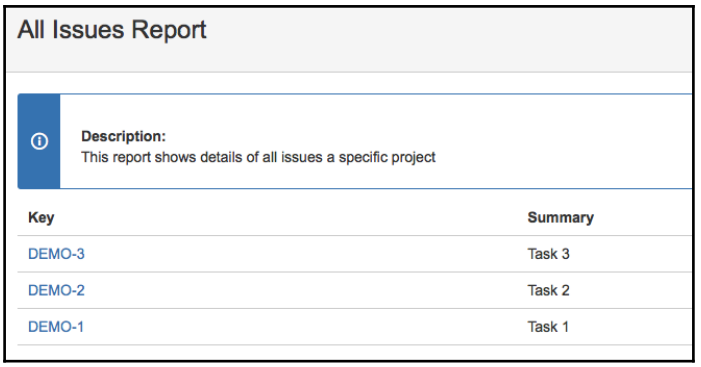
After the plugin is deployed, you can see the report among other JIRA reports in the **Reports** section, as shown in the following screenshot:



After clicking on the report, the input screen is displayed, which is constructed using the properties entered in the plugin descriptor, the **Project** drop-down list in our case:



Clicking on **Next**, the report will be generated using the report class and will be rendered using the view template as follows:



See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Reports in Excel format

In the previous recipe, we saw how to write a simple report. We will now see how to modify the report plugin to include Excel reports.

Getting ready

Create the report plugin, as mentioned in the previous recipe.

How to do it…

The following are the steps to include the provision of exporting the report to Excel:

1. Add the velocity resource type for the Excel view in the plugin descriptor if not added already:

```
<resource type="velocity" name="excel"
  location="templates/allissues/allissues-report-excel.vm" />
```

2. Override the `isExcelViewSupported` method in the report class to return true. In our case, we add this in the `AllIssuesReport.java`:

```
@Override
public boolean isExcelViewSupported() {
  return true;
}
```

   This method returns false by default, as it is implemented that way in the `AbstractReport` class.

3. Override the `generateReportExcel` method returning the Excel view. This is very similar to the `generateReportHtml` we implemented in the previous recipe. The only difference is the view returned. The method looks as follows:

```
@Override
public String generateReportExcel(ProjectActionSupport action,
Map reqParams) throws Exception {
    return descriptor.getHtml("excel", getVelocityParams(action,
    reqParams));
}
```

   Here, the `getVelocityParams` method is exactly the same as what is used in the `generateReportHtml` method in the previous recipe. It retrieves the list of issues and populates the map of velocity parameters with the variable name issues.

4. Create the Excel velocity template. The template is created using HTML tags and velocity syntax, just like the other templates. In our example, it will be `allissues-report-excel.vm` under the folder `templates/allissues/` under resources. This is where the view can be customized for Excel.

   In our example, all we have is a list of issues with its summary and key. Hence, we can even use the same template for Excel. It appears as follows:

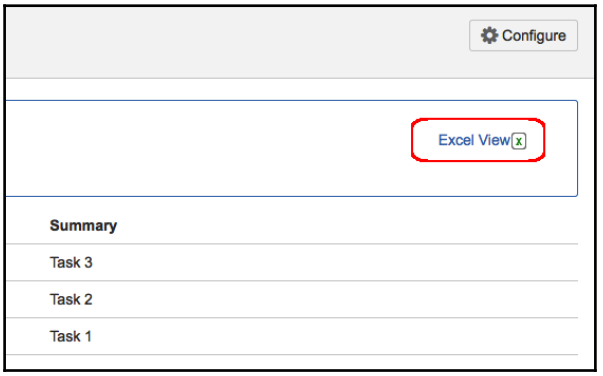
```
<table id="allissues-report-table" border="0" cellpadding="3"
  cellspacing="1" width="100%">
  <tr class="rowNormal">
    <th>Key</th>
    <th>Summary</th>
  </tr>
  #foreach ($issue in $issues)
  <tr class="rowNormal">
    <td>$issue.key</td>
    <td>$issue.summary</td>
  </tr>
  #end
```

```
</table>
```

5. Package the plugin and deploy it.

How it works…

Once the Excel view is added into the reports, a link **Excel View** will appear on the right-hand top side of the generated reports, as shown in the following screenshot:



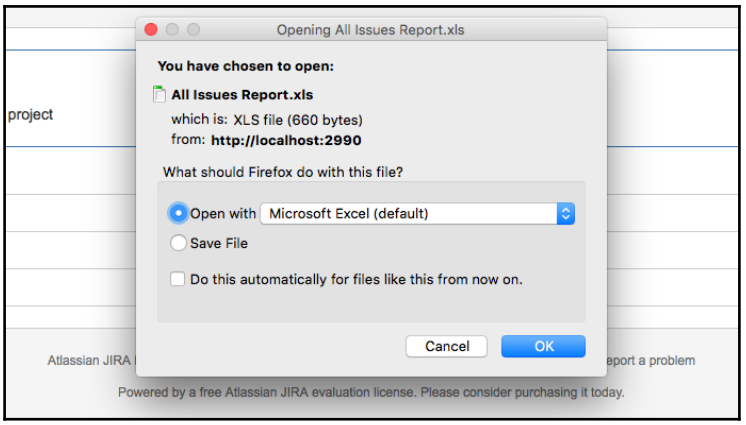
On clicking the link, the `generateReportExcel` method is executed, which in turn will generate the report and render the Excel view using the appropriate template that is defined in the plugin descriptor.

You may have noticed that when you click on the **Excel View** link, the Excel report that opens is of the name `ConfigureReport!excelView.jspa`, and we need to rename that to `.xls` to make it Excel-friendly.

To do it automatically, we need to set the **content-disposition** parameter in the response header, as shown:

```
final StringBuilder contentDispositionValue = new StringBuilder(50);
contentDispositionValue.append("attachment;filename="");
contentDispositionValue.append(getDescriptor().getName()).append(".xls";");
final HttpServletResponse response = ActionContext.getResponse();
response.addHeader("content-disposition",
contentDispositionValue.toString());
```

This snippet is added in the `generateReportExcel` method before returning the excel view using the descriptor. The report will now open as a `.xls` file and can then be opened in Excel without any renaming:





Please refer to `http://support.microsoft.com/kb/260519` and `http://jira.atlassian.com/browse/JRA-8484` for some details about this.

See also

- The *Writing a JIRA report* recipe of this chapter

Data validation in JIRA reports

Whenever we take user inputs, it is always a good idea to validate them to make sure the input is in the format that is expected. The same applies to reports also. JIRA reports, as we have seen in the previous recipes, accept user inputs based on which reports are generated. In the example we used, a project is selected and the details of issues in the selected project are displayed.

In the previous example, the likelihood of a wrong project being selected is as low, as the project is selected from a valid list of available projects. But still, the final URL that generates the report can be tampered with to include a wrong project ID, and so it is best to do the validation no matter how the input is taken.

Getting ready

Create the report plugin, as explained in the first recipe.

How to do it…

All we need here is to override the validate method to include our custom validations. The following are the steps:

1. Override the `validate` method in the report class we created in the previous recipe.
2. Extract the input parameters from the request parameters, which is an argument to the `validate` method:

   ```
   final String projectid = (String) reqParams.get("projectId");
   ```

   `reqParams` here is an argument of the `validate` method:

   ```
   public void validate(ProjectActionSupport action, Map reqParams)
   ```

3. Check the validity of the input parameter. In our example, the input parameter is the `projectId`. We can check if it is valid by verifying if a project exists with the given ID. The following condition returns true if it is an invalid project ID:

   ```
   if (ComponentAccessor.getProjectManager()
   .getProjectObj(pid) == null)
   ```

4. If the parameter is invalid, add an error to the action with the appropriate error message:

   ```
   action.addError("projectId", "Invalid project Selected");
   ```

   Here we pass the field name to the `addError` method so that the error message appears at the top of the field.You can use internationalization here as well to include appropriate error messages.

5. Add similar validation for all the interested parameters. The following is what the method looks like in our example:

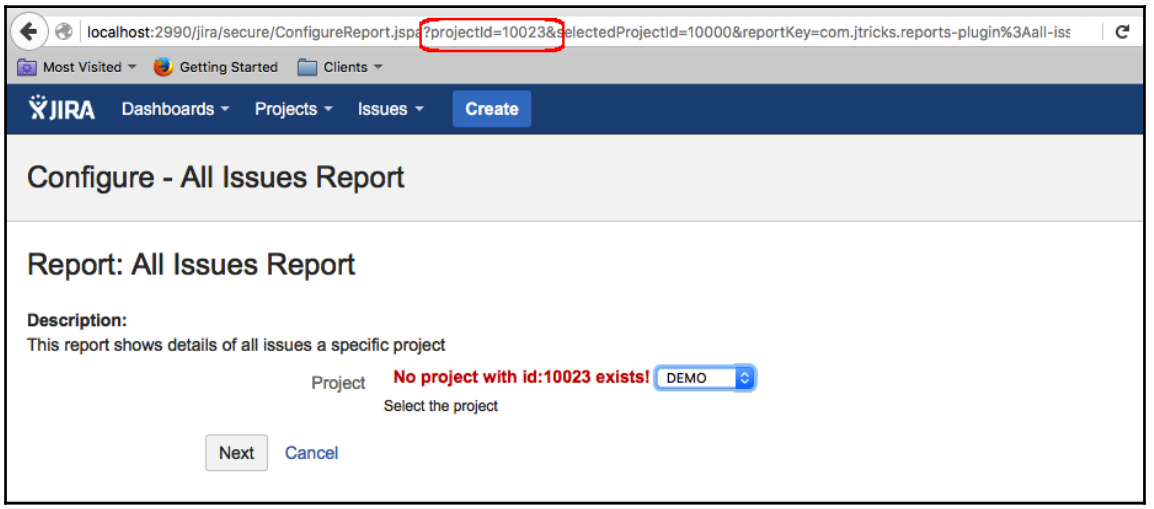
   ```
   @Override
   public void validate(ProjectActionSupport action, Map reqParams) {
     final String projectid = (String) reqParams.get("projectId");
     final Long pid = new Long(projectid);
     if (this.projectManager.getProjectObj(pid) == null) {
   ```

```
            action.addError("projectId", "No project with id:" +
            pid + " exists!");
          }
          super.validate(action, reqParams);
        }
```

6. Package the plugin and deploy it!

How it works…

Just before the report is generated, the `validate` method is executed. If there is any error, the user is taken back to the input screen with the error highlighted as follows:



This example shows an error when the report URL is tampered with to include an invalid project with ID: `10023`.

See also

- The *Writing a JIRA report* recipe of this chapter

Restricting access to reports

It is possible to restrict access to JIRA reports based on predefined criteria, such as making the report visible only to a certain group of people, or showing the report only in certain projects, and so on. Let us quickly have a look at how to code permissions for a JIRA report.

Getting ready

Create the report plugin, as explained in the first recipe.

How to do it…

All we need to do here is to implement the `showReport` method on the report. Let us assume we want to restrict the report only to JIRA Administrators. The following are the steps:

1. Override the `showReport` method in the report class we created in the previous recipes.
2. Implement the logic to return `true` only if the condition is satisfied. In our example, the report should be visible only to JIRA Administrators, and hence we should return `true` only if the current user is a JIRA Administrator:

```
@Override public boolean showReport() {
    ApplicationUser user = this.authContext.getLoggedInUser();
    return this.userUtil.getJiraAdministrators().contains(user);
}
```

3. Package the plugin and deploy it.

How it works…

If the user is an administrator, he/she will see the report link under the **Reports** section. If not, the report link won't be visible. We can include similar condition checks and evaluate them in the `showReport` method before returning true.

See also

- The *Writing a JIRA report* recipe of this chapter

Object configurable parameters for reports

We have seen how to write JIRA reports and we also had a brief look at how JIRA lets us configure the input parameters. The example we have chosen in the previous recipe, on creating JIRA reports, explained the usage of the `select` type. In this recipe, we will see the various property types supported and some examples on how to configure them.

There are a number of property types supported in JIRA. The full list supported by your JIRA version can be found in the `com.atlassian.configurable.ObjectConfigurationTypes` class. For JIRA 7.1.*, the following are the types supported for reports:

Type	**Input HTML type**
`string`	Textbox
`long`	Textbox
`hidden`	NA. Hidden to the user
`Date`	Textbox with calendar popup
`user`	Textbox with user picker
`text`	Text area
`select`	Select list
`multiselect`	Multiselect list
`checkbox`	Checkbox
`filterpicker`	Filter picker
`filterprojectpicker`	Filter or project picker
`cascadingselect`	Cascading select list dependent on a parent select list

How to do it…

Let us quickly look at each property and how it is used:

- **string**: The `string` property is used to create a Text Box. The Java data type is **string**. All you need here is to add the `property` tag with the type as `string`:

```
<property>
  <key>testString</key>
  <name>Test String</name>
  <description>Example String property</description>
  <type>string</type>
  <default>test val</default>
</property>
```

  Each of the property types, including the `string` property, can have a default value populated using the `default` tag, as shown.

- **long**: The `long` property is used to create a Text Box. The Java data type is again **string**:

```
<property>
  <key>testLong</key>
  <name>Test Long</name>
  <description>Example Long property</description>
  <type>long</type>
</property>
```

- **select**: The select property is used to create a Select List. The Java data type is string. We have seen an example of this in the previous recipe. There are two ways you can populate the values of a select property:

  Using a value generator class: The class should implement the `ValuesGenerator` interface and return a map of key/value pairs. The *key* will be the value returned to the report class, whereas the *value* is the display value to the user. Let us use the same example from the previous recipe here:

```
<property>
  <key>projectId</key>
  <name>Project</name>
  <description>report.allissues.project.description</description>
  <type>select</type>
  <values class=
   " com.jtricks.jira.reports.ProjectValuesGenerator "/>
</property>
```

`ProjectValuesGenerator` implements the `getValues()` method as follows:

```
public class ProjectValuesGenerator implements ValuesGenerator{
  public Map<String, String> getValues(Map userParams) {
    Map<String, String> projectMap = new HashMap<String, String>();
    List<Project> allProjects =
      ComponentAccessor.getProjectManager().getProjectObjects();
    for (Project project : allProjects) {
      projectMap.put(project.getId().toString(),
      project.getName());
    }
    return projectMap;
  }
}
```

Using pre-defined key/value pairs in the property: The following is an example:

```
<property>
  <key>testSelect</key>
  <name>Test Select</name>
  <description>Example Select Property</description>
  <type>select</type>
  <values>
    <value>
      <key>key1</key>
      <value>Key 1</value>
    </value>
    <value>
      <key>key2</key>
      <value>Key 2</value>
    </value>
    <value>
      <key>key3</key>
      <value>Key 3</value>
    </value>
  </values>
</property>
```

- **multiselect**: The `multiselect` property is used to create a Multi Select List. It is the same as the select property. The only difference is that the type name is `multiselect`. Here the Java type will be a **string** if only one value is selected, and it will be an array of strings `String[]` if more than one value is selected.

- **hidden**: The `hidden` property is used to pass a hidden value. The Java data type is `string`:

```
<property>
  <key>testHidden</key>
  <name>Test Hidden</name>
  <description>Example Hidden property</description>
  <type>hidden</type>
  <default>test hidden val</default>
</property>
```

  We must provide a value using the `default` tag, as the user won't be able to enter a value.

- **date**: The `date` property is used to create a date picker. The Java data type is **string**. We should then parse it to the `Date` object in the report:

```
<property>
  <key>testDate</key>
  <name>Test Date</name>
  <description>Example Date property</description>
  <type>date</type>
</property>
```

- **user**: The `user` property is used to create a user picker. The Java data type is **string** and it will be the username:

```
<property>
  <key>testUser</key>
  <name>Test User</name>
  <description>Example User property</description>
  <type>user</type>
</property>
```

- **text**: The `text` property is used to create a text area. The Java data type is **string**:

```
<property>
  <key>testText</key>
  <name>Test Text Area</name>
  <description>Example Text property</description>
  <type>text</type>
</property>
```

- **checkbox**: The `checkbox` property is used to create a checkbox. The Java data type is **string** and the value will be `true` if selected. If the checkbox is unchecked, the value will be `null`:

```
<property>
  <key>testCheckbox</key>
  <name>Test Check Box</name>
  <description>Example Checkbox property</description>
  <type>checkbox</type>
</property>
```

- **filterpicker**: The `filterpicker` property is used to create a filter picker. The Java data type is string and it will hold the ID of the selected filter:

```
<property>
  <key>testFilterPicker</key>
  <name>Test Filter Picker</name>
  <description>Example Filter Picker property</description>
  <type>filterpicker</type>
</property>
```

- **filterprojectpicker**: Used to create a filter or project picker. The Java data type is string, and it will be the ID preceded by the filter (if a filter is selected) and project (if a project is selected):

```
<property>
  <key>testFilterProjectPicker</key>
  <name>Test Filter or Project Picker</name>
  <description>
    Example Filter or Project Picker property
  </description>
  <type>filterprojectpicker</type>
</property>
```

- **cascadingselect**: Used to create a cascading select, based on another select box:

```
<property>
  <key>testCascadingSelect</key>
  <name>Test Cascading Select</name>
  <description>Example Cascading Select</description>
  <type>cascadingselect</type>
  <values class="com.jtricks.CascadingValuesGenerator"/>
  <cascade-from>testSelect</cascade-from>
</property>
```

Here, the cascading select `testCascadingSelect` depends on the select property named `testSelect`. We have seen the `testSelect` property with the key/value pairs. The next important thing is the value generator class. As with the other value generator classes, this one also generates a map of key/value pairs.

Here, the key in the key/value pair should be the value that will be returned to the user. The value should be an instance of a `ValueClassHolder` class, which is a static class. The `ValueClassHolder` class will look like the following:

```
private static class ValueClassHolder {
    private String value;
    private String className;

    public ValueClassHolder(String value, String className) {
      this.value = value;
      this.className = className;
    }

    public String getValue() {
      return value;
    }

    public String getClassName() {
      return className;
    }

    public String toString() {
      return value;
    }
}
```

The `value` in the `ValueClassHolder` will be the display value of the cascading select options to the user. The `className` attribute will be the `key` of the parent select option.

In our example, the parent select property is `testSelect`. It has three keys – `key1`, `key2`, and `key3`. The `getValues()` method will, therefore, look as follows:

```
public Map getValues(Map arg0) {
    Map allValues = new LinkedHashMap();
    allValues.put("One1", new ValueClassHolder("First Val1", "key1"));
    allValues.put("Two1", new ValueClassHolder("Second Val1", "key1"));
    allValues.put("Three1", new ValueClassHolder("Third Val1", "key1"));
    allValues.put("One2", new ValueClassHolder("First Val2", "key2"));
    allValues.put("Two2", new ValueClassHolder("Second Val2", "key2"));
    allValues.put("One3", new ValueClassHolder("First Val3", "key3"));
    return allValues;
}
```

If you take a single line, for example, `allValues.put("One1", new ValueClassHolder("First Val1", "key1"))`, it will have the key/value pair `One1/First Val1` when the select list has the key `key1` selected!

After selecting the appropriate values, they can be retrieved in the report class, as shown in the following lines of code:

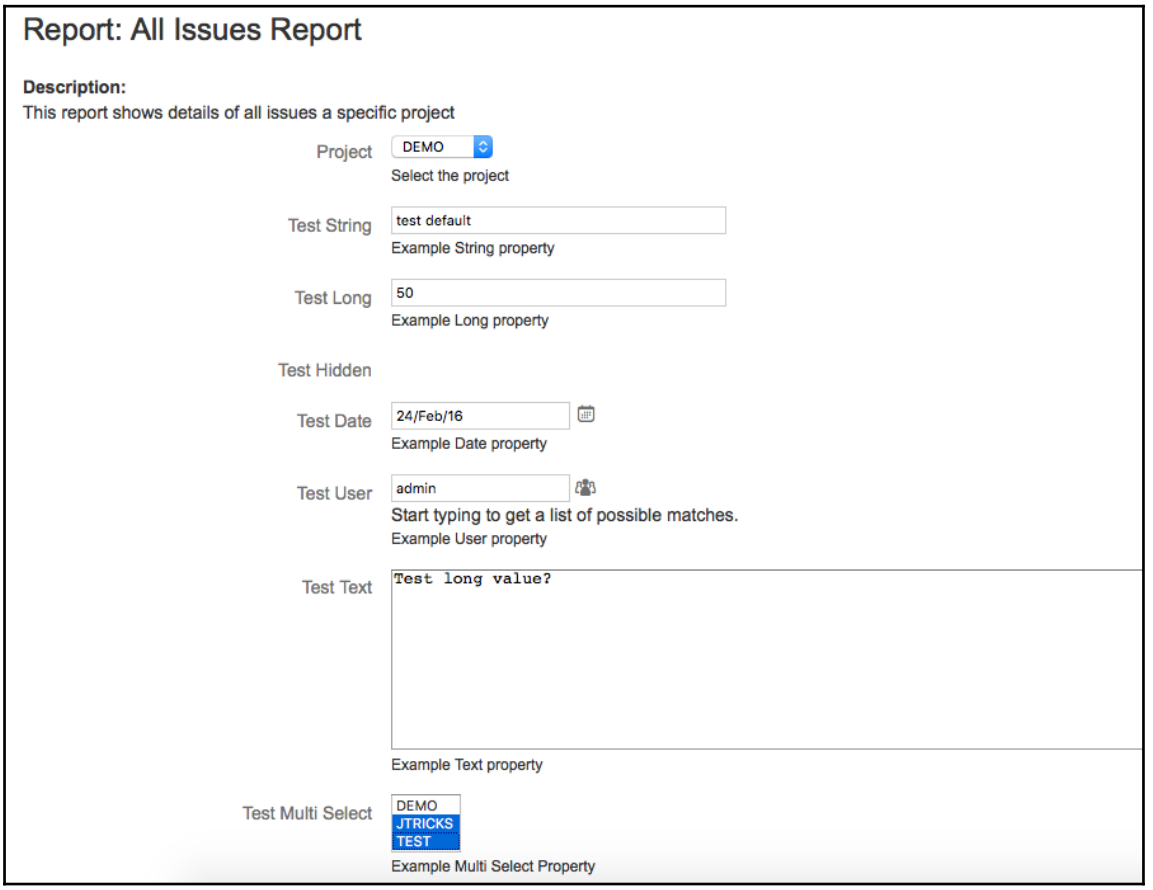
```
final String testString = (String) reqParams.get("testString");
final String testLong = (String) reqParams.get("testLong");
final String testHidden = (String) reqParams.get("testHidden");
final String testDate = (String) reqParams.get("testDate");
final String testUser = (String) reqParams.get("testUser");
final String testText = (String) reqParams.get("testText");
final String[] testMultiSelect = (String[])
reqParams.get("testMultiSelect");
final String testCheckBox = (String) reqParams.get("testCheckBox");
final String testFilterPicker = (String) reqParams.get("testFilterPicker");
final String testFilterProjectPicker = (String)
reqParams.get("testFilterProjectPicker");
final String testSelect = (String) reqParams.get("testSelect");
final String testCascadingSelect = (String)
reqParams.get("testCascadingSelect");
```

Special mention should be given to the `filterprojectpicker`. The value will be `filter-10000` if a filter with the ID 10000 is selected. The value will be `project-10000` if a project with ID `10000` is selected.

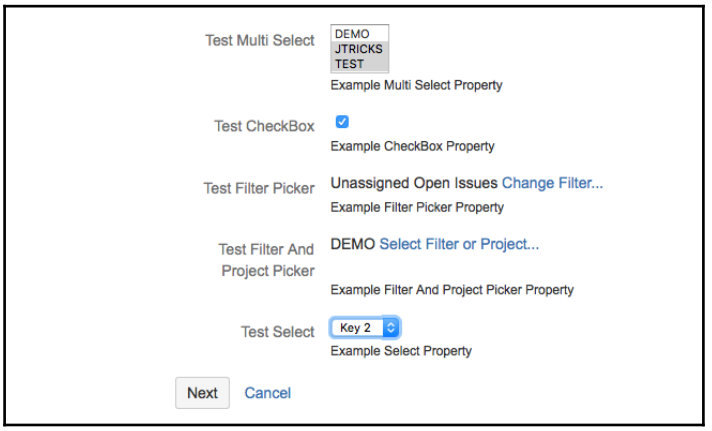
How it works…

When the report input screen is presented to the user, the properties mentioned in the plugin descriptor are converted into the appropriate HTML elements, as discussed. We can then retrieve their values in the report class and process them to generate the report.

The following screenshot shows how properties such as String, Long, Hidden, Date, User, Text and Multi Select fields appear on the input screen:



The following screenshot shows properties such as CheckBox, Filter Picker, Filter and Project Picker, and Select fields on the input screen:



If you print the extracted values into the console in the report class, it will appear as follows:

```
[INFO] [talledLocalContainer] Object Configurable Properties Demo
[INFO] [talledLocalContainer] ************************************
[INFO] [talledLocalContainer] Test String:test default
[INFO] [talledLocalContainer] Test Long:50
[INFO] [talledLocalContainer] Test Hidden:test hidden val
[INFO] [talledLocalContainer] Test Date:24/Feb/16
[INFO] [talledLocalContainer] Test User:admin
[INFO] [talledLocalContainer] Test Text:Test long value?
[INFO] [talledLocalContainer] Test Multi Select:[10101, 10100]
[INFO] [talledLocalContainer] Test Checkbox:true
[INFO] [talledLocalContainer] Test Filter Picker:10200
[INFO] [talledLocalContainer] Test Filter Project Picker:
[INFO] [talledLocalContainer] Test Select:key2
[INFO] [talledLocalContainer] ************************************
```

Hopefully, that gives you a fair idea of how to use object configurable parameters in JIRA reports.



As of JIRA 7.0.x, there is a bug that is preventing cascading selects from rendering properly. The bug was originally reported for Firefox and IE but is reproducible in other browsers as well. See `https://jira.atlassian.com/browse/JRA-22613` for the latest status.

See also

- The *Writing a JIRA report* recipe of this chapter

Writing JIRA gadgets

Gadgets are a big leap in JIRA's reporting features! The fact that JIRA is now an OpenSocial container lets its users add useful gadgets (both JIRA's own and third-party) into its dashboard. At the same time, gadgets written for JIRA can be added in other containers, such as iGoogle, Gmail, and so on!

In this recipe, we will have a look at writing a very simple gadget, one that says "Hello from JTricks". By keeping the content simple, it will let us concentrate more on writing the gadget!

Before we start writing the gadget, it is probably worth understanding the key components of a JIRA gadget:

- Gadget XML is the most important part of a JIRA gadget. It holds the specification of the gadget and includes the following:
    - **Gadget characteristics**: It includes title, description, author's name, and so on.
    - **Screenshot and a thumbnail location**: Please note that the screenshot is not used within Atlassian containers such as JIRA or Confluence. We can optionally add it if we want them to be used in other OpenSocial containers. The thumbnail will be used in the gadget listing.
    - **Required features:** The gadget container must provide for the gadget.
    - **User preferences:** This will be configured by the gadget users.
    - **Gadget content:** This is created using HTML and JavaScript.
- Image files referred to in the gadget XML.
- An `i18n` property file used for internationalization in the gadget.
- Optional CSS and JavaScript file used to render the display in the **Content** section of the gadget.

We will see each of them in the recipe.

Getting ready

Create a skeleton plugin using Atlassian plugin SDK.

How to do it…

The following are the steps to write our first gadget, one that shows the greeting from JTricks!

1. Modify the plugin descriptor with the gadget module. The following is how it is done:

```
<gadget key="hello-gadget" name="Hello Gadget"
  location="gadgets/hello-gadget.xml">
  <description>Hello Gadget!</description>
</gadget>
```

   As you can see, this has a unique `key` and it points to the `location` of the gadget XML. You can have as many gadget definitions as you want in your `atlassian-plugin.xml` file, but in our example, we stick with one.

2. Define any resources that you will need for your gadgets, such as image files, JS and CSS files, and so on. You can do this by defining downloadable resources, as explained at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/downloadable-plugin-resources`.
3. Define the `i18n` properties file that will be used in the gadget also as a downloadable resource:

```
<resource type="download" name="i18n/messages.xml"
  location="i18n/messages.xml">
  <param name="content-type" value="text/xml; charset=UTF-8"/>
</resource>
```

4. Add the screenshot and thumbnail images under the `src/main/resources/images` folder (or the appropriate folder defined in the resource definition). The thumbnail image should be of the size 120 x 60 pixels.

5. Add the `i18n` properties file under the `src/main/resources/i18n` folder. The name of the file we defined is `messages.xml`.

   This file is an XML file wrapped within the `messagebundle` tag. Each property in the file is entered as an XML tag, as shown next:

```
<msg name="gadget.title">Hello Gadget</msg>
```

The `msg` tag has a `name` attribute, which is the property, and the corresponding value is enclosed in the `msg` tag. We used three properties in our example and the entire file in our example looks like the following:

```
<messagebundle>
  <msg name="gadget.title">Hello Gadget</msg>
  <msg name="gadget.title.url">http://www.j-tricks.com</msg>
  <msg name="gadget.description">Example Gadget from J-Tricks</msg>
</messagebundle>
```

6. Write the gadget XML.

The gadget XML has a **module** element at the root of the XML. It has mainly three elements underneath – `ModulePrefs`, `UserPref`, and `Content`. We will write each of them in this example. The entire set of attributes and elements and other details of the gadget specification can be read at `https://developer.atlassian.com/display/GADGETS/Creating+your+Gadget+XML+Specification`.

a. `ModulePrefs` element: This element holds the information about the gadget. It also has two child elements – `Require` and `Optional`, that are used to define the required or optional features for the gadget. The following is how the `ModulePrefs` element looks in our example after it is populated with all the attributes:

```
<ModulePrefs
title_url="__MSG_gadget.title.url__"
description="__MSG_gadget.description__"
author="Jobin Kuruvilla"
author_email= author@gmail.com
screenshot='#staticResourceUrl("com.jtricks.reports-plugin:
  reports-plugin-resources", "screenshot.png")'
thumbnail='#staticResourceUrl("com.jtricks.reports-plugin:
  reports-plugin-resources", "thumbnail.png")'
height="150">
```

As you can see, it holds information such as `title`, `title URL` (to which the gadget title will link to), `description`, `author name` and `email`, `height` of the gadget, and URLs to screenshot and thumbnail images.Anything that starts with `__MSG_` and ends with `__` is a property that is referred from the `i18n` properties file.The `height` of the gadget is optional and 200 by default.

The images are referenced using `#staticResourceUrl`, where the first argument is the fully qualified gadget resource key which is of the form `${atlassian-plugin-key}:${resource-module-key}`. In our example, the plugin key is `com.jtricks.gadgets` and the web resource module is `reports-plugin-resources`.

b. Add the `Optional` gadget directory feature inside `ModulePrefs`. This is currently supported only in JIRA:

```
<Optional feature="gadget-directory">
  <Param name="categories">
    Other
  </Param>
</Optional>
```

In the example, we added the category as `Other`.

Other values supported for category are: `JIRA`, `Confluence`, `FishEye`, `Crucible`, `Crowd`, `Clover`, `Bamboo`, `Admin`, `Charts`, and `External Content`.

You can add the gadget to more than one category by adding all the categories within the `Param` element, each in a new line.

c. Include `Required` features, if needed, under the XML tag `require`. A full list of supported features can be found at `https://developer.atlassian.com/display/GADGETS/Including+Features+into+your+Gadget`.

We will omit this for now, to keep our first gadget really simple!

d. Add the `Locale` element to point to the `i18n` properties file:

```
<Locale messages=
"__ATLASSIAN_BASE_URL__/download/resources/
 com.jtricks.reports-plugin/i18n/messages.xml"/>
```

Here the property `__ATLASSIAN_BASE_URL__` will be automatically substituted with JIRA's configured base URL when the gadget is rendered. The path to the property file here is `__ATLASSIAN_BASE_URL__/download/resources/com.jtricks.reports-plugin`, where `com.jtricks.reports-plugin` is the Atlassian plugin key. The path to the XML file `/i18n/messages.xml` is what was defined in the resource module earlier.

e. Add **user preferences** if required, using the `UserPref` element. We will omit this one as well, as the `Hello Gadget` doesn't take any inputs from the user.

f. Add the `Content` for the gadget. This is where the gadget is rendered using HTML and JavaScript. In our example, we just need to provide the static text "Hello from JTricks" and it is fairly easy.

The entire content is wrapped within the `<![CDATA[` and `]]>`, so that they won't be treated as XML tags. The following is how it looks in our example:

```
<Content type="html" view="profile">
  <![CDATA[ Hello From JTricks ]]>
</Content>
```

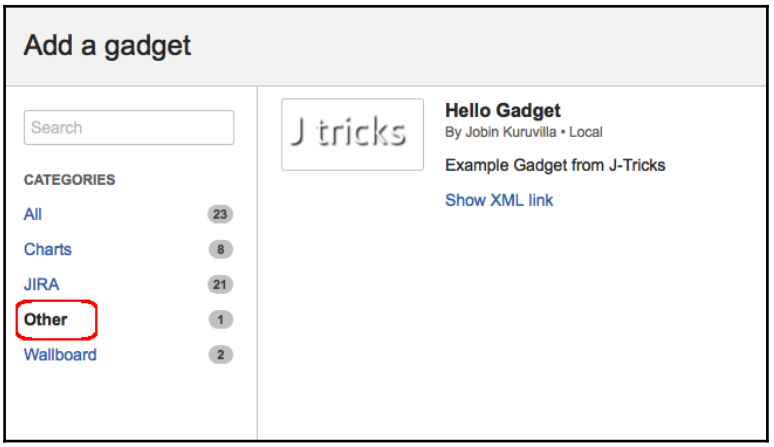
Our gadget XML is now ready and looks like the following block of code:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <Module>
    <ModulePrefs
    title_url="__MSG_gadget.title.url__"
    description="__MSG_gadget.description__"
    author="Jobin Kuruvilla"
    author_email="author@gmail.com"
    screenshot='#staticResourceUrl
      ("com.jtricks.reports-plugin:reports-plugin-resources",
      "screenshot.png")'
    thumbnail='#staticResourceUrl
      ("com.jtricks.reports-plugin:reports-plugin-resources",
      "thumbnail.png")'
    height="150">
      <Optional feature="gadget-directory">
        <Param name="categories">
           Other
        </Param>
      </Optional>
      <Require feature="dynamic-height"/>
        <Locale messages="__ATLASSIAN_BASE_URL__/download
         /resources/com.jtricks.reports-plugin
        /i18n/messages.xml"/>
    </ModulePrefs>
    <Content type="html" view="profile">
      <![CDATA[ Hello From JTricks ]]>
    </Content>
  </Module>
```

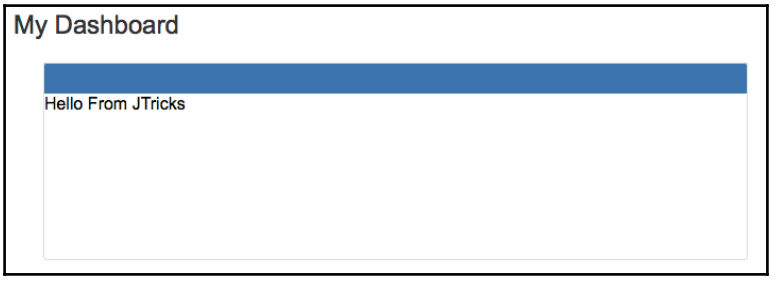
7. Package the plugin, deploy it, and test it.

How it works…

Once the plugin is deployed, we need to add the gadget in the JIRA dashboard. The following is how it appears in the **Add Gadget** screen. Note the thumbnail is the one we have in the plugin and also note that it appears in the **Other** section:



Once it is added, it appears as follows in the **Dashboards** section:



There's more…

We can modify the look and feel of the gadgets by adding more HTML or gadget preferences! For example, `<font color="red">Hello From JTricks</font>` will make it appear in red.

We can adjust the size of the gadget using the **dynamic-height** feature. We should add the following under the `ModulePrefs` element:

```
<Require feature="dynamic-height"/>
```

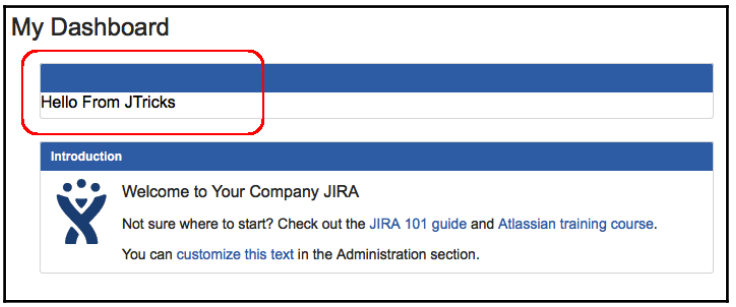
We should then invoke `gadgets.window.adjustHeight();` whenever the content is reloaded. For example, we can do it in a window onload event, as shown next:

```
<script type="text/javascript" charset="utf-8">
  function resize(){
    gadgets.window.adjustHeight();
  }
  window.onload=resize;
</script>
```

The updated `gadget xml` file will look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <Module>
    <ModulePrefs
       title_url="__MSG_gadget.title.url__"
       description="__MSG_gadget.description__"
       author="Jobin Kuruvilla"
       author_email="author@gmail.com"
       screenshot='#staticResourceUrl("com.jtricks.reports-plugin:
        reports-plugin-resources", "screenshot.png")'
       thumbnail='#staticResourceUrl("com.jtricks.reports-plugin:
        reports-plugin-resources", "thumbnail.png")'
       height="150">
      <Optional feature="gadget-directory">
        <Param name="categories">
          Other
        </Param>
      </Optional>
      <Require feature="dynamic-height"/>
      <Locale messages="__ATLASSIAN_BASE_URL__/download/resources
       /com.jtricks.reports-plugin/i18n/messages.xml"/>
    </ModulePrefs>
    <Content type="html" view="profile">
      <![CDATA[
        <script type="text/javascript" charset="utf-8">
          function resize(){
            gadgets.window.adjustHeight();
          }
          window.onload=resize; </script> Hello From JTricks ]]>
    </Content>
</Module>
```

The gadget should now appear as follows:



Note that the size is now adjusted to just fit the text!

Invoking REST services from gadgets

In the previous recipe, we saw how to write a gadget with static content. In this recipe, we will have a look at creating a gadget with dynamic content or the data that is coming from the JIRA server.

JIRA uses REST services to communicate between the gadgets and the server. We will see how to write REST services in the coming chapters. In this recipe, we will use an existing REST service.

Getting ready

Create the **Hello Gadget**, as described in the previous recipe.

How to do it…

Let us consider a simple modification to the existing **Hello Gadget** to understand the basics of invoking REST services from gadgets. We will try to greet the current user by retrieving the user details from the server instead of displaying the static text: **Hello From JTricks**.

JIRA ships with some built-in REST methods, one of which is to retrieve the details of the current user. The method can be reached at the URL: `/rest/gadget/1.0/currentUser`. We will use this method to retrieve the current user's full name and then display it in the gadget greeting. If the user's name is **Jobin Kuruvilla**, the gadget will display the message as **Hello, Jobin Kuruvilla**.



More such REST methods can be found in the REST API and the same for the latest JIRA version can be found at `http://docs.atlassian.com/jira/REST/latest/`.

As we are only changing the content of the gadget, the only modification is required in the gadget XML. Only the `Content` element needs to be modified, which will now invoke the REST service and render the content.

The following are the steps:

1. Include the common Atlassian gadget resources, in the `Content` element:

```
#requireResource("com.atlassian.jira.gadgets:common")
#includeResources()
```

   `#requireResource` will bring in the JIRA gadget JavaScript framework into the gadget's context. `#includeResources` will write out the HTML tags for the resource in place. Check out `https://developer.atlassian.com/display/GADGETS/Using+Web+Resources+in+your+Gadget` for more details.

2. Construct a gadget object as follows:

```
var gadget = AJS.Gadget
```

   The gadget object has four top-level options:

   a. `baseUrl`: An option to pass the base URL. It is a mandatory option, and we use `__ATLASSIAN_BASE_URL__` here, which will be rendered as JIRA's base URL.

   b. `useOauth`: An optional parameter. Used to configure the type of authentication, which must be a URL. `/rest/gadget/1.0/currentUser` is commonly used.

   c. `config`: Another optional parameter. Only used if there are any configuration options for the gadget.

   d. `view`: Used to define the gadget's view.

In our example, we don't use any configuration options. But for authentication, we must have the required feature: `oauthpopup`. This is added as shown under **ModulePrefs**:

```
<Require feature="oauthpopup" />
```

We will now look at the `baseUrl` and `view` options. The following is how the gadget is created using JavaScript:

```
<script type="text/javascript">
  (function () {
    var gadget = AJS.Gadget({
      baseUrl: "__ATLASSIAN_BASE_URL__",
      view: {
        ...
      }
    });
  })();
</script>
```

3. Populate the gadget view. The `view` object has the following properties:

   a. `enableReload`: Optional. Used to reload the gadget at regular intervals.

   b. `onResizeReload`: Optional. Used to reload the gadget when the browser is resized.

   c. `onResizeAdjustHeight`: Optional and used along with the dynamic-height feature. This will adjust the gadget height when the browser is resized.

   d. `template`: Creates the actual view.

   e. `args`: An array of objects or function that returns an array of objects. It has two attributes: `Key` – used to access the data from within the template, and `ajaxOptions` – a set of request options used to connect to the server and retrieve data.

In our example, we will use the `template` and `args` properties to render the view. First, let us see `args` because we use the data retrieved here in the `template`. `args` will look like the following:

```
args: [{
```

```
    key: "user",
    ajaxOptions: function() {
      return {
        url: "/rest/gadget/1.0/currentUser"
      };
    }
  }]
```

As you can see, we invoked the `/rest/gadget/1.0/currentUser` method and used the key `user` to refer the data we retrieved while rendering the view. `ajaxOptions` uses the jQuery AJAX options, details of which can be found at `http://api.jquery.com/jQuery.ajax#options`.

The key `user` will now hold the user details from the REST method, as follows:

```
{
  "username":"jobinkk",
  "fullName":"Jobin Kuruvilla",
  "email":"author@gmail.com"
}
```

The `template` function will now use this `args` object (defined earlier) and its `key`, `user` to render the view as follows:

```
 template: function(args) {
   var gadget = this;
   var userDetails = AJS.$("<h1/>").text("Hello, "
   +args.user["fullName"]);
   gadget.getView().html(userDetails);
}
```

Here, `args.user["fullName"]` will retrieve the user's `fullName` from the REST output. Username or e-mail can be retrieved in a similar fashion.

`AJS.$` will construct the view as `<h1>Hello, Jobin Kuruvilla</h1>`, where `Jobin Kuruvilla` is the `fullName` retrieved.

The entire `Content` section will look as shown in the following lines of code:

```
<Content type="html" view="profile">
  <![CDATA[
    #requireResource("com.atlassian.jira.gadgets:common")
    #includeResources()
      <script type="text/javascript">
      (function () {
        var gadget = AJS.Gadget({
            baseUrl: "__ATLASSIAN_BASE_URL__",
```

```
            view: {
              template: function(args) {
                 var gadget = this;
                 var userDetails = AJS.$("<h1/>").text("Hello,
                  "+args.user["fullName"]);
                 gadget.getView().html(userDetails);
            },
            args: [{
              key: "user",
              ajaxOptions: function() {
                return {
                  url: "/rest/gadget/1.0/currentUser"
                };
               }
            }]
          }
        });
      })();
    </script>
  ]]>
</Content>
```

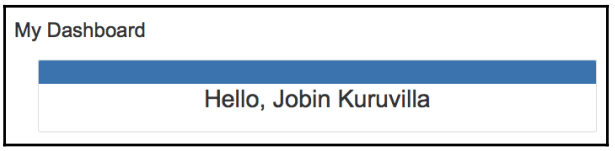
4. Package the gadget and deploy it.



Don't forget to add the `oauthpopup` required feature. More about authentication in gadgets can be found at `https://developer.atlassian.com/display/GADGETS/Using+Authentication+in+your+Gadget`.

How it works…

After the modification to the gadget XML, the gadget will now display the method as follows:



See also

- The *Writing JIRA 4 gadgets* recipe in this chapter

Configuring user preferences in gadgets

In the previous two recipes, we saw how to create gadgets from static content and dynamic content. In this recipe, we will go one step further and display the gadget content, based on user input.

The user will configure the gadget during its creation or modify it later and the gadget content will vary depending on the configuration parameters.

Getting ready…

Create the **Hello User Gadget**, populated with dynamic content, as described in the previous recipe.

How to do it…

In this recipe, we will let the user choose whether to display the name in the greeting message or not. There will be a property on the gadget named `displayName`. If it is set to `true`, the gadget will display the username and the greeting message will be **Hello, Jobin Kuruvilla**. If the `displayName` is set to false, the greeting message will be **Hello**!

The following are the steps to configure user preferences:

1. Include the `setprefs` and the `views` features under the `ModulePrefs` element:

```
<Require feature="setprefs" />
<Require feature="views" />
```

   `setprefs` is required to persist user preferences, whereas `views` determines whether the current user can edit the preferences or not.

2. Include the gadget, the `common` locale, under `ModulePrefs`, along with our custom `Locale` element:

```
#supportedLocales("gadget.common")
```

This is required to get the gadget configuration language properly.

3. Include the required `UserPref` elements. This element defines the various user preferences. The element supports the following fields:

    a. `name`: Required. Name of the user preferences. The value of this can then be accessed using `gadget.getPref("name")`.

    b. `display_name`: Display name of the field. By default, it will be the same as the name.

    c. `urlparam`: Optional string to pass as the parameter name for content `type="url"`.

    d. `datatype`: Data type of the field. Valid options include: `string`, `bool`, `enum`, `hidden`, or `list`. Default is string.

    e. `required`: Marks the field as required. Default is false.

    f. `default_value`: Sets a default value.

    In our example, we add the `displayName` property as follows:

```
<UserPref name="displayName" datatype="hidden" default_value="true"/>
```

    The field is marked as `hidden` so that it won't appear in the OpenSocial gadget configuration form!

4. Modify the creation of `AJS.Gadget` to include the `config` property. `config` is normally of the form:

```
...
  config: {
    descriptor: function(){
    ...
    },
    args: {
    Function, Array
    }
  },
...
```

Here, `descriptor` is a function that returns a new configuration descriptor. `args` is an array of objects or a function that returns one similar to `view`.

In our example, we define a function to return a descriptor with the configuration details of the `displayName` property. It looks like the following:

```
config: {
  descriptor: function (args) {
    var gadget = this;
    return {
      fields: [ {
        userpref: "displayName",
        label: gadget.getMsg("property.label"),
        description:gadget.getMsg("property.description"),
        type: "select",
        selected: gadget.getPref("displayName"),
        options:[ {
          label:"Yes",
          value:"true"
        },
        {
          label:"No",
          value:"false"
        }
      ]
    }]
   };
  }
}
```

Here, there is only one field: `displayName`. It is of the type `select` and has a `label` and `description`, both populated from the `i18n` property file using the `gadget.getMsg` method. The `Selected` attribute is populated with the current value – `gadget.getPref("displayName")`. `Options` are given as an array, as shown in the preceding snippet.

More details on the various other field types and their properties can be found at `https://developer.atlassian.com/display/GADGETS/Field+Definitions`.

5. Add the new `i18n` properties to the message bundle:

```
        <msg name="property.label">Display Name?</msg>
        <msg name="property.description">Example Property from J-
Tricks</msg>
```

6. Include the `UserPref – isConfigured`:

```
<UserPref name="isConfigured" datatype="hidden"
 default_value="false"/>
```

   The user preferences are set every time the gadget loads, and we use this property, which is specially designed to prevent this.

   When this property is used, `AJS.gadget.fields.nowConfigured()` should be added as an additional field under the `config descriptor`.

7. Modify the view to display usernames based on the configured property.

   The `template` function is modified as follows:

```
    if (gadget.getPref("displayName") == "true")
      var userDetails = AJS.$("<h1/>").text("Hello,
"+args.user["fullName"]);
    } else {
      var userDetails = AJS.$("<h1/>").text("Hello!");
    }
```

   As you can see, the configured property is retrieved using `gadget.getPref("displayName")`. If it is `true`, the username is used.

   The entire `Content` section now looks like the following lines of code:

```
<Content type="html" view="profile">
  <![CDATA[
    #requireResource("com.atlassian.jira.gadgets:common")
    #includeResources()
    <script type="text/javascript">
    (function () {
      var gadget = AJS.Gadget({
      baseUrl: "__ATLASSIAN_BASE_URL__",
      config: {
        descriptor: function (args) {
        var gadget = this;
        return {
        fields: [ {
          userpref: "displayName",
          label: gadget.getMsg("property.label"),
          description: gadget.getMsg("property.description"),
          type: "select",
          selected: gadget.getPref("displayName"),
          options:[ {
            label:"Yes",
```

```
              value:"true"
             },
             {
              label:"No",
              value:"false"
             }]
            },
            AJS.gadget.fields.nowConfigured()
          ]};
         }
        },
        view: {
           template: function(args) {
             var gadget = this;
             if (gadget.getPref("displayName") == "true")
             {
              var userDetails = AJS.$("<h1/>").text("Hello,
               "+args.user["fullName"]);
             } else {
                var userDetails = AJS.$("<h1/>").text("Hello!");
              }
              gadget.getView().html(userDetails);
             },
             args: [{
              key: "user",
              ajaxOptions: function() {
               return {
                 url: "/rest/gadget/1.0/currentUser"
               };
              }
             }]
            }
          });
         })();
      </script>
    ]]>
  </Content>
```

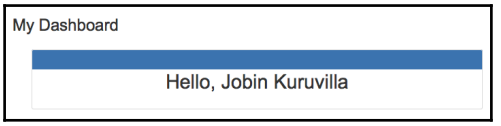
8. Package the gadget and deploy it.

How it works…

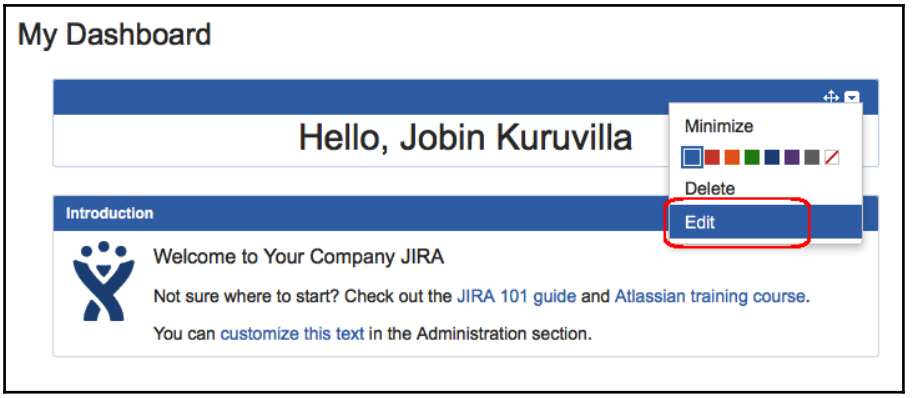
Once the user configurable properties are added, the gadget on its creation will ask the user to configure the `displayName` property, as shown next. The default value will be `true` (label :`Yes`) as we configured it:



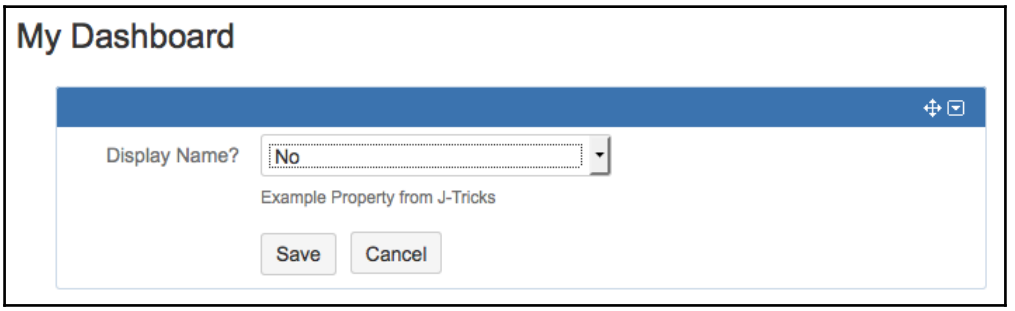
When **Yes** is selected, it appears exactly as it was in the previous recipe:



If you click on the gadget options now, you can see the **Edit** option, as shown in the following screenshot:



The following screenshot appears when you click on **Edit**:



On selecting **No**, the message is displayed without the username, as shown in the following screenshot:



There's more…

One of the most popular user preferences in JIRA gadgets, and therefore worth a special mention, is its ability to auto refresh itself at a configured interval. JIRA has a predefined feature that helps us to do it.

There are only a couple of things you need do to implement this feature:

1. Add the `refresh` UserPref:

```
<UserPref name="refresh" datatype="hidden" default_value="false"/>
```

2. Include the `enableReload: true` property in the `view`:

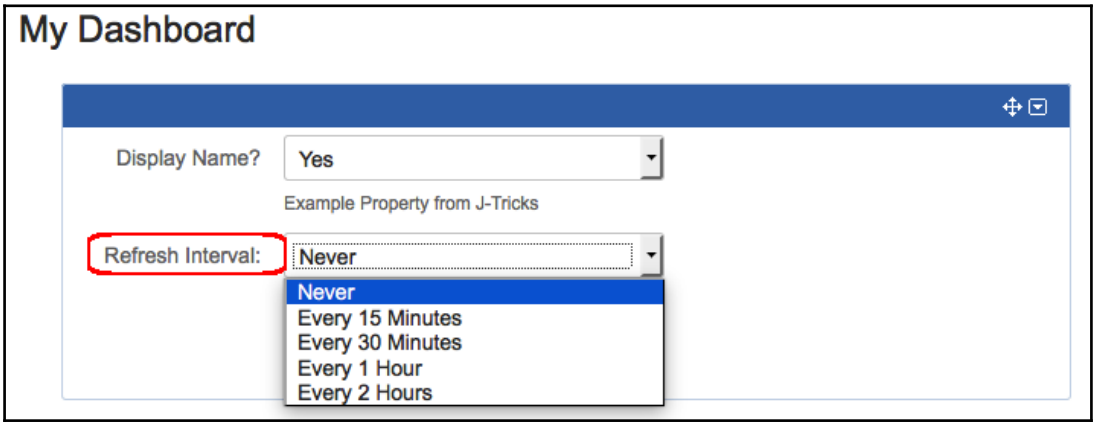
```
view: {
  enableReload: true,
  template: function(args) {
      ...
  },
  args: [{ ... }] }
```

You will now see an extra **Refresh** action on the gadget properties, as shown in the next screenshot:



This can be used to refresh the gadget at any time.

On clicking on **Edit**, the automatic refresh interval can be selected, as shown in the following screenshot:



See also

- The *Writing JIRA 4 gadgets* recipe of this chapter
- The *Invoking REST services from gadgets* recipe of this chapter

Accessing gadgets outside of JIRA

We have seen how to write a gadget and add it onto the JIRA dashboard. But have we made use of all the advantages of an OpenSocial gadget? How about adding them onto other OpenSocial containers such as Confluence?

In this recipe, we will see how to add a JIRA gadget into Confluence. The process is pretty much similar for other containers as well, except for the specific instructions of adding a gadget in the other container.

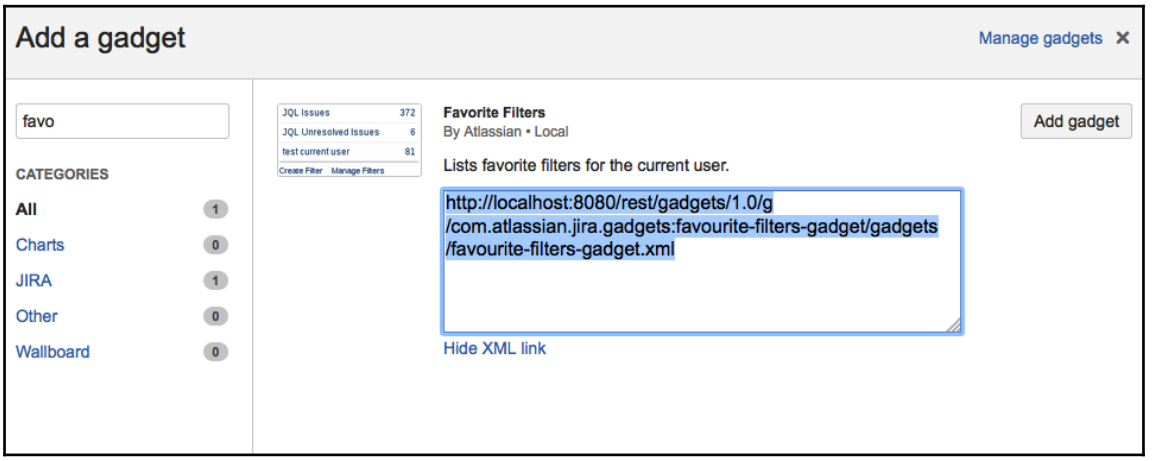
Getting ready…

Make sure JIRA is connected to Confluence using Application Links, as described at `https://confluence.atlassian.com/doc/linking-to-another-application-360677690.html`.

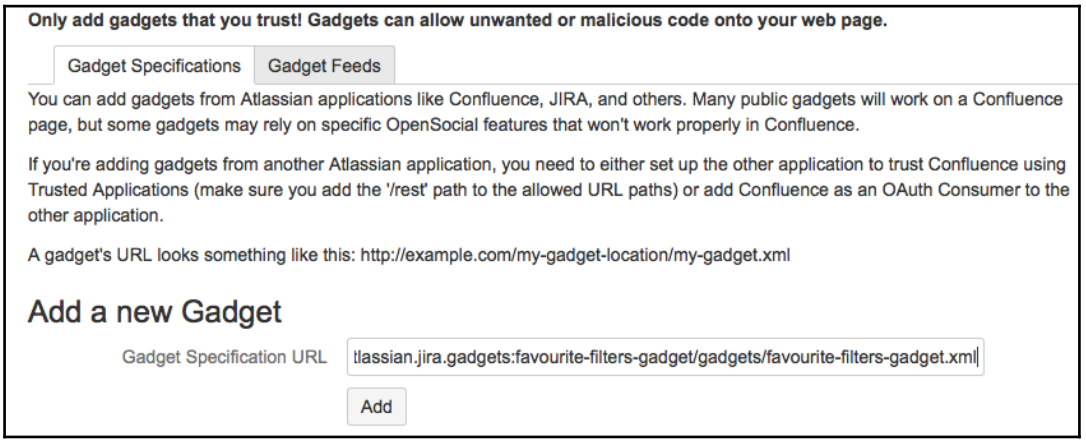
How to do it…

The following is a quick step-by-step procedure to add a gadget to Confluence:

1. Identify the gadget URL for the gadget that we are going to add. We can find this URL from the JIRA gadgets directory, as shown in the next screenshot. In the example, we choose to add the **Favorite Filters** gadget:



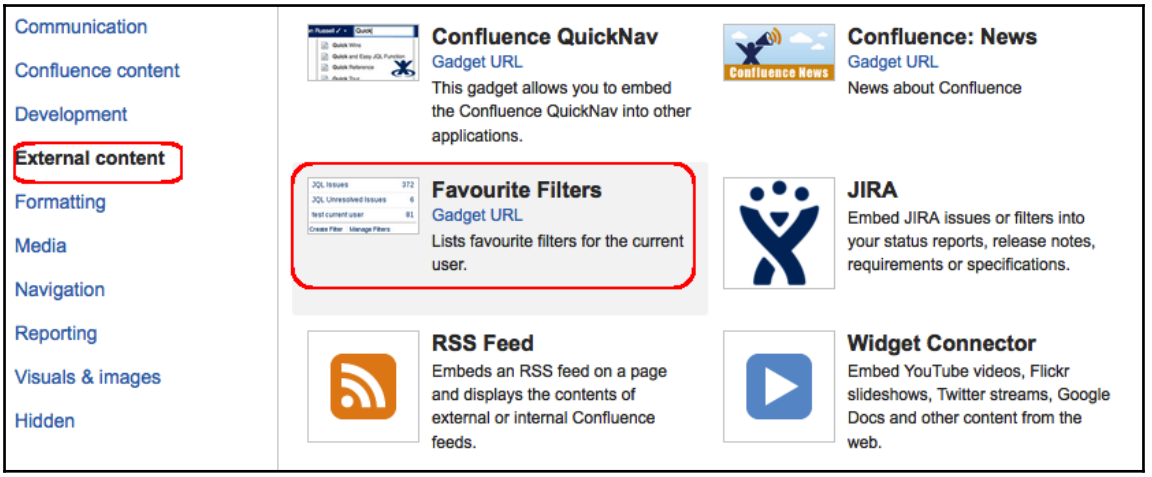
2. In Confluence, navigate to **Administration** | **General Configuration** | **External Gadgets**. Enter the URL, as shown in the next screenshot:



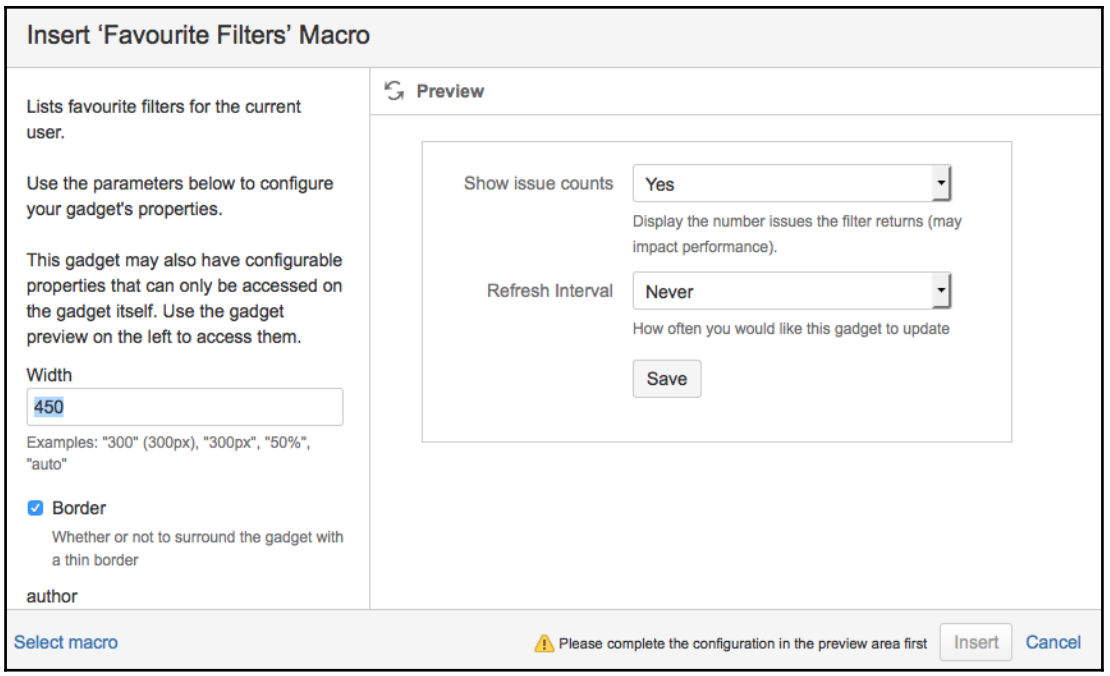


Note that this is the only process that will be different for different containers. We need to enter this URL in the appropriate place for each different container.

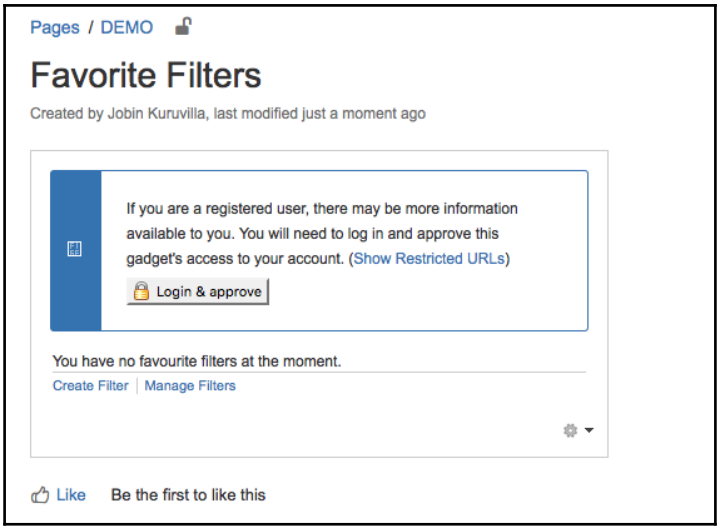
3. Once added, the gadget is available for use in any Confluence page you want. You can pick the gadget from the list of macros, as shown here:



4. When the gadget is added, you will have to configure the options, as shown here:

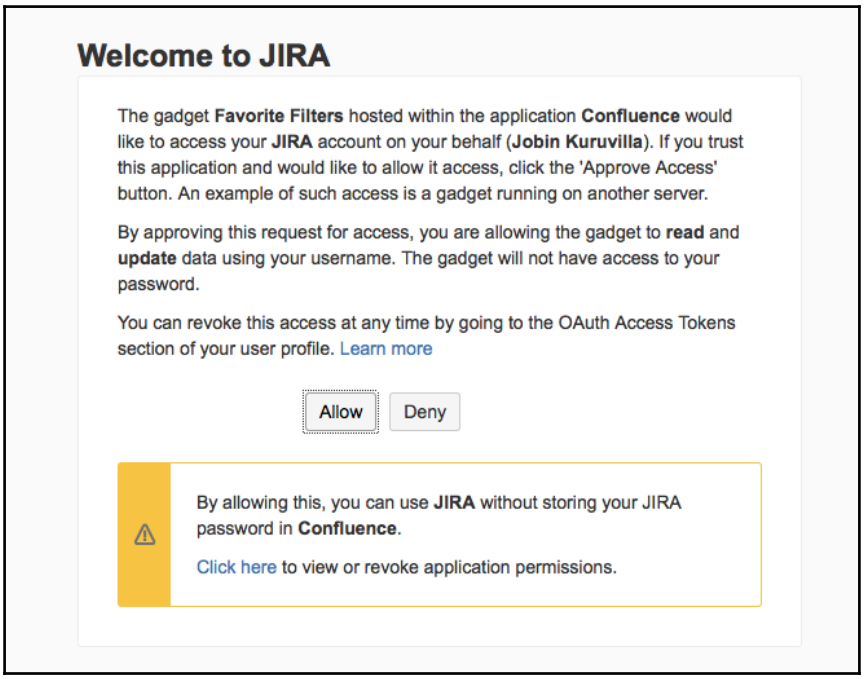


5. After inserting the gadget in the page, it renders the gadget, as shown here:



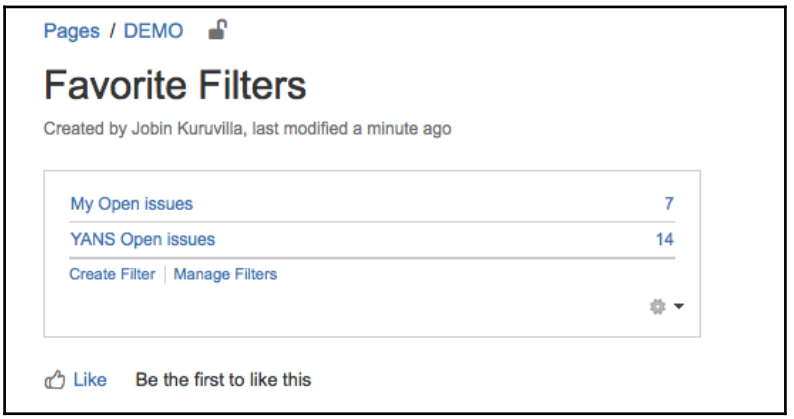
As you can see, Confluence prompts the user to log in to the JIRA application, to authorize the user to see JIRA data.

6. Click on **Login & approve** to navigate to the JIRA authorization screen, as shown in the following screenshot:



7. Once you click on Allow, the approval is done. If you were not logged in prior to JIRA, you will be asked enter the username/password of your JIRA instance.

8. After the approval, the gadget will start showing the actual data from JIRA, as shown here:



How it works…

The way it works is identical to that of its behavior in JIRA dashboards. The gadget will communicate with JIRA using the REST APIs and the data is rendered using the HTML and JavaScript code under the `view` section in the gadget XML's `Content` element.

You can add other JIRA gadgets also using the above technique and it unleashes the power of JIRA reporting inside confluence.

See also

- The *Writing JIRA gadgets* recipe in this chapter
- The *Invoking REST services from gadgets* recipe of this chapter

6
The Power of JIRA Searching

In this chapter, we will cover:

- Writing a JQL function
- Sanitizing JQL functions
- Adding a search request view
- Smart querying using quick search
- Searching in plugins
- Parsing a JQL query in plugins
- Linking directly to search queries
- Indexing and de-indexing issues programmatically
- Searching on issue entity properties
- Managing filters programmatically
- Subscribing to a filter

Introduction

JIRA is known for its search capabilities. It also allows us to extend these capabilities in a way that impresses its users! In this chapter, we will look at customizing the various searching aspects of JIRA, such as JQL, searching in plugins, managing filters, and so on.

JQL, **JIRA Query Language**, brings to the table advanced searching capabilities. The users can search for issues in their JIRA instance and then exploit all the capabilities of the issue navigator.

While JIRA has a **Simple or Basic Searching** that can be used for most searching requirements, **JQL**, or **Advanced Searching**, introduces support for logical operations, including AND, OR, NOT, NULL, and EMPTY. It also introduces a set of JQL functions, which can be used effectively to search based on predefined criteria.

JQL is a structured query language that lets us find issues using a simple SQL-like syntax. It is simple because of its auto-complete features, and it is powerful because of the custom JQL functions. As Atlassian puts it:

> *"JQL allows you to use standard Boolean operators and wild cards to perform complex searches, including fuzzy, proximity, and empty field searches. It even supports extensible functions, allowing you to define custom expressions like currentUser() or lastSprint() for dynamic searches."*

A query in *Advanced Search* consists of a **field**, followed by an **operator**, followed by a **value** or **function**. To find out all issues in a project, we can use:

```
project = "TEST"
```

Here, `project` is the field, = is the operator, and `TEST` is the value.

Similarly, we can find all issues assigned to the current user using the following:

```
assignee = currentUser()
```

Here, `assignee` is the field, = is the operator, and `currentUser()` is a JQL function.

At this point in time, JQL doesn't support the comparison of two fields or two functions in a single query. But we can use logical operators and keywords to introduce more control, as follows:

```
project = "TEST" AND assignee = currentUser()
```

This query will display issues that are in the project `TEST` and that have the current user as the `assignee`. A more detailed explanation on **Advanced searching**, along with the full reference to the **keywords**, **operators**, **fields**, and **functions** used can be found at `https://confluence.atlassian.com/jira/advanced-searching-179442050.html`.

Writing a JQL function

As we have seen, a **JQL function** allows us to define custom expressions or searchers. JIRA has a set of built-in JQL functions, the details of which can be found at `https://confluence.atlassian.com/jira/advanced-searching-functions-338363497.html#AdvancedSearchingFunctions-function`. In this recipe, we will look at writing a new JQL function.

JQL functions provide a way for values within a JQL query to be calculated at runtime. It takes optional arguments and produces results based on these arguments at runtime.

In our example, let us consider creating a function `projects()`, which can take a list of project keys and return all issues in the supplied projects. For example:

```
project in projects("TEST", "DEMO")
```

It will be equivalent to this:

```
project in ("TEST","DEMO")
```

It will also be equivalent to this:

```
project = "TEST" OR project = "DEMO"
```

We are introducing this new function just for the sake of this recipe. A simple function makes it easier to explain the concepts without worrying much about the logic of what it does!

Getting ready

Create a skeleton plugin using the Atlassian plugin SDK.

How to do it…

JIRA uses the **JQL Function Module** to add new JQL functions to the Advanced Search. The following is the step-by-step process for our example:

1. Modify the plugin descriptor to include the JQL function module:

```
<jql-function name="Projects Function"
 i18n-name-key="projects-function.name" key="projects-function"
 class="com.jtricks.jira.jql.ProjectsFunction">
  <description key="projects-function.description">
```

```
        The Projects Function Plugin
    </description>
    <!--The name of the function-->
    <fname>projects</fname>
    <!--Whether this function returns a list or a single value-->
    <list>true</list>
  </jql-function>
```

As with any other plugin modules, a JQL function module also has a unique**key**. The other major attribute of the function module is the function**class**. In this example, `ProjectsFunction` is the function class. The root element, `jql-function`, has two other elements: `fname` and `list`:

a. `fname` holds the JQL function name that is visible to the user. This will be used in the JQL query.

b. `list` indicates whether the function returns a list or not. In our example, we return a list of projects, and hence, we use the value `true` to indicate that it is a list. A list can be used, along with operators `IN` and `NOT IN`, whereas a scalar (single item) can be used with operators `=`, `!=`, `<`, `>`, `<=`, `>=`, `IS`, and `IS NOT`.

2. Implement the function class. The class name here is the name used in the module description, `ProjectsFunction` in this case. The class should extend the `AbstractJqlFunction` class. We now need to implement the major methods explained here:

a. `getDataType`: This method defines the return type of the function. In our example, we take a list of project keys and return valid projects, and hence we will implement the method to return the `PROJECT` datatype, as follows:

```
public JiraDataType getDataType() {
  return JiraDataTypes.PROJECT;
}
```

Check out the `JiraDataTypes` (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/JiraDataTypes.html`) class to see other supported data types.

b. `getMinimumNumberOfExpectedArguments`: This returns the smallest number of arguments that the function may accept. The auto-population of the method in the issue navigator takes this into consideration and puts sufficient double quotes within brackets when the function is selected.

For example, in our case, we need at least one project key in the function name, and hence we use `return 1`, as follows:

```
public int getMinimumNumberOfExpectedArguments() {
  return 1;
}
```

The prepopulated function will then look like `projects("")`.

c. `validate`: This method is used for validation of the arguments that we pass. In our example, we need to check if the method has at least one argument and also make sure that all the arguments passed are valid project keys. The `validate` method looks like the following:

```
  @Override
  public MessageSet validate(ApplicationUser user,
  FunctionOperand operand, TerminalClause terminalClause) {
   List<String> projectKeys = operand.getArgs();
    MessageSet messages = new MessageSetImpl();
    if (projectKeys.isEmpty())
    {
       messages.addErrorMessage
        ("Atleast one project key needed");
    } else {
    for (String projectKey : projectKeys) {
     if (projectManager.getProjectObjByKey
     (projectKey) == null) {
       messages.addErrorMessage
        ("Invalid Project Key:" + projectKey);
     }
    }
  }
 return messages;
}
```

Here we instantiate a new `MessageSet` and add error messages to it, if the validation fails. We must always return a `MessageSet`, even if it is empty. Returning `null` is not permitted. We can also add warning messages, which don't prevent the JQL execution, but warn the user about something.



The most important argument in the `validate` method is `FunctionOperand`, as it holds the arguments of the function which can be retrieved as `operand.getArgs()`. The other argument, `terminalClause`, is JIRA's representation of the JQL condition we are validating for.We can extract the name, operator, and function from the argument using `terminalClause.getName`, `terminalClause.getOperator`, and `terminalClause.getOperand`, respectively.

The `AbstractJqlFunction` has a validation method in it to check the number of arguments. So if we know the expected number of arguments (which is not the case in our example, as we can have any number of projects passed in this example), we can validate it using:

```
MessageSet messages = validateNumberOfArgs(operand, 1);
```

This code adds an error if the number of arguments is not `1`.

d. `getValues`: This is the method that takes the arguments and returns the date type as a list or scalar depending on the function. In our example, the `getValues` method returns a list of literals that has the project ID.

The method is implemented as follows in our example:

```
@Override
public List<QueryLiteral> getValues(QueryCreationContext
context, FunctionOperand operand, TerminalClause
terminalClause) {
   List<QueryLiteral> literals =
      new LinkedList<QueryLiteral>();
   List<String> projectKeys = operand.getArgs();
   for (String projectKey : projectKeys) {
      Project project =
      projectManager.getProjectObjByKey(projectKey);
        if (project != null) {
           literals.add(new QueryLiteral
           (operand, project.getId()));
        }
   }
   return literals;
}
```

The arguments `operand` and `terminalClause` are the same as what we have seen in the validate method. The `QueryCreationContext` argument holds the context in which the query is executed. `QueryCreationContext.getUser()` will retrieve the user who executed the query, and the `QueryCreationContext.isSecurityOverriden()` method indicates whether or not this function should actually perform security checks.

The function should always return a list of `QueryLiteral` objects. Even when the function returns a scalar instead of a list, it should return a list of `QueryLiteral`, which can be created as follows:

```
Collections.singletonList(new QueryLiteral(operand, some_value))
```

A `QueryLiteral` represents either a `String`, `Long`, or `EMPTY` value. These three represent JQL's distinguishable types. Construct it with no value and it will represent EMPTY, construct it with a String and it represents a String, construct it with a Long and it represents a Long.

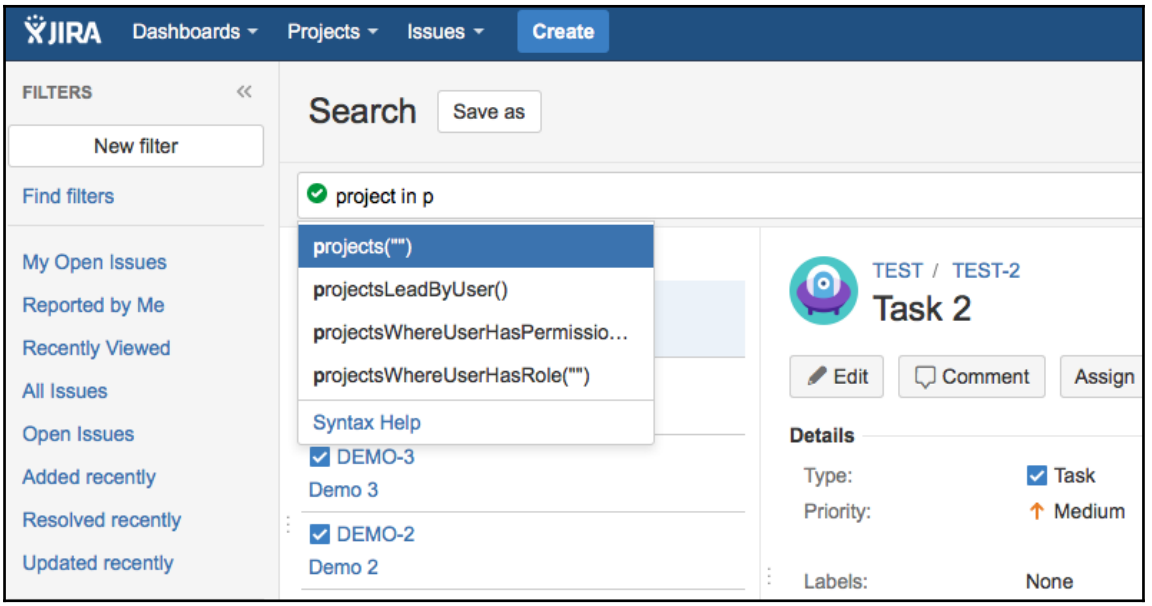
In our example, we use the project ID (LONG) that is unique across projects. For projects, we can even use the key (STRING) or name (STRING), as they are also unique. However, it may not work with fields such as *Fix For Version*, as you might find two *Fix For Versions* with the same name. It is recommended to return the ID, wherever possible, to avoid such unambiguous search results.

To summarize, we find out the project objects using the project keys supplied by the user and return a list of `QueryLiterals`, created using the project IDs.

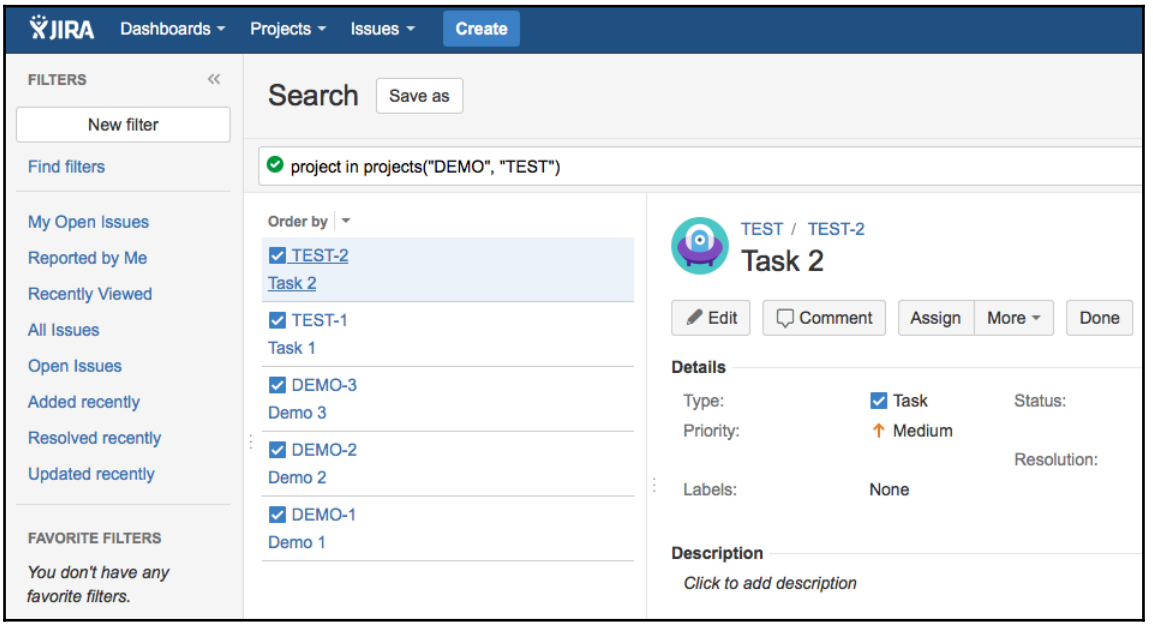
3. Package the plugin and deploy it.

How it works…

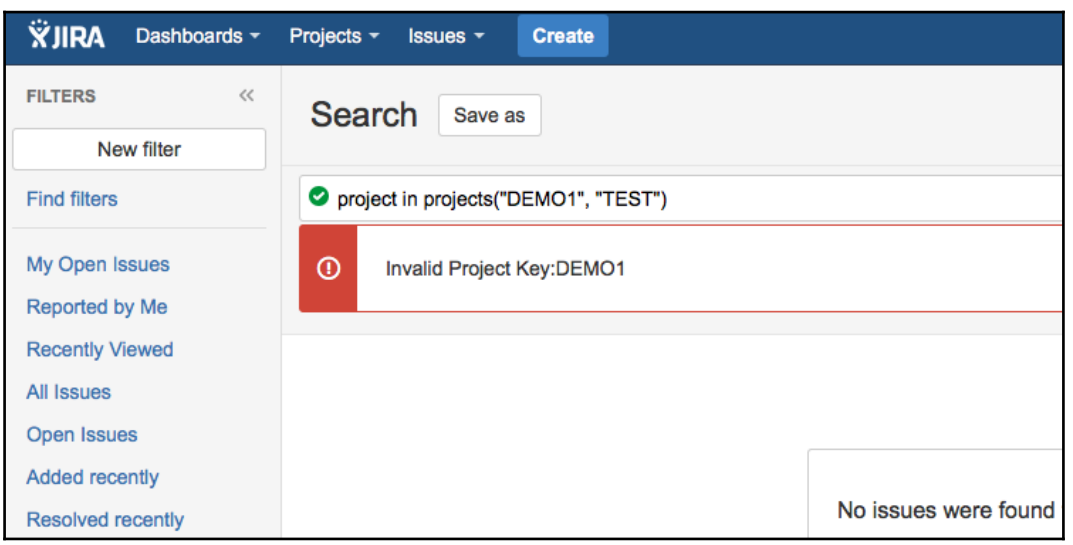
Once the plugin is deployed, we can go to the **Issue Navigator** and open the **Advanced** search to start using our brand new function! When you start typing **project inp**, JIRA auto-populates the available options, including our new function, as follows:



Once the function with appropriate arguments is added, the search is executed and results are shown, as follows:



When an invalid project key is given as the argument, our `validate` method populates the error message, as shown in the following screenshot:



See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Sanitizing JQL functions

If you don't want your JQL function to violate the strict security aspects of your JIRA instance, sanitizing the JQL functions is a must! So, what does this actually mean?

Imagine a filter created by you to find out issues in a pre-defined set of projects. What will happen if you share the filter with a friend of yours who is not supposed to see the project or know that the project existed? The person with whom you shared it won't be able to modify the issues in the protected project due to JIRA's permission schemes, but they will surely see the name of the project in the JQL query that is used in the filter.

This is where sanitizing the JQL function will help. In essence, we just modify the JQL query to protect the arguments in line with the permission schemes. Let us see an example of doing that by sanitizing the JQL function we created in the previous recipe.

Getting ready

Develop the JQL function, as explained in the Writing a JQL function recipe.

How to do it…

In our JQL function, we use the project keys as the arguments. To explain the function sanitization, we will look to replace the keys with project IDs whenever the user doesn't have permission to browse a project. The following is the step-by-step process showing you how to do it:

1. Modify the JQL function class to implement the `ClauseSanitisingJqlFunction` interface:

```
public class ProjectsFunction extends AbstractJqlFunction
implements ClauseSanitisingJqlFunction{
```

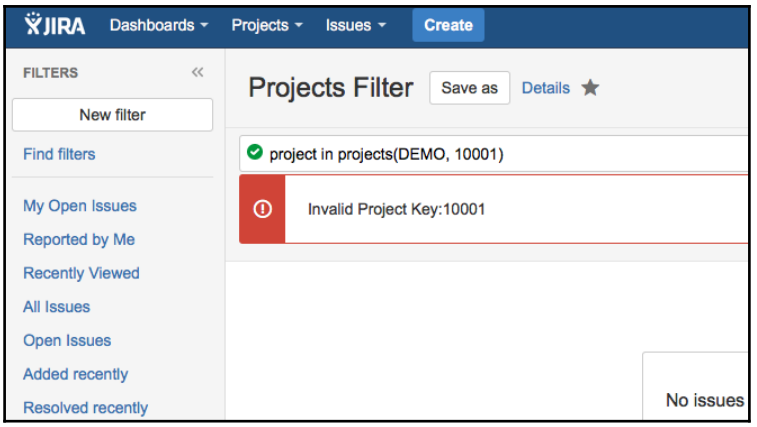
2. Implement the `sanitiseOperand` method. In this method, we read all the existing arguments of the JQL function, from the `FunctionOperand` argument, and modify them to include project IDs instead of keys, wherever the user doesn't have **Browse** permissions:

```
@Override
public FunctionOperand sanitiseOperand(ApplicationUser user,
FunctionOperand functionOperand) {
  final List<String> pKeys = functionOperand.getArgs();
  boolean argChanged = false;final List<String>
  newArgs = new ArrayList<String>(pKeys.size());
  for (final String pKey : pKeys) {
      Project project = projectManager.getProjectObjByKey(pKey);
      if (project != null && !permissionManager.hasPermission
      (ProjectPermissions.BROWSE_PROJECTS, project, user)) {
          newArgs.add(project.getId().toString());
          argChanged = true;
      }
      else {
             newArgs.add(pKey);
      }
  }
  if (argChanged) {
     return new FunctionOperand(functionOperand.getName(),
     newArgs);
  } else {
      return functionOperand;
  }
}
```

3. Package and deploy the modified plugin.

How it works…

Once the plugin is deployed, if a user doesn't have the permission to browse a project, they will see the project ID instead of the key that was originally entered when the filter was created. The following is a sample screenshot of how the query will look in that case. In this case, I just removed myself from the **Browse** permission of the `TEST` project, and you can see that the query is modified to replace the key `TEST` with its unique ID, which doesn't reveal much information:



This is only an example, and we can sanitize even more complex queries to hide sensitive data from users without the required privileges.

See also

- *The Writing a JQL function recipe in this chapter*

Adding a search request view

One of the customizable features in JIRA is its **Issue Navigator**. It lets us search based on numerous criteria, and choose the fields that need to be shown, in a way we want to see them!

The normal or the default view in the issue navigator is the tabular view to display the issues and the fields we have chosen by configuring the issue navigator. JIRA also gives us a few other options to see the search results in different formats—to export them into Excel, Word, or XML—with the help of the predefined search request views.

In this recipe, we will see how we can add a simple HTML view in to JIRA's issue navigator. Such search views enable us to see the search results in any format we like, and the HTML view we are creating is just an example. To achieve this, we need to use the **Search Request View Plugin** module.

Getting ready

Create a plugin skeleton using Atlassian Plugin SDK.

How to do it…

As mentioned before, we use the Search Request View Plugin module to create custom search views. In our example, let us create a simple HTML view that just displays the issue key and summary.

The following is the step-by-step process:

1. Define the plugin descriptor with the search request view module:

```
<search-request-view key="simple-searchrequest-html"
name="Simple HTML View"
class="com.jtricks.jira.search.SimpleSearchRequestHTMLView"
state='enabled' fileExtension="html"
contentType="text/html">
  <resource type="velocity" name="header"
  location="templates/searchrequest-html-header.vm"/>

  <resource type="velocity" name="body"
  location="templates/searchrequest-html-body.vm"/>

  <resource type="velocity" name="footer"
  location="templates/searchrequest-html-footer.vm"/>

  <order>200</order>
</search-request-view>
```

   As usual, the module has a unique key. The following are its other attributes:

   a. `name`: The name that will appear in the Issue Navigator for the View.

   b. `class`: The search request view class. This is where we populate the velocity contexts with the necessary information.

   c. `contentType`: The `contentType` of the file that is generated. `text/html`, `text/xml`, `application/rss+xml`, `application/vnd.ms-word`, `application/vnd.ms-excel`, and so on.

d. `fileExtension`: The following are the extensions of the file generated: `.html`, `.xml`, `.doc`, `.xls`, and so on.

e. `state`: Enabled or disabled. This field determines whether the module is enabled at startup or not.

The `search-request-view` element also has a few child elements to define the velocity templates required for the various views and to determine the `order` in which the views will appear. Modules with lower `order` values are shown first. JIRA uses an order of 10 for the built-in views. A lower value will put the new view above the built-in views and a higher value will put the new view at the bottom:

2. Implement the Search Request View class. The Search Request View class must implement the `SearchRequestView` interface. To make things easier, we can extend the `AbstractSearchRequestView` class that already implements this interface. When we do that, we have just one method, `writeSearchResults`, which needs to be implemented. This method takes a writer argument, which we can use to generate the output using the various template views we define. For example:

```
writer.write(descriptor.getHtml("header", headerParams));
```

It will identify the velocity template with the view named `header` and will use the variables on the map-`headerParams`-to render the template. We can similarly define as many templates as we want and write them to create the view that we need.

In our example, we have three views defined – **header**, **body**, and **footer**. These views can be named in any way we want, but the same names that we define in the `atlassian-plugin.xml` should be used in the Search Request View class.In our class implementation, we use the three views to generate the simple HTML view. We use the **header** and **footer** views in the beginning and the end and will use the **body** view to generate the issue view for each individual issue in search results. The following is how we do it:

a. Generate a map with the parameters required to populate the header. In our example, let us populate the map with a filter name and the logged-in user name:

```
final Map<String, String> headerParams = new
HashMap<String, String>();
```

```
headerParams.put("filtername", searchRequest.getName());
ApplicationUser user = authenticationContext.getLoggedInUser();
headerParams.put("user", user.getDisplayName());
```

b. Render the **header** view using the preceding `headerParams` map, and write it using the `writer` object, as mentioned earlier.

```
writer.write(descriptor.getHtml("header", headerParams));
```

c. Retrieve the JQL `Query` from the `SearchRequest` and use it to retrieve the issues matching query, as shown here:

```
SearchResults results = this.searchService.search(user,
searchRequest.getQuery(), PagerFilter.getUnlimitedFilter());
List<Issue> issues = results.getIssues();
```

More about `SearchService` will be covered in the following recipes.

d. Iterate over the issues and render the **body** view as shown here:

```
for (Issue issue : issues) {
   writer.write(descriptor.getHtml("body",
   MapBuilder.build("issue", issue)));
}
```

Here, the map is populated only with the issue object.

e. Populate the **footer** view using the user name:

```
writer.write(descriptor.getHtml("footer",
MapBuilder.build("user", user.getDisplayName())));
```

The method will now look as follows:

```
@Override
public void writeSearchResults(final SearchRequest
searchRequest, final SearchRequestParams searchRequestParams,
final Writer writer) throws SearchException {
  try {

       // Write the header using filter name
       and user name final Map<String, String>
       headerParams = new HashMap<String, String>();

      headerParams.put("filtername", searchRequest.getName());

      ApplicationUser user =
      authenticationContext.getLoggedInUser();
```

```
            headerParams.put("user", user.getDisplayName());

            // Write the header using headerParams
            writer.write(descriptor.getHtml("header", headerParams));

            // Write the body using issue details
            SearchResults results = this.searchService.search(user,
            searchRequest.getQuery(),
            PagerFilter.getUnlimitedFilter());

            List<Issue> issues = results.getIssues();
            for (Issue issue : issues) {
                  writer.write(descriptor.getHtml("body",
                  MapBuilder.build("issue", issue)));
            }

            // Finally, lets write the footer using the user
            writer.write(descriptor.getHtml("footer",
            MapBuilder.build("user", user.getDisplayName())));
      }
      catch (IOException e) {
           throw new RuntimeException(e);
      }
}
```

3. Write the velocity templates. As we saw, we are using three views:

   a. Header – The velocity template is `templates/searchrequest-html-header.vm`. The following is how it looks:

```
Hello $user , Have a look at the search results!<br><br>
#set($displayName = 'Anonymous')
#if($filtername)
   #set($displayName = $textutils.htmlEncode($filtername))
#end <b>Filter</b> : $displayName <br><br>
<table>
```

   We just greet the user and display the filter name here. It also has a `<table>` tag, which is used at the beginning of the issue table. The table will be closed in the footer.

   b. Body – The velocity template is `templates/searchrequest-html-body.vm`. The following is how it looks:

```
<tr>
 <td><font color="green">$!issue.key</font></td>
 <td>$!issue.summary</td>
</tr>
```

Whatever appears here is common to all the issues. Here we create a table row for each issue and display the key and summary appropriately.

c. Footer – The velocity template is `templates/searchrequest-html-footer.vm`. The following code shows how it looks:

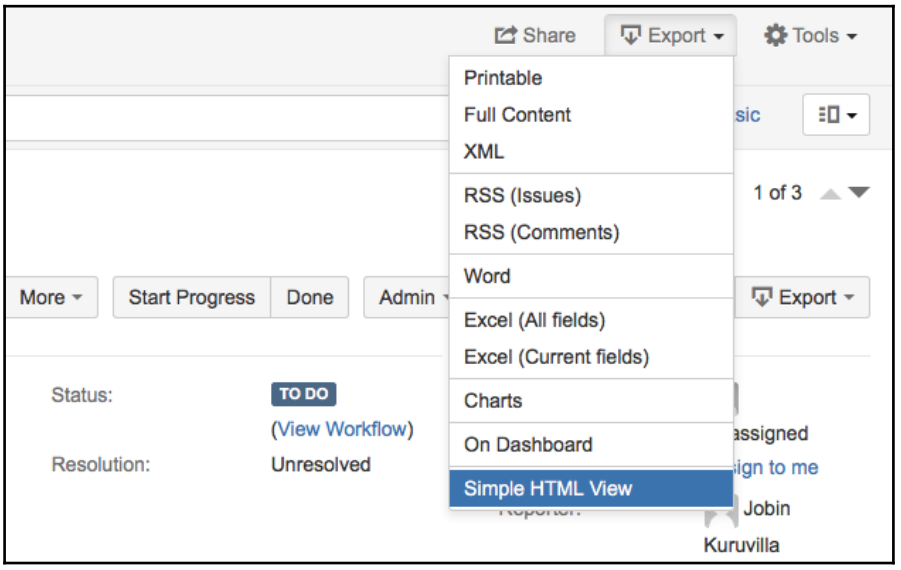
```
</table> <br><br>...And that's all we have got now , $user!
```

We just close the table and wind up with a message.

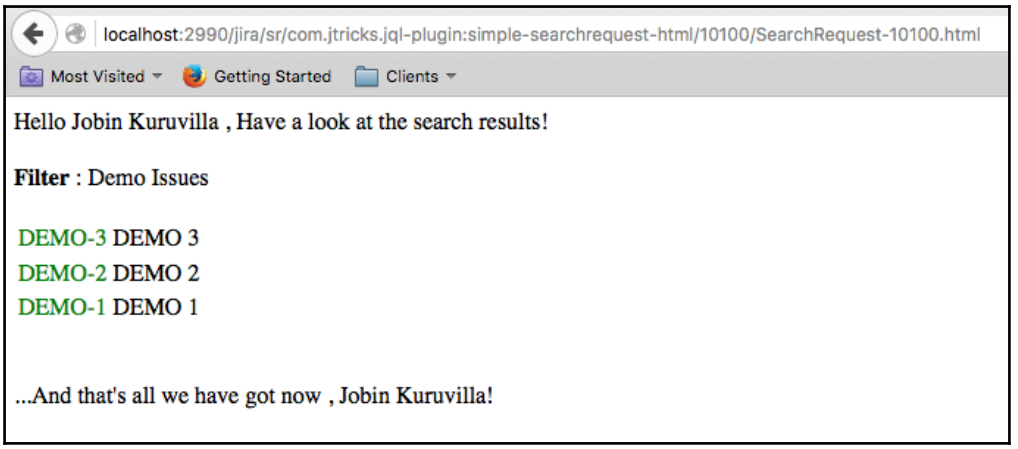
4. Package the plugin and deploy it.

How it works…

Once the plugin is deployed, we will find a new view in the issue navigator named **Simple HTML View**:



On selecting the view, the current search results will appear as in the following screenshot.



As you can see, our simple view is rendered using the velocity templates, at run time, using the user name, filter name, and the issue list. It is now left to our creativity to make it more beautiful, or use an entirely different content type instead of HTML. An example of how an XML view is generated can be found in the JIRA documentation at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/search-request-view-plugin-module`.

There's more…

What if you want to render a separate header, body, and footer for each issue? And, what if you don't want to go through the trouble of searching the issues and iterating over the results?

That is where the **single issue views** will help.

Using Single Issue Views to render search views

The following are the steps to implement a search request view using single issue views:

1. Define a `single-issue-view`, as shown here:

```
<single-issue-view key="issue-html" name="HTML"
class="com.jtricks.jira.search.IssueHTMLView"
state='enabled' fileExtension="html" contentType="text/html">

  <resource type="velocity" name="body"
```

```
    location="templates/single-html-view.vm" />
    <resource type="velocity" name="header"
    location="templates/single-html-header.vm" />

    <resource type="velocity" name="footer"
    location="templates/single-html-footer.vm" />

    <order>100</order>
  </single-issue-view>
```

2. Implement the `IssueHTMLView` class, defined in the class attribute. This class will extend the `AbstractIssueView` class and will have to implement the `getBody()` and `getContent()` methods, as shown here:

```
public class IssueHTMLView extends AbstractIssueView {
  @Override
  public String getBody(Issue issue,
  IssueViewRequestParams params) {

    return descriptor.getHtml("body",
    MapBuilder.build("issue", issue));
  }
  @Override
  public String getContent(Issue issue,
  IssueViewRequestParams params) {
    String header = getHeader(issue);
    String body = getBody(issue, params);
    String footer = getFooter(issue);
    return header + body + footer;
  }

  public String getHeader(Issue issue) {
    return descriptor.getHtml("header",
    MapBuilder.build("issue", issue));
  }
  public String getFooter(Issue issue) {
    return descriptor.getHtml("footer",
    MapBuilder.build("issue", issue));
  }
}
```

As you can see, we are using the appropriate views to render the **header**, **footer** and **body** content and then combine them to return the issue view in the `getContent()` **method.**

3. Implement the velocity templates for the issue view.

   a. `single-html-view.vm`: Renders the **body** of the issue using the summary and reporter name:

```
<tr>
  <td>$!issue.summary</td>
  <td>, Reported by <b>$!issue.reporter.displayName</b></td>
</tr>
```

   b. `single-html-header.vm`: Renders the header of the issue using its summary:

```
<tr>
  <td colspan="2">
   <font color="green"><u>$!issue.key</u></font>
  </td>
</tr>
```

   c. `single-html-footer.vm`: Renders the footer of the issue using the issue assignee:

```
<tr>
   <td colspan="2">
    <i>
      #if($!issue.assignee)
        Currently assigned to
        <b>$!issue.assignee.displayName</b>
      #else Currently unassigned
      #end
    </i>
   </td>
</tr>
```

4. Modify the search request view definition to have only its own **header** and **footer** views. The **body** of the search request view will be rendered using the individual issue views:

```
<search-request-view key="searchrequest-html"
name="HTML View"
class="com.jtricks.jira.search.SearchRequestHTMLView"
state='enabled' fileExtension="html"
contentType="text/html">

   <resource type="velocity" name="header"
   location="templates/searchrequest-html-header.vm"/>
```

```
    <resource type="velocity" name="footer"
    location="templates/searchrequest-html-footer.vm"/>

    <order>200</order>
</search-request-view>
```

5. Modify the search request view class, `SearchRequestHTMLView` in the example, to use `SingleIssueWriter` and `SearchRequestViewBodyWriterUtil` classes, as shown here:

```
// Write the body using issue details
final SingleIssueWriter singleIssueWriter =
new SingleIssueWriter() {

   public void writeIssue(final Issue issue,
   final AbstractIssueView issueView, final Writer writer)
   throws IOException {

      writer.write(issueView.getContent(issue,
      searchRequestParams));
   }
};
final IssueHTMLView htmlView =
SearchRequestViewUtils.getIssueView(IssueHTMLView.class);
searchRequestViewBodyWriterUtil.writeBody(writer, htmlView,
searchRequest, singleIssueWriter,
searchRequestParams.getPagerFilter());
```

Here, `SingleIssueWriter` writes the view using the issue view defined in Step 2. As you can see, it invokes the `getContent()` method in the issue view. If you do not need a separate header and footer for each issue, you may very well use the `getBody()` method, instead of `getContent()`.
The updated class will look like the following:

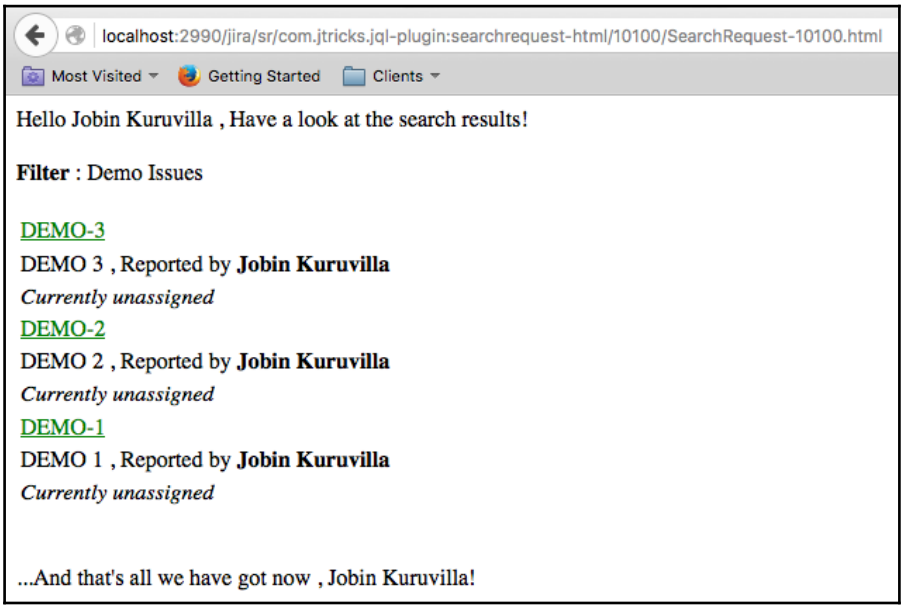
```
@Scanned
public class SearchRequestHTMLView extends
AbstractSearchRequestView {
   private final JiraAuthenticationContext
   authenticationContext;

   private final SearchRequestViewBodyWriterUtil
   searchRequestViewBodyWriterUtil;

   public SearchRequestHTMLView(@ComponentImport
   JiraAuthenticationContext authenticationContext,
   @ComponentImport SearchRequestViewBodyWriterUtil
   searchRequestViewBodyWriterUtil) {
```

```
    this.authenticationContext = authenticationContext;

    this.searchRequestViewBodyWriterUtil =
    searchRequestViewBodyWriterUtil;
}
@Override
public void writeSearchResults(final SearchRequest
searchRequest, final SearchRequestParams
searchRequestParams, final Writer writer)
throws SearchException {
    try {
        // Write the header using filter name and user name
        final Map<String, String> headerParams =
       new HashMap<String, String>();

       headerParams.put("filtername", searchRequest.getName());

       ApplicationUser user =
       AuthenticationContext.getLoggedInUser();

       headerParams.put("user", user.getDisplayName());

       // Write the header using headerParams
       writer.write(descriptor.getHtml
       ("header", headerParams));

       // Write the body using issue details
       final SingleIssueWriter singleIssueWriter = new
       SingleIssueWriter() {
          public void writeIssue(final Issue issue,
          final AbstractIssueView issueView,
          final Writer writer) throws IOException {
             writer.write(issueView.getContent(issue,
             searchRequestParams));
          }
       };
       final IssueHTMLView htmlView =
       SearchRequestViewUtils.getIssueView
       (IssueHTMLView.class);

       searchRequestViewBodyWriterUtil.writeBody(writer,
       htmlView, searchRequest, singleIssueWriter,
       searchRequestParams.getPagerFilter());

       // Finally, lets write the footer using the user
       writer.write(descriptor.getHtml("footer",
       MapBuilder.build("user", user.getDisplayName())));
```

```
            }
            catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

6. Package the plugin and deploy it.

Once the updated search view is used, you can see that the individual issues are rendered using the new issue view, as shown here:



See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

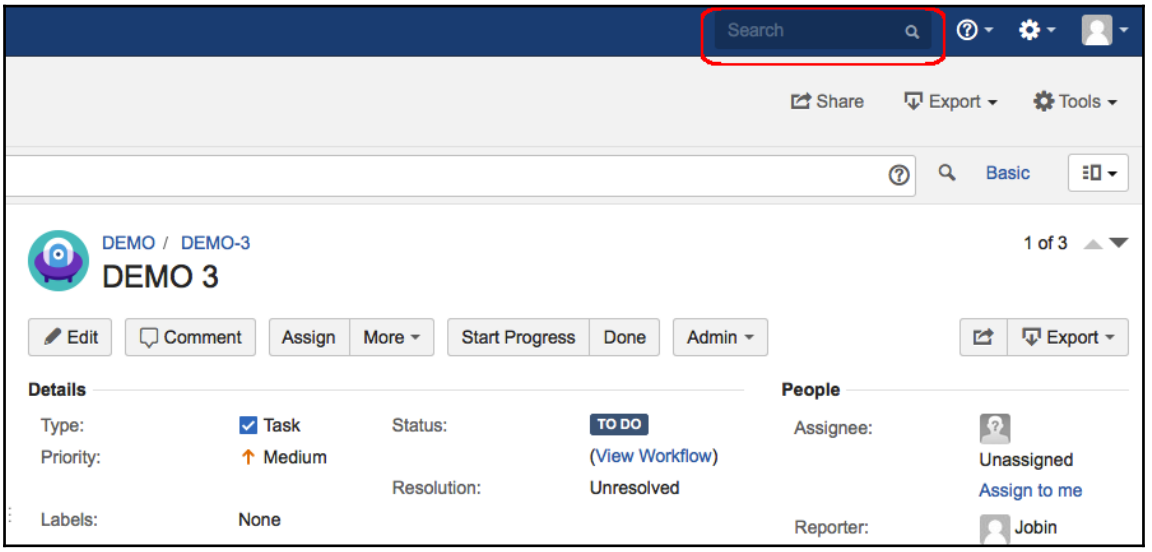
Smart querying using quick search

The name says it all! JIRA allows smart querying using its **Quick Search** functionality and it enables users to find critical information with ease. There is a pre-defined set of search keywords that JIRA recognizes and we can use them to search smart and fast!

In this recipe, we will look at how we can do smart querying on some of the JIRA fields.

How to do it…

Before we start, the **Quick Search** box is located at the right-hand top corner of JIRA, as shown:



The following is how we can search on some of the fields as of JIRA 7. Don't forget to check how many of them are supported in your version of JIRA!

- **Issue key**: If you already know the issue key that you want to see, it doesn't get any better! Just type the issue key in the **Quick Search** box and JIRA will take you to the view issue page.
  There's more! If you are browsing a project or viewing an issue, and you want to see another issue for which the key is known, all you need to type is the number in the unique key (just the numerical part). There isn't even a need to type in the full key.

For example, **TEST-123** will take you to that issue directly. Typing **125** will then take you to **TEST-125**!

- **Project**: If you type in the project key, the quick search will show you all the issues in that particular project. The project name can also be used, as long as there are no spaces in it.
  For example, `TEST` will return all issues in the project `TEST` or a project with the key `TEST`. "TEST Project" will not display issues in the project with the name "Test Project", as the quick search interprets it as two different keywords.

- **Assignee**: The keyword `my` can be used to find all issues assigned to me.
- **Reporter**: The keyword `r:` is followed by `me`, or the *reporter name* can find all issues reported by me or the specified user. `r:none` is also supported and it returns issues without any reporter.
  For example, `r:me` will retrieve all issues reported by me, whereas `r:admin` will retrieve all issues reported by the user – admin.
- **Date fields**: Quick Search can be done based on the three major date fields on the issue – **created**, **updated**, and **duedate**. The keywords used are `created`, `updated`, and `due`, respectively. The keyword should be followed by **:** and the date range, without any spaces.
  The date range can use one of the following keywords – `today`, `tomorrow`, `yesterday`, or a single date range (for example, "-5d") or two date ranges (for example, "-2w,1w"). The date ranges cannot have spaces in them. Valid date/time abbreviations are: "w" (week), "d" (day), "h" (hour), and "m" (minute).
  For example:`created:today` will retrieve all issues created on the date.`updated:-5d` will retrieve all issues updated in the last five days.`due:-2w,1w` will retrieve all issues due in the last two weeks and in the next week.
  You can also use the keyword `overdue` to retrieve all issues that are overdue (have a past due date):
- **Priority**: The quick search can be done using the priority values **blocker**, **critical**, **major**, **minor**, and **trivial**. Just typing the value will retrieve all the issues that have the given priority value
  .For example, all issues with the priority `major` can be retrieved by searching with `major`.
- **IssueType**: Issue type names can be used in the Quick search as long as it doesn't have any spaces in it. Even plurals will work.
  For example, typing `bug` or `bugs` will retrieve all the issues with the issue type of bug.

- **Resolution**: You can use the resolution values in quick search to find issues having those values as resolution.
  For example, `fixed`, `duplicate`, and so on.
- **Versions**: Quick Search can find issues with known Affected Versions or Fix for versions using the keywords `v:` or `ff:` followed by the value, without any space. There shouldn't be any spaces between `v:` and the version name. It can also use wild card search. The search will also find all issues with version values that contain the string you specify, followed immediately by a space.
  For example:
  `v:2.0` will find issues in versions 2.0, 2.0 one, 2.0 beta, and so on, but it wouldn't find issues in version 2.0.1.
  `v:2.*` will find issues in versions 2.0, 2.0 one, 2.0.1, 2.2, and so on.The same applies to fixes for versions. The prefix only changes to ff:

- **Components**: Quick search can find issues with component names using the prefix `c:` followed by the component name. It will retrieve all issues where the component has the value somewhere in its name, not necessarily starting with it. For example, `c:jql` will find all issues in components that have the word `jql` in it. It will work for the components jql, jql performance, advanced jql, and so on.

There's more…

Quick Search can also be used to search for any word within the issue(s) you are looking for, provided the word is in the summary, description, or comments of the issue. It is called **Smart Search**.

If you think you want to use any of these keywords without using Smart search, the query can be run without smart search when the results are displayed.

Smart Querying can have multiple keywords combined to narrow down the search. It can even be combined with Free Text Search.

For example, **my open bugs** will retrieve all bugs that are opened and assigned to me. It is equivalent to the JQL:

```
issuetype = Bug AND assignee = currentUser() AND status = Open
```

my open bugs jql will retrieve all bugs that are opened and assigned to me and have the word “jql” in their summary, description, or comments. It is equivalent to:

```
 (summary ~ jql OR description ~ jql OR comment ~ jql) AND issuetype = Bug
AND assignee = currentUser() AND status = Open
```

my open bugs jql performance is equivalent to:

```
(summary ~ "jql performance" OR description ~ "jql performance" OR comment
~ "jql performance") AND issuetype = Bug AND assignee = currentUser() AND
status = Open
```

More on advanced searching or JQL can be found at
`https://confluence.atlassian.com/jira/advanced-searching-179442050.html`.

Searching in plugins

With the invention of JQL, JIRA Search APIs have changed drastically from 3.x versions. Searching in plugins is now done using APIs supporting JQL. In this recipe, we will see how to search for issues within our plugins using those APIs.

How to do it…

For the sake of concentrating on the search APIs, we will look at writing a simple method, `getIssues()`, that returns a list of issue objects based on some search criteria.

The essence of searching is to build a `Query` object using `JqlQueryBuilder`. A `Query` object will have a `where` clause and an `order by` clause, which are built using the `JqlClauseBuilder`. We can also incorporate conditions in between clauses, using `ConditionBuilders`.

For now, let us assume we want to find all the issues in a particular project (project ID: 10000, Key: DEMO) and assigned to the current user within our plugin. The JQL equivalent for this is:

project = “DEMO” and assignee = currentUser()

The following are the steps to do this programmatically:

1. Create a `JqlQueryBuilder` object.
   `JqlQueryBuilder` is used to build the query that is used to perform issue searching. The following is how a `JqlQueryObject` is created:

   ```
   JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
   ```

2. Create a `where` clause that returns a `JqlClauseBuilder`. A query is constructed with one or more JQL clauses, with different conditions added in between.builder.where()returns a `JqlClauseBuilder` object for our `QueryBuilder`, on which we can then add multiple clauses.
3. Add the project clause to search for a project with its ID or key as argument. The project clause will return a `ConditionBuilder`:

```
builder.where().project("DEMO")
```

4. Add the `assignee` clause using the `AND` condition on the `ConditionBuilder`:

```
builder.where().project("DEMO").and().assigneeIsCurrentUser();
```

We can have numerous clauses added like this using the different conditions. Let us see some examples in the 'There's More…' section.

5. Add ordering, if you have any, using the `Order By` clause. We can sort based on assignee as follows:

```
builder.orderBy().assignee(SortOrder.ASC);
```

`SortOrder.DESC` can be used for descending orders.

6. Build the `Query` (`com.atlassian.query.Query`) object:

```
Query query = builder.buildQuery();
```

The `Query` object is immutable; once it is created it cannot be changed. The `JqlQueryBuilder` represents the mutable version of a `Query` object. We can create a Query from an already existing Query by calling `JqlQueryBuilder.newBuilder(existingQuery).`

7. Get an instance of the `SearchService`. It could be injected in the constructor of your plugin using dependency injection, or can be retrieved from the `ComponentAccessor` class as follows:

```
SearchService searchService =
ComponentAccessor.getComponent(SearchService.class);
```

8. Search using the query to retrieve the SearchResults (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/issue/search/SearchResults.html`):

```
SearchResults results = searchService.search(user, query,
PagerFilter.getUnlimitedFilter());
```

   Here we used `PagerFilter.getUnlimitedFilter()` to retrieve all the results. It is possible to limit the results to a particular range, say from 20 to 80 results, using the method `PagerFilter.newPageAlignedFilter(index, max)`. This will be useful when pagination is done, such as in the case of the issue navigator.

9. Retrieve the issues from the search results:

```
List<Issue> issues = results.getIssues();
```

The entire method will look as follows:

```
// Search for issues in DEMO project, assigned to current user private
List<Issue> getIssuesInProject(ApplicationUser user) throws
SearchException {
     JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
     builder.where().project("DEMO").and().assigneeIsCurrentUser();
     builder.orderBy().assignee(SortOrder.ASC);
     Query query = builder.buildQuery();

     SearchService searchService =
     ComponentAccessor.getComponent(SearchService.class);

     SearchResults results = searchService.search(user, query,
     PagerFilter.getUnlimitedFilter()); return results.getIssues();
}
```

Hopefully, that is a good starting point from which you can write more complex queries!

There's more…

As promised earlier, let us look at writing complex queries with a couple of examples.

We can extend the aforementioned search to include multiple projects, assignees, and a custom field. The JQL representation of the query will be as follows:

```
project in ("TEST", "DEMO") and assignee in ("jobinkk", "admin")
and "Customer Name" = "Jobin"
```

The `where` clause is written as follows:

```
builder.where().project("TEST", "DEMO").and().assignee().in("jobinkk",
"admin").and().customField(10000L).eq("Jobin");
```

Here, `10000L` is the ID of the custom field **Customer Name**.

We can group the conditions using `sub()` and `endsub()` to write even more complex queries:

```
project in ("TEST", "DEMO") and (assignee is EMPTY or reporter is EMPTY)
```

This can be written as follows:

```
builder.where().project("TEST",
"DEMO").and().sub().assigneeIsEmpty().or().reporterIsEmpty().endsub();
```

Similarly, we can write more complex queries.

See also

- The *Writing a JQL funcion* recipe in this chapter

Parsing a JQL query in plugins

In the previous recipe, we saw how to build a query to search within JIRA. In this recipe, we will see searching again, but without building a query using the APIs. We will use the JQL query as it is written in the Issue Navigator in advanced mode and search using the same.

How to do it…

Suppose we know the query that we want to execute. Let us assume it is the same we saw in the previous recipe: `project = "DEMO" and assignee = currentUser()`.

The following is how we do it:

1. Parse the JQL query:

```
String jqlQuery = "project = "DEMO" and assignee = currentUser()";
SearchService.ParseResult parseResult =
searchService.parseQuery(user, jqlQuery);
```

2. Check if the parsed result is valid or not:

```
if (parseResult.isValid())
{
    // Carry On
} else {
   // Log the error and exit!
}
```

3. If the result is valid, get the `Query` object from the `ParseResult`:

```
Query query = parseResult.getQuery();
```

4. Search for the issues and retrieve the `SearchResults`, as we have seen in the previous recipe:

```
SearchResults results = searchService.search(user, query,
PagerFilter.getUnlimitedFilter());
```

5. Retrieve the list of issues from the search results:

```
List<Issue> issues = results.getIssues();
```

How it works…

Here, the `parseQuery` operation in `SearchService` converts the `String` JQL query into the `Query` object we normally construct using `JqlQueryBuilder`. The actual parse operation is done by `JqlQueryParser` behind the scenes.

See also

- The *Searching in plugins* in this chapter

Linking directly to search queries

Haven't you wondered how we can link to a query from a template or JSP from a custom page or plugin page? In this recipe, we will see how we can create a link, programmatically and otherwise, to use in various places.

How to do it…

Let us first look at creating a search link programmatically. Perform the following steps:

1. Create the `Query` object using `JqlQueryBuilder`, as we have seen in the previous recipe.
2. Create a `IssueSearchParamaters` object as shown here:

```
    IssueSearchParameters params =

SearchService.IssueSearchParameters.builder().query(query).build();
```

3. Get an instance of the `SearchService`. It could be injected in the constructor of your plugin using dependency injection, or can be retrieved from the `ComponentAccessor` class as follows:

```
SearchService searchService =
ComponentAccessor.getComponent(SearchService.class);
```

4. Retrieve the query path from the `Query` object and `IssueSearchParamaters` using `SearchService`, as shown:

```
String queryPath = searchService.getIssueSearchPath(user, params);
```

5. Construct the link using the context path. In JSPs, you can do it as shown next:

```
<a href="<%= request.getContextPath()
 <ww:property value="/queryPath"/>">Show in Navigator</a>
```

   Here, `getQueryPath()` in the `Action` class returns the preceding `queryPath`.And in velocity templates:

```
<a href="${requestContext.baseUrl}$queryPath">
  Show in Navigator
</a>
```

   Here, `$queryPath` is the preceding `queryPath` in context!

How it works…

The `getIssueSearchPath()` method in `SearchService` returns the `queryPath` in a manner in which it can be used in a URL. It starts with `/issues/?jql=`, followed by the actual query as a web URL:

```
${requestContext.baseUrl}$queryPath will be then
${requestContext.baseUrl}/issues/?jql=someQuery
```

There's more…

Linking to a Quick Search is also pretty easy and useful. We can even store such searches in our browser favorites. All we need to do is find out the URL by replacing `%s` in JIRA's URL as follows:

```
http:///secure/QuickSearch.jspa?searchString=%s
```

For example, if your JIRA instance is `http://localhost:8080/` and you want to Quick Search for all the issues where you are the assignee, the relevant quick search string will be: **my open**.

The URL will then be:

```
http://localhost:8080/secure/QuickSearch.jspa?searchString=my+open
```



Please note that the spaces in Quick Search are replaced by `+`, while substituting `%s`.

Other examples:

`http://localhost:8080/secure/QuickSearch.jspa?searchString=my+open+critical` retrieves all open critical issues assigned to you
`http://localhost:8080/secure/QuickSearch.jspa?searchString=created:-1w+my` retrieves all the issues assigned to you, created in the past week

Index and de-index issues programmatically

As we have seen in the JIRA architecture explained in `Chapter 2`, *Understanding Plugin Framework*, searching in JIRA is based on Apache Lucene. The Lucene indexes are stored in File System and are used as the basis for the search queries executed in JIRA. Whenever an issue is updated, more records are created, or existing records are updated for that particular issue in the filesystem.

It is possible to programmatically index selected or all issues, or de-index an issue. Also, we can switch OFF or ON indexing selectively in our plugins if needed. An example where re-indexing is needed is when a custom post function updates a field on the issue. De-indexing might be needed if you want to hide the issue from search results, for example, while archiving. In this recipe, we will see both of these.

How to do it…

Most of the indexing operations can be done with the help of `IssueIndexingService`. An instance of `IssueIndexingService` can be created either by injecting in the constructor, or as follows:

IssueIndexingService indexingService =
ComponentAccessor.getComponent(IssueIndexingService.class);

The following are the important operations supported by `IssueIndexingService`:

- `reIndexAll()` – Indexes all the issues in JIRA. A good method if you want a custom admin operation to do indexing as well!
- `reIndex(Issue issue)` – To selectively index an issue by passing the `Issue` object.
- `deIndex(Issue issue)` – Method to de-index an issue. Once this is done, the issue won't appear in the search results.Be aware that when the issue is later updated, or a comment is added on the issue, JIRA automatically indexes again. So don't rely on calling this just once to permanently hide your issue from searches. To do so, the `IssueIndexer` should be overridden so that it won't index the issue again.
- `reIndexIssueObjects(Collection<? extends Issue> issueObjects)` - Indexes a collection of issues.



Check out the Java Docs at `https://docs.atlassian.com/jira/latest/com/atlassian/jira/issue/index/IssueIndexingService.html`for more available methods on the `IssueIndexingService`.

If we want to make sure that indexing is turned ON when we make a major update on an issue, we can do the following:

```
// Store the current state of indexing
boolean wasIndexing = ImportUtils.isIndexIssues();
// Set indexing to true
ImportUtils.setIndexIssues(true);
// Update the issue or issues
.................
// Reset indexing
ImportUtils.setIndexIssues(wasIndexing);
```

Here we use `ImportUtils` to save the current indexing state and turn it ON. After the update to issue(s) is done, indexing is turned back to whatever it was!

See also

- The *Searching in plugins* recipe in this chapter

Searching on issue entity properties

JIRA has a nice feature where we can store key/value pairs on JIRA entities like issues or projects. To make it even better, JIRA allows us to search issues using the entity properties set on them using JQL or REST API.

JIRA uses a plugin module named **Index Document Configuration** to expose these issue properties as properties that can be indexed. Once indexed, they can be searched using JQL, just like any other issue fields.

In this recipe, we will see how we can use the **Index Document Configuration** plugin module to search on issue properties.

Getting ready

First, we need to populate an existing issue in JIRA with some properties. These properties have a key, and the value can be a JSON object.

Let us assume that we want to set a property named **color**, and it has different attributes – **name** and **density**. In this case, the property key will be **color** and the value will be {“name”:”name of color”, “density” : “an integer value”}.

The property can be set using an add-on, or using JIRA's REST API. To make it easier, let us go with REST API. We can set the property using the following REST command:

```
curl -D- -u admin:admin -X PUT -H "Content-type: application/json"
http://localhost:2990/jira/rest/api/2/issue/DEMO-3/properties/color -d
'{"name":"red", "density" : 2}'
```

In this example, we are setting the color name as `red` and density as `2` on issue **DEMO-3**. Change the user name and password as appropriate.

How to do it…

Searching the issue properties using JQL is done in the following format:

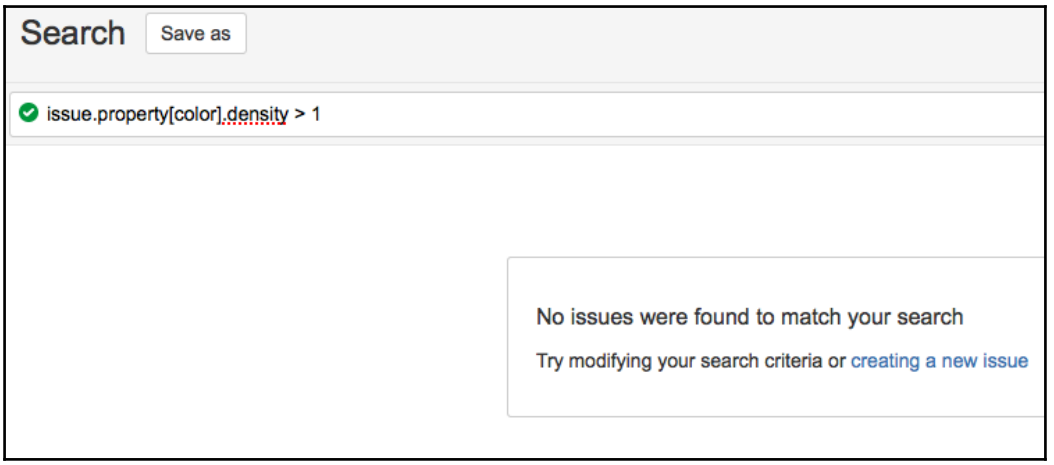
```
Issue.property[propertyKey].attributeName OPERATOR attributeValue
```

Here, the **OPERATOR** will be a valid JQL operator, referenced at `https://confluence.atlassian.com/jira/advanced-searching-operators-reference-433395091.html`, depending on the type of the property.

In our example, density is a number and we can search it as follows:

```
issue.property[color].density > 1
```

When we search using the preceding JQL, we will not get a result, as shown here:



For the searching to work, we need to define an **Index Document Configuration** module, as explained in the following:

1. Modify the plugin descriptor to include the **Index Document Configuration** module. This module has the following attributes:

    a. entity-key: The type of the property. As of now, the only available value is **IssueProperty**.

    b. key: The unique plugin identifier.

    The module also has a child element named **key**. Each **key** represents a property key that needs to be indexed. In our case, we only have one key – `color`. Each **key** has the following attribute:

    a. property-key: The property key from which the data is indexed.

    The **key** also has a child element named **extract**. Each **extract** references a value in the JSON object. In our example, `name` and `density` will have separate extracts. Each **extract** has the following attributes:

    a. path: The path of the JSON object to be indexed. It will be a key in the JSON object. In our example, paths are `name` and `density`.

b. type: The type of the referenced value. It can be one of the following:

i. number: Indexed as a number and allows numerical operators in JQL.

ii. text: Tokenized before indexing and allows fuzzy searching of words.

iii. string: Indexed as is. Allows exact searching of words.

iv. date: indexed as a date and allows date operators.

This is an important attribute, as the value is indexed based on this attribute and this impacts the search operators we can use for this JSON property.

c. alias: An alias name for the property in JQL searches.

In our example, the module is defined as follows:

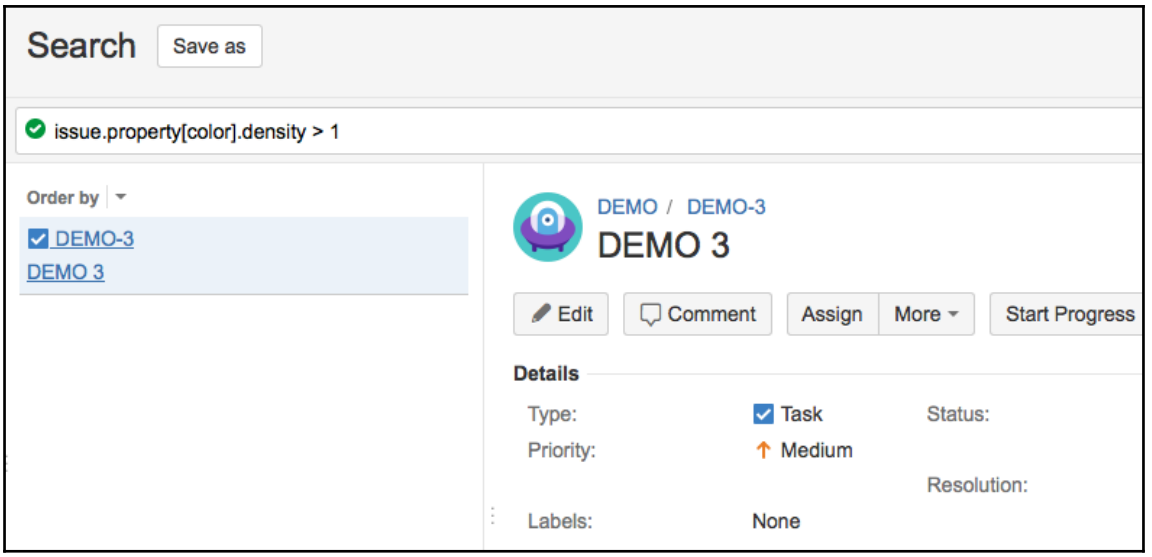
```
<index-document-configuration entity-key="IssueProperty"
  key="jira-issue-property-indexing">
  <key property-key="color">
    <extract path="name" type="string" />
    <extract path="density" type="number" />
  </key>
</index-document-configuration>
```

As you can see, `name` is defined as a string and `density` as a number.

2. Package the plugin and deploy it.

How it works…

Once the **Index Document Configuration** module is defined and deployed, we can search on the issue properties using the defined extracts. The earlier search that returned no issues will now return the appropriate search results, as shown here:



Similarly, we can define as many indexes as we want for the issue properties and use them in our searches.

There's more…

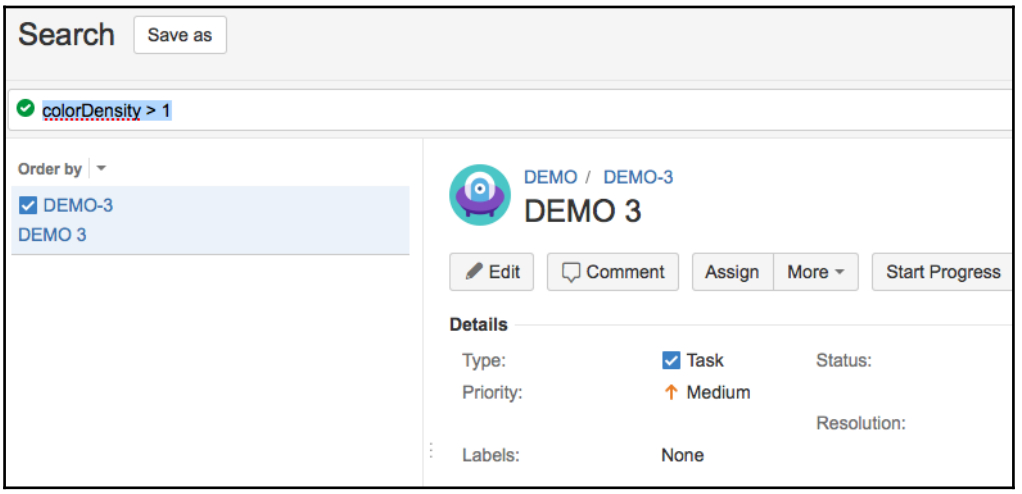
As you might have noticed, searching on issue properties is really cool, but the syntax is not very user-friendly. This is where the **extract alias** can help.

Let us modify our index document configuration as follows:

```
<index-document-configuration entity-key="IssueProperty"
 key="jira-issue-property-indexing">
  <key property-key="color">
    <extract path="name" type="string" />
    <extract path="density" type="number" alias="colorDensity"/>
  </key>
</index-document-configuration>
```

As you might have noticed, we added an alias **colorDensity** for the density **extract**. This gives us two main advantages.

It allows us to search using the user friendly alias, that is, instead of `issue.property[color].density > 1`, we can use `colorDensity > 1`, as shown here:



It allows JQL auto-completion and makes it easier for users to find it.

See also

- The *Creating a skeleton plugin* recipe in `Chapter 1`, *Plugin Development Process*
- The *Deploying your plugin* recipe in `Chapter 1`, *Plugin Development Process*

Managing filters programmatically

Be you a beginner in JIRA or a professional, one of the features used often is creating and managing filters. The fact that we can save the searches, share them, and subscribe to them adds a lot of value to JIRA. So, how do we programmatically create and manage filters?

In this recipe, we will learn how to manage filters programmatically.

How to do it…

We will see the various aspects of managing the filters one-by-one.

Creating a filter

Most of the operations on managing filters are done using `SearchRequestService`. For creating a filter, the following are the steps:

1. Create the query to be saved as a filter. The query can be created using `JqlQueryBuilder`, as we have seen in the previous recipes.
2. Create a `SearchRequest` object from the query, using the appropriate name, description, and so on:

```
SearchRequest searchRequest = new SearchRequest(query,
loggedInUser, "Test Filter", "Test Description");
```

3. Create a JIRA Service Context. If you are in an action class, you can get the service context by calling `getJiraServiceContext()` and if not, an instance can be created as follows:

```
JiraServiceContext ctx = new JiraServiceContextImpl(user);
```

   Here, `user` is the user for whom the filter should be created.

4. Get an instance of `SearchRequestService`. It can be either injected in the constructor or as follows:

```
SearchRequestService searchRequestService =
ComponentAccessor.getComponent(SearchRequestService.class);
```

5. Create the filter:

```
final SearchRequest newSearchRequest =
searchRequestService.createFilter(ctx, searchRequest, favorite);
```

   Here, `favorite` is a Boolean, which can be set to true if you want the filter to be made a favorite.

Updating a filter

Updating a filter is very similar to creating a filter. Here we have to retrieve the `SearchRequest` using its ID and then update the attributes such as name, description, and so on, using the setter methods.

Once the `SearchRequest` is modified, we can use `SearchRequestService` to update the filter:

```
SearchRequest updatedFilter = searchRequestService.updateFilter(ctx,
newSearchRequest, favorite);
```

Deleting a filter

JIRA takes the filter ID as the input for deleting a filter. Before we actually delete the filter, we need to validate the deletion as follows:

```
searchRequestService.validateForDelete(ctx, filterId);
```

If there are any errors, it will be added into the action's error collection. We can then check for the errors, and delete the filter if there are no errors:

```
if(!ctx.getErrorCollection().hasAnyErrors())
{
         searchRequestService.deleteFilter(ctx, filterId);
}
```

We can also delete all the filters of a user by using the following:

```
deleteAllFiltersForUser(JiraServiceContextserviceCtx, ApplicationUser
user);
```

Retrieving filters

The `SearchRequestService` also has a few methods to retrieve favorite filters, filters owned by a user, non-private filters, and so on. Key methods are listed as follows:

```
Collection<SearchRequest> getFavouriteFilters(ApplicationUser user);
Collection<SearchRequest> getOwnedFilters(ApplicationUser user);
Collection<SearchRequest> getNonPrivateFilters(ApplicationUser user);
Collection<SearchRequest> getFiltersFavouritedByOthers(ApplicationUser
user);
```

The method names are self-explanatory.

Sharing a filter

In order to share a filter, we need to retrieve the relevant filter and set the permissions on it using the following:

```
searchRequest.setPermissions(permissions);
```

Here, permissions is a set of `SharePermission` objects. The `SharePermission` objects can be created from a `JSONArray` using the `SharePermissionUtils` utility class. The `JSONObject` can have three keys – `Type`, `Param1`, and `Param2`.

The `Type` can have the following values: `global`, `group`, or `project`:

- When `Type` is `global`, `Param1` and `Param2` are not required
- When it is `group`, `Param1` is populated with the `groupname`
- When it is `project`, `Param1` is the ID of the project and `Param2` is the ID of the project role

An example of JSON arrays is as follows:

```
[{"type":"global"}]
[{"type":"group","param1":"jira-
administrators"},{"type":"project","param1":"10000","param2":"10010"}]
```

If you want to share a filter with everyone, permissions can be set as follows:

```
newSearchRequest.setPermissions(SharePermissionUtils.fromJsonArrayString("[
{"type":"global"}]"));
```

See also

- The *Searching in plugins* recipe in this chapter

Subscribing to a filter

We have seen various methods of managing filters. While filters are a great way to save searches and access them quickly at a later point in time, filter subscriptions are even better! The subscriptions help us to see the issues of interest at regular intervals, without even logging in to JIRA.

How do we subscribe to a filter programmatically? In this recipe, we will focus on subscribing to a filter in our plugins.

How to do it…

For the subscription of filters, JIRA provides a manager class implementing the `FilterSubscriptionService` interface. This class provides the important methods needed for managing filter subscriptions.

There are three important parameters for filter subscriptions:

1. **Cron Expression**: This is the most important part of a subscription. It tells us when the subscription has to run, or in other words, it defines the schedule of a subscription. Cron expressions consist of the following fields, separated by spaces:

Field	Allowed Values	Allowed Special Characters
Second	0-59	, – * /
Minute	0-59	, – * /
Hour	0-23	, – * /
Day-of-Month	1-31	, – * / ? L W C
Month	1-12 Or JAN-DEC	, – * /
Day-of-week	1-7 Or SUN-SAT	, – * / ? L C #
Year (Optional)	1970-2099	, – * /

The special characters denote the following:

Special Character	Usage
,	**List of values. For example, "MON,WED,FRI" means "every Monday, Wednesday, and Friday".**
–	Range of Values. For example, "MON-WED" means "every Monday, Tuesday, Wednesday".
*	All possible values. For example, * in the `Hour` field means "every hour of the day".

Special Character	Usage
/	Increments to the give value. For example, 1/3 in `Hour` field means "every three hours during the day, starting from 1.00 AM".
?	No particular value. This is useful when you need to specify a value for only one of the two fields, `Day-of-month` or `Day-of-week`, but not the other.
L	Last possible value. It has different meanings based on the context. For example: • L in `Day-of-week` means "Last day of every week" • 7L means "last Saturday of the month" • L in `Day-of-month` means "last day of the month" • LW means "last weekday of the month"
W	Weekday (MON-FRI) nearest to the given day of the month. For example, 1W means "nearest working day to the 1st of the month" – useful when you want to get the first working day of the month! It cannot be used with a range of days.
#	N'th occurance of a given day of the week. For example, MON#3 means "3rd Monday of the month".

We need to create a valid Cron expression based on the subscription we want to set up. The following are some examples based on these rules:

- `0 7 30 * * ?` – 7:30 AM Every Day
- `0 0/15 15 * * ?` – Every 15 minutes starting at 3.00PM and ending at 3:59 PM

You can find more examples in the Atlassian documentation for filter subscriptions at `http://confluence.atlassian.com/display/JIRA/Receiving+Search+Results+via+Email`.

2. **Group Name**: This is the group that we want to subscribe the filter. If the value is null, it will be considered as a personal subscription and the user in the context will be used.
3. **Email On Empty**: It is a Boolean value, which is **true** if you want the subscription to send an e-mail, even when it has no results.

Now let us see the steps to subscribe to a known filter:

1. Get an instance of the `FilterSubscriptionService`. You can either inject the class in the constructor, or get it using the `ComponentAccessor` class, as follows:

```
FilterSubscriptionService filterSubscriptionService =
ComponentAccessor.getComponent(FilterSubscriptionService.class);
```

2. Define the Cron expression based on the aforementioned rules:

```
String cronExpression = "0 0/15 * * * ? *";
// Denotes every 15 minutes
```

3. Define the group name. Use `null` if it is a personal subscription:

```
String groupName = "jira-administrators";
```

4. Create a JIRA Service Context. If you are in an action class, you can get the service context by calling `getJiraServiceContext()`, and if not, an instance can be created as follows:

```
JiraServiceContext ctx = new JiraServiceContextImpl(user);
```

   Here, `user` is the user for whom the filter is subscribed, in case it is a personal subscription.

5. Define whether an e-mail should be sent or not, even when the number of results is zero:

```
boolean mailOnEmpty = true;
```

6. Validate the Cron expression:

```
filterSubscriptionService.validateCronExpression
(ctx, cronExpression);
```

   If there are any errors, the Error Collection in `JiraServiceContext` will be populated with an error message.

7. If there are no errors, use the `FilterSubscriptionService` class to store the subscription:

```
if (!ctx.getErrorCollection().hasAnyErrors()){
  filterSubscriptionService.storeSubscription(ctx,
  filterId, groupName, cronExpression, emailOnEmpty);
}
```

Here, `filterId` is the ID of the filter we want to subscribe to, and can be obtained as `searchRequest.getId()`.

The subscription should now be saved and the mails will be sent based on the schedule defined by the Cron expression.

We can also update an existing subscription using `FilterSubscriptionService`, using the following method:

```
filterSubscriptionService.updateSubscription(ctx, subId, groupName,
cronExpression, emailOnEmpty);
```

Here, `subId` is the existing subscription ID.

How it works…

Each subscription we create is stored as a scheduled job in the system, which run based on the Cron expression we have defined while storing the subscription.

There's more…

If you want to use a web form like the one used in JIRA to create filter subscriptions, and you don't want to write the Cron expression, you can create a `CronEditorBean` using the parameters from the web form.



The various attributes supported in the form can be found from the CronEditorBean class. The Java Docs can be found at `http://docs.atlassian.com/software/jira/docs/api/latest/com/atlassian/jira/web/component/cron/CronEditorBean.html`.

Once the `CronEditorBean` is created, it can be parsed into a Cron expression, as follows:

```
String cronExpression = new
CronExpressionGenerator().getCronExpressionFromInput(cronEditorBean);
```

See also

- The *Searching in plugins* recipe in this chapter

7
Programming Issues

In this chapter, we will cover:

- Creating an issue from your plugin
- Creating subtasks on an issue
- Updating an issue
- Deleting an issue
- Adding new issue operations
- Conditions on issue operations
- Working with attachments
- Time tracking and worklog management
- Working with comments on issues
- Programming Change Logs
- Programming Issue Links
- JavaScript tricks on issue fields
- Creating issues and comments from e-mail

Introduction

We have so far seen how to develop custom fields, workflows, Reports & Gadgets, JQL functions, and other pluggable things associated with them. In this chapter, we will learn about programming "issues", namely, creating, editing, or deleting issues, creating new issue operations, and managing the various other operations available on issues via JIRA APIs.

As you might already know, an "issue" in JIRA represents a ticket. Different organizations use JIRA to track different types of "issues". It could be a Defect, a User Story, a Helpdesk Ticket, an Inventory Request, and so on. Creating and modifying issues is the most important functionality in JIRA. Hence programming issues becomes an important step for a plugin developer.

Creating an issue from a plugin

In this recipe, we will see how to create an issue from a plugin programmatically. We will be using `IssueService`to take advantage of its validation and error handling capabilities.

How to do it…

Following are the steps to create an issue using the `IssueService`:

1. Create an instance of the `IssueService` class. You can either inject it in the constructor, or get it from the `ComponentAccessor`, as shown:

```
IssueService issueService = ComponentAccessor.getIssueService();
```

2. Create the issue input parameters. In this step, we will set all the values that are required to create the issue using the `IssueInputParameters` class:

   a. Create an instance of the `IssueInputParameters` class:

```
IssueInputParameters issueInputParameters =
 issueService.newIssueInputParameters();
```

   b. Populate the `IssueInputParameters` with the values required to create the issue, as shown in the next few lines of code:

```
issueInputParameters
 .setProjectId(10000L)
 .setIssueTypeId("10000")
 .setSummary("Test Summary")
 .setReporterId("admin")
 .setAssigneeId("admin")
 .setDescription("Test Description")
 .setStatusId("10000")
 .setPriorityId("2")
 .setFixVersionIds(10000L);
```

c. Make sure all the required values, like project, issue type, summary, and other mandatory values, required when the issue is created using the user interface, are set on the `IssueInputParameters`.

d. Make sure the values set on `IssueInputParameters` are valid for your JIRA instance. For example, the project, issue type ID, priority ID, Fix version IDs, and so on, should have appropriate values.

3. Validate the input parameters using `IssueService`:

```
CreateValidationResult createValidationResult =
issueService.validateCreate(user, issueInputParameters);
```

Here, the `user` is the one creating the issue. The validation is done based on the user permissions, and the `createValidationResult` variable will have errors if the validation fails due to permission issues or due to invalid input parameters.

4. If the `createValidationResult` is valid, create the issue using `IssueService`:

```
if (createValidationResult.isValid()) {
    IssueResult createResult =
    issueService.create(user,createValidationResult);
}
```

Here, we use the `createValidationResult` object to create the issue, as it already has the processed input parameters. If the result is not valid, handle the errors as shown in the following code:

```
if (!createValidationResult.isValid()) {
 Collection<String> errorMessages =
 createValidationResult.getErrorCollection().getErrorMessages();

 for (String errorMessage : errorMessages) {
    System.out.println(errorMessage);
 }
  Map<String, String> errors =
  createValidationResult.getErrorCollection().getErrors();
  Set<String> errorKeys = errors.keySet();
  for (String errorKey : errorKeys) {
    System.out.println(errors.get(errorKey));
  }
}
```

Here, we just print the error to the console if the result is invalid. The `errorMessages` will have all non-field-specific errors like permission issue-related errors and so on, but any field-specific errors, like input validation errors, will appear in the `errors` map, where the key will be the field name. We should handle both the error types as appropriate.

5. After the creation of an issue, check if the `createResult` object is valid or not. If not, handle it appropriately. The `createResult` object will have errors only if there is a severe problem with JIRA (for example, one can't communicate with the database, the workflow has changed since you invoked validate, and so on):

```
if (!createResult.isValid()) {
  Collection<String> errorMessages =
  createResult.getErrorCollection().getErrorMessages();

  for (String errorMessage : errorMessages) {
    System.out.println(errorMessage);
  }
}
```

6. If `createResult` is valid, then the issue is created successfully and you can retrieve it as follows:

```
MutableIssue issue = createResult.getIssue();
```

How it works…

By using `IssueService`, JIRA now validates the inputs we give using the rules we have set up in JIRA via the user interface, such as the mandatory fields, permission checks, individual field validations, and so on. Behind the scenes, it uses the `IssueManager` class, which creates the issue from an empty issue object.

There's more…

As mentioned previously, `IssueService` uses the `IssueManager` class to create the issues. It can be used directly in our code, but this is not recommended, as it overrides all the validations and such.

But then again, what if you want to override those validations due to some reason? For example, to skip permission checks or field screen validations inside the plugin? In such cases, we might still need `IssueManager`.

Creating the issue using IssueManager

Follow these steps:

1. Initialize an issue object using the `IssueFactory` class:

```
MutableIssue issue = ComponentAccessor.getIssueFactory().getIssue();
```

2. Let all the fields required on the issue object:

```
issue.setProjectId(10000L);
issue.setIssueTypeId("5");
issue.setAssigneeId("admin");
```

3. Create the issue using `IssueManager`:

```
Issue createdIssue =
ComponentAccessor.getIssueManager()
.createIssueObject(user, issue);
```

4. Handle `CreateException` to capture any errors.

See also

- The *Dealing with custom fields on an issue* recipe in `Chapter 3`, *Working with Custom Fields*

Creating subtasks on an issue

In this recipe, we will demonstrate how to create a subtask from a JIRA plugin. It is very similar to issue creation, but there are some notable differences.

Subtasks are useful for splitting up a parent issue into a number of tasks, which can be assigned and tracked separately. The progress on an issue is generally a sum of progress on all its subtasks, although people use it for a lot of other purposes too.

How to do it…

There are two steps in creating a subtask:

1. Create an issue object. A subtask object is nothing but an issue object in the backend. The only difference is that it has a parent issue associated with it. So, when we create a subtask issue object, we will have to define the parent issue in addition to what we normally do while creating a normal issue.
2. Link the newly-created subtask issue to the parent issue.

Let's see the steps in more detail:

1. Create the subtask issue object similar to how we created the issue in the previous recipe. Here, the `IssueInputParameters` is constructed (after changing the methods, like `setIssueTypeId()`, appropriately).For this issue, we will use the `validateSubTaskCreate` method instead of `validateCreate`, which takes an extra parameter `parentId`:

```
CreateValidationResult createValidationResult =
issueService.validateSubTaskCreate(user, parent.getId(),
issueInputParameters);
```

   Here, parent is the issue object on which we are creating the subtask.

2. Create the subtask issue after checking for errors, as we have seen before:

```
if (createValidationResult.isValid()) {
    IssueResult createResult = issueService.create(user,
    createValidationResult);
}
```

3. Create a link between the newly-created subtask issue and the parent issue:

   a. Get an instance of `SubTaskManager`. You can either inject it in the constructor or get it from `ComponentAccessor`:

```
SubTaskManager subTaskManager =
ComponentAccessor.getSubTaskManager();
```

   b. Create the subtask link:

```
subTaskManager.createSubTaskIssueLink(parent,
createResult.getIssue(), user);
```

4. The subtask should now be created with a link back to the original parent issue.

See also

- The *Creating an Issue from your plugin* recipe in this chapter

Updating an issue

In this recipe, let's look at editing an existing issue. Users can edit the issue to update one or more fields on the issue, and there are screen schemes or field configurations to define what a user can see while editing an issue. Moreover, there is the "Edit" project permission to limit editing to selected users, groups, or roles.

Programmatically editing an issue also takes these things into account.

How to do it…

Let's assume that we have an existing issue object. We will just modify the `Summary` to a new summary. Following are the steps to do the same:

1. Create the `IssueInputParameters` object with the input fields that need to be modified:

```
IssueInputParameters issueInputParameters =
issueService.newIssueInputParameters();;
issueInputParameters.setSummary("Modified Summary");
```

   If you do not want to retain the existing values and just want the summary on the issue to be updated, you can set the `retainExistingValuesWhenParameterNotProvided` flag as shown:

```
issueInputParameters
.setRetainExistingValuesWhenParameterNotProvided(false);
```

2. Validate the input parameters using `IssueService`:

```
UpdateValidationResult updateValidationResult =
issueService.validateUpdate(user, issue.getId(),
issueInputParameters);
```

   Here, the issue is the existing issue object.

3. If `updateValidationResult` is valid, update the issue:

```
if (updateValidationResult.isValid()) {
    IssueResult updateResult = issueService.update(user,
    updateValidationResult);
}
```

   If it is not valid, handle the errors as we did while creating the issue.

4. Validate the `updateResult` and handle the errors, if any. If it is not valid, the updated issue object can be retrieved as follows:

```
MutableIssue updatedIssue = updateResult.getIssue();
```

Deleting an issue

In this recipe, let us look at deleting an issue programmatically.

How to do it…

Let us assume that we have an existing issue object. For deletion as well, we will use the `IssueService` class. Following are the steps to do it:

1. Validate the delete operation on the issue using `IssueService`:

```
DeleteValidationResult deleteValidationResult =
issueService.validateDelete(user, issue.getId());
```

   Here, the issue is the existing issue object that needs to be deleted.

2. If `deleteValidationResult` is valid, invoke the delete operation:

```
if (deleteValidationResult.isValid()) {
    ErrorCollection deleteErrors = issueService.delete(user,
    deleteValidationResult);
}
```

3. If the `deleteValidationResult` is invalid, handle the errors appropriately.
4. Confirm whether the deletion was successful by checking `deleteErrors` ErrorCollection:

```
if (deleteErrors.hasAnyErrors()){
```

```
        Collection<String> errorMessages =
        deleteErrors.getErrorMessages();

        for (String errorMessage : errorMessages) {
          System.out.println(errorMessage);
        }
    } else {
        System.out.println("Deleted Succesfully!");
    }
```

Adding new issue operations

In this recipe, we will look at adding new operations to an issue. The existing issue operations include **EditIssue**, **CloneIssue**, and so on, but most of the time people tend to look for similar operations with variations or entirely new operations that they can perform on an issue.

Prior to JIRA 4.1, the issue operations were added using the Issue Operations Plugin Module (`http://confluence.atlassian.com/display/JIRADEV/Issue+Operations+Plugin+Module`). But since JIRA 4.1, new issue operations are added using the **Web Item Plugin Module** (`http://confluence.atlassian.com/display/JIRADEV/Web+Item+Plugin+Module`).

A **WebItem** plugin module is a generic module that is used to define links in various application menus. One such menu is the issue operations menu. We will see more about the web items module and how it can be used to enhance the UI later in this book. In this recipe, we will only concentrate on using the web-item module to create issue operations.

Getting ready

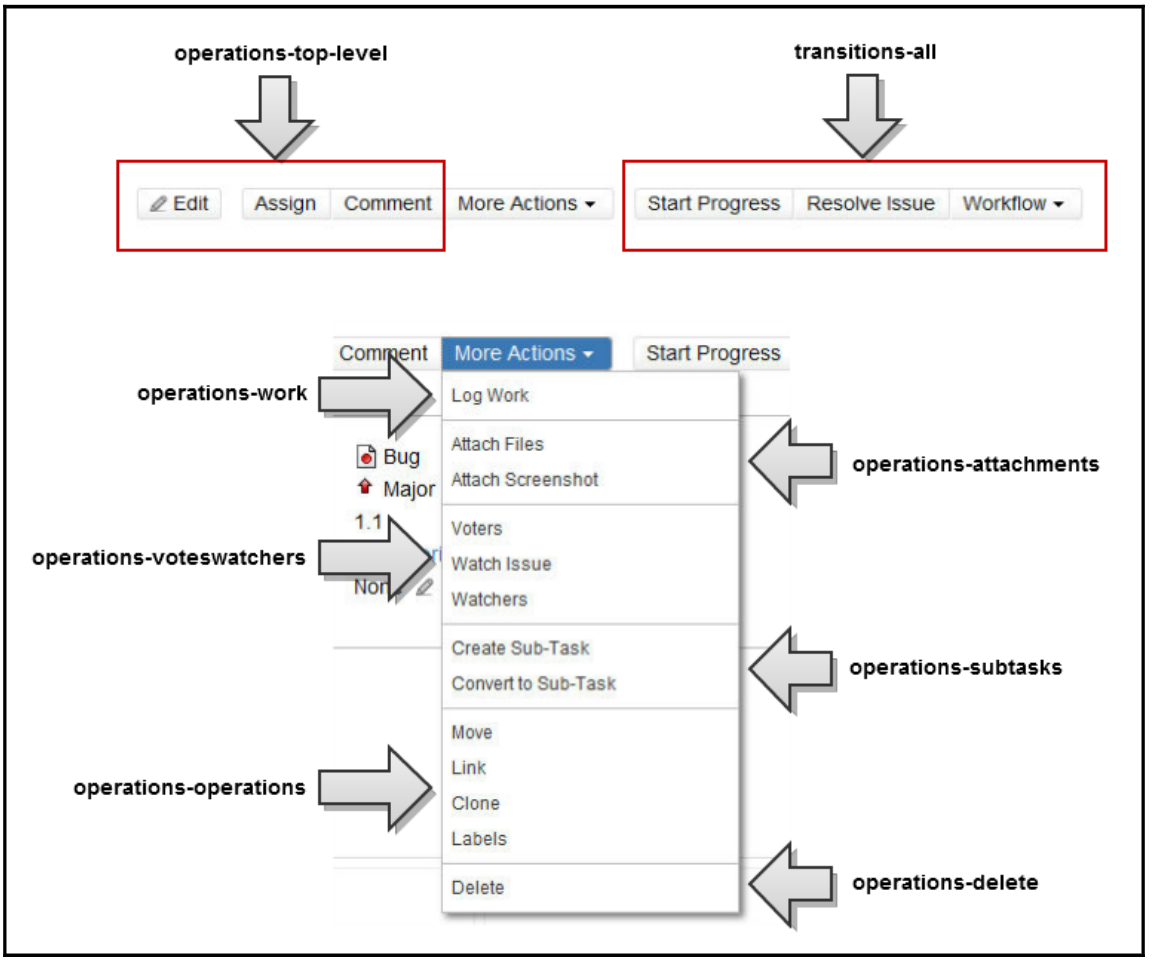
Create a skeleton plugin using Atlassian Plugin SDK.

How to do it…

Creating a web item is pretty easy! All we need to do is to place it in the appropriate section. There are already defined web sections in JIRA, and we can add more sections using the **WebSection** module if needed.

Let us create a new operation that lets us administer the project of an issue when we are on the **View Issue** page. All we need here is to add an operation that takes us to the **Administer Project** page. Following are the steps to create the new operation:

1. Identify the web section where the new operation should be placed.For issue operations, JIRA already has multiple web sections defined. We can add our new operation on any one of the sections. The following is a diagram from the Atlassian documentation detailing each of the available web sections for the issue operations:



2. For example, if we want to add a new operation along with **Move**, **Link**, and so on, we need to add the new web item under the **operations-operations** section. If you are rather hoping to add it right at the top, along with **Edit**, **Assign**, and **Comment**, the section must be **operations-top-level**. We can reorder the operation using the `weight` attribute.
3. Define the web item module in the plugin descriptor with the section identified in the previous step! For our example, the module definition in `atlassian-plugin.xml` will look like the following:

```
<web-item name="Manage Project"
i18n-name-key="manage-project.name" key="manage-project"
section="operations-operations" weight="100">

    <description key="manage-project.description">
        Manages the project in which the issue belongs
    </description>

    <label key="manage-project.label"></label>
    <link linkId="manage-project-link">
        /plugins/servlet/project-config/${issue.project.key}
    </link>
</web-item>
```

As you can see, it has a unique key and a human-readable name. The section here is `operations-operations`. The weight attribute is used to reorder the operations as we saw earlier, and here we use weight as 100 to put it at the bottom of the list.

The `label` is the name of the operation that will appear to the user. We can add a `tooltip` as well, which can have a friendly description of the operation. The next part, that is, `link` attribute, is the most important one, as that links us to the operation that we want to perform. Essentially, it is just a link and hence you can use it to redirect to anywhere-the Atlassian site, for example.

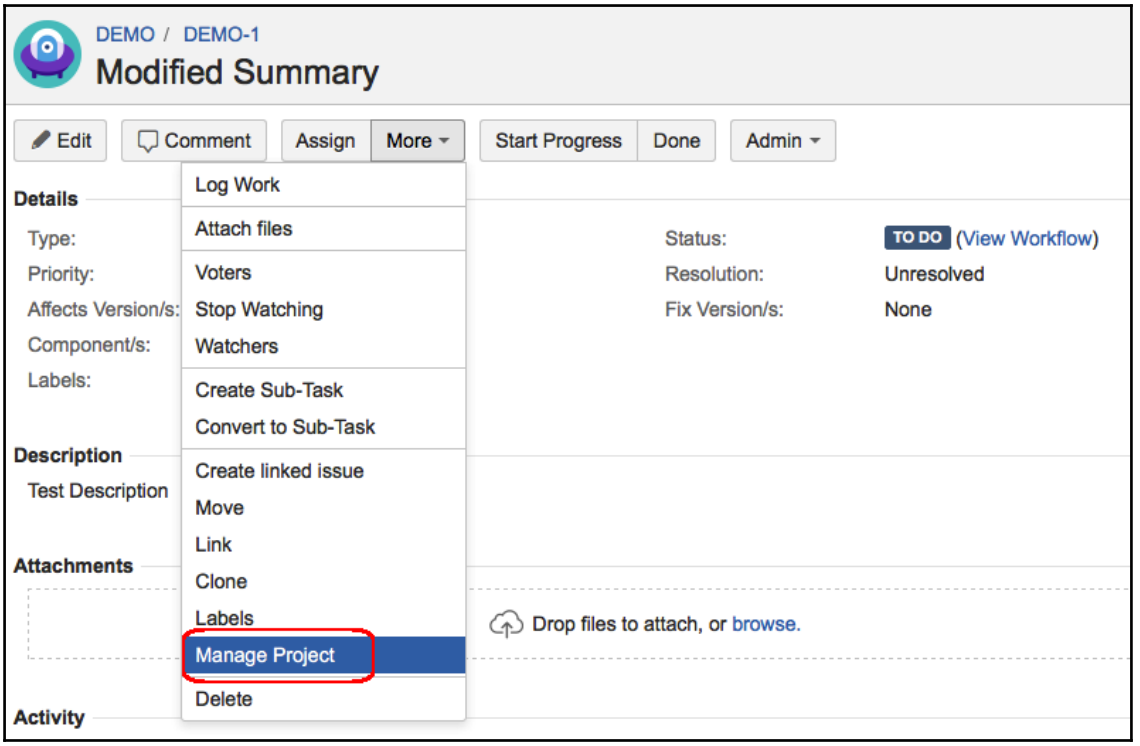
In our example, we need to take the user to the *administer* project area. Luckily, in this case, we know the servlet to be invoked, as it is an existing servlet in JIRA. All we need to do is to invoke the `project-config` servlet by passing the project key as well. The issue object is available on the view issue page as `$issue` and hence we can retrieve the project ID on the link as `${issue.project.key}`.

In cases where we need to do new things, we will have to create an action or servlet by ourselves and point the link to that action/servlet. We will see more about creating new actions and extending actions later in the book.

4. Package the plugin and deploy it.

How it works…

At runtime, you will see a new operation on the **ViewIssue** page on the **More** actions drop-down menu, as shown in the next screenshot:



After clicking on the link, the **Administer Project** screen will appear, as expected. As you might notice, the URL is populated with the correct `key` from the expression `${issue.project.key}`.

Also, just change the section or weight and see how the operation appears at various places on the screen!

There's more…

Prior to JIRA 4.1, The **IssueOperations** module was used in creating new issue operations. It is outside the scope of the book, though you can find the details in the Atlassian documentation at:
`http://confluence.atlassian.com/display/JIRADEV/Issue+Operations+Plugin+Module`.

See also

- The *Conditions on issue operations* recipe in this chapter

Conditions on issue operations

When new operations are created, it is often a requirement to hide them or show them, based on the permissions or state of the issue or something similar. JIRA allows conditions to be added while defining the web items, and the web item won't show up when the conditions are not satisfied.

In this recipe, let us lock down the new issue operation, created in the previous recipe, to a specific user named *admin*.

Getting ready…

Create the **ManageProject** issue operation, as explained in the previous recipe.

How to do it…

Following are the steps to add a new condition to an issue operation's web item:

1. Create the `condition` class. Since the issue operation is a web item, the easiest thing to do is to extend the `com.atlassian.jira.plugin.webfragment.conditions.AbstractWebCondition` class.While extending `AbstractWebCondition`, we will have to implement the `shouldDisplay` method, as shown here:

```
@Override
public boolean shouldDisplay(ApplicationUser user,
```

```
JiraHelper helper) {
    return user!= null && user.getName().equals("admin");
}
```

Here, a `true` value is returned only if the user is *admin*.

2. Include the `condition` in the web item. The updated web-item will look like the following:

```
<web-item name="Manage Project"
    i18n-name-key="manage-project.name"
    key="manage-project" section="operations-operations"
    weight="100">

    <description key="manage-project.description">
        Manages the project in which the issue belongs
    </description>

    <label key="manage-project.label"></label>
    <link linkId="manage-project-link">
        /plugins/servlet/project-config/${issue.project.key}
    </link>

    <condition class="com.jtricks.jira.webfragment.conditions
    .UserCondition"/>
</web-item>
```

It is possible to invert a condition by using the invert flag, as shown:

```
<condition class="com.jtricks.jira.webfragment
.conditions.UserCondition" invert="true"/>
```

Once inverted, this condition will return true for everyone other than *admin*.

Condition elements can also take optional parameters, as shown:

```
<condition class="com.atlassian.jira.plugin.webfragment
.conditions.HasProjectPermissionCondition">
    <param name="permission">create</param>
</condition>
```

The parameters can be retrieved in the `condition` class by overriding the `init(Map params)` method. Here, `params` is a map of string key/value pairs that hold these parameters. In the preceding example, the map will have `permission` as the key. The value passed, `create`, can be accessed using the key and can then be used in passing or failing the condition.

It is also possible to combine multiple conditions using the `conditions` element. The `conditions` element will have multiple `condition` elements connected through a logical AND (default) or OR condition.

For example, if we want to make our example operation available when one of the two conditions is satisfied, we can do it using an `OR` condition, as shown:

```
<conditions type="OR">
    <condition class="com.jtricks.jira.webfragment
    .conditions.ParamCondition"/>

        <param name="paramKey">paramValue</param>

    </condition> <condition class="com.jtricks.jira.webfragment
    .conditions.UserCondition"/>
</conditions>
```

3. Package the plugin and deploy it.

How it works…

Once the plugin is deployed, we can go and check the operation on the **View Issue** Page, as we did in the previous chapter. If you are the user with name *admin*, you will see the operation. If not, the operation won't be shown.

Working with attachments

The attachments feature is a useful feature in JIRA, and it sometimes helps to manage the attachments on an issue through the JIRA APIs. In this recipe, we will learn how to work with attachments using the JIRA API.

There are three major operations that can be done on attachments: Create, Read, and Delete. We will see each of them in this recipe.

Getting ready…

Make sure the attachments are enabled in your JIRA instance. You can do this from **Administration**|**System** | **Advanced**|**Attachments**, as mentioned at `http://confluence.atlassian.com/display/JIRA/Configuring+File+Attachments`.

How to do it…

All the operations on the attachments can be performed using the `AttachmentManager` API. The `AttachmentManager` can be retrieved either by injecting it in the constructor or from the `ComponentAccessor` class, as shown:

```
AttachmentManager attachmentManager =
ComponentAccessor.getAttachmentManager();
```

Creating an attachment

An attachment can be created on an issue using the `createAttachment` method on the `AttachmentManager`. First we need to create a `CreateAttachmentParamsBean` object, as shown here:

```
CreateAttachmentParamsBean attachmentBean =
new CreateAttachmentParamsBean.Builder(new File(fileName),
newFileName, "text/plain", user, issue).build();
```

The following are the arguments:

1. The `fileName` here needs to be the full path to the file on the server. You can also create a file object by uploading from the client machine, depending on the requirement.
2. `newFileName` is the name with which the file will be attached to the issue, and it can be different from the original filename.
3. The third parameter is the `contentType` of the file. In this case, we are uploading a text file and hence the content type is **text/plain**.
4. `user` is the user who is attaching the file.

5. `issue` is the issue to which the file will be attached.

   If you also want to set a list of properties on an attachment as a key/value pair, `CreateAttachmentParamsBean.Builder` can take a map containing the key/value properties. Similarly, `createdTime`, which is of the type `java.util.Date`, can be set on the builder to change the time of creation of the attachment. See Javadocs for more details.

   The attachment properties will be stored in the database using `PropertySet`.

   Once the attachment bean is created, the attachment is created on the issue as follows:

```
this.attachmentManager.createAttachment(attachmentBean);
```

Reading attachments on an issue

`AttachmentManager` has a method to retrieve the list of attachments of type `com.atlassian.jira.issue.attachment.Attachment` available on an issue. The following is how we do it:

```
List<Attachment> attachments =
this.attachmentManager.getAttachments(issue);

for (Attachment attachment : attachments) {
    System.out.println("Attachment: "+attachment.getFilename()
    +" attached by "+attachment.getAuthorKey());
}
```

The object attachment holds all the information of the attachment, including any properties set during the creation of the attachment.

Deleting an attachment

All you need to do here is to retrieve the attachment object that needs to be deleted and invoke the `deleteAttachment` method on `AttachmentManager`:

```
this.attachmentManager.deleteAttachment(attachment);
```

Here, attachment is an attachment that can be retrieved using the `getAttachment(id)` method or by iterating on the list of attachments retrieved previously.

There's more…

`AttachmentManager` also has other useful methods, like `attachmentsEnabled()`, `isScreenshotAppletEnabled()`, `isScreenshotAppletSupportedByOS()`, and so on, to check whether the respective functionalities are enabled or not.

Check out: `http://docs.atlassian.com/jira/latest/com/atlassian/jira/issue/AttachmentManager.html` for a full list of available methods.

Time tracking and worklog management

Time tracking is one of the biggest pluses for any issue tracking system. JIRA's time tracking is highly configurable and gives plenty of options to manage the work done and the remaining time.

Even though the time tracking in JIRA can be done using the JIRA UI, many users want to do it from the customized pages or third-party applications or plugins. In this recipe, we will see how to do time tracking using the JIRA APIs.

Before we start, each of the operations on worklogs, namely, create, edit, or delete, have different modes. Whenever one of these operations is performed, we can adjust the remaining amount of work to be done in the following ways:

- Let JIRA adjust the remaining work automatically.For example, if the remaining estimate is 2 hours and if we log 30 minutes, JIRA will automatically adjust the remaining estimate to 1 hour 30 minutes.
- Enter a new remaining estimate time while performing the operations.For example, if the remaining estimate is 2 hours and if we log 30 minutes, we can force JIRA to change the remaining estimate to 1 hour (instead of the automatically calculated 1 hour 30 minutes).
- Adjust the remaining estimate, or in other words, reduce a specific amount of time from the remaining estimate.For example, if the remaining estimate is 2 hours and if we log 30 minutes, we can force JIRA to reduce the remaining estimate by 1 hour 30 minutes (instead of automatically reducing the logged 30 minutes). When we do that, the remaining estimate will come out to be 30 minutes.
- Leave the remaining estimate as it is.

Getting ready…

Make sure time tracking is turned on, as explained at `http://confluence.atlassian.com/display/JIRA/Configuring+Time+Tracking`. It can be enabled from the **Administration** | **System** | **Issue Features** | **Time Tracking** menu.

How to do it…

Worklogs in JIRA can be managed using the `WorklogService` class. It does all the major operations, like creating worklogs, updating them, or deleting them, and that too, in all the four different modes, we have seen earlier.

We will see how to create worklogs, or in other words, log work in the following four modes:

- Auto adjusting the remaining estimate
- Logging work and retaining the remaining estimate
- Logging work with a new remaining estimate
- Logging work and adjusting the remaining estimate by a value

Auto adjusting the remaining estimate

1. Create the JIRA Service Context for the user who is logging work:

```
        JiraServiceContext jiraServiceContext = new
JiraServiceContextImpl(user);
```

2. Create a `WorklogInputParametersImpl.Builder` object to create the parameters needed for the worklog creation:

```
final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue)
.timeSpent(timeSpent).startDate(new Date())
.comment(null).visibility(Visibilities.publicVisibility());
```

   Here, `issue` is the issue on which work is logged, and `timeSpent` is the time that we are going to log in. `timeSpent` is a String that represents the format in which time is entered in JIRA , that is, `*w *d *h *m` (representing weeks, days, hours, and minutes, where * can be any number).

`startDate` is the date from when the work was started. We can also optionally add **comments** and set the worklog visibility to certain **groups** or **project roles**! **Visibility** is set using the `Visibility` bean, which can be constructed using the `Visibilities` class, as shown in the preceding example.

3. Create the `WorklogInputParameters` object from the builder and validate it using the `WorklogService`:

```
WorklogResult result =
this.worklogService.validateCreate(jiraServiceContext,
builder.build());
```

4. Create the worklog using `WorklogService`, as shown here:

```
Worklog worklog =
this.worklogService
.createAndAutoAdjustRemainingEstimate(jiraServiceContext,
result, false);
```

Here, as you can see, the method invoked is `createAndAutoAdjustRemainingEstimate`, which will create the worklog and automatically adjust the remaining estimate on the issue.

The method takes as input the service context we created, the `WorklogResultobject` after validating the input parameters, and a Boolean, which will be used to dispatch an event, if needed. When the Boolean value is true, the **Work Logged On Issue** event is fired.

With this, the work will be logged on the issue.

Logging work and retaining the remaining estimate

Here, the first three steps are similar to what was discussed in the *Auto-adjusting the remaining estimate* section. The only difference is that the method invoked on `WorklogService` is `createAndRetainRemainingEstimate` instead of `createAndAutoAdjustRemainingEstimate`.

The full code is as shown:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);

final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue).timeSpent(timeSpent)
.startDate(new Date()).comment(null)
```

```
.visibility(Visibilities.publicVisibility());
WorklogResult result = this.worklogService
.validateCreate(jiraServiceContext, builder.build());
Worklog worklog = this.worklogService.createAndRetainRemainingEstimate
(jiraServiceContext, result, false);
```

Logging work with a new remaining estimate

Here, the first two steps are similar to what was discussed in the *Auto-adjusting the remaining estimate* section:

1. Create the JIRA Service Context for the user who is logging work:

```
JiraServiceContext jiraServiceContext =
new JiraServiceContextImpl(user);
```

2. Create a `WorklogInputParametersImpl.Builder` object to create the parameters needed for the worklog creation:

```
final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue)
.timeSpent(timeSpent) .startDate(new Date()).
comment(null.visibility(Visibilities.publicVisibility());
```

3. Create the New Estimate Input Parameters from the Builder object:

```
final WorklogNewEstimateInputParameters params =
builder.newEstimate(newEstimate).buildNewEstimate();
```

   Here, we specify the `newEstimate`, which is a string representation similar to `timeSpent`. The `newEstimate` will be set as the remaining estimate on the issue.

4. Create the `WorklogResult` from `WorklogNewEstimateInputParameters` using `WorklogService`:

```
WorklogResult result =
this.worklogService.validateUpdateWithNewEstimate
(jiraServiceContext, params);
```

   The result here will be an instance of `WorklogNewEstimateResult`, which will be used in the next step.

5. Create the worklog using `WorklogService`:

```
Worklog worklog =
this.worklogService.createWithNewRemainingEstimate
(jiraServiceContext, (WorklogNewEstimateResult) result, false);
```

Here, the method used is `createWithNewRemainingEstimate`, which sets the `newEstimate` as the remaining estimate on the issue, after logging the work using `timeSpent`. As you might have noticed, the result object is converted to `WorklogNewEstimateResult`.

Logging work and adjusting the remaining estimate by a value

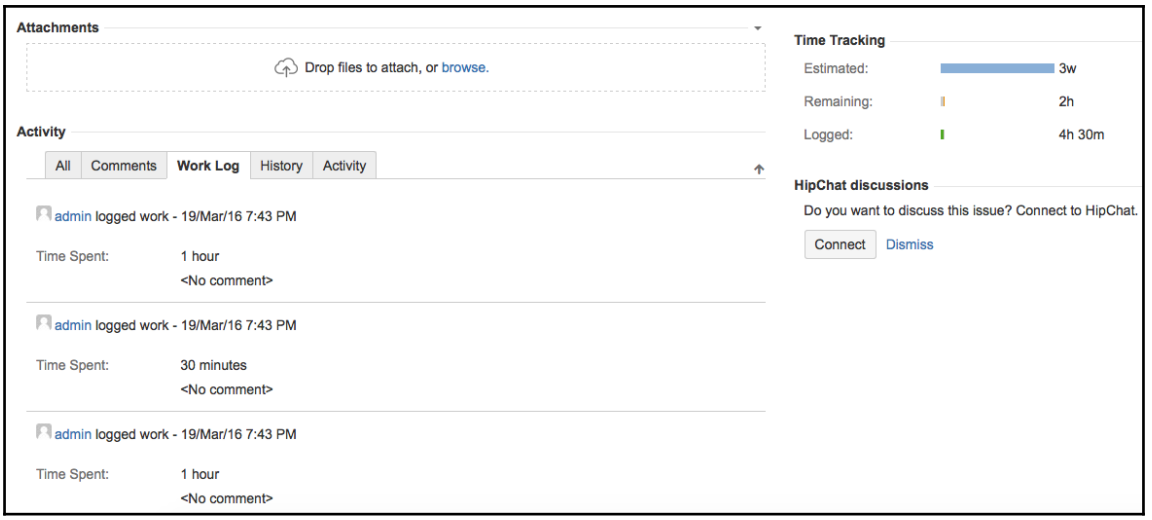
Here, the process is very similar to the preceding section. The only difference is that the `adjustmentAmount` method is used on Builder instead of `newEstimate`, and `validateCreateWithManuallyAdjustedEstimate` is used on `WorklogService` to create the worklog. Also, the `WorklogResult` is an instance of `WorklogAdjustmentAmountResult`.

The code is as follows:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);

final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue).timeSpent(timeSpent)
.startDate(new Date()).comment(null)
.visibility(Visibilities.publicVisibility());
final WorklogAdjustmentAmountInputParameters params =
builder.adjustmentAmount(estimateToReduce)
.buildAdjustmentAmount();
WorklogResult result =
worklogService.validateCreateWithManuallyAdjustedEstimate
(jiraServiceContext, params);
Worklog worklog = this.worklogService
.createWithManuallyAdjustedEstimate
(jiraServiceContext, (WorklogAdjustmentAmountResult) result, false);
```

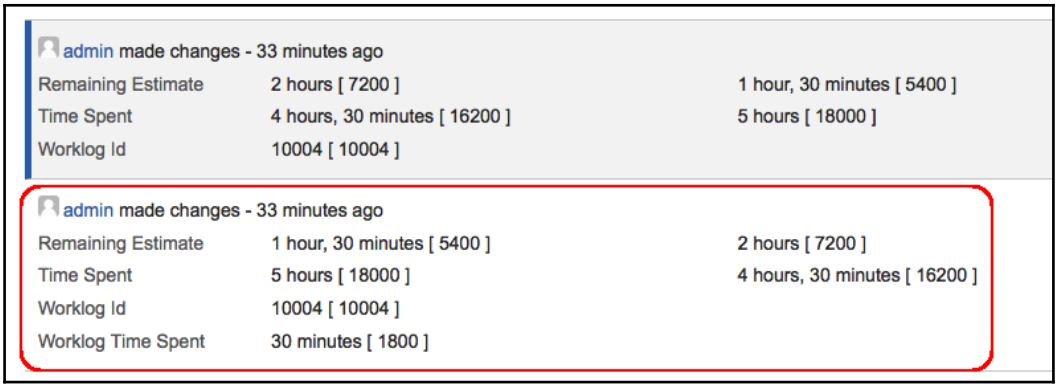
How it works…

Once we create or update the worklogs using the `WorklogService` API, the changes will be reflected on the issue under the **WorkLog** tab, as shown in the following screenshot:



You can also see that the graphical representation of time tracking reflects these changes.

When a worklog is deleted, it appears on the **Change history** as shown:



There's more

Similarly, worklogs can be updated using `WorklogService` as well.

Updating worklogs

Updating worklogs is similar to creating them in many ways. Here, we pass the ID of the `Worklog` object to be updated instead of the issue we pass while creating a worklog. And, of course, the methods invoked on `WorklogService` are different. The following is the code to update a given worklog for the first mode, where the remaining estimate is auto-adjusted:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);

final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.worklogId(worklog.getId())
.timeSpent(timeSpent).startDate(new Date()).comment(null)
.visibility(Visibilities.publicVisibility());
WorklogResult result = this.worklogService.validateUpdate
(jiraServiceContext, builder.build());
Worklog updatedLog =
this.worklogService.updateAndAutoAdjustRemainingEstimate
(jiraServiceContext, result, false);
```

As you can see, a *builder* is created by passing the worklog ID, which is unique across issues. The `WorklogResult` here is created using the `validateUpdate` method and the worklog is finally updated using the `updateAndAutoAdjustRemainingEstimate` method.

The other modes are also similar to how we created the worklogs. Let us quickly see how to update a worklog with a new remaining estimate:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);
final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.worklogId(worklog.getId())
.timeSpent(timeSpent).startDate(new Date())
.comment(null).visibility(Visibilities.publicVisibility());
final WorklogNewEstimateInputParameters params =
builder.newEstimate(newEstimate).buildNewEstimate();
WorklogResult result = this.worklogService
.validateUpdateWithNewEstimate(jiraServiceContext, params);
Worklog updatedLog = this.worklogService
.updateWithNewRemainingEstimate(jiraServiceContext,
(WorklogNewEstimateResult) result, false);
```

The preceding code looks pretty familiar, doesn't it? It is similar to creating a worklog with a new estimate, except that we call the respective update methods, as discussed before.

We can update a worklog by retaining the estimate and also adjust it by a specified amount of time from the remaining estimate in the same way.

Deleting worklogs

Deleting a worklog is slightly different and maybe easier than creating or updating one, as it doesn't involve building the input parameters.

Auto Adjusting remaining estimate

All we need here is the worklog ID and to create the JIRA Service Context. The code is as shown here:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);
WorklogResult worklogResult = worklogService.validateDelete
(jiraServiceContext, worklog.getId());
worklogService.deleteAndAutoAdjustRemainingEstimate
(jiraServiceContext, worklogResult, false);
```

Here, the `validateDelete` method takes the worklog ID as input and creates a `WorklogResult`, which is then used in the `deleteAndAutoAdjustRemainingEstimate` method.

Deleting a worklog and retaining the remaining estimate

This is done in almost the same way as mentioned in the previous section, expect that the `deleteAndRetainRemainingEstimate` method is used instead of `deleteAndAutoAdjustRemainingEstimate`:

```
JiraServiceContext jiraServiceContext =
new JiraServiceContextImpl(user);

WorklogResult worklogResult = worklogService.validateDelete
(jiraServiceContext, worklog.getId());

worklogService.deleteAndRetainRemainingEstimate
(jiraServiceContext, worklogResult, false);
```

Deleting a worklog with a new remaining estimate

As mentioned before, we don't create the input parameters while deleting worklogs. Instead, the `newEstimate` is used to create `WorklogResult`, which is an instance of `WorklogNewEstimateResult`, while validating. The code is as follows:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);

WorklogResult worklogResult = worklogService.validateDeleteWithNewEstimate
(jiraServiceContext, worklog.getId(), newEstimate);
```

```
worklogService.deleteWithNewRemainingEstimate
(jiraServiceContext, (WorklogNewEstimateResult) worklogResult, false);
```

Deleting a worklog and adjusting the remaining estimate

This is also pretty much the same as mentioned in the previous section, except for the method names:

```
JiraServiceContext jiraServiceContext = new JiraServiceContextImpl(user);

WorklogResult worklogResult =
worklogService.validateDeleteWithManuallyAdjustedEstimate
(jiraServiceContext, worklog.getId(), adjustmentAmount);

worklogService.deleteWithManuallyAdjustedEstimate
(jiraServiceContext, (WorklogAdjustmentAmountResult) worklogResult, false);
```

Here, `adjustmentAmount` is the value that is used to increase the remaining estimate on the issue.

Working with comments on issues

In this recipe, we will see how to manage commenting on issues using the JIRA API.

Let us have a look at the three major operations—creating, editing, and deleting comments. We will also have a look at how to restrict the comment visibility to a specific group of people or to a project role.

How to do it…

JIRA uses the `CommentService` class to manage the comments on an issue.

Creating comments on issues

A comment can be added on to an issue in a few simple steps:

1. Create `commentParams` of type `CommentService.CommentParameters`, with the appropriate values. You can do it with the help of `CommentService.CommentParameters.CommentParametersBuilder`, as shown here:

```
CommentService.CommentParameters commentParams =
new CommentService.CommentParameters
.CommentParametersBuilder().issue(issue)
.body(commentString).build();
```

Here, `commentString` is the comment we are adding and `issue` is the issue on which the comment is added.

2. Validate the comment creation using the params created in Step 1 and the `user` creating the comment:

```
CommentCreateValidationResult commentResult =
this.commentService.validateCommentCreate(user, commentParams);
```

3. Create the comment using the validation result:

```
Comment comment = this.commentService.create(user,
commentResult, true);
```

Here, `user` is the user creating the comment and the third parameter is a Boolean determining whether to dispatch an **Issue Commented** event. You can pass **false** if you do not want to send notifications on comment creation.

With this, the comment is created on the JIRA issue.

Creating comments on an issue and restricting it to a project role or group

If we need to restrict the **visibility** of the comments, we need to use the `visibility()` method on `CommentService.CommentParameters.CommentParametersBuilder`. Here, we need to create a `Visibility` bean, with the appropriate **project role** or **group level** set on it.

Visibility for a project role or group can be created using the `Visibilities` utility class, as follows:

```
Visibility groupVisibility = Visibilities.groupVisibility(group);
Visibility roleVisibility = Visibilities.roleVisibility(roleId);
```

It is also possible to combine those and you can find all the available methods at `https://docs.atlassian.com/jira/latest/com/atlassian/jira/bc/issue/visibility/Visibilities.html`.

Once the visibility is defined, we can use it while creating the `CommentService.CommentParameters` object:

```
CommentService.CommentParameters commentParams = new
CommentService.CommentParameters.CommentParametersBuilder()
.issue(issue).body(commentString).visibility(visibility).build();
```

Updating comments

Updating a comment is very similar to the create operation. The methods used are different, but the sequence of operations is the same. Following are the steps:

1. Create `commentParams` of type `CommentService.CommentParameters`, with the values to be modified on the comment:

   ```
   CommentService.CommentParameters commentParams =
   new CommentService.CommentParameters
   .CommentParametersBuilder().body(modifiedComment).build();
   ```

   Here, `modifiedComment` is the new comment value.

2. Validate the comment update using the params created in step 1 and the `user` updating the comment:

   ```
   CommentUpdateValidationResult commentResult =
   this.commentService.validateCommentUpdate(user,
   comment.getId(),commentParams);
   ```

3. Update the comment using the validation result:

   ```
   Comment comment = this.commentService.update(user,
   commentResult, true);
   ```

   Here, `user` is the user updating the comment and the third parameter is a Boolean determining whether to dispatch an **Issue Comment Edited** event. You can pass **false** if you do not want to send notifications on comment edits.

You may modify other parameters, like visibility, just like we did during the create operation.

Deleting comments

Deleting a comment is the easiest of all. It can be done using a simple call, as shown here:

```
this.commentService.delete(new JiraServiceContextImpl(user), comment,
false);
```

`comment` is the comment object to be deleted. The last Boolean argument determines whether to dispatch the event or not.

Programming change logs

Tracking changes to an issue is very important. JIRA stores all the changes that are done on an issue as change logs, along with the information of who made the change and when. Sometimes, when we do custom development, we will have to update the Change History ourselves when there are some changes on the issue by our plugin.

Change Histories are logged as change groups, which are groups of one or more change items, made by a user at any one time. Each change item will be a change made on any single field.

In this recipe, we will see how to add change logs on an issue using the JIRA API.

How to do it…

Each change item in JIRA is created as a `ChangeItemBean`. `ChangeItemBean` can be of two different types-one for **system** fields where the field type is `ChangeItemBean.STATIC_FIELD`, and another for **custom** fields where the field type is `ChangeItemBean.CUSTOM_FIELD`.

The following are the steps to add a Change History:

1. Create a `ChangeItemBean` for the change that needs to be recorded for every item that is changed:

```
ChangeItemBean changeBean =
new ChangeItemBean(ChangeItemBean.STATIC_FIELD,
IssueFieldConstants.SUMMARY, "Old Summary", "New Summary");
```

Here, the first attribute is the `fieldType` and the second one is the name of the field. For system fields of type `ChangeItemBean.STATIC_FIELD`, the name can be retrieved from `IssueFieldConstants` class.
For example, `IssueFieldConstants.SUMMARY` represents the issue summary.

The third and fourth arguments are the *oldvalue* and the *newvalue* of the field, respectively.

As we know, some of the JIRA fields have an ID value and a String value.
For example, the issue Status has the status **name** and the corresponding status **ID**. In such cases, we can use an overridden constructor that also takes the old ID and new ID as shown here:

```
ChangeItemBean changeBean = new
ChangeItemBean(ChangeItemBean.STATIC_FIELD,
IssueFieldConstants.STATUS,"1", "Open", "3", "In Progress");
```

For custom fields, we use the field type `ChangeItemBean.CUSTOM_FIELD` and the custom field name. Everything else is the same:

```
ChangeItemBean changeBean = new
ChangeItemBean(ChangeItemBean.CUSTOM_FIELD, "My Field",
"Some Old Value", "Some New Value");
```

It is worth noting that the field name can be manipulated to give any value when the `fieldType` is `ChangeItemBean.CUSTOM_FIELD`. It is probably a useful feature when you want to programmatically add change logs that are not directly related to a field; say, for adding a subtask:

```
ChangeItemBean changeBean = new
ChangeItemBean(ChangeItemBean.CUSTOM_FIELD, "Some Heading",
"Some Old Value", "Some New Value");
```

2. Create a change holder and add the change items into it:

```
IssueChangeHolder changeHolder = new DefaultIssueChangeHolder();
changeHolder.addChangeItem(changeBean);
```

3. Create and store the `changelog` using the items in the `changeHolder` using `ChangeLogUtils` class:

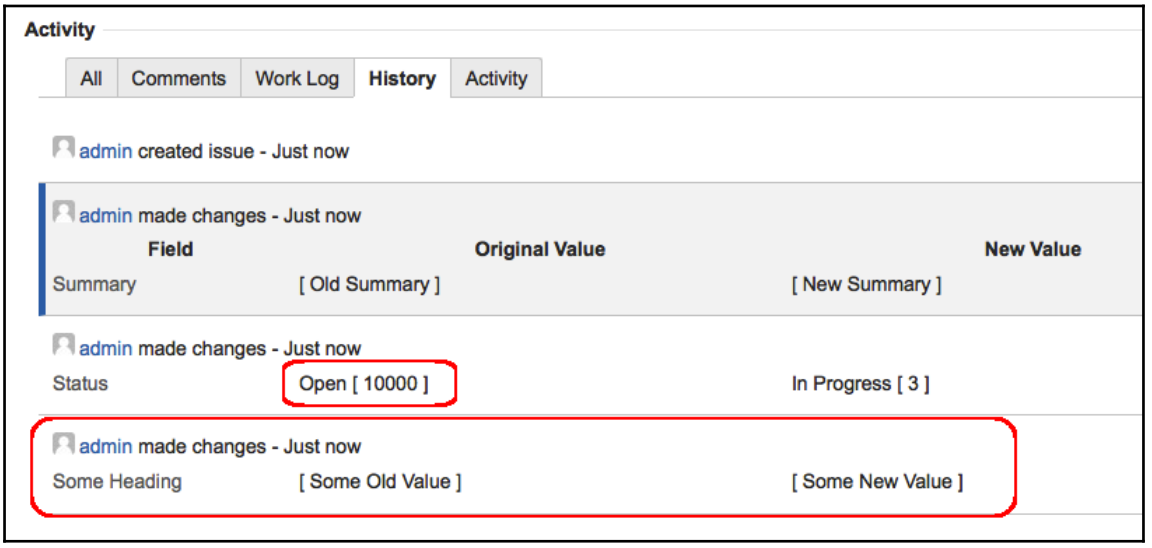
```
GenericValue changeLog = ChangeLogUtils.createChangeGroup(user,
issue.getGenericValue(), issue.getGenericValue(),
changeHolder.getChangeItems(), false);
```

Here, `user` is the user making the change. The second and third arguments are the original issue and the issue after changes. You can give both the same if the change items are explicitly created and added to `changeHolder`. But if we are modifying an issue using the setter methods, an easier way might be to pass the original issue object, along with the modified issue object (object after setter methods are invoked), and set the last argument as `true`, which determines whether a list of change items needs to be generated from the before and after objects. In that case, we don't need to explicitly create `changeItems`, and hence the third argument can be an empty list.

We can still pass additional `changeItems` if needed, as the third argument, in which case both the passed `changeItems` and generated `changeItems` (from issue *before* and *after* modification) will be created!

How it works…

Once the change logs are added, they will appear in the issues change log panel, as shown in the following screenshot:



Notice that the highlighted *changelog* is added even though there is no field named **Some Heading**. Also, see how both the ID and name are shown for the **Status** field!

Programming issue links

Issue linking is another important feature in JIRA. It helps us to define the relationship between issues. In this recipe, we will see how to create links between issues and to break them using the JIRA APIs.

Before we start, an issue link type has an **inward** and an **outward** description. For every issue link, there will be a **source** issue and a **destination** issue. From a source issue, we can look up the destination issues by looking up the outward links. Similarly, from a destination issue, we can look up the source issues by looking up inward links.

Getting ready…

Make sure the Issue Linking feature is turned ON in JIRA and valid link types are created. This can be done from **Administration** | **System** | **Issue Features** | **Issue Linking**, as explained at
`http://confluence.atlassian.com/display/JIRA/Configuring+Issue+Linking`.

How to do it…

Issue Links are managed in JIRA with the help of the `IssueLinkManager` class. The following are the steps to create an issue link between two given issues:

1. Get the `IssueLinkType` object for the link type we are going to create. This can be retrieved using the `IssueLinkTypeManager` class. The `IssueLinkTypeManager` class can be retrieved from the `ComponentAccessor`, or can be injected in the constructor:

```
IssueLinkTypeManager issueLinkTypeManager =
ComponentAccessor.getComponent(IssueLinkTypeManager.class);

IssueLinkType linkType =
issueLinkTypeManager.getIssueLinkTypesByName
("Duplicate").iterator().next();
```

   Here we are getting the `Duplicate` issue link type. Even though the `getIssueLinkTypesByName` method returns a collection, there will be only one link with the same name.

2. Create the issue link using the `IssueLinkManager` class. The `IssueLinkManager` class can also be retrieved from the

`ComponentAccessor`class or injected in the constructor:

```
IssueLinkManager issueLinkManager =
ComponentAccessor.getIssueLinkManager();

issueLinkManager.createIssueLink(sourceIssue.getId(),
destIssue.getId(), linkType.getId(), null, user);
```

Here we pass the source and destination issue IDs, in the order mentioned, along with the link type ID. The fourth parameter is the sequence, which is of type `long`, used to order the links on the user interface. `user` is the user who is performing the link action.

There's more…

Let's now see how to delete them or just display links.

Deleting Issue Links

Following are the steps:

1. Retrieve the `IssueLinkType`, as we did earlier:

```
IssueLinkTypeManager issueLinkTypeManager =
ComponentAccessor.getComponent(IssueLinkTypeManager.class);

IssueLinkType linkType =
issueLinkTypeManager.getIssueLinkTypesByName
("Duplicate").iterator().next();
```

2. Get the `IssueLink` to be deleted using the `IssueLinkManager` class:

```
IssueLink issueLink = issueLinkManager
.getIssueLink(sourceIssue.getId(), destIssue.getId(),
linkType.getId());
```

Here, the `sourceIssue` and `destIssue` are the source and destination issues, respectively.

3. Delete the link using the `IssueLinkManager` class:

```
issueLinkManager.removeIssueLink(issueLink, user);
```

Retrieving Issue Links on an issue

We can retrieve the inward or outward links on an issue or all the linked issues using different methods on the `IssueLinkManager` class.

All inward links can be retrieved as shown:

```
List<IssueLink> links = issueLinkManager.getInwardLinks(issue.getId());
for (IssueLink issueLink : links) {
    System.out.println(issueLink.getIssueLinkType().getName()
    +": Linked from "+issueLink.getSourceObject().getKey());
}
```

Here, `issue` is the destination object and we are getting all the inward issue links and displaying the source issue key.

Similarly, outward links can be retrieved as shown:

```
links = issueLinkManager.getOutwardLinks(issue.getId());
for (IssueLink issueLink : links) {
    System.out.println(issueLink.getIssueLinkType().getName()
    +": Linked to "+issueLink.getDestinationObject().getKey());
}
```

Here, issue is the source object and we are getting all the outward issue links and displaying the destination issue key.

All the linked issues can be retrieved in a single method, as shown:

```
LinkCollection links = this.issueLinkManager
.getLinkCollection(issue, user);
Collection<Issue> linkedIssues = links.getAllIssues();
```

JavaScript tricks on issue fields

JIRA provides a lot of options to manage the various fields on an issue. Field configuration schemes, screen schemes, and so on, help the JIRA admins to show or hide fields, mark them as mandatory, and so on, differently for different issue types and projects.

Irrespective of how configurable these schemes are, there are still areas where we need to perform custom development. For example, if we need to show or hide fields, based on the values of another field, then JIRA doesn't have any in-built options to do so.

Then what is the best way to deal with this? It is always possible to create a new composite custom field that can have multiple fields driven by each other's behavior. But probably an easier way-that doesn't require developing a plugin-is to drive this using JavaScript. And to make things better, JIRA offers jQuery library, that can be used to write neat JavaScript code!

However, using JavaScript to handle field behavior can create problems. It limits the behavior to the browser, it is client-side, and is dependent on whether JavaScript is enabled or not. But given its advantages and ease of use, most users prefer to do it. In this recipe, we will see a small example of using JavaScript to show or hide the values of a custom field based on the issue's priority value!

How to do it…

Let us assume that we have a custom field named **Reason for High Priority**. The field should be shown only if the priority of the issue is **Highest**.

Following are the simple steps to achieve it using JavaScript:

1. Write the JavaScript to achieve the functionality.In our example, we need to show the **Reason for High Priority** field only when the priority is **Highest**. Let us write the JavaScript for these purposes as an example:

   a. Identify the ID value for priority. We can get it by looking at the URL while editing the priority, or from the JIRA database by looking at the `priority` table.

   b. Identify the ID of the custom field. We can get this in a similar fashion, either by looking at the URL while editing the custom field, or from the `customfield` table.

   c. Write the JavaScript to show or hide the field depending on the priority value. Here, we use JIRA's jQuery library, which has a predefined namespace `AJS`, a short name for Atlassian JavaScript!:

```
<script type="text/javascript">
    JIRA.bind(JIRA.Events.NEW_CONTENT_ADDED,
    function (e, context) {
        hideOrShowPriority();
    });
    AJS.$(document).ready(function () {
        hideOrShowPriority();
    });
```

```
    AJS.$('#priority').change(function () {
         hideOrShowPriority();
    });
    function hideOrShowPriority(){
        var priorityVal = AJS.$('#priority').val();
        if (priorityVal == '1'){
            AJS.$("#customfield_10000")
            .closest('div.field-group').show();
        } else {
            AJS.$("#customfield_10000")
            .closest('div.field-group').hide();
        }
    }
</script>
```

Here, 10000 is the `id` of the `customfield` and hence `customfield_10000` represents the unique custom field ID! Also, 2 is the `id` of the `priority` system field.

In the example, we have handled three events:

i. A page load event:

```
AJS.$(document).ready(function () { });
```

This event handles new page loads.

ii. The NEW_CONTENT_ADDED event:

```
JIRA.bind(JIRA.Events.NEW_CONTENT_ADDED,
function (e, context) { });
```

This event handles DOM changes due to other JavaScript changes or by other plugins.

iii. Priority field onChange() event:

```
AJS.$('#priority').change(function () { });
```

This event handles changes to the priority field.

On all three events, we get the priority value, and hide or show the div surrounding the custom field, as shown here:

```
AJS.$("#customfield_10000").closest('div.field-group').hide();
```

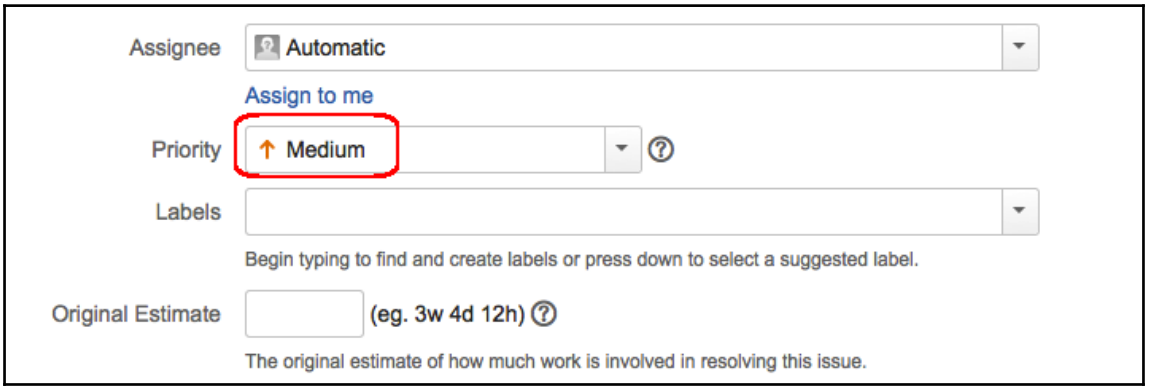
2. Add the preceding JavaScript to the description of the custom field.If the field is used under different field configurations, make sure to edit the description under the appropriate field configuration.

The field behavior will be effective on the next reload after the JavaScript is added on to the field description.

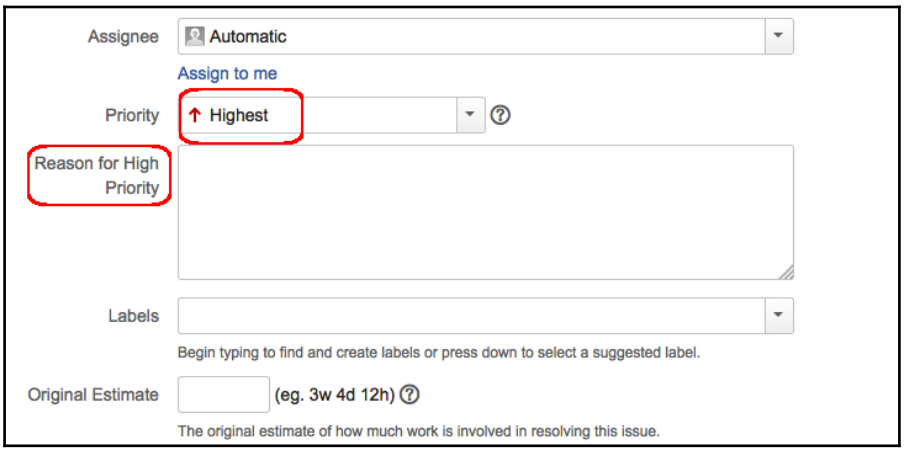
How it works…

Whenever the field is rendered under the velocity view in the edit mode, the field description is executed along with all the JavaScript code in there!

Once the script is added in the relevant field configuration screen, the field will not appear for priority values other than **Highest**, as shown in the next screenshot:



Here, the **Priority** is **Medium** (value 3), and hence the field **Reason for High Priority** is not available. But the moment the priority is changed to **Highest**, we can see the field appearing back on the page:



The JavaScript can now be modified to do a lot of other useful stuff! Don't forget to modify the scripts according to your needs, specifically, your browser and your version of JIRA.

Creating issues and comments from e-mail

It is possible to automatically create issues or comments in JIRA based on incoming e-mail messages. This feature is very useful in scenarios such as helpdesks, where the users normally send an e-mail to a designated e-mail address and the support team picks up the issues from JIRA based on those e-mails!

Once configured correctly, any new e-mail that comes in will create a corresponding issue in JIRA, and the replies to the e-mail notifications on that issue will be created as comments on that issue. It is also possible to attach documents on the issue by attaching them on the e-mail, provided attachments are enabled in JIRA. If external user management is not enabled, it is also possible to create a user account-if they don't already have an account.

In this recipe, we will see how we can configure JIRA to enable this feature.

How to do it…

The following are the steps to enable issue creation from e-mail:

1. Create an e-mail account on the server-typically, one e-mail account for each JIRA project. This mailbox should be accessible via POP, IMAP, or on the local file system. JIRA will periodically scan this mailbox and create issues or comments based on the e-mail.
2. Navigate to JIRA's **Administration** | **System** | **Mail**| **IncomingMail.**
3. Click on **Add POP / IMAP mail server** button.
4. Enter the details for the POP or IMAP mail server created in *Step1* and click on **Add**.
5. Click on **Add incoming mail handler** button:

    a. Name: Name of the mail handler.

    b. Server: Pick one from the servers configured previously, or select the Local Files option for an external mail service that writes messages to the file system.

    c. Delay: Choose a delay for the handler to run and scan the mails.

    d. Handler: Select one of the handlers from the list. Details of available handlers can be found at `https://confluence.atlassian.com/display/JIRA/Creating+Issues+and+Comments+from+Email#CreatingIssuesandCommentsfromEmail-messagehandlers`. Let us pick the **Create a new issue or add a comment to an existing issue** handler.

    e. Folder Name: For IMAP Server, specify the folder name if it is a folder other than **Inbox**. For the Local Files option, specify the subdirectory within the **import/mail** directory in JIRA Home.

6. Click **Next** to add the **handler parameters** specific to the handle selected. This is the most important part, where we specify the parameters that will be used while creating the issue. Following is the list of important parameters for the handler we picked:

    a. `Project`: Project where the issue should be created.

    b. `Issue Type`: Type of the issue to be created. For example, if we want the issue to be created as a bug, select **Bug**.

c. `Strip Quotes`: If enabled, it strips quoted text, for example, text from previous replies, from the e-mail body.

d. `Catch Email Address`: If added, JIRA will process only e-mails sent to this address. It is used when there are multiple aliases for the same e-mail inbox.

e. `Bulk`: Determines how to handle "bulk" e-mails. Possible options are:

i. `ignore`: Ignore the e-mail and do nothing.

ii. `forward`: Forward the e-mail to the address set in the "Forward Email" text field.

iii. `delete`: Delete the e-mail permanently.

iv. `accept`: Accept the email for processing.

f. `Forward Email`: Error notifications and un-handled e-mails (used in conjunction with bulk forward handle parameter) will be forwarded to this address.

g. `Create Users`: If set to true, accounts will be created for new senders. This option is not compatible with the **Default Reporter** option.

h. `Default Reporter`: Can be used to create issue with the specified reporter when the sender does not match with an existing user. This option will not be available if **Create Users** is checked.

i. `Notify Users`: Only used if **Create Users** is checked. Indicates whether users should get a mail notification for the new accounts created.

j. `CC Assignee`: If set, the new issue will be assigned to a matching user in the **To** field, or **Cc** field or **Bcc** field-in the given order-depending on where the user is matched. The user should have **Assignable User** project permission.

k. `CC Watchers`: If set, matching users in **To**, **Cc**, and **Bcc** fields are added as **watchers** on the issue. Even the new users created by the **Create Users** option can be added as a watcher using this option.

7. Finish the handler creation.

JIRA is now configured to receive mails to the newly added mailbox.

How it works…

The handler we set up scans the mailbox every *N* minutes as configured in the *delay* and picks up the new incoming messages. When a new message is received, JIRA scans through the subject to see if there are any mentions of an already existing issue. If there is one, the e-mail is added as a comment on the mentioned issue, with the e-mail body as the comment text. If there is no mention of an issue in the subject, JIRA still checks whether the e-mail is a reply to another e-mail that already created an issue or not. If so, the e-mail body is again added as a comment on that issue. This is done by checking the `in-reply-to` header in the e-mail.

If JIRA still couldn't find any matching issues, a new issue is created in the project and of the type configured in the handle parameters. The e-mail subject will become the issue summary and the e-mail body the description.

Any attachments on an e-mail, new, or replies, will be added as attachments on the issue.

More information about the creation of issues and comments from an e-mail and on how the other handlers work can be found at `http://confluence.atlassian.com/display/JIRA/Creating+Issues+and+Comments+from+Email`.



It is also worth checking the plugin exchange for plugins with extended mail handlers that are capable of adding more details on the issue during creation, like custom field values. Some of them have far better filtering mechanisms as well.

You can also write a custom mail handler, using the `message-handler` plugin module, by following the tutorial at `https://developer.atlassian.com/jiradev/jira-platform/guides/email/tutorial-custom-message-mail-handler-for-jira`.

8
Customizing the UI

In this chapter, we will cover:

- Changing the basic look and feel
- Adding new web sections in the UI
- Adding new web items in the UI
- Use of decorators and other metadata tags
- Adding conditions for web fragments
- Creating new velocity context for web fragments
- Adding a new drop-down menu on the top navigation bar
- Dynamic creation of web items
- Adding new tabs in the View Issue screen
- Adding new tabs in the Browse Project screen
- Adding new links in the project-centric view
- Adding new panels in the project-centric view
- Adding sub-navigation in the project-centric view
- Adding issue link renderers
- Displaying dynamic notifications/warnings on issues
- Re-ordering Issue Operations in the ViewIssue page
- Re-ordering fields in the ViewIssue page

Introduction

One of the many good things about JIRA is that it has a simple but powerful user interface. A lot has changed between the older versions and 7.x in terms of the user interface, and it still continues to be one that keeps the users happy and plugin developers interested.

While the existing JIRA interface works for many people, there are cases where we need to modify bits and pieces of it, add new UI elements, remove some, and so on.

Normally, when we think of modifying a web application's user interface, the first thought that comes to our mind is to go and modify the JSPs, VMs, and many others involved. While that is true, in some cases for JIRA as well, a lot of the user-interface changes can be introduced without even touching the JIRA code. JIRA helps us to do that with the help of a number of UI-related plugin modules.

In this chapter, we will be looking at various recipes for enhancing the JIRA UI with the various plugin modules available, and also, in some cases, by modifying the JSPs or other files involved.

Note that the look and feel can be changed to a big extent only by modifying the CSS files and other templates involved. But here we are talking about adding new web fragments, such as new sections and links, in the various parts of the UI without actually modifying the core JIRA files or with little modification of them.



If we modify the JIRA files, it should be noted that maintaining the changes over various JIRA versions, i.e. during upgrades, would be more difficult than usual and worth considering carefully before proceeding!

Changing the basic look and feel

As mentioned earlier, any big changes to the look and feel of JIRA can be achieved only by modifying the CSS files, JSPs, templates, and other tools involved. But JIRA lets its administrators make slightly simpler changes like changing the logo, color scheme, and so on with simple configurations. In this recipe, we will see some examples on how easy it is to make those changes.

There are mainly six things that can be configured to change JIRA's appearance:

- **Logo**: Understandably, this is one thing everyone wants to change
- **Site Title**: You can choose to show the Site Title alongside the logo
- **Favicon:** Similar to the logo, you can upload a favicon
- **Colors**: JIRA has a nice theme of colors revolving around the color blue. However, we can easily change these colors to suit our taste, or rather the company's taste!



Specify the hexadecimal notations (HEX values) of the colors you are interested in if the color scheme needs to be changed.

- **Gadget colors:** For each gadget in JIRA, we can set a different color chosen from a predefined set of colors. We can easily change the predefined list of colors through simple configuration.



Similar to colors, use the HEX values here as well.

- **Date and Time formats:** The Date and Time formats in JIRA can be modified easily to suit our needs, provided it is a valid format supported by Java's **SimpleDateFormat**).

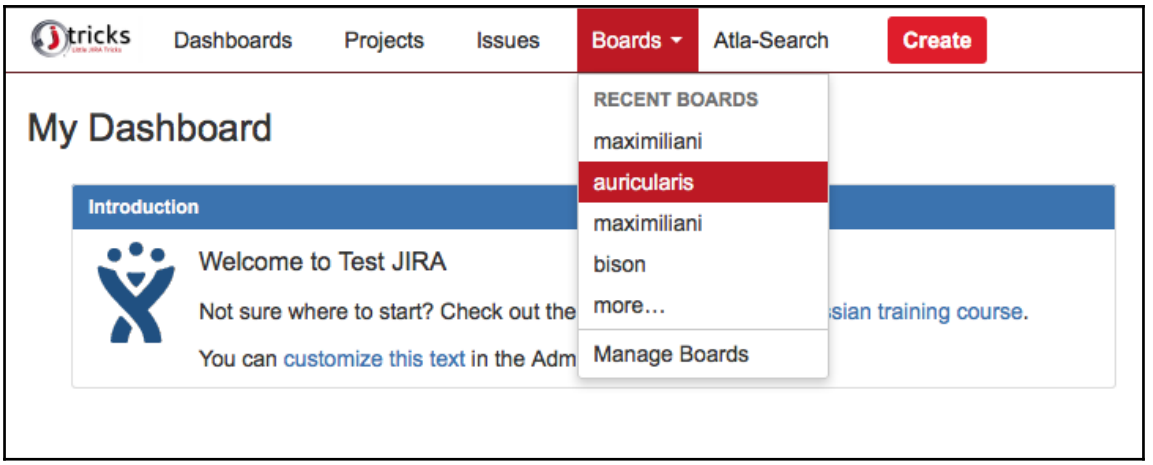
How to do it…

Following are the steps required to make changes to the basic JIRA look and feel.

1. Log in to JIRA as an administrator.
2. Navigate to **Administration** | **System** | **User Interface** | **Look and Feel**.
3. Click on **Edit Configuration**.
4. Make the appropriate changes
5. Click on **Update**.

Repeat the cycle until the desired result is achieved. We can always revert back to the default settings by clicking on **Reset Defaults** while editing the configurations.

With some simple changes, the JIRA UI can look a lot different to how it normally looks. The following screenshot is a small example:



It's a small, yet powerful change!

Adding new web sections in the UI

A web *fragment* is a link or a section of links in a particular location of the JIRA web interface. It can be a menu in JIRA's top navigation bar, a new set of issue operations, or a new section in the**AdminUI** section.

There are two types of plugin modules used to add new web fragments in JIRA, namely, the **WebSection** plugin module and the **WebItem** plugin module. A **WebSection** is a collection of links that is displayed together in a particular location of the JIRA user interface. It may be a group of buttons on the issue operations bar or a set of links separated by lines.

In this recipe, we will see how to add a new web section to JIRA.

How to do it…

Following are the steps required to add a new web section to JIRA:

1. Identify the *location* where the new sections should be added.



JIRA has a lot of identified locations in its user interface and it lets us add new web sections in any of these locations. A complete list of the available locations can be found at `https://developer.atlassian.com/display/JIRADEV/Web+Fragments`.

2. Add the new `web-section` module into the `atlassian-plugin.xml`:

```
<web-section name="J-Tricks Section"
i18n-name-key="j-tricks-section.name" key="j-tricks-section"
location="admin_plugins_menu" weight="1000">
  <description key="j-tricks-section.description">
    The J-Tricks Section Plugin
  </description>
  <label key="j-tricks-section.label"/>
</web-section>
```

As with all other plugin modules, it has a unique module **key**. Here, the two other important attributes of the web-section elements are **location** and . **location**weight. location defines the location in the UI where the section should appear and **weight** defines the order in which it should appear.

In the preceding example, location is `admin_plugins_menu`, which will create a new web section under the **Administration** | **Add-ons** menu, just like the existing section **Atlassian Marketplace**.

The `web-section` module also has a set of child elements. The **condition** or **conditions** element can be used to define conditions, one or more, details of which we will see in the following recipes. The **context-provider** element can be used to add a new context provider, which will then define the velocity context for the web section. **label** is what will be displayed to the user. **param** is another element that can be used to define key/value parameters and is handy if we want to use additional custom values from the UI. The **resource** element can be used to include resource files like JavaScript or CSS files and the **tooltip** element will provide a tooltip for the section.

label is the only mandatory element.

Elements such as **label** and **tooltip** can have optional key value parameters, as shown in the following code:

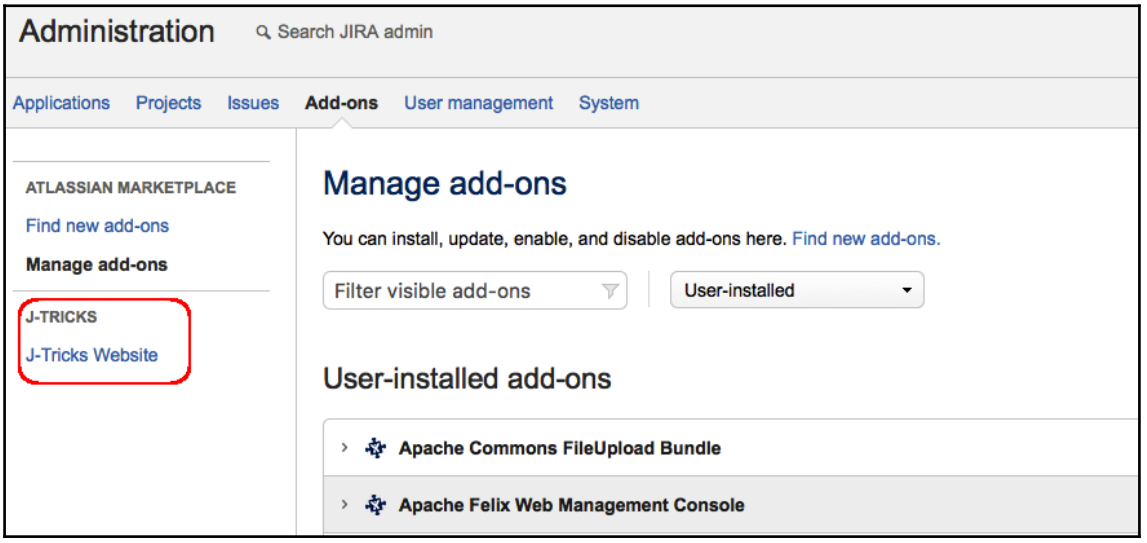
```
<label key="some.valid.key">
  <param name="param0">$somevariable</param>
</label>
```

As you can see in the example, **label** takes a **key/value** parameters where the value is dynamically populated from a velocity variable. The param will be passed to the text as **{0}** and will substitute that position in the label. Here, the parameters allow one to insert values into the label using Java's `MessageFormat` syntax, the details of which can be found at `http://download.oracle.com/javase/7/docs/api/java/text/MessageFormat.html`. Parameter names must start with **param** and will be mapped in alphabetical order to the substitutions in the format string, that is, `param0` is `{0}`, `param1` is `{1}`, `param2` is `{2}`, so on and so forth.

3. Deploy the plugin.

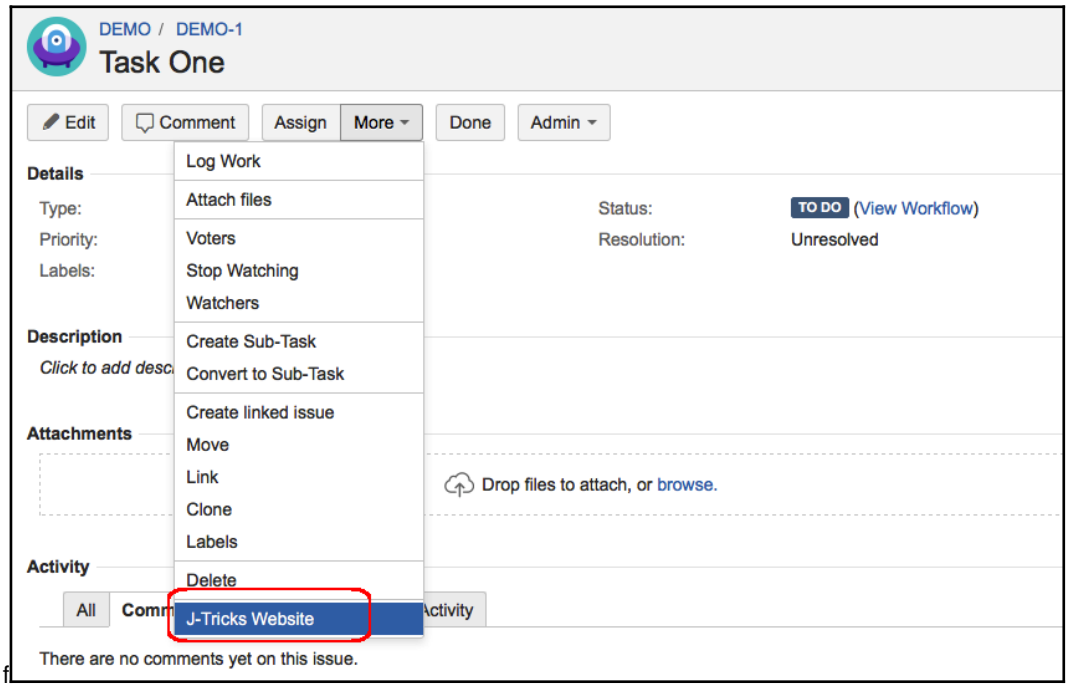
How it works…

Once the plugin is deployed, we can see that a new section is created in the Admin screen of JIRA, as shown in the following screenshot. The web item, which appears in the new section, is explained in detail in the next recipe.



We can add the section to many different places just by changing the **location** attribute. If we change the **location** alone to `opsbar-operations`, the new section will appear on the **View issue** page, as shown in the next screenshot.

The web item's **section** attribute must be changed to match the new location as well, i.e.`j-tricks -section` in our example.



Note that the web section **label** may not always be visible because in some cases the section is used to just group the links. For example, in the case of issue operations, the section is just used to group the links together, as shown in the preceding screenshot.

See also

- The *Adding new web items* in the recipe in this chapter

Adding new web items in the UI

A *webitem* is a new link that can be added to various places in the JIRA UI. A link will typically go under a *websection*. A link can simply point to a URL or can be used to invoke an action. In this recipe, we will see how to add a new web item to JIRA.

How to do it…

Following are the steps required to add a new web item to JIRA:

1. Identify the *web section* where the new link should be added.We have already seen how to create a new **web section**. A link is then added into a section created as above or into a predefined JIRA section. We can add the link directly to a location if it is a **non-sectioned** one. For **sectioned** locations, it is the location key, followed by a slash ('/'), and the key of the web section in which it should appear. For example, if we want to place a link in the web section created before, the **section** element will have the value `admin_plugins_menu/j-tricks-section`.

2. Add the new **web item** module into the `atlassian-plugin.xml`:

```
<web-item name="J-Tricks Link" i18n-name-key="j-tricks-link.name"
key="j-tricks-link" section="admin_plugins_menu/j-tricks-section"
weight="1000">
   <description key="j-tricks-link.description">
   The J-Tricks Link Plugin</description>
   <label key="j-tricks-link.label"></label>
   <link linkId="j-tricks-link-link">
     http://www.j-tricks.com/
   </link>
</web-item>
```

   A web item module also has a unique **key**. The other two important attributes of a web-item are **section** and **weight**. **section** defines the web section where the link is placed, as mentioned above, and **weight** defines the order in which the link will appear.

   A web item also has all the elements of a web section: **condition**/**conditions**, **context-provider**, **description**, **param**, **resource**, and **toolitp**. In addition, a web item also has a **link** element that defines where the web item should link to. The **link** could be an action, a direct link, and so on, and can be created using velocity parameters dynamically, as shown in the examples here:

```
<link linkId="create_link" absolute="false">
   /secure/CreateIssue!default.jspa
</link>
<link linkId="google_link">
   http://www.google.com
</link>
<link linkId="profile_link" absolute="false">
  /secure/ViewProfile.jspa?name=$user.name
</link>
```

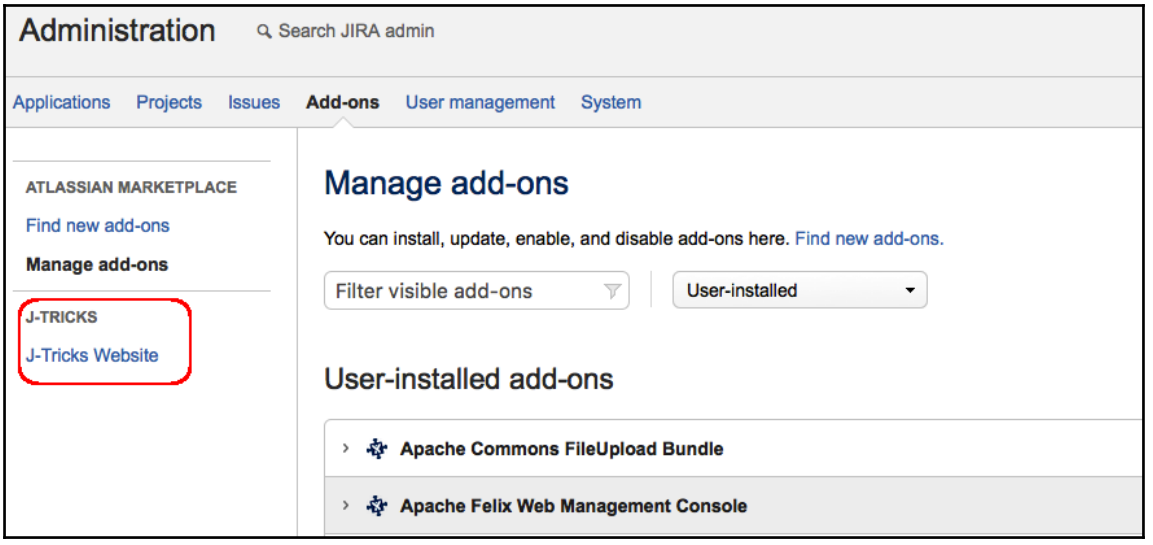
In the third example, `user` is a variable available in the velocity context!
An **icon** element is used when we need to add an icon alongside the link:

```
<icon height="16" width="16">
   <link>/images/avatar.gif</link>
</icon>
```

3. Deploy the plugin.

How it works…

Once the plugin is deployed, we can see that a new web item is shown in the web section we created previously under the **Administration** screen of JIRA.



We can add the item under various different places, just by changing the **section** attribute. We have seen an example while creating a new issue operation in the previous recipe.

See also

- The *Adding new web sections in the UI* recipe in this chapter

Use of decorators and other metadata tags

We have seen web sections and web items. In most cases, these web items point to custom actions created via plugins and will most certainly have views rendered using custom velocity templates or jsps. So, how do we provide a user experience similar to other standard JIRA pages?

Atlassian User Interface, AUI, provides all the necessary libraries needed to build a user experience that is in line with default JIRA pages. You can find more details on AUI at `https://docs.atlassian.com/aui/latest/`. But is that enough?

Everyone knows how the usage of proper decorators can get you the desired look and feel. Have you been paying attention to the JIRA decorators as well?

Getting ready

Let us assume that we are developing a simple **webwork** plugin and have pointed the **web-item** we developed in the previous recipe to the new action. This is how the respective modules look in the `atlassian-plugin.xml`.

```
<web-section name="J-Tricks Section"
i18n-name-key="j-tricks-section.name" key="j-tricks-section"
location="admin_plugins_menu" weight="1000">

  <description key="j-tricks-section.description">
    The J-Tricks Section Plugin
  </description>
  <label key="j-tricks-section.label"/>
</web-section>

<web-item name="J-Tricks Link"
i18n-name-key="j-tricks-link.name" key="j-tricks-link"
section="admin_plugins_menu/j-tricks-section" weight="1000">

  <description key="j-tricks-link.description">
    The J-Tricks Link Plugin
  </description>
  <label key="j-tricks-link.label"/>
  <link linkId="j-tricks-link-link">
    http://localhost:2990/jira/secure/JTricksDemoAction!default.jspa
  </link>
</web-item>

<webwork1 key="j-tricks-demo-action" name="J-Tricks Demo Action"
```

```
i18n-name-key="j-tricks-demo-action.name">
   <description key="j-tricks-demo-action.description">
       The J-Tricks Demo Action Plugin
   </description>
   <actions>
     <action name="com.jtricks.jira.webwork.JTricksDemoAction"
     alias="JTricksDemoAction">
        <view name="input">
          /templates/j-tricks-demo-action/jtricksdemoaction/success.vm
        </view>
     </action>
   </actions>
</webwork1>
```

As you can see, we have a `webwork1` action with an **input** view. We also have a web section, named `j-tricks-section`, under **Administration** | **Add-ons** and a web item under the new section which points to our action. On clicking on the new item, the input view is rendered. All the usual stuff!

Let us also create the `input.vm` file as a standard page using AUI, as shown below:

```
<html>
   <head>
     <title>J-Tricks Demo</title>
   </head>
   <body>
     <div class="aui-page-panel">
       <div class="aui-page-panel-inner">
         <section class="aui-page-panel-content">
         <h2>JTricks Demo</h2>
         <p>How is the look and feel?</p>
        </section>
      </div>
    </div>
  </body>
</html>
```

The new page created will appear as follows:



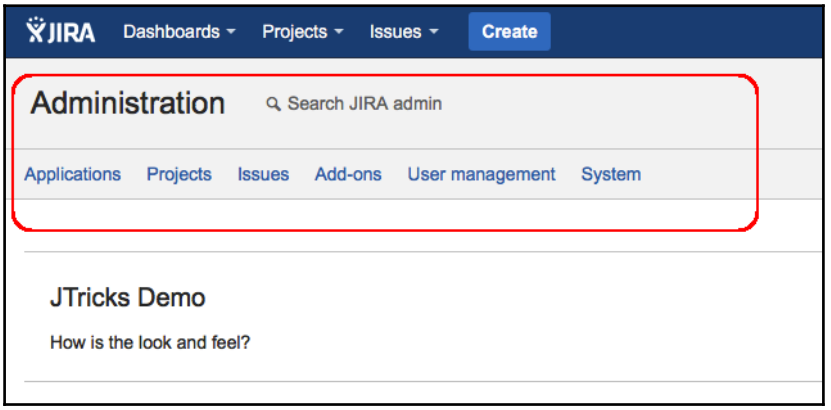
In this recipe, we will see how we can use JIRA decorators to modify the look and feel of this page.

How to do it…

As you probably noticed, the **web section** in our snippet resides under `admin_plugins_menu` and we expect to see the look and feel of an administration screen. But all we see is a plain page, barring the top navigation bar! Where are all the administration navigation items? This is where the decorators help. Let us add the admin content decorator in the **head** of the **html** code, as shown below:

```
<html>
  <head>
   <title>J-Tricks Demo</title>
     <meta name="decorator" content="atl.admin" />
  </head>
   <body>
      ......
   </body>
</html>
```

As soon as the admin decorator is added, the page is rendered as shown below:

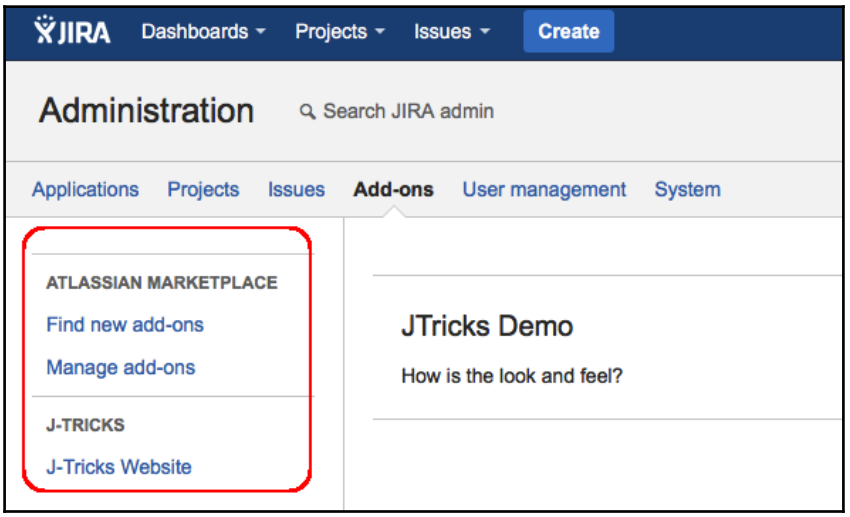


The single line of code, an HTML meta tag that defines the decorator, adds the complete administration navigation, as shown in the above screenshot.

What if you want to show the current active section? This, again, requires adding another meta tag that defines the active admin section.

```
<meta name="admin.active.section"
content="admin_plugins_menu/j-tricks-section" />
```

The `admin.active.section` has the section value, `admin_plugins_menu/j-tricks-section`, under the `content` attribute. The modified page will appear as follows:

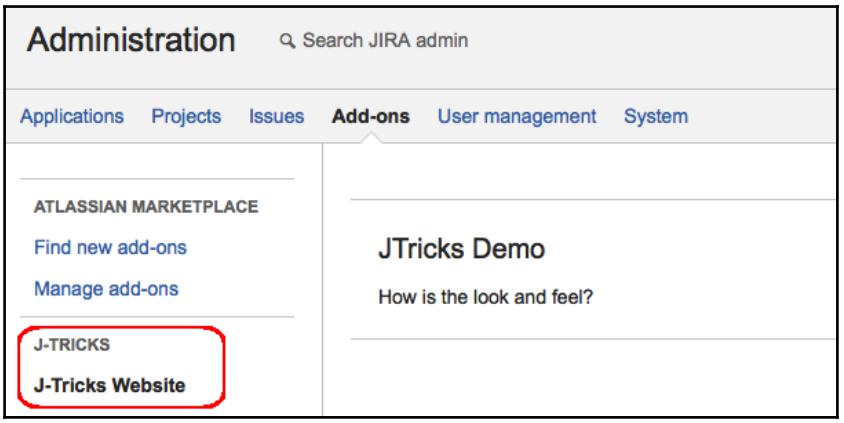


As you can see, the add-ons sections now appear on the left side of the page.

But the page itself is rendered using the **J-Tricks Website** link and it makes sense to highlight it while we are on the page. It is, again, another meta tag that comes to our rescue. The meta tag `admin.active.tab` has the web item's `linkId` in its content attribute, as shown below:

```
<meta name="admin.active.tab" content="j-tricks-link-link">
```

The modified page will appear as follows:



As you can see, the **J-Tricks Website** tab is now highlighted.

Similarly, you can use these tags in anycustom pages, as appropriate, to alter the look and feel of the entire page.

How it works…

JIRA defines the complete list of decorators under `atlassian-jira/WEB-INF/decorators.xml` and it renders the appropriate JSP files at runtime, based on the decorator defined on the page.

You can also check out the appropriate jsp file to find out the list of supported meta tags.

For example, the `atl.admin` decorator points to `/decorators/admin.jsp`, as shown here:

```
<decorator name="atl.admin" page="/decorators/admin.jsp"/>
```

A quick scan in the `/decorators/admin.jsp` file will reveal the use of properties like `meta.admin.active.section`, `meta.admin.active.tab`, `meta.projectKey`, etc. You can use these meta tags in the pages to create the appropriate user experience.

Only four of the decorators are available for v2 plugins, and you can find the list of them, with details, at `https://developer.atlassian.com/docs/common-coding-tasks/using-standard-page-decorators`.

See also

- The *Adding new web sections in the UI* recipe in this chapter
- The *Adding new web items in the UI* recipe in this chapter

Adding conditions for web fragments

As we saw in the previous recipes, adding a web fragment is pretty easy. However, the job doesn't always end there. In many cases, we would want to limit the web item based on a set of conditions.

For example, an **Edit** link on an issue should only appear for people with edit permission on an issue. The **Administration** link should appear only if the user is a JIRA administrator. In this recipe, let us look at how we can implement conditions for displaying web fragments.

How to do it…

It is possible to add one or more conditions to a web section or web item. In the latter case, the `conditions` element is used, which is a collection of `condition/conditions` elements and a `type` attribute. The type attribute is either the logical AND or OR.

For example, the following condition specifies that the user should have either the**admin** permission or use permission in a project before he/she can see the web fragment that has the condition on it.

```
<conditions type="OR">
     <condition class="com.atlassian.jira.plugin.webfragment
     .conditions.JiraGlobalPermissionCondition">
       <param name="permission">admin</param>

     </condition>

     <condition class="com.atlassian.jira.plugin.webfragment
     .conditions.JiraGlobalPermissionCondition">

       <param name="permission">use</param>

    </condition>
</conditions>
```

Possible values of permission, as of 7.0, are `admin`,`use`, `sysadmin`, `project`, `browse`, `create`, `edit`, `update` (same as edit), `scheduleissue`, `assign`, `assignable`, `attach`, `resolv`, `close`, `transition`, `comment`, `delete`, `work`, `worklogdeleteall`, `worklogdeleteown`, `worklogeditall`, `worklogeditown`, `link`, `sharefilters`, `groupsubscriptions`,`move`, `setsecurity`, `pickusers`, `viewversioncontrol`, `modifyreporter`, `viewvotersandwatchers`, `managewatcherlist`,`bulkchange`, `commenteditall`,`commenteditown`, `commentdeleteall`, `commentdeleteown`, `attachdeleteall`, `attachdeleteown`, and `viewworkflowreadonly`. This list can be found from the `com.atlassian.jira.security.Permissions` class.

Let us consider a simple example of how to write a condition and display the web items based on it. In this example, we will display a web item in the top navigation bar if, and only if, the user has logged in and belongs to the `jira-developers` group. The following are the steps:

1. Write the condition class. The class should extend the `AbstractWebCondition` class and override the following abstract method.

```
public abstract booleanshouldDisplay(ApplicationUser user,
JiraHelperjiraHelper);
```

2. In our example, all we need to check is that the user is not null and is a member of the group `jira-developers`. The class is implemented as follows:

```
public class DeveloperCondition extends AbstractWebCondition{
    @Override
    public booleanshouldDisplay(ApplicationUser user,
    JiraHelper helper) {
       return user != null && ComponentAccessor.getGroupManager()
        .getGroupNamesForUser(user).contains("jira-developers");
      }
}
```

3. Add the new condition class in the `web-item`:

```
<web-item key="jtricks-condition-menu"
 name="JTricks Condition Menu"
 section="system.top.navigation.bar" weight="160">
  <description>J Tricks Web site with condition</description>
  <label>JTricks Conditional Menu</label>
  <tooltip>J Tricks Web site</tooltip>
  <link linkId="jtricks-condition-menu">
     http://www.j-tricks.com
  </link>
  <condition class="com.jtricks.ui.conditions.DeveloperCondition"/>
```

```
</web-item>
```

As you can see, the section here is `system.top.navigation.bar`, which will place the new link on the Top Navigation bar. The link will only be visible if the condition `DeveloperCondition` returns true.
We can easily invert a condition using the invert flag as follows:

```
<condition class=
 "com.jtricks.ui.conditions.DeveloperCondition"
  invert="true"/>
```

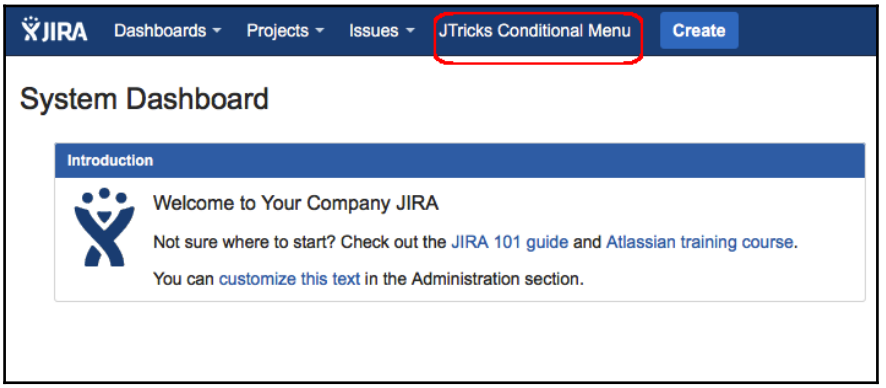
This will display the link if the user is not logged in or not in the group of JIRA developers!

4. Deploy the plugin.

How it works…

Once the plugin is deployed, we can see that the new **JTricks Conditional Menu** is rendered in the top navigation bar only when the user is logged in and in the group of `jira-developers`.

The following screenshot shows the dashboard of a user who is logged in and in the group of`jira-developers`:



If the user is not logged in, or if the user is not in the `jira-developers` group, the condition is not satisfied and the above menu is not shown.

Creating new velocity context for web fragments

As we have mentioned in the previous recipes, it is possible to add velocity variables while constructing a JIRA web fragment. JIRA supports a list of variables by default, which include `user`, `req`, `baseurl`, and so on. The full list and the details of these variables can be found at `https://developer.atlassian.com/display/JIRADEV/Velocity+Contexts`.

In this recipe, we will see how to add more variables to the velocity context with the use of the `context-provider` element.

How to do it…

The `context-provider` element adds to the velocity context available to the web section and web item modules. Only one `context-provider` can be added for an item. The following steps show how we can make use of a context provider:

1. Create the new `ContextProvider`class.
   The class must implement `com.atlassian.plugin.web.ContextProvider`. To make things easy, it is enough to extend the `AbstractJiraContextProvider` class and override the following abstract method in it:

   ```
   public abstract Map getContextMap(ApplicationUser user,
   JiraHelperjiraHelper);
   ```

   The following is what the class looks like if you want to add the full name of the user as a separate variable in the velocity context.

   ```
   public class UserContextProvider
   extends AbstractJiraContextProvider {
     @Override
     public Map getContextMap(ApplicationUser user,
     JiraHelper helper) {
        return MapBuilder.build("userName",
        user.getDisplayName());
     }
   }
   ```



Please note that the `$user` variable is already available in the velocity context of web fragments and so the full name can be retrieved easily using `$user.getDisplayName()`. This is just a simple example of how to use the context providers.

2. Use the variable that is added into the velocity context appropriately while constructing the web section/item.
   In the example, let us create a new web section with the user's full name in the admin section with a single web item in it to link to the user's website.

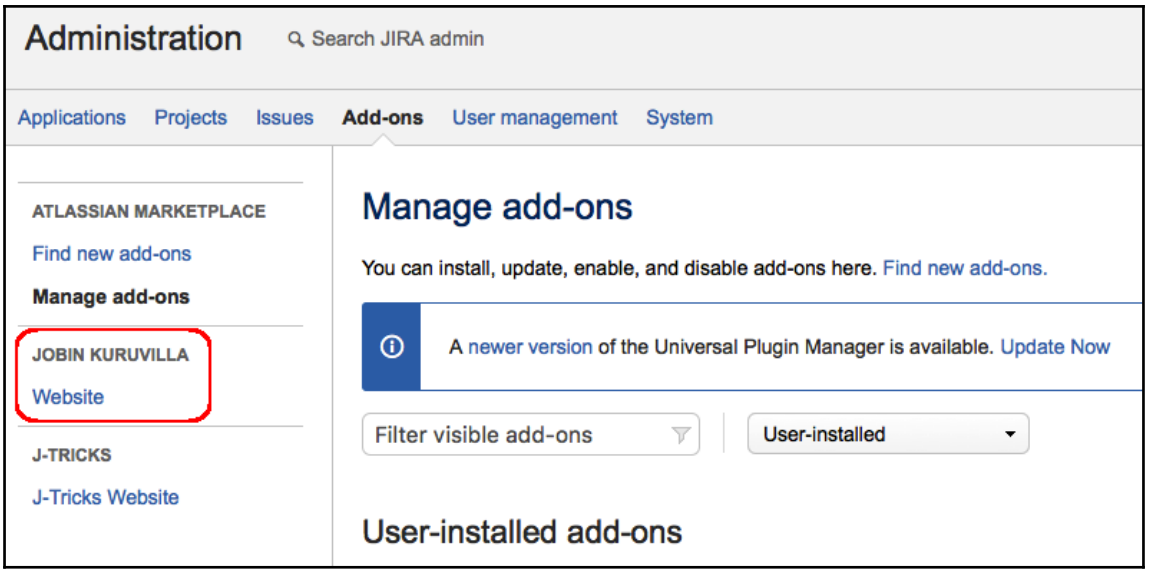
```
<web-section key="jtricks-admin-context-section"
 name="JTricks Context Section" location="admin_plugins_menu"
 weight="910">
   <label>$userName</label>
   <context-provider
   class="com.jtricks.ui.context.UserContextProvider" />
</web-section>

<web-item key="jtricks-admin-context-link"
name="JTricks Context Link"
section="admin_plugins_menu/jtricks-admin-context-section"
weight="10">
   <label>Website</label>
   <link linkId="jtricks.admin.context.link">
      http://www.j-tricks.com
   </link>
</web-item>
```

   As you can see, the web section refers to `$userName` in its label.

3. Deploy the plugin.

How it works…

Once the plugin is deployed, we can see that the new web section is created under the JIRA Admin UI, as shown in the following screenshot. The `$userName` variable is dynamically replaced by the current user's full name.



Adding a new drop-down menu on the top navigation bar

In this recipe, we will show how to use the web section and web item modules quickly to add a new drop-down menu in JIRA's top navigation bar.

How to do it…

Here, we first need a *webitem* to be placed in the system's top navigation bar and then have a *websection* declared under it. The web section can then have a list of web items created under it that will then form the links on the drop-down menu.

Following are the steps to do it:

1. Create a new web item under the system's top navigation bar:

```
<web-item key="jtricks-menu" name="JTricks Menu"
section="system.top.navigation.bar" weight="150">
   <description>J Tricks Web site</description>
   <label>J Tricks</label>
   <tooltip>J Tricks Web site</tooltip>
```

```
    <link linkId="jtricks-menu">http://www.j-tricks.com</link>
</web-item>
```

As you can see, the section is `system.top.navigation.bar`. It can have a link that is pointed to somewhere, in this case, the **J-Tricks website**.



Here, an important thing to notice is that the `linkId` should be same as the key. In this case, both are `jtricks-menu`.

2. Define a web section located under the above web item:

```
<web-section key="jtricks-section" name="JTricks Dropdown"
location="jtricks-menu" weight="200"></web-section>
```



Make sure the `location` is pointing to the `key` of the first web item, which is also its `linkId`.

3. Now add the various web-items under the above web section:

```
<web-item key="jtricks-item" name="Jtricks Item"
section="jtricks-menu/jtricks-section" weight="210">
   <description>J Tricks Tutorials</description>
   <label>J Tricks Tutorials</label>
   <tooltip>Tutorials from J Tricks</tooltip>
   <link linkId="jtricks.link">
      http://www.j-tricks.com/tutorials
   </link>
</web-item>
```

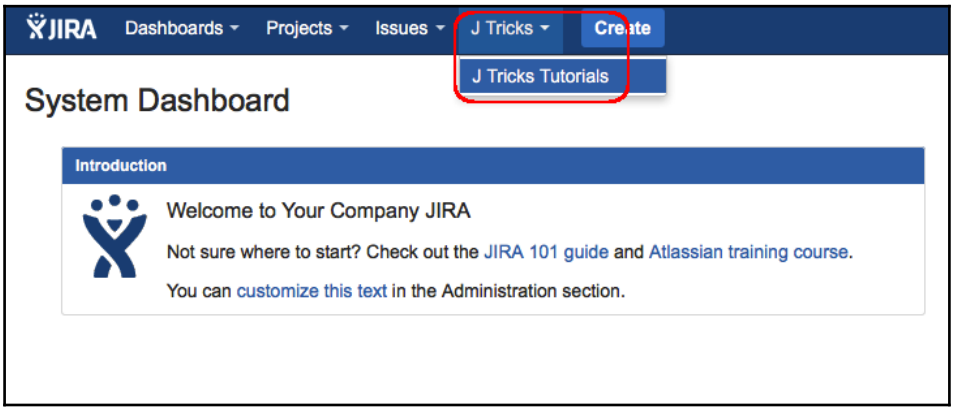


Note that the section is pointed to `jtricks-menu/jtricks-section`, which is similar to a localized section. Here, `jtricks-menu` is the key for the first web item and `jtricks-section` is the key for the previous web section.

4. Deploy the plugin.

How it works…

Once the plugin is deployed, we can see that the new web fragments are created in the top navigation banner. We have a web item, **JTricks**, and a list of links grouped under it, in a section, as shown in the following screenshot:



Dynamic creation of web items

We have now seen quite a few recipes on how to create web items and place them at different places in the UI. But in each case, we knew what links were needed. How about creating these links on the fly?

In this recipe, we will see how to create web items dynamically.

Getting ready

Create a new web item, **Favourites**, in the system top navigation bar, as discussed in the previous recipe.

How to do it…

Let us assume we want to create some links in the system top navigation bar. We have seen the same thing in the previous recipe, but that works only when we know the links in advance. Let us consider a new scenario where the user sees different sets of links when he/she is logged in and not logged in! We can use conditions to check user status, but let us use dynamic link creation for the sake of this recipe.

The following is a step-by-step process to do the same:

1. Create the **Favourites** web section in the system top navigation bar.

```
<web-item key="favourites-menu" name="Favourites Menu"
section="system.top.navigation.bar" weight="900">
   <description>Favourites Menu</description>
   <label>Favourites</label>
   <tooltip>My Favourite Links</tooltip>
   <link linkId="favourites-menu">
     http://www.j-tricks.com
   </link>
</web-item>

<web-section key="favourites-section" name="Favourites Dropdown"
location="favourites-menu" weight="200">
</web-section>
```

   Here, we did exactly what we saw in the previous recipe. A web-item is created in the top navigation bar, under which a web section is created.

2. Define a `web-item-provider` in the `atlassian-plugin.xml`. A web item provider implements the `WebItemProvider` interface and it lets us create a set of links dynamically. The `web-item-provider` is defined as follows:

```
<web-item-provider key="favourites-factory"
name="Favourites Link Factory"
section="favourites-menu/favourites-section"
class="com.jtricks.ui.links.FavouritesLinkProvider" />
```

   As you can see, there is a unique `key` and a human readable `name` associated with this module. Then, there is the `section` attribute, which points to the section defined in step 1, and the `class` attribute, which points to the `WebItemProvider`.

3. Create the web item provider class. The class should implement `WebItemProvider`, as shown here:

```
public class FavouritesLinkProvider
implements WebItemProvider {
    @Override
    public Iterable<WebItem>getItems(Map<String,
    Object> context) {
         .....
    }
}
```

4. Implement the `getItems` method. In this method, we will retrieve the current user and return a list of links, depending on whether the user is null or not:

```
@Override
public Iterable<WebItem>getItems(Map<String, Object> context) {
   final ApplicationUser user =
   (ApplicationUser) context.get("user");
   final List<WebItem> links = Lists.newArrayList();
   if (user != null) {
     links.add(new WebFragmentBuilder(10).id("issue_lnk_id1")
     .label("Favourites 1").title("My Favourite One")
     .webItem("favourites-menu/favourites-section")
     .url("http://www.google.com").build());

     links.add(new WebFragmentBuilder(20).id("issue_lnk_id2")
     .label("Favourites 2").title("My Favourite Two")
     .webItem("favourites-menu/favourites-section")
     .url("http://www.j-tricks.com").build());
   } else {
     links.add(new WebFragmentBuilder(10).id("issue_lnk_id1")
     .label("Favourite Link").title("My Default Favourite")
     .webItem("favourites-menu/favourites-section")
     .url("http://www.google.com").build());
   }
   return links;
}
```

Here, each `WebItem` is a web fragment defined using the following:

a. id: A unique ID of the web-item

b. label: Label of the web item

c. title: A tooltip for the link

d. webitem: The web section key, under which the web item should be placed

e. url: URL of the web item

5. Package the plugin and deploy it.

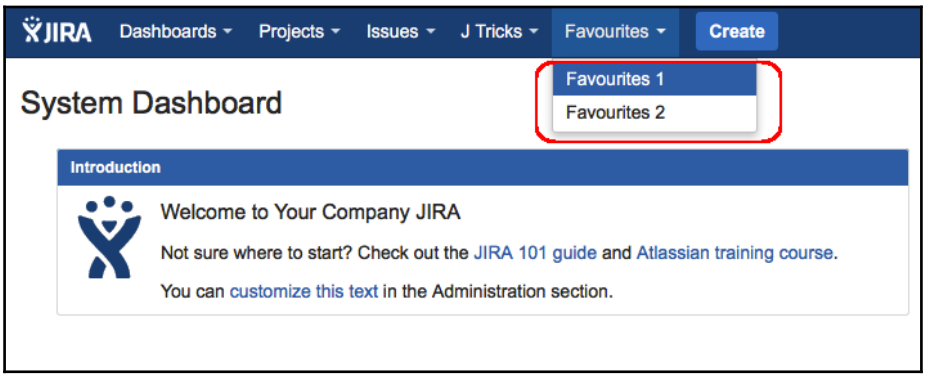
It is also possible to dynamically create web sections using the same technique. We just need to use the `webSection` function, instead of `webItem`, while building the `WebFragmentBuilder`.

How it works…

Once the plugin is deployed, we can see that the new web fragments are created in the top navigation banner. If the user is not logged in, the **Favourites** menu is shown with the default link, as shown in the following screenshot:



Once the user is logged in, he/she will see a different set of links, as per the `getItems` method.



The same approach can be used to create dynamic links/sections based on a different criterion and, of course, at various different places in the UI.

Adding new tabs in the View Issue screen

We have seen how to enhance the UI by adding new sections and links to various locations. In this recipe, we will see how to add a new tab panel under the view issue page, similar to the existing panels, such as Comments, Change History, and so on.

Getting ready

Create a new skeleton plugin using the Atlassian Plugin SDK.

How to do it…

Adding a new tab panel to the **ViewIssue** page can be done by the **IssueTab PanelPluginModule**. Following are the steps required to create a new issue tab panel that displays some static text with a greeting to the logged-in user.

1. Define the**Issue Tab Panel** in the `atlassian-plugin.xml`:

```
<issue-tabpanel key="jtricks-issue-tabpanel"
i18n-name-key="issuetabpanel.jtricks.name" name="Issue Tab Panel"
class="com.jtricks.ui.tabs.JTricksIssueTabPanel">
```

```
    <description>A sample Issue Tab Panel</description>
    <label>JTricks Panel</label>
    <resource type="velocity"
    name="view" location="templates/issue/issue-panel.vm" />
    <order>100</order>
    <sortable>true</sortable>
  </issue-tabpanel>
```

Here, the plugin module has a unique key and should define the class that implements the tab panel. It also has a list of elements, explained as follows:

a. description: A description of the tab panel

b. label: A human-readable label for the panel

c. resource: Defines the velocity template that renders the tab panel view

d. order: Defines the order in which the panels will appear on the view issue page

e. sortable: Defines whether the contents of the panel are sortable or not; for example, sorting comments or the change history elements

2. Implement the Issue Tab Panel class. The class should extend the `AbstractIssueTabPanel` class, which in turn implements the `IssueTabPanel` interface. We need to implement the `showPanel` and `getActions` methods:

   a. Implement the `showPanel` method to return **true** if the panel can be displayed to the user. This method can have complex logic to check whether the user can see the tab or not, but in the example we have, we just return true:

   ```
   public booleanshowPanel(Issue issue, ApplicationUserremoteUser) {
           return true;
   }
   ```

   b. Implement the `IssueAction` classes that need to be returned in the `getActions` method. It is in the **Action** classes that we populate the velocity context to render the view and also return the time performed to facilitate sorting – if *sortable = true*. Sorting is done based on time returned in the `getTimePerformed()` method.
   In the example, let us create a single Action class as follows:

   ```
   public class JTricksAction extends AbstractIssueAction {
         public JTricksAction(IssueTabPanelModuleDescriptor
         descriptor) {
   ```

```
            super(descriptor);
        }
        @Override
        public Date getTimePerformed() {
            return new Date();
        }
        @Override
        protected void populateVelocityParams(Map params) {
             params.put("user",
             ComponentAccessor.getJiraAuthenticationContext()
            .getLoggedInUser().getDisplayName());
        }
}
```

As you can see, the action class must extend the `AbstractIssueAction` class, which in turn implements the `IssueAction` interface.

In the `getTimePerformed` method, it just returns the current date. `populateVelocityParams` is the important method where the velocity context is populated. In our example, we just include the current user's full name with *key* as **user**.

c. Implement the `getActions` method in the Tab Panel class to return a list of issue actions. In our example, we just return a list that contains the new `JTricksAction`:

```
public List getActions(Issue issue,
ApplicationUserremoteUser) {
     List<JTricksAction>panelActions =
     new ArrayList<JTricksAction>();

     panelActions.add(new JTricksAction(descriptor));
     return panelActions;
}
```

Here, the **descriptor** is an instance variable of the super class. All we do here is create an instance of the `IssueAction` class and return a list of such actions.

3. Create the `view` template in the location specified earlier:

```
Hey $user, sample Issue Tab Panel!
```

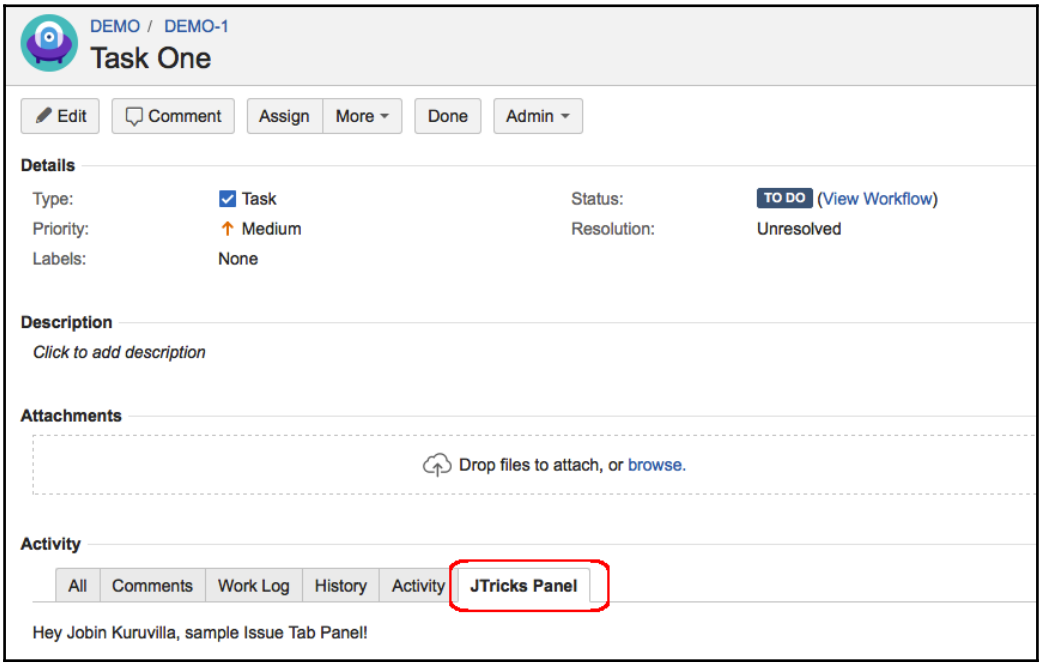
This is all we need for the **user** here is populated into the context by the Action class.

4. Package the plugin and deploy it.

How it works…

Once the plugin is deployed, a new tab panel will appear in the **ViewIssue** page, as shown in the following screenshot.

As you can see, the greeting message there is populated using the velocity context and the attributes in it.



There's more…

From JIRA 5.0, it is possible to load the issue tab panels asynchronously. This improves the user experience if a custom tab takes more time to load.

Loading issue tab panel asynchronously

All we need to do is to add the `<supports-ajax-load>` configuration element in the tab panel definition in `atlassian-plugin.xml`. The updated definition will be as follows:

```
<issue-tabpanel key="jtricks-issue-tabpanel"
i18n-name-key="issuetabpanel.jtricks.name" name="Issue Tab Panel"
class="com.jtricks.ui.tabs.JTricksIssueTabPanel">
      <description>A sample Issue Tab Panel</description>
      <label>JTricks Panel</label>
      <resource type="velocity" name="view"
      location="templates/issue/issue-panel.vm" />

      <order>100</order>
      <sortable>true</sortable>
      <supports-ajax-load>true</supports-ajax-load>
</issue-tabpanel>
```

This will load the panel asynchronously. However, if you have any web resources used in this panel, you should be careful because the resources might not be available in the **View Issue** page by the time the tab panel has loaded.

In such a scenario, use the `jira.view.issue` context in the `web-resource` definition. When a web resource module is defined with the context, all the resources defined under that web resource module will be available in all pages using that context.

Following is an example:

```
<web-resource key="jtricks-resource" name="JTricks tab java script">
     <context>jira.view.issue</context>
     <dependency>jira.webresources:viewissue</dependency>
     <resource type="download" name="myJS.js" location="script/myJS.js"/>
</web-resource>
```

Here, the `myJS.js` file will always be available in the **View Issue** page.

Adding new tabs in the Browse Project screen

In this recipe, we will see how to add a new tab in the **BrowseProject** screen using the `project-tabpanel` module.



The use of this module is deprecated in JIRA 7 but we still have it in the book for the sake of plugins in older versions or to help plugin migrations to JIRA7.

Getting ready

Create a new skeleton plugin using Atlassian Plugin SDK.

How to do it…

Following are the steps required to create a new project tab panel:

1. Define the **Project Tab Panel** in the `atlassian-plugin.xml`:

```
<project-tabpanel key="jtricks-project-panel"
i18n-name-key="projectpanels.jtricks.name"
name="Project Tab Panel"
class="com.jtricks.ui.tabs.JTricksProjectTabPanel">
    <description>A sample Project Tab Panel</description>
    <label>JTricks Panel</label>
    <order>900</order>
    <resource type="velocity"
    name="view" location="templates/project/project-panel.vm" />
</project-tabpanel>
```

   Here, the plugin module has a unique `key` and should define the class that implements the tab panel. It also has a list of elements, explained as follows:

   a. `description`: A description of the tab panel

   b. `label`: A human-readable label for the panel

   c. `resource`: Defines the velocity template that renders the tab panel view

   d. `order`: Defines the order in which the panels will appear on the browse project screen

2. Implement the **Project Tab Panel** class.The class should extend the `AbstractProjectTabPanel` class, which in turn implements the `ProjectTabPanel` interface. We need to implement only the `showPanel` method.
   The `showPanel` method should return **true** if the panel can be displayed to the user. This method can have complex logic to check whether the user can see the tab or not, but in the example we have, we just return `true`:

```
public booleanshowPanel(BrowseContextctx) {
    return true;
}
```

3. Use the `createVelocityParams` method to add more variables to the velocity context, if needed. These variables can be used by the velocity templates used in the plugin module definition.For example, the current user's **name** can be added to the context, as shown here:

```
@Override
protected Map<String,
Object>createVelocityParams(BrowseContextctx) {
      Map<String, Object>params =
      super.createVelocityParams(ctx);

     params.put("user",
     ctx.getUser() != null ?
     ctx.getUser().getDisplayName() : "Anonymous");

    return params;
}
```

4. Create the `view` template in the location specified earlier. The template we defined is as follows:

```
Hey <b>$!user</b>, This is a sample Project Tab Panel from
<a href="http://www.j-tricks.com">J Tricks</a>
```

   As you can see, the template uses the `user` variable populated in step 3.

5. Package the plugin and deploy it.

How it works…

Once the plugin is deployed, a new tab panel will appear in the **BrowseProject** page, as shown in the following screenshot:



Unlike the previous versions, the new panel appears under the **Add-ons** section, as the `project-tabpanel` modules are now deprecated.

We can add links and panels directly in the **Browse Project** section using the project-centric navigation; you will learn that in the next few recipes.

Adding new links in the Project-centric view

As you saw in the previous recipe, the `project-tabpanel` module is deprecated and the panels created using the same are now grouped under the **Add-ons** section.

In JIRA7 and higher, items can be directly added into the project-centric view using the `web-item` module. We have already seen the web item module earlier in this chapter. We just need to create a web item with the appropriate **section** in the plugin module definition.

In this recipe, we will see how to add a simple link to the project-centric navigation.

Getting ready

Create a new skeleton plugin using Atlassian Plugin SDK.

How to do it…

As mentioned earlier, we just need to define a `web-item` module with the appropriate section and link defined in it.

For the project-centric navigation, the section is `jira.project.sidebar.plugins.navigation`.

The web-item module is defined as follows:

```
<web-item key="jtricks-projectcentric-link"
i18n-name-key="projectcentriclink.jtricks.name"
section="jira.project.sidebar.plugins.navigation" weight="300">

    <label>JTricks Website</label>
    <link>http://www.j-tricks.com</link>
    <param name="iconClass"
    value="aui-icon aui-icon-small aui-iconfont-link"/>
</web-item>
```



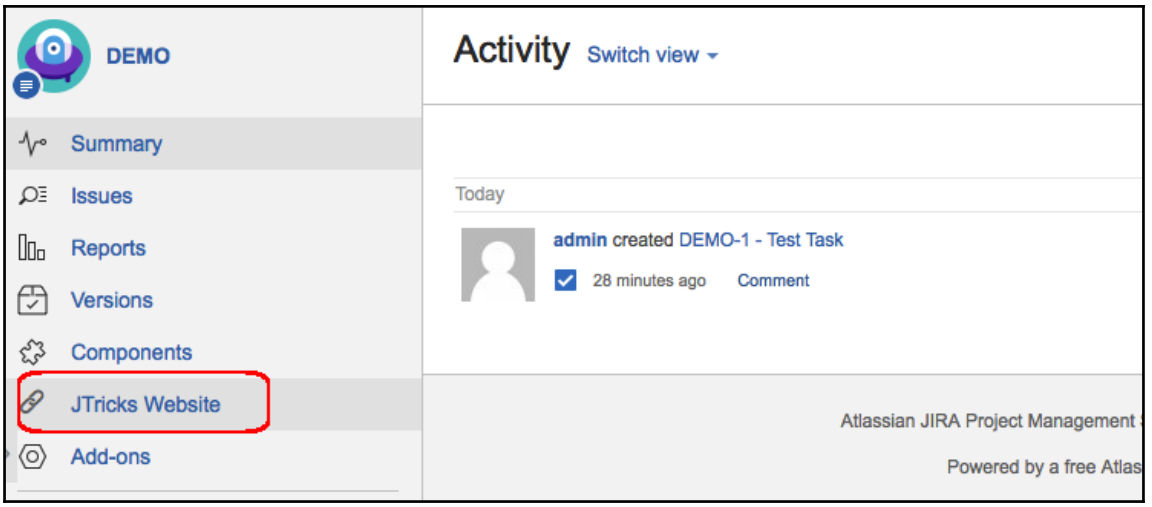
Please note the `iconClass` param, as this gives us a way to alter the link visibility using different CSS classes.

In the above example, we have used the CSS classes, exposed by **AUI** (Atlassian User Interface), and took advantage of the default icons that come with JIRA. We use the link icon, and the full list of icons, and the respective icon classes, can be found at `https://docs.atlassian.com/aui/latest/docs/icons.html`.

We can, of course, use our own CSS classes, defined in the plugin web resources, to alter the visibility as we like.

How it works…

Once the plugin is deployed, the new link will appear in the **BrowseProject** page, as shown in the following screenshot:



As you can see, the new link uses the link icon from AUI.

See also

- The *Adding new web items in the UI* recipe in this chapter

Adding new panels in the project-centric view

In the previous recipe, we saw how to add simple links to the project-centric navigation. This is good enough for some cases, but what if we need to show a web panel with proper logic?

This is where we can add a web panel and link to the web panel from a web item, added in the project-centric navigation.

Getting ready

Create a new skeleton plugin using **AtlassianPluginSDK**.

How to do it…

Following are the steps required to add a new web panel in the project-centric view.

1. Define a web panel using the web-panel plugin module, as shown here:

```
<web-panel key="jtricks-projectcentric-panel-key"
location="com.atlassian.jira.jira-projects-plugin:jtrickspanel">
    <resource type="velocity" name="view"
    location="templates/project/project-panel.vm" />
    <context-provider
    class="com.jtricks.ui.tabs.context
    .ProjectPanelContextProvider"/>
</web-panel>
```

   Similar to the web-item module, we can add conditions, resources, and context providers on the web-panel module. More details on the web-panel module can be found at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/web-panel-plugin-module`.

   As you can see, we have defined a `view` template and a `context-provider` in the above example. The template will be used to render the **view** and context provider will be used to populate variables into the velocity context.



   Make a note of the `location` element. This should be a unique value and we will use it later in this recipe.

2. Create the context provider class. This class will implement the `ContextProvider` interface, as shown here:

```
public class ProjectPanelContextProvider implements
ContextProvider{
    @Override
    public void init(Map<String, String>params) throws
    PluginParseException {
      //Nothing here but we can initiate things here, if needed!
    }
    @Override
    public Map<String,
    Object>getContextMap(Map<String, Object> context) {
        ApplicationUser user =
        ComponentAccessor.getJiraAuthenticationContext()
```

```
            .getLoggedInUser();

            context.put("user", user != null ?
            user.getDisplayName() : "Anonymous");

            return context;
        }
    }
```

As you can see, we need to implement the `init` and `getContextMap` methods. We can initiate the variables in the `init` method and add variables to the context using the `getContextMap` method.

In our example, we have added the logged-in user's name as the `user` variable.

3. Create the `view` template. In our example, let us create a simple view using the `user` variable populated to the context in the earlier step:

```
Hey <b>$!user</b>, This is a sample Project Tab Panel from
<a href="http://www.j-tricks.com">J Tricks</a>
```

4. Now that the web panel is created, we need to define the `web-item` module in the project-centric navigation that points to the new web panel:

```
<web-item key="jtricks-projectcentric-panel"
i18n-name-key="projectcentricpanel.jtricks.name"
section="jira.project.sidebar.plugins.navigation" weight="310">
    <label>JTricks Web Panel</label>

    <link>/projects/$pathEncodedProjectKey?selectedItem=
    com.atlassian.jira.jira-projects-plugin:jtrickspanel</link>

    <param name="iconClass"
    value="aui-icon aui-icon-small aui-iconfont-file-code"/>
</web-item>
```

As you can see, the section points to `jira.project.sidebar.plugins.navigation`, as we saw in last recipe. We are also using yet another CSS class from AUI library to render the link using a file icon.

But the most important change here is in the `link` element. The `link` points to the project page with the selected item pointing to the location defined in the web-panel module, as defined in step 1. The link is constructed as `/projects/pojectKey/selectedItem=webPanelLocation`.

The `projectKey` is dynamically constructed in the link as `$pathEncodedProjectKey`. `webPanelLocation` is `com.atlassian.jira.jira-projects-plugin:jtrickspanel`, as defined in step 1.

5. Package the plugin and deploy it.

How it works…

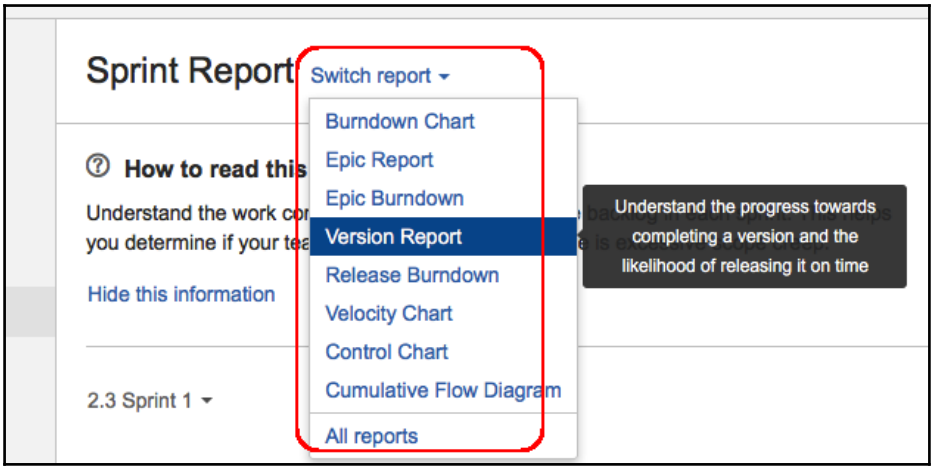
Once the plugin is deployed, the new webitem link will appear in the project-centric navigation. On clicking on the link, the view is rendered using the web panel, as shown in the below screenshot:



Adding sub-navigation in Project-centric view

In the earlier recipe, we saw how new web panels are added to the project-centric navigation. The web panels can be completely designed by the plugin developer and the developer can add extra navigation in those web panels, as he/she likes.

Having said that, JIRA also gives us an easy way to add**sub-navigation** into these web panels. For example, you can see sub-navigation in the **Reports** section in the project-centric view, as shown below:



How can we do this easily in a plugin? In this recipe, we will work on adding sub-navigation to a web panel under the project-centric view.

Getting ready

Create a web panel and link to it as we saw in the earlier recipe. Here is the definition in the `atlassian-plugin.xml` file:

```
<web-item key="jtricks-projectcentric-panel-navigation"
i18n-name-key="projectcentricpanelnavigation.jtricks.name"
section="jira.project.sidebar.plugins.navigation" weight="320">
    <label>JTricks Web Panel Navigation</label>
    <link>
        /projects/$pathEncodedProjectKey?selectedItem=
        com.atlassian.jira.jira-projects-plugin
```

```
            :jtricksnavigationpanel
        </link>
        <param name="iconClass"
        value="aui-icon aui-icon-small aui-iconfont-file-pdf"/>
    </web-item>
    <web-panel key="jtricks-projectcentric-panel-keyall"
    location="com.atlassian.jira.jira-projects-
    plugin:jtricksnavigationpanel">
       <resource type="velocity"
        name="view" location="templates/project
        /project-navigation-panel.vm" />
    </web-panel>
```

As you can see, there is a `web-panel` definition, without any context providers to keep it simple, and there is a `web-item` definition that points to it.

How to do it…

Following are the steps required to add sub-navigation in the web panel defined above.

1. Define a web resource in the `atlassian-plugin.xml`, with a JavaScript file defined as resource in it. If you have used the **Atlassian Plugin SDK** to create the skeleton plugin, you should already have a **web resource** definition and an empty JavaScript file defined in it. If so, you can reuse that web resource and JavaScript file. Here is a sample one:

```
<web-resource key="ui-plugin-resources"
name="ui-plugin Web Resources">
    <dependency>com.atlassian.auiplugin:ajs</dependency>
       ............
    <resource type="download" name="ui-plugin.js"
    location="/js/ui-plugin.js" /> <context>ui-plugin</context>
</web-resource>
```

2. Add a dependency to `com.atlassian.jira.jira-projects-plugin:subnavigator`, which contains the essential JavaScript definitions for implementing **sub-navigation**.
   The updated`web-resource` will be as follows:

```
<web-resource key="ui-plugin-resources"
name="ui-plugin Web Resources">
    <dependency>com.atlassian.auiplugin:ajs</dependency>
    <dependency>
        com.atlassian.jira.jira-projects-
```

```
            plugin:subnavigator
        </dependency>
        ............
        <resource type="download" name="ui-plugin.js"
        location="/js/ui-plugin.js" />
        <context>ui-plugin</context>

    </web-resource>
```

3. Make sure the plugin defines the appropriate context for the `web-resource`. We can import the already defined context, `ui-plugin`, into the velocity templates or add the `atl.general` context, which will make the resources available on all the pages:

```
<web-resource key="ui-plugin-resources"
name="ui-plugin Web Resources">
     ............
     <context>atl.general</context>
</web-resource>
```



More about web resource contexts can be read at `https://developer.atlassian.com/jiradev/jira-platform/building-jira-add-ons/jira-plugins2-overview/jira-plugin-module-types/web-resource-plugin-module#WebResourcePluginModule-WebResourceContexts`.

4. Go to the JavaScript file and add the `JIRA.Projects.Subnavigator` definition. Following are the important elements:

- `id`: An identifier for the component. We will use the web panel location as the ID, as it will help us dynamically fetch it later.
- `triggerPlaceholder`: A CSS selector or DOM element where the dropdown2 trigger will be placed.
- `contentPlaceholder`: A CSS selector or DOM element where the dropdown2 content (aka the options) will be placed.
- `titlePlaceholder`: A CSS selector or DOM element where the dropdown2 title will be placed (optional). If not provided, the component will not handle the title.
- `itemGroups`: An array with the list of items to choose from. It will be an array of arrays. Each inner group of items will be placed together into separate sections; for example the array of **itemGroups** passed to the component should contain an array for each section of items that needs to be rendered.
- `selectedItem`: The identifier of the item that will appear as selected. We will need to dynamically identify the selected item and set it to this attribute.

- `changeViewText`: Text used for the trigger (optional). If not provided, it will show the selected item.
- `hideSelectedItem`: Whether we should hide the selected item from the dropdown2 (defaults to **true**).

5. In our example, the definition appears as shown here:

```
var jtricksnavigator = new JIRA.Projects.Subnavigator({
   id: "jtricksPanel",
   triggerPlaceholder: jtriggerPlaceholder,
   contentPlaceholder: jcontentPlaceholder,
   titlePlaceholder: jtitlePlaceholder,
     itemGroups: [
          [{
               id: "com.atlassian.jira.jira-projects-
                plugin:jtricksnavigationpanel1",
               label: "Panel One",
               description: "Sample description for panel one",
               link: AJS.contextPath()
                + "/projects/"
                + JIRA.API.Projects.getCurrentProjectKey()
                + "?selectedItem=com.atlassian.jira
                .jira-projects-plugin:jtricksnavigationpanel1"
           },
           {
                id: "com.atlassian.jira.jira-projects-
                plugin:jtricksnavigationpanel2",
                label: "Panel Two",
                description: "Sample description for panel two",
                link: AJS.contextPath()
                + "/projects/"
                + JIRA.API.Projects.getCurrentProjectKey()
                + "?selectedItem=com.atlassian.jira
                .jira-projects-plugin:jtricksnavigationpanel2"
           }],
           [{
                id: "com.atlassian.jira.jira-projects-
                plugin:jtricksnavigationpanel",
                label: "All Panels",
                description: "Sample description for panels",
                link: AJS.contextPath()
                + "/projects/"
                + JIRA.API.Projects.getCurrentProjectKey()
                + "?selectedItem=com.atlassian.jira
                .jira-projects-plugin:jtricksnavigationpanel"
            }]
     ], selectedItem: selectedPanelId,
```

```
        changeViewText: "Select Panel", hideSelectedItem: false
});
```

6. Populate the variables in the above code as appropriate. In our example, the full JavaScript, including the variable population, is as follows:

```
AJS.$(document).ready(function() {
     var jtriggerPlaceholder = AJS.$("#panel-subnav-trigger");
     var jcontentPlaceholder = AJS.$("#panel-subnav-content");
     var jtitlePlaceholder = AJS.$("#panel-subnav-header");

     //Get the selected panel id from browser url
     var selectedPanelId = "com.atlassian.jira.jira-projects-
     plugin:jtricksnavigationpanel";
     var url = window.location.href;
     if (url.indexOf("jtricksnavigationpanel1") > -1) {
         selectedPanelId = "com.atlassian.jira.jira-projects-
         plugin:jtricksnavigationpanel1";
     } else if (url.indexOf("jtricksnavigationpanel2") > -1) {
         selectedPanelId = "com.atlassian.jira.jira-projects-
         plugin:jtricksnavigationpanel2";
     }
     var jtricksnavigator = new JIRA.Projects.Subnavigator({
         ..........
         //as in above step
     });
     AJS.$("#subnav-trigger-jtricksPanel")
     .removeClass("subnav-trigger");
     jtricksnavigator.show();
 });
```

7. Some of the variables are populated from HTML elements in the view template, which we will see in the next step.You also probably noticed a step to remove the `subnav-trigger` class to adjust the view look and feel. Use it if appropriate. You can also define your own CSS classes and use them.
`jtricksnavigator.show()` creates the navigation menus using the above definition.

8. Define the required web panels. Since we have three options, one for the main panel and two for the sub panels, defined into two sections in the `itemGroups` element, we need to create two more web panel definitions, as given here:

```
<web-panel key="jtricks-projectcentric-panel-key1"
location="com.atlassian.jira.jira-projects-
plugin:jtricksnavigationpanel1">
```

```
        <resource type="velocity" name="view"
        location="templates/project/project-navigation-panel1.vm" />

    </web-panel>

    <web-panel key="jtricks-projectcentric-panel-key2"
     location="com.atlassian.jira.jira-projects-
     plugin:jtricksnavigationpanel2">

         <resource type="velocity" name="view"
         location="templates/project/project-navigation-panel2.vm" />

    </web-panel>
```

9. Create the resource views for all the web panels using the HTML element IDs used in the JavaScript earlier. We only need four items in our panel views:

- `panel-subnav-header` holds the selected item's name
- `panel-subnav-trigger` to place the dropdown2 trigger
- `panel-subnav-content` to place the dropdown2 options
- Actual content of the panel

10. The velocity templates for our view definitions are as follows:

project-navigation-panel.vm:

```
<h1>
 <span id="panel-subnav-header"></span>
 <span id="panel-subnav-trigger"></span>
</h1>
<span id="panel-subnav-content"></span>
<br> Please select a panel from the dropdown
     to see how the sub-navigation works!
```

project-navigation-panel1.vm:

```
<h1>
  <span id="panel-subnav-header"></span>
  <span id="panel-subnav-trigger"></span>
</h1>
<span id="panel-subnav-content"></span>
<br> This is a Navigation Panel One from
<a href="http://www.j-tricks.com">J Tricks</a>
```

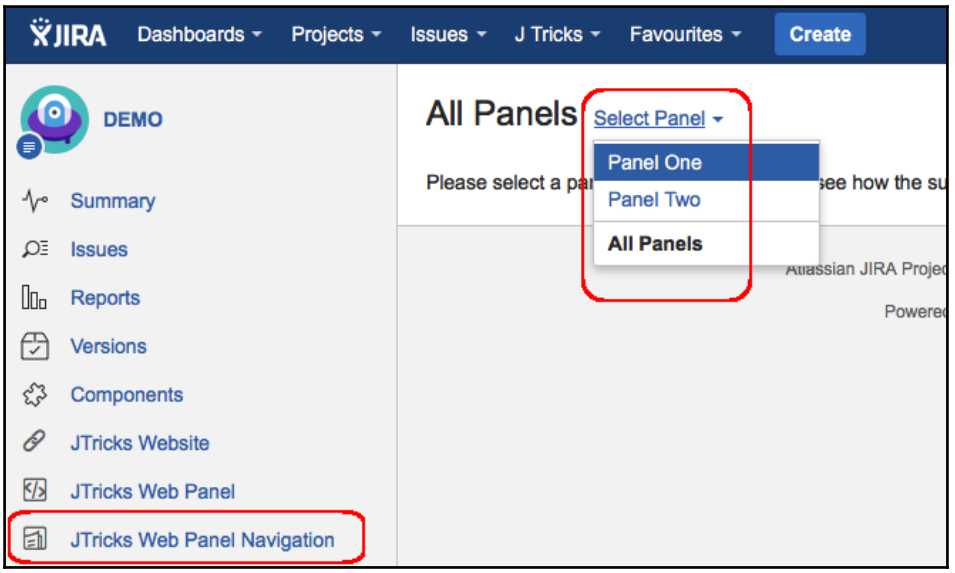
project-navigation-panel.vm:

```
<h1>
  <span id="panel-subnav-header"></span>
  <span id="panel-subnav-trigger"></span>
</h1>
<span id="panel-subnav-content"></span>
<br> This is a Navigation Panel Two from
<a href="http://www.j-tricks.com">J Tricks</a>
```

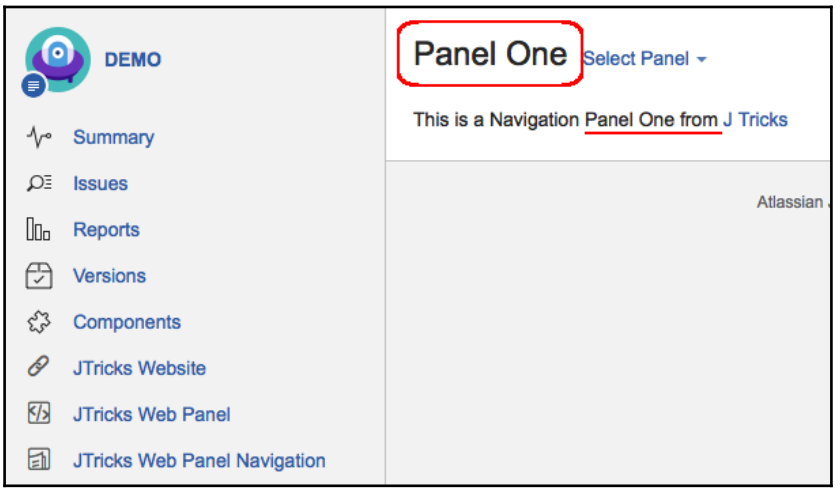
11. Package the plugin and deploy it.

How it works…

Once the plugin is deployed, the new tab panel is shown as below.



The default panel is the main panel, as the main panel's ID is retrieved from the `web-item` link. When we click on the options, the appropriate panel is selected, as shown below:



As you probably noticed, the **title** has also changed, along with the actual **content**.

In the background, all the work is done by the `Subnavigator.js` file, which is added into the page context by the `com.atlassian.jira.jira-projects-plugin:subnavigator` dependency. We only have the small task of initiating the `JIRA.Projects.Subnavigator` object, with the appropriate parameters, and calling the `show()` method.

Adding issue link renderers

One of the great features in JIRA is the ability to create links between JIRA issues and other remote entities. The remote entity can be anything that is accessible via a URL, for example a confluence page, a remote JIRA issue, a ticket in another system, etc.

But the beauty of the feature doesn't end there. In addition to providing a way to create remote links in the user interface, JIRA also lets you create new remote links of custom types and you can then add issue link renderers to render those links in a way you like. The renderer can get more information from the remote system and load it while rendering the view issue page, even asynchronously. In this recipe, we will see how to create an issue renderer for a custom remote link type.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK. Let us consider an interesting example where a given user's twitter status and number of followers are displayed in an issue as a remote link.

In order to communicate with Twitter using Java, there are different mechanisms required, but in this recipe, let us use **Twitter4j**. More details about Twitter4j can be found at `http://twitter4j.org/en/index.html`. The required libraries can be pulled to the plugin project by adding the following dependency in pom.xml.

```
<dependency>
    <groupId>org.twitter4j</groupId>
    <artifactId>twitter4j-core</artifactId>
    <version>[3.0,)</version>
</dependency>
```

You will also have to register an application at `https://apps.twitter.com/`to communicate with Twitter. After the app is registered, we can generate key and access tokens for our app to communicate with Twitter over **OAuth**. We will need the **Consumer Key**, **Consumer Secret**, **Access Token,** and **Access Token Secret** for communicating with Twitter in our code.

Following is the code to get an instance of Twitter using the above attributes:

```
ConfigurationBuildercb = new ConfigurationBuilder();
cb.setDebugEnabled(true).setOAuthConsumerKey("***************")
   .setOAuthConsumerSecret("***************")
   .setOAuthAccessToken("***************")
   .setOAuthAccessTokenSecret("***************");
TwitterFactorytf = new TwitterFactory(cb.build());
Twitter twitter = tf.getInstance();
```

We can now use the Twitter object to retrieve the authorized data from the given account.

That's it for the fun part, and now we move on to the serious bit!

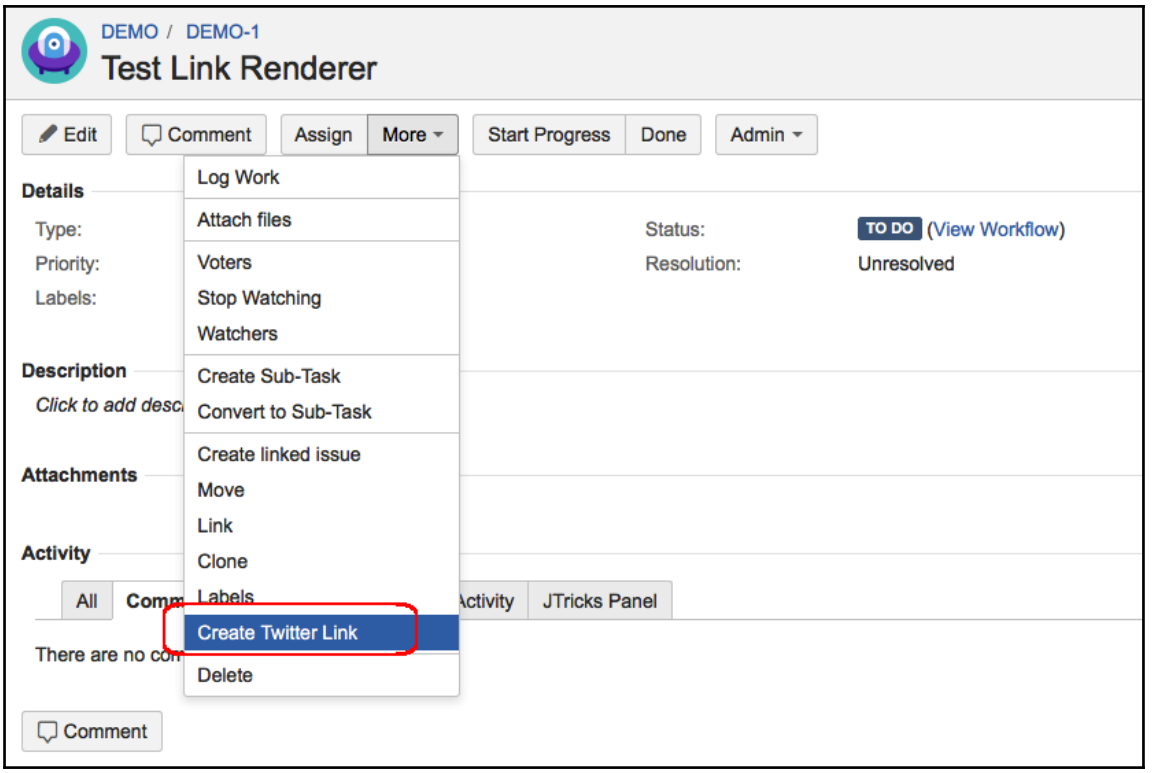
How to do it…

In order to set the stage for issue link rendering, let us create a new remote link type in JIRA with the application type as **Twitter**. This can be done using the `RemoteIssueLinkService` interface, as explained in the below steps.

1. Define a webwork action and add it under issue operations using a web-item module, as in the following snippet:

```
<web-item key="remote-twitter-item"
name="Twitter Link" section="operations-operations" weight="800">
   <label>Create Twitter Link</label>
   <link linkId="remote-twitter.link">
      /secure/TwitterLink.jspa?key=$!{issue.key}
   </link>
</web-item>

<webwork1 key="remote-twitter-action" name="Remote Twitter Action"
class="java.lang.Object">
    <description>Action to create remote twitter link</description>
    <actions>
        <action name="com.jtricks.jira.webwork.TwitterLinkAction"
        alias="TwitterLink">
        </action>
    </actions>
</webwork1>
```

This adds a new issue operation to create a twitter link, as shown here:

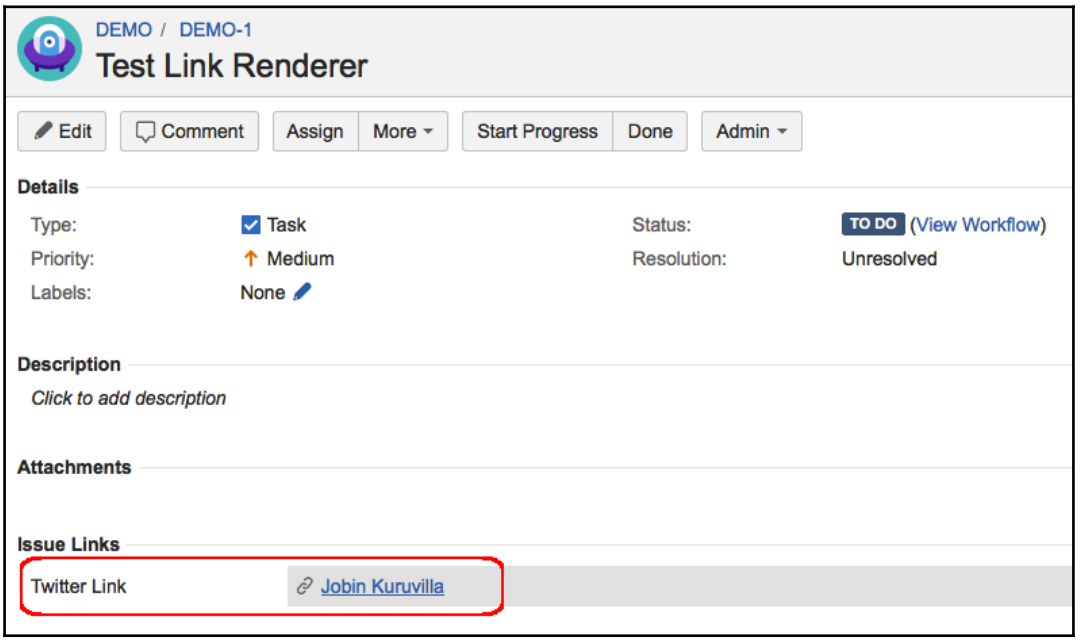


2. Add the `TwitterLinkAction` class to create the twitter link, using the `RemoteIssueLinkService` interface. In this class, we will use the `Twitter` object to retrieve the details required to create the new application type, as shown here:

```
final RemoteIssueLinkremoteIssueLink = new
RemoteIssueLinkBuilder().url("https://twitter.com/" +
twitterId).title(twitter.showUser(twitterId).getName())
.globalId(twitterId).issueId(issue.getId())
.relationship("Twitter Link").applicationName("Twitter")
.applicationType("Twitter").build();

RemoteIssueLinkService.CreateValidationResultvalidationResult =
this.remoteIssueLinkService.validateCreate(jiraAdmin,
remoteIssueLink);
if (validationResult.isValid()) {
    RemoteIssueLinkResult result =
    this.remoteIssueLinkService.create(jiraAdmin,
    validationResult);
}
```

3. As you can see, we have used the application type as **Twitter** and have populated the other attributes using the `Twitter` instance, as appropriate.

Once the remote link is created, it will appear as follows:



This is the default rendering of an issue link in JIRA. We will now implement a new issue link rendering to modify the rendering of the preceding link:

Following is the step-by-step process required to implement an issue link renderer that will transform this link to something better and with more detail.

1. Define the **issue link renderer** module in `atlassian-plugin.xml`:

```
<issue-link-renderer key="twitterLinkRenderer"
application-type="Twitter"
class="com.jtricks.ui.renderer.TwitterLinkRenderer">
    <resource name="initial-view" type="velocity"
    location="templates/issue/twitterlink.vm"/>
</issue-link-renderer>
```

As with any other plugin module, `issue-link-renderer` has a unique **key**. It also has a human readable **name** and can have a **i18n-name-key** for localization. The next two attributes, **class** and **application-type**, are the important ones. **class** will be used to create the renderer context and **application-type** will define the type of links that will use the new renderer.

The module also has a `resource` element that will be used to render the view. The **type** of resource will be `velocity` and the **name** will be `initial-view` or `final-view`, which are used for initial viewing and asynchronous loading respectively.

In the preceding declaration, **application-type** is `Twitter` and we have only the **initial-view** defined. The renderer class is `TwitterLinkRenderer`.

2. Develop the renderer class. The class should implement `IssueLinkRenderer` interface and may extend `AbstractIssueLinkRenderer` to reduce the number of methods to be implemented.When we extend `AbstractIssueLinkRenderer`, there is only one method, `getInitialContext`, to be implemented. The code goes like this:

```
public class TwitterLinkRenderer extends AbstractIssueLinkRenderer{
    public static final String DEFAULT_ICON_URL =
    "/download/resources/com.jtricks.ui-plugin
    :ui-plugin-resources/images/twitter.jpg";

    @Override
    public Map<String, Object>
    getInitialContext(RemoteIssueLinktwitterLink,
    Map<String, Object> context) {
        final I18nHelperi18n = getValue(context, "i18n",
        I18nHelper.class);

        final String baseUrl = getValue(context, "baseurl",
        String.class);

        //Get Twitter status here
        return createContext(twitterLink, i18n, baseUrl, status);
    }

    private <T> T getValue(Map<String, Object> context,
    String key, Class<T>klass) {
         Object obj = context.get(key);
         if (obj == null) {
            throw new IllegalArgumentException
            (String.format("Expected '%s' to exist in
```

```
                the context map", key));
            }
            return klass.cast(obj);
        }
        private static Map<String,
        Object>createContext(RemoteIssueLinkremoteIssueLink,
        I18nHelperi18n, String baseUrl, String status) {
            ImmutableMap.Builder<String,
            Object>contextBuilder = ImmutableMap.builder();
            String tooltip = "Some tooltip";
            final String iconUrl = DEFAULT_ICON_URL;
            final String iconTooltip = "Some icon tooltip";
            putMap(contextBuilder, "id",
            remoteIssueLink.getId());

            putMap(contextBuilder, "url", remoteIssueLink.getUrl());
            putMap(contextBuilder, "title",
            remoteIssueLink.getTitle());
            putMap(contextBuilder, "iconUrl", iconUrl);
            putMap(contextBuilder, "iconTooltip", iconTooltip);
            putMap(contextBuilder, "tooltip", tooltip);
            putMap(contextBuilder, "status",
            status.length() > 50 ? status.substring(0, 50)
            + "..." : status); return contextBuilder.build();
    }
    private static void putMap(ImmutableMap.Builder<String,
    Object>mapBuilder, String key, Object value) {
            if (value != null) {
                mapBuilder.put(key, value);
            }
    }
}
```

The method only populates the context required for rendering the velocity template. Basically, it populates the context map with a number of key/value pairs. We will be using them in the velocity template in the next step.

You might also have noticed that the `DEFAULT_ICON_URL` points to a downloadable resource. You can either use a custom image declared as a resource in the plugin or go with a JIRA image.

Getting the twitter status, the latest tweet, can be done easily using the `Twitter` instance.

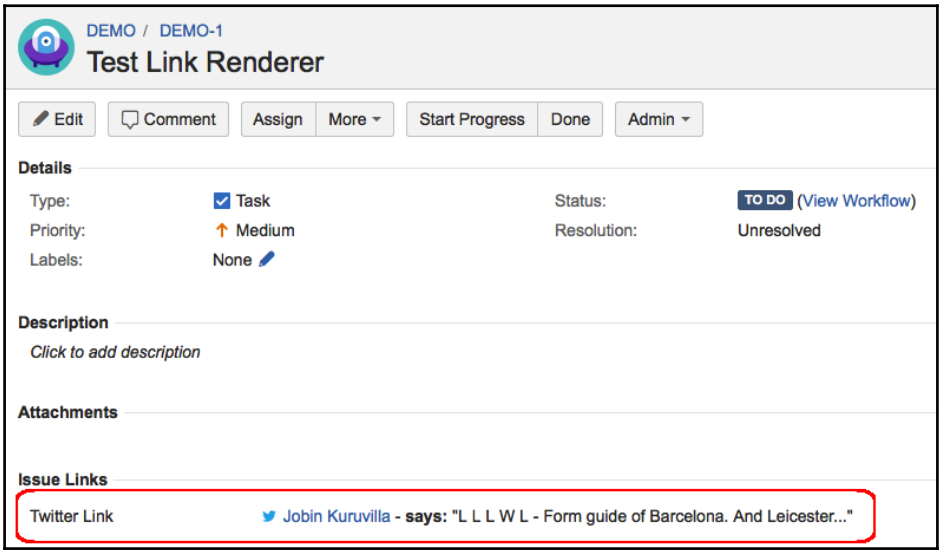
```
ResponseList<Status> response =
twitter.getUserTimeline(twitterLink.getGlobalId());
status = response.get(0).getText();
```

3. Create a renderer velocity template. In this example, we only have `initial-view` defined and the velocity template for the same is `twitterlink.vm`. We can create it using the variables available in the context. An example would be:

```
<p>
  #renderIssueLinkIcon(${iconUrl}
    ${iconTooltip} ${iconTooltip} ${textutils})
    <span >
      <a href="${url}" class="link-title">
      ${textutils.htmlEncode($title)}</a>
    </span>
  #if ($status) -
    <span >
      <span class="status"><b>says:</b> "$status"</span>
    </span>
  #end
</p>
```

   Here, we render the view with **icon**, the **title** (which was nothing but the tweeter's name), the **url** to tweeters account, and the current **status**.
   `renderIssueLinkIcon` is a macro provided by JIRA to render the icon. You can, of course, use your own code here!

With this, the remote link will be rendered slightly differently, although it points to the same location, and it also shows the latest tweet of the tweeter!



What if you want to show more details but load them asynchronously to improve the page loading experience? That is where `final-view` comes in. In that case, we can modify the plugin as follows:

1. Include `final-view` in the plugin definition in `atlassian-plugin.xml`:

```
<issue-link-renderer key="twitterLinkRenderer"
application-type="Twitter"
class="com.jtricks.ui.renderer.TwitterLinkRenderer">

    <resource name="initial-view" type="velocity"
    location="templates/issue/twitterlink.vm"/>
    <resource name="final-view" type="velocity"
    location="templates/issue/twitterlink-final.vm"/>

</issue-link-renderer>
```

   Here, we added `twitterlink-final` as the `final-view`.

2. Modify the renderer class to override the `getFinalContext` and `requiresAsyncLoading` methods.

   a. `getFinalContext`: This method populates the variables required in the final context. Let us assume we want to show everything we showed before and also the number of followers for the tweeter. The code will be:

```
@Override public Map<String,
Object>getFinalContext(RemoteIssueLinktwitterLink,
Map<String, Object> context) {
    final I18nHelperi18n = getValue(context,
    "i18n", I18nHelper.class);
    final String baseUrl = getValue(context,
    "baseurl", String.class);

    //Get twitter status as before
    ImmutableMap.Builder<String, Object>contextBuilder =
    ImmutableMap.builder();
    contextBuilder.putAll(createContext(twitterLink, i18n,
    baseUrl, status));

    //Also add count of twitter followers
    try {
        putMap(contextBuilder, "followers",
        twitter.getFollowersIDs(-1).getIDs().length);
    }
    catch (TwitterException e) {
         putMap(contextBuilder, "followers", "?");
```

```
            e.printStackTrace();
        }
        return contextBuilder.build();
    }
```

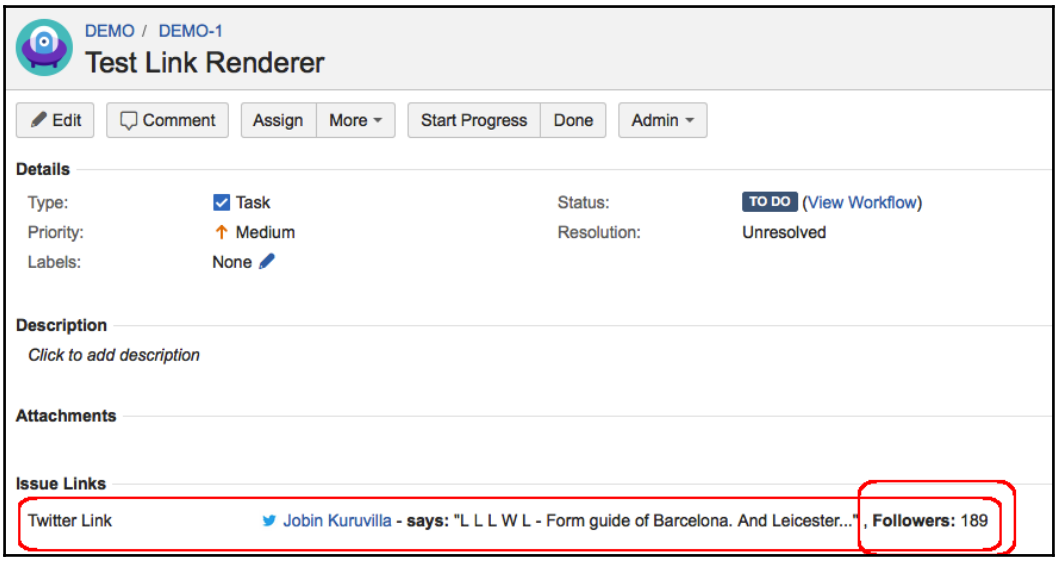
b. `requiresAsyncLoading`: This method will just return true.

3. Create the velocity template required for `final-view`, **twitterlink-final** in this case:

```
<p>
  #renderIssueLinkIcon(${iconUrl} ${iconTooltip}
  ${iconTooltip} ${textutils})
    <span >
       <a href="${url}"
       class="link-title">${textutils.htmlEncode($title)}</a>
    </span>
  #if ($status) -
    <span >
        <span class="status"><b>says:</b> "$status"
        </span>
    </span>
  #end
  #if ($followers) ,
    <span >
        <span class="followers">
            <b>Followers:</b> $followers
        </span>
    </span>
  #end
</p>
```

As you can see, it is pretty much similar to the first template, except for the part where the follower count is added.

With that, the link is rendered as follows:



You will notice that the follower count is retrieved asynchronously and is added after the page has loaded.

How it works…

Be it twitter or any other app, JIRA looks for the **application type** of the remote link and checks if there are any renderers available for that type. The rendering part is done in two steps:

1. The `initial-view` is rendered along with the rendering of issue page. It is wise to provide only the basic information here, namely what is included in the remote link, to avoid delay in loading the issue details.
2. After the `initial-view` is rendered, an asynchronous call is made to render the `final-view`. This is where you can perform costlier operations.

In our example, we have loaded the twitter status in the `initial-view`, but ideally that should also be done in the `final-view`. That will reduce the overhead of communicating with Twitter during the initial load.

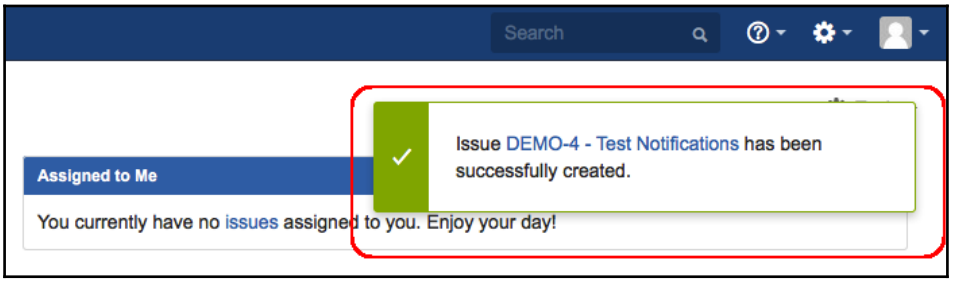
We wanted to demonstrate the loading of status and follower count separately to show you the difference between `initial-view` and `final-view`!

See also

- The *Programming Issue Links* recipe in `Chapter 7`, Programming Issues

Displaying dynamic notifications/warnings on issues

While performing various operations in JIRA, you have probably noticed popup **error/warning/info** messages which notify the user about the status of the operation just performed. For example, a notification message is displayed to the user as soon as an issue is created, as shown here:



How are these messages displayed? How hard is it to do something similar in a plugin?

As you might have guessed, JIRA does this with the help of built-in CSS classes or Javascript functions. **AUI** (**Atlassian User Interface**) components include **messages** to render such notifications on the page itself, while the **AJS** (**Atlassian JavaScipt**) framework exposes those as built-in functions.

JIRA has an interesting feature, the Announcement Banner, which can be used to make announcements to its user community via JIRA itself. The announcement banner supports HTML tags and can be used to inject JavaScript snippets into pages, if needed. This is a great incentive, given that most of the JIRA data is now accessible via REST API using JavaScript!

Power users of JIRA sometimes want to take advantage of this to display warnings or notifications while they are viewing different pages in JIRA, based on various criteria.

In this recipe, we will see how to add a warning or error message on a JIRA page, using AJS functions in the announcement banner. To keep it simple, we will ignore complex logic and focus on the display part alone.

Getting ready

Let us assume that we want to display a message to the user when the user visits his/her profile page.

How to do it…

As mentioned earlier, we will use the announcement banner to inject the JavaScript into the pages. This can, of course, be done using the web-resource plugin module as well. Using the web-resource plugin module is recommended, as it gives us more flexibility on the pages where JavaScript is enabled.

However, we will use the announcement banner in this recipe to concentrate on AJS functions. See earlier chapters to find out how we can use the web-resource plugin module to define downloadable resources like JavaScript.

You can navigate to **Administration** | **System** | **User Interface** | **Announcement banner** to add the JavaScript into the JIRA pages.

In order to display a simple notification, all that is needed is the following JavaScript snippet.

```
JIRA.Messages.showSuccessMsg('Your message goes here');
```

The same thing can be written as:

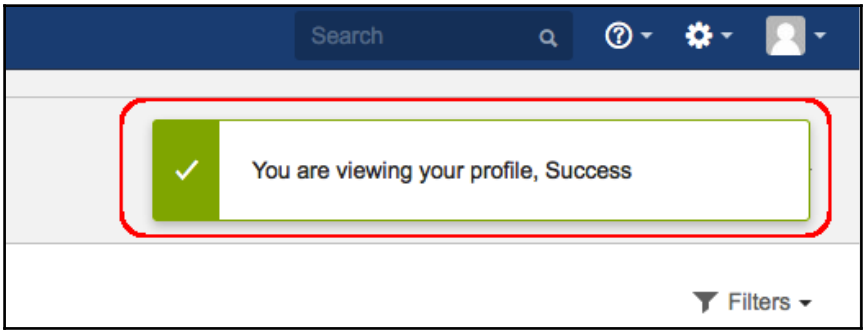
```
JIRA.Messages.showMsg('Your message goes here', {type: SUCCESS});
```

As you can see, the second snippet has a generic method that takes the type as `SUCCESS`. This is encapsulated as a single call, `showSuccessMsg`, in the first snippet. For now, let us stick with `showSuccessMsg`.

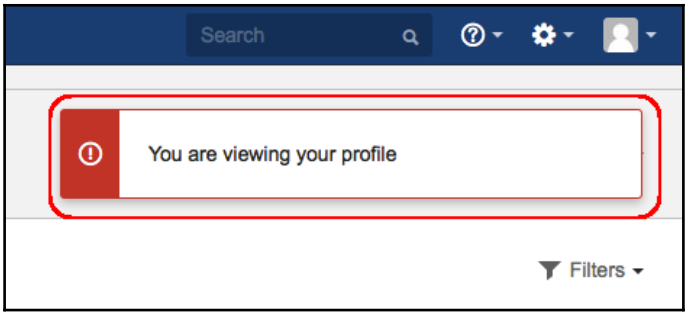
To display a message to the user, when he/she visits the profile page, we can use the following JavaScript.

```
<script> var pathname = window.location.pathname;
    if (pathname.indexOf("ViewProfile.jspa") >= 0) {
        JIRA.Messages.showSuccessMsg('You are viewing your profile,
        Success');
    }
</script>
```

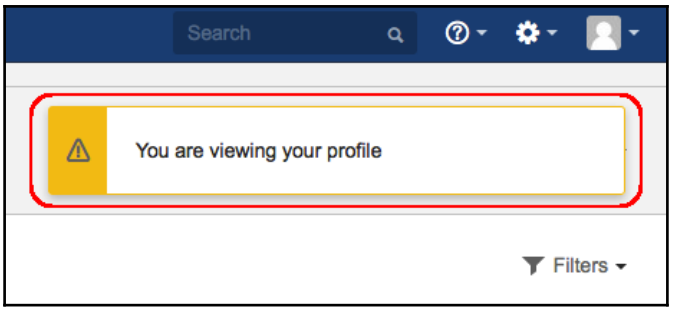
The message will appear on the screen when you visit the profile, as shown below.



We can replace the `showSuccessMsg` function with `showErrorMsg` or `showWarningMsg` to get the appropriate message format, as shown in the following screenshot:



You can also use `showWarningMsg`, as shown in the following screenshot:



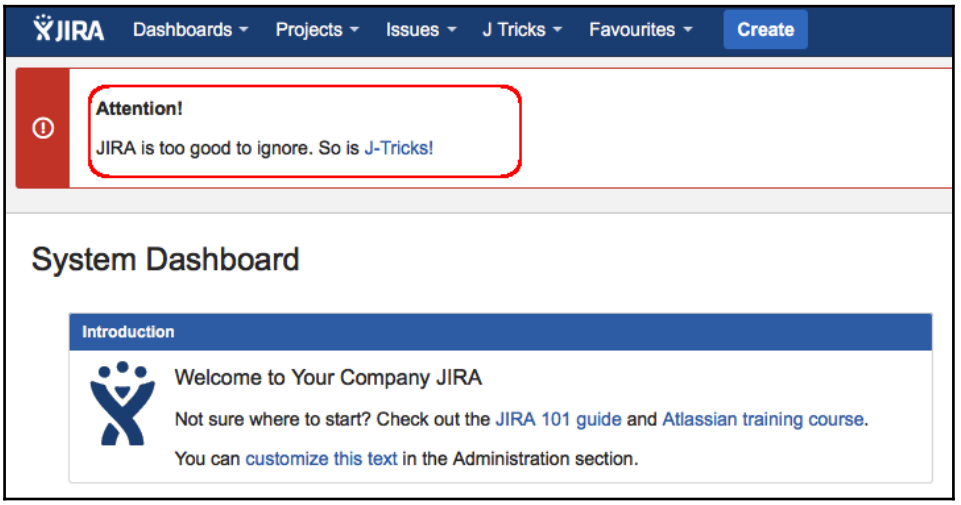
It is also possible to add options like `closeable` and `timeout`, details of which can be found in the `atlassian-jira/includes/jira/common/messages.js` file under the JIRA installation folder. We hope you find it useful.

There's more…

It is also possible to display such notification messages using the AUI Messages component. For example, the `aui-message` and `aui-message-error` classes can be used to create an error message, as shown below:

```
<div class="aui-message aui-message-error">
    <p class="title">
        <strong>Attention!</strong>
    </p>
    <p>
        JIRA is too good to ignore.
         So is <a href="http://www.j-tricks.com">J-Tricks</a>
    </p>
</div>
```

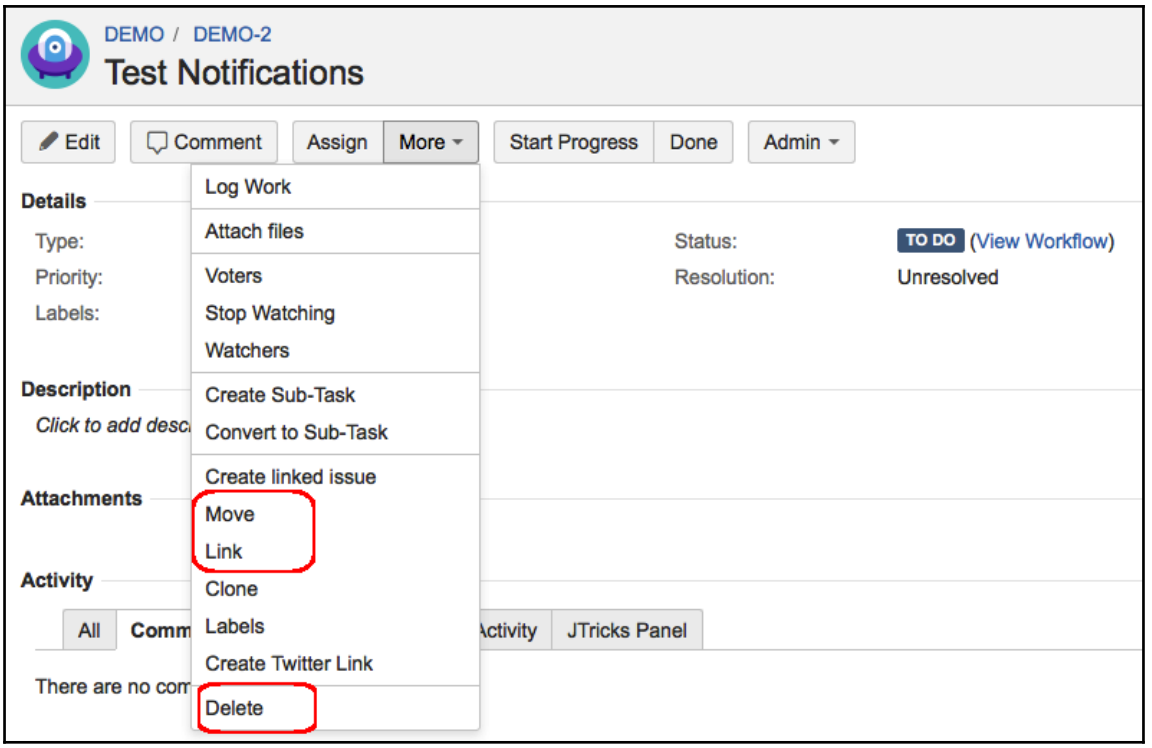
When the above snippet is added in the announcement banner, it is rendered as shown here:



Similar to **JIRA.Messages**, we can use different classes to get a different look and feel. More details about **AUI Messages** can be read at `https://docs.atlassian.com/aui/latest/docs/messages.html`.

Re-ordering Issue Operations in the View Issue page

In the previous chapter, we saw how to create new issue operations. All the existing issue operations in JIRA have a predefined order associated with them. Currently, in JIRA, the actions are ordered as shown in the following screenshot:



In this recipe, we will see how we can reorder those actions without actually doing any coding! For example, let us assume we want to move the **Delete** option so it is first in the list and then move the **Link** operation above **Move**!

How to do it…

Following is the step-by-step process required to reorder the issue operations:

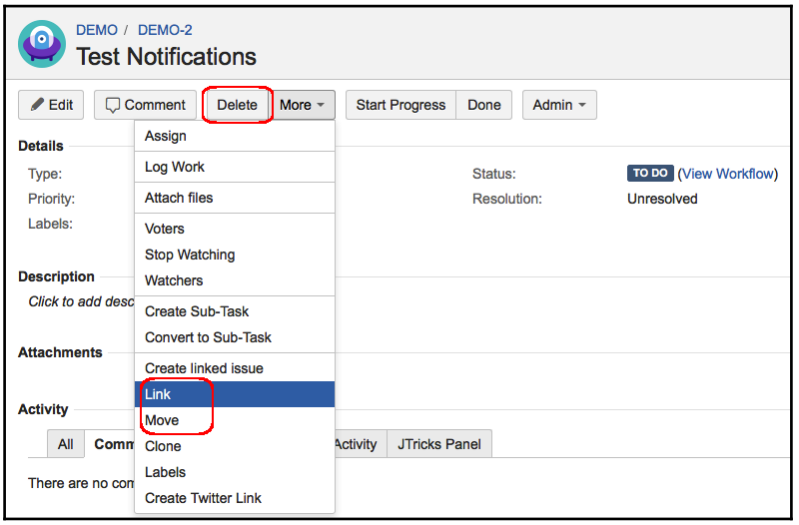
1. Go to the `system-issueoperations-plugin.xml` file residing under the `atlassian-jira/WEB-INF/classes` folder. This is the file where all the issue operations are defined.
2. Identify the web fragments to be reordered. In our example, we need to reorder **Delete**, **Move**, and **Link**.
   Move and **Link** are `web-item` elements that belong to the same `web-section` named `operations-operations`. **Delete**, on the other hand, has its own `web-section` named `operations-delete`.
   The weight attribute is relative only within the same level; i.e. changes in the

weight attribute of **Move** and **Link** will not impact changed to **Delete**.

3. Modify the **weight** attribute on `operations-delete` web section to 5. 10 is the lowest **weight** value, assigned to the top level web section, named `operations-top-level`. Changing the **weight** value of `operations-delete` to 5 will send it above `operations-top-level`.
4. Similarly, modify the **weight** value of the `link-issue` web item to 8, which is a lesser value compared to the **weight** of the `move-issue` web item, which is 10.
5. Save the file and restart JIRA.

How it works…

As mentioned earlier, the **weight** attribute is relative within the same level, and hence the above changes will reorder the operations, as shown here:



As you can see, **Delete** now appears as the first operation, while **Link** moved above **Move**!

See also

- The *Adding new Issue Operations* recipe in `Chapter 7`, Programming Issues

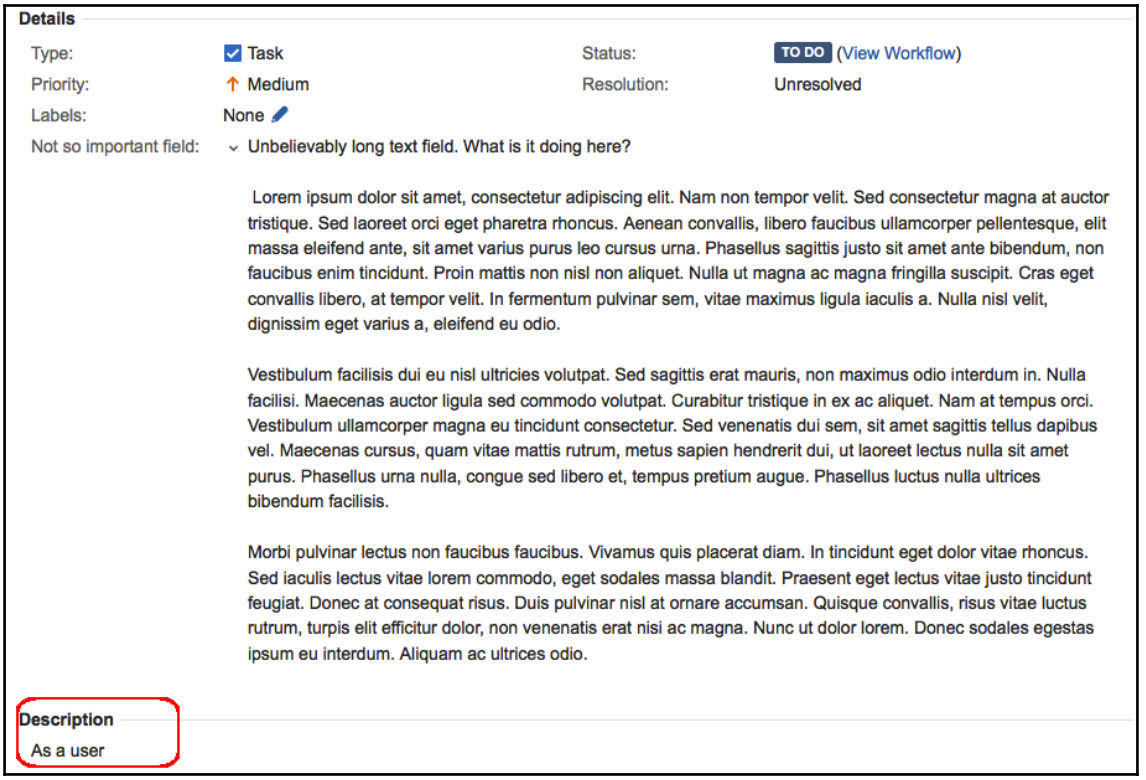
Re-ordering fields in the View Issue page

JIRA has highly configurable screen schemes and field configuration schemes. These schemes let us use different screens for different operations and allow us to configure the screens to show/hide fields, make fields mandatory, change the order of fields, etc.

While this works on most screens, JIRA's **View Issue** page is an exception. We can hide/show fields from this screen but there is no easy way to reorder the fields on this screen using configuration options.

For example, in the view issue page, the summary of the issue is followed by standard issue fields like status, priority, versions, components, and so on. It is then followed by custom fields, and then comes the description of the issue. This can sometimes be a pain, for example in cases where description is the most important field.

Following is how the view issue page looks when you have a large custom field:



As you can see, the **Not so important field** is an unlimited text field, which has a huge value, and the **Description** field is way down on the screen. In this recipe, we will see how we can bring the **Description** field above the **Details** section.

How to do it…

In earlier versions of JIRA, the view issue page was rendered by `/secure/views/issue/viewissue.jsp`, but in JIRA5+, this is done by a bundled plugin named `jira-view-issue-plugin.xxx.jar`. This makes it slightly difficult, as we already saw in the "**Modifying Atlassian bundled plugins**" recipe in `Chapter 2`, *Understanding the Plugin Framework*. We have already seen how to modify a bundled plugin there, but in this particular example, the following are the required steps.

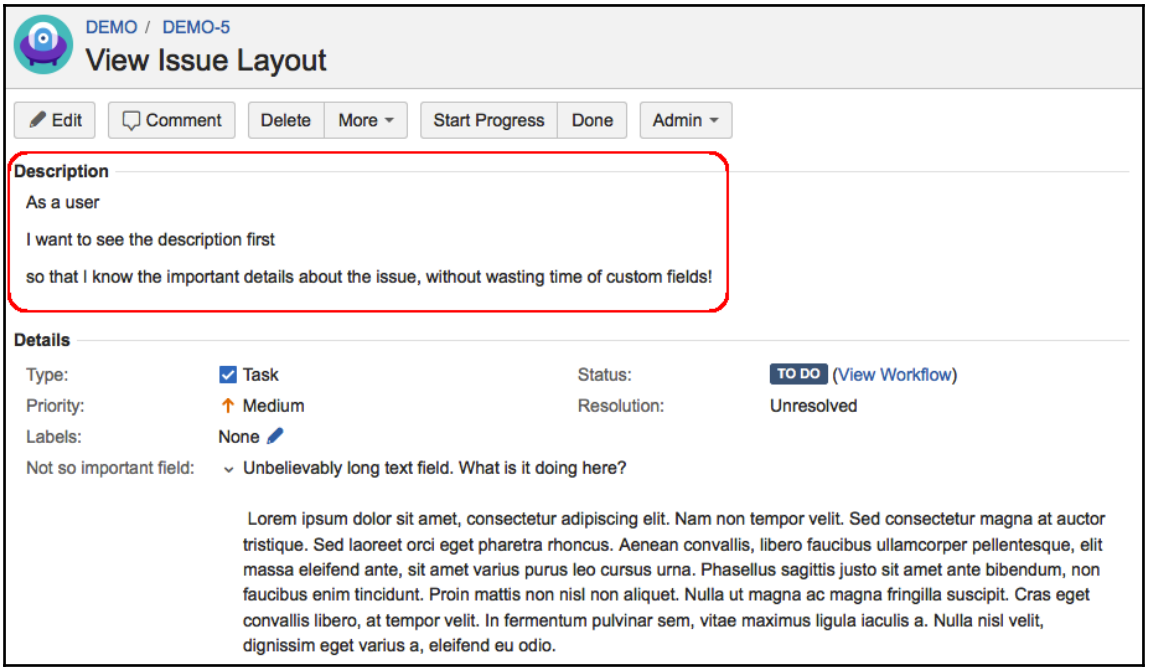
1. Go to the `atlassian-plugin.xml` file of the bundled plugin.
2. Identify the definition of the **description** module. It looks like this:

```
<!-- Description panel -->
<web-panel key="descriptionmodule"
location="atl.jira.view.issue.left.context" weight="200">

    <context-provider class="com.atlassian.jira.plugin
    .viewissue.DescriptionBlockContextProvider"/>

    <resource name="view" type="velocity"
    location="viewissue/descriptionblock.vm"/>

    <label key="common.concepts.description"/>

    <condition class="com.atlassian.jira.plugin
    .webfragment.conditions.IsFieldHiddenCondition" invert="true">

        <param name="field">description</param>
    </condition>
</web-panel>
```

As you can see, this a web panel and the order in which it appears onscreen is defined by the **weight** attribute.

3. Modify the **weight** attribute to have a value that is less than that of `details-module`. The weight of **details**module is 100, as you can see in the `atlassian-plugin.xml` file, and hence we will have to modify the weight of the **description** module to something less than 100; say, 50.
4. Save everything and put it back as described in the "Modifying Atlassian bundled plugins" recipe in `Chapter 2`, *Understanding the Plugin Framework*.
5. Restart JIRA.

Once this is done, the modified **View Issue** page will look like this:



See also

- The *Modifying Atlassian bundled plugins* recipe in `Chapter 2`, *Understanding Plugin Framework*

9
Remote Access to JIRA

In this chapter, we will cover:

- Writing Java client for REST API
- Working with issues
- Working with attachments
- Remote time tracking
- Working with comments
- Remote user and group management
- Progressing an issue in workflow
- Searching issues
- Managing versions
- Managing coRemote administration methods mponents
- Remote administration methods
- Exposing services and data entities as REST APIs
- Using the REST API browser
- Working with JIRA Webhookse, a single permission is added, that is, to administer the project. We can add the rest in a similar fashion.

Introduction

We have seen various ways to enhance JIRA functionality in the previous chapters, but how do we communicate with JIRA from another application? What are the various methods of integrating third-party applications with JIRA? Or, in simple words, how does JIRA expose its functionalities to the outside world?

JIRA used to expose its functionalities via **REST**, **SOAP**, or **XML/RPC** interfaces. But in JIRA7, SOAP and XML/RPC interfaces are removed, after giving enough notice to developers, and hence we need to use REST APIs for remote communication. If you have existing integrations that use the SOAP or XML/RPC APIs, they need to be migrated to REST API calls.

Not all the JIRA functionality is exposed via the REST interface but JIRA also lets us extend them. In this chapter, we will learn how to communicate with JIRA using REST APIs and add how to expose more functionality via REST with the help of plugins.

This chapter covers only a few examples and should not be treated as the final list of the supported methods. A more detailed explanation of all these interfaces can be found at `https://docs.atlassian.com/jira/REST/latest/`.

Also, for most of this chapter, we will use the **JIRA REST Java Client (JRJC)** library, as it provides an easy way to interact with JIRA using a Java client. But **JRJC** doesn't expose all the methods supported by **REST** APIs. For such methods, we will see the appropriate **REST** API definitions and sample request/response data and you can use them, as appropriate, in your code.

Writing Java client for REST API

In this recipe, we will quickly see how we can create a Java client to communicate with JIRA using the REST APIs.

Getting ready

Create a simple Java project. You can use the maven archetype, `maven-archetype-quickstart`, to create the project or use your favorite IDE to generate one.

The following is the *non-interactive* command for generating a simple project using `maven-archetype-quickstart`:

```
    mvnarchetype:generate -DgroupId=com.jtricks
    -DartifactId=rest-client -DarchetypeArtifactId=maven-archetype-
quickstart
    -DinteractiveMode=false
```

You can easily generate an Eclipse project using the following command:

```
    mvn eclipse:eclipse
```

How to do it…

In order to connect to JIRA using REST APIs, Atlassian has developed the **JRJC** library. It provides a thin layer of abstraction on top of the REST API and related HTTP(S) communication, and gives a domain object model to represent the JIRA entities, such as issues, priorities, resolutions, statuses, users, and so on.

The **REST** API and the **JRJC** library are quickly evolving, with new methods added in every version. The current status of the library can be viewed at `https://ecosystem.atlassian.net/wiki/display/JRJC/Home`.

We will be using **JRJC** to connect to our JIRA instance using the standalone Java program. The following are the steps:

1. Create a Maven project and add the **JRJC** dependency to the `pom.xml` file:

```
<dependency>
  <groupId>com.atlassian.jira</groupId>
  <artifactId>jira-rest-java-client-core</artifactId>
  <version>4.0.0</version>
</dependency>
```

   Make sure you use the appropriate version of JRJC. All the versions can be found in the Maven repository at `https://maven.atlassian.com/content/repositories/atlassian-public/com/atlassian/jira/jira-rest-java-client-core/`. If you are not using Maven, the full dependencies are listed in the Atlassian documentation at `https://ecosystem.atlassian.net/wiki/display/JRJC/Project+Dependencies`.

   Depending on the version of JRJC, you may end up adding a few dependencies in your `pom.xml`. For version 4.0.0, the following are the dependencies added:

```
<dependency>
   <groupId>com.atlassian.fugue</groupId>
   <artifactId>fugue</artifactId>
   <version>1.1</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.9</version>
</dependency>
```

2. Create a Java project by running `mvneclipse:eclipse` if you are using Eclipse or create the project using your favorite IDE and add all the dependencies listed earlier in the class path.
   Once done, create a standalone Java class.
3. Create a connection to the JIRA server:

```
finalAsynchronousJiraRestClientFactory factory = new
AsynchronousJiraRestClientFactory();
final URI uri = new URI("http://localhost:8080/jira");
final JiraRestClientjiraRestClient =
factory.createWithBasicHttpAuthentication(uri,
"username", "*****");
```

   Here, we instantiate the `AsynchronousJiraRestClientFactory` class and use the `createWithBasicHttpAuthentication` method to instantiate the `JiraRestClient` by passing the username and password. There is also an `AnonymousAuthenticationHandler` that can be used, if you do not want to pass a username and password. Only operations supported by anonymous access can be invoked in that case.

   RESTful architecture promotes stateless connection and hence there is no notion of the user session. This means the credentials will be sent back and forth in plain text, just encoded with Base64, for each request and so it is not safe to use it outside a firewall or company network. Outside a firewall, it is recommended to use HTTPS protocol to communicate.

4. Retrieve the appropriate client from `JiraRestClient`. For example, you can retrieve `IssueRestClient` for issue operations and `UserRestClient` for user operations:

```
IssueRestClientissueClient = jiraRestClient.getIssueClient();
UserRestClientuserClient = jiraRestClient.getUserClient();
```

   Similarly, other clients can be retrieved.

5. Use the clients to invoke the supported methods.

With that, the client is ready! Given the fact that JIRA REST API is evolving so quickly, JRJC has a lot of potential and is worth investing time in.

Working with issues

So far, we have seen how to write the REST client using JRJC. Now it is time to move on to some real examples. In this recipe, we will take a look at using REST APIs for various issue operations, such as creating issues, browsing issues, and so on.

Getting ready

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

Let us start with creating issues using the JRJC client, before moving onto updating/editing issues and browsing issues.

Creating Issues

The following are the steps to create an issue with the standard fields populated on it:

1. As mentioned in the *Writing Java client for REST API* recipe, initialize the REST client:

```
final AsynchronousJiraRestClientFactory factory = new
AsynchronousJiraRestClientFactory();
final URI uri = new URI("http://localhost:8080/jira");
final JiraRestClientjiraRestClient =
factory.createWithBasicHttpAuthentication(uri,
"username", "*****");
```

2. Retrieve the `IssueRestClient` from `JiraRestClient`, as we are dealing with issues in this recipe:

```
IssueRestClientissueClient = jiraRestClient.getIssueClient();
```

3. Initialize the `IssueInputBuilder` object using the project key, issuetype Id, and summary:

```
IssueInputBuilderissueInputBuilder = new
IssueInputBuilder("DEMO", 1L, "Test Summary");
```

   Here,1L is the **id** of issuetype.

4. Populate the standard fields on the `IssueInputBuilder`, as appropriate:

```
issueInputBuilder.setPriorityId(2L).setComponentsNames
(Arrays.asList("Apples", "Oranges"))
.setAffectedVersionsNames(Arrays.asList
 ("1.0", "1.1")).setReporterName("test")
.setAssigneeName("jobinkk")
.setDueDate(new DateTime());
```

Make sure the priority **id**, component values, version values, usernames, time, and so on are all valid values in your JIRA instance. Priority, in the preceding example, uses the **id** and not the **name**.



Also, you can see that fields such as components and versions take a list of values instead of a single value. Both of them use the **name** instead of the **id**.
Whether to use **name** or **id**, for various fields, can be found at Javadocs (`https://docs.atlassian.com/jira-rest-java-client-api/`).

5. Set the custom field values on the issue. The way to set values into `IssueInputBuilder` differs based on the type of field. The following are some examples for the most-used field types:

   a. Single value fields: We just need to pass a value:

```
issueInputBuilder.setFieldValue("customfield_10000",
"Test text Val");
```

   b. Multiuser picker field: For a multiuser picker, you need to send a list of user objects:

```
User jobin = userClient.getUser("jobinkk").get();
User testUser = userClient.getUser("test").get();
issueInputBuilder.setFieldValue("customfield_10000",
  Arrays.asList(jobin, testUser));
```

   c. Single select fields with options: For a multi-valued field, such as a select list, we need to first build a `ComplexIssueInputFieldValue` object and then set the field value using it:

```
issueInputBuilder.setFieldValue("customfield_10000",
ComplexIssueInputFieldValue.with("value", "Three"));
```

d. Multi select fields with options: For a multi select list, you need to set a list of `ComplexIssueInputFieldValue` objects:

```
List<ComplexIssueInputFieldValue>fieldList = new
ArrayList<ComplexIssueInputFieldValue>();
String[] valuesList = new String[] { "Alpha", "Gamma" };
for (String aValue : valuesList) {
  Map<String, Object>mapValues = new HashMap<String, Object>();
  mapValues.put("value", aValue);

  ComplexIssueInputFieldValuefieldValue = new
  ComplexIssueInputFieldValue(mapValues);

  fieldList.add(fieldValue);
}
issueInputBuilder.setFieldValue("customfield_10000",
fieldList);
```

e. Cascading select: For cascading select, the parent value should be put into the map as usual and the child value should have its own `ComplexIssueInputFieldValue` object as shown in the following code:

```
Map<String, Object>cascadingValues = new
HashMap<String, Object>();
cascadingValues.put("value", "Parent 2");

cascadingValues.put("child",
ComplexIssueInputFieldValue.with("value", "Child 22"));

issueInputBuilder.setFieldValue("customfield_10000", new
ComplexIssueInputFieldValue(cascadingValues));
```

In all the examples, customfield_10000 denotes the custom field ID. Other field types, such as number fields, checkboxes, and so on, will fit into one of the above categories.

5. Build the `IssueInput` object from the builder:

```
IssueInputissueInput = issueInputBuilder.build();
```

6. Create the issue using the `IssueRestClient` object:

```
Promise<BasicIssue>createdIssue =
jiraRestClient.getIssueClient().createIssue(issueInput);
BasicIssue issue = createdIssue.get();
```

Updating issues

Updating an issue is very similar to create. It involves creating an `IssueInputBuilder` object and setting only the fields that need to be updated.

We can then use the `IssueRestClient` to call the `updateIssue` method, as shown below:

```
IssueInputBuilderupdateInputBuilder = new IssueInputBuilder();
updateInputBuilder.setAssignee(testUser);
issueClient.updateIssue(issueKey, updateInputBuilder.build());
```

In the above example, we are only setting the `assignee` field.

Browsing issues

Browsing an existing issue couldn't be any simpler:

```
final Promise<Issue>newIssue =
jiraRestClient.getIssueClient().getIssue(issueKey); Issue browsedIssue =
newIssue.get();
```

We can then retrieve the fields from the issue object as shown below:

```
Iterable<Field> fields = browsedIssue.getFields();
```

While reading field values, standard fields are straightforward using getter methods whereas custom fields differ based on the type.

If the field is a multiuser field, you can retrieve the values as shown:

```
JSONArray array = (JSONArray) field.getValue();
for (inti = 0; i<array.length(); i++) {
    JSONObjectobj = array.getJSONObject(i);
    System.out.println("Value:" + obj.getString("displayName"));
}
```

You can use different keys instead of `displayName` if you want to retrieve other user properties.

For multiselect fields, use `value` instead of `displayName`:

```
JSONArray array = (JSONArray) field.getValue();
for (inti = 0; i<array.length(); i++) {
    JSONObjectobj = array.getJSONObject(i);
    System.out.println("Value:" + obj.getString("value"));
}
```

For single select, the field value will be a JSON object instead of an array and everything else will be the same:

```
JSONObjectobj = (JSONObject) field.getValue();
System.out.println("Value:" + obj.getString("value"));
```

For cascading select, the parent value and child value are retrieved as shown below:

```
JSONObjectobj = (JSONObject) field.getValue();
System.out.println("Parent Val:" + obj.getString("value"));
System.out.println("Child Val:" +
obj.getJSONObject("child").getString("value"));
```

For normal fields:

```
Object value = field.getValue();
```

Hopefully, that covers the major issue operations using REST.

Working with attachments

In this recipe, we will see how to add attachments to an issue via REST and browse existing attachments.

Getting ready

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe. Make sure attachments are enabled on the JIRA instance by checking at Administration | **System** | **Advanced** | **Attachments**.

How to do it…

There are three different methods exposed by JRJC to add attachments to an issue. The following are the three options and how they are used.

Using input stream and a new filename

1. Create an `InputStream` object from the filepath:

```
InputStream in = new
```

```
FileInputStream("/Users/jobinkk/Desktop/test.txt");
```

2. Get the issue to attach the file. You can use the `IssueRestClient` object as explained previously:

```
final Promise<Issue> issue =
jiraRestClient.getIssueClient().getIssue(key);
Issue browsedIssue = issue.get();
```

3. Get the attachment URI from the issue:

```
URI attachmentURI = browsedIssue.getAttachmentsUri();
```

4. Add the attachment:

```
issueClient.addAttachment(attachmentURI , in, "file1.txt");
```

Here, the last argument, `file1.txt`, is the name with which the file appears on the issue.

Using the AttachmentInput object

This is very similar to the previous case. The only difference is that an `AttachmentInput` object is created before calling the `addAttachments` method. Note that the method name is `addAttachments`, with an "s" at the end:

1. Create an `InputStream` object from the file path:

```
InputStream in = new FileInputStream
("/Users/jobinkk/Desktop/test.txt");
```

2. Get the issue to attach the file. You can use the `IssueRestClient` object as explained earlier:

```
final Promise<Issue> issue = issueClient.getIssue(key);
Issue browsedIssue = issue.get();
```

3. Get the attachment URI from the issue:

```
URI attachmentURI = browsedIssue.getAttachmentsUri();
```

4. Create the `AttachmentInput` object:

```
AttachmentInputattachmentInput = new
AttachmentInput("file2.txt", in1);
```

5. Add the attachment:

```
issueClient.addAttachments(attachmentURI , attachmentInput);
```

Using file and a new filename

Here, we use a `File` object instead of creating `InputStream` and use it in the `addAttachments` method:

1. Create a File object from the file path:

```
File file = new File("/Users/jobinkk/Desktop/test.txt"));
```

2. Get the issue to attach the file. You can use the IssueRestClient object as explained earlier:

```
final Promise<Issue> issue = issueClient.getIssue(key);
Issue browsedIssue = issue.get();
```

3. Get the attachment URI from the issue:

```
URI attachmentURI = browsedIssue.getAttachmentsUri();
```

4. Add the attachment:

```
issueClient.addAttachments(attachmentURI, file);
```

Browsing attachments

Browsing attachments on an issue is even easier. You just need to invoke the `getAttachments` method:

```
Iterable<Attachment> attachments = attachedIssue.get().getAttachments();
for (Attachment attachment : attachments) {
        System.out.println("Name:" + attachment.getFilename()
        + ", added by:"
        + attachment.getAuthor().getDisplayName()
        + ", URI:" + attachment.getSelf());
}
```

Remote time tracking

Time tracking in JIRA is a great feature that allows users to track the time they spent on a particular issue. It lets users log work as and when they spend time on an issue and JIRA will keep track of the original estimated time, actual time spent, and the remaining time. It also lets users adjust the remaining time to be spent on the issue, if needed!

While JIRA has a great user interface to let users log work, there are times-when integrating with third-party products-it is necessary to log work using remote APIs. In this recipe, we will see logging work using the REST API.

Getting ready…

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe. Make sure that time tracking is enabled on the JIRA instance by checking at **Administration** | **System** | **Issue Features** | **Time Tracking**.

How to do it…

There are different ways to log work on an issue depending on what we need to do with the remaining estimate on the issue, the visibility of the worklog, and so on. In all the cases, we need to retrieve the issue where the time is logged and the user who logs the time:

```
final Promise<Issue> issue = jiraRestClient.getIssueClient().getIssue(key);
Issue browsedIssue = issue.get();
User jobin = jiraRestClient.getUserClient().getUser("jobinkk").get();
```

Then we need to create a `WorklogInput` object using the required arguments. While adding a worklog, the first argument-self URI-should be `null`. For worklog updates, you can pass the URI of the worklog to be updated instead of `null`.

A simple `WorklogInput` object can be created as follows:

```
WorklogInputworklogInput  = new WorklogInput(null, browsedIssue.getSelf(),
jobin, null, "Some Comment1", new DateTime(), 60, null);
```

Here, `60 minutes` is logged by user `jobin`, along with a comment.

If we want to restrict the visibility of the worklog to a certain group or project role, a `Visibility` object can be passed as the last argument. For example:

```
WorklogInputworklogInput  = new WorklogInput(null, browsedIssue.getSelf(),
 jobin, null, "Some Comment1", new DateTime(), 60,
new Visibility(Type.GROUP, "jira-developers"));
```

If you want to adjust the remaining estimate, you can use the overloaded constructor that takes two additional parameters, adjustment type and the adjustment value. Various possible values for adjustment type and their meanings are as follows:

- `AdjustEstimate.LEAVE`: This leaves the remaining estimate without any changes.
- `AdjustEstimate.AUTO`: This is the default option. It recalculates the remaining estimate based on the logged work.
- `AdjustEstimate.MANUAL`: It decreases the remaining estimate by the provided value.
- `AdjustEstimate.NEW`: It sets the provided value as the new remaining estimate.

The `AUTO` and `LEAVE` options ignore the last argument. Using the `AUTO` option is equivalent to leaving out the last two arguments. A few examples of using adjust estimate types are as follows:

Logging 60 minutes and leaving the remaining estimate as it is:

```
WorklogInputworklogInput  = new WorklogInput(null,
browsedIssue.getSelf(), jobin, null, "Some Comment1",
new DateTime(), 60, null, AdjustEstimate.LEAVE, null);
```

Logging 60 minutes and setting the remaining estimate as 240 minutes:

```
WorklogInputworklogInput  = new WorklogInput(null,
browsedIssue.getSelf(), jobin, null, "Some Comment1",
new DateTime(), 60, null, AdjustEstimate.NEW, "240");
```

Logging 60 minutes and decreasing the remaining estimate by 30 minutes:

```
WorklogInputworklogInput  = new WorklogInput(null,
browsedIssue.getSelf(), jobin, null, "Some Comment1",
new DateTime(), 60, null, AdjustEstimate.MANUAL, "30");
```

Once the `worklogInput`object is constructed, you can use `addWorklog` method:

```
jiraRestClient.getIssueClient().addWorklog(browsedIssue.getWorklogUri(),
worklogInput).claim();
```

In all the examples, `minutesSpent` is an integer whereas`adjustEstimateValue` is a **string**, although both represent minutes.



Notice the `claim()` method invoked on the `Promise` object returned by the `addWorklog` method. This will make the call **synchronous-like**.

Working with comments

In this recipe, we will see how to manage comments on an issue.

Getting ready

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

Adding a comment on an issue using REST API is pretty easy!

1. Retrieve the issue on which the comment needs to be added and retrieve the user who needs to make the comment:

```
final Promise<Issue> issue =
jiraRestClient.getIssueClient().getIssue(key);
Issue browsedIssue = issue.get(); User jobin =
jiraRestClient.getUserClient().getUser("jobinkk").get();
```

2. Create a `Comment` object using issue URI, comment text, author, time of the comment, and so on. You can optionally add visibility information as well, similar to what we have seen while adding worklogs. Leave the visibility null if the comment is visible to all:

```
Comment comment = new Comment(null, "Test Comment",
jobin, null, new DateTime(), null,
new Visibility(Type.GROUP, "jira-developers"), null));
```

3. Add the comment using the `addComment` method:

```
jiraRestClient.getIssueClient()
.addComment(browsedIssue.getCommentsUri(), comment).claim();
```

For updating comments, use the same methods but pass the comment URI, update author name, and update time instead of null values.



Notice the `claim()` method invoked on the `Promise` object returned by the `addComment` method. This will make the call **synchronous-like**.

Remote user and group management

Let's now have a look at user and group management using remote APIs. This is really useful when we need to manage users and groups from a third-party application.

As mentioned at the beginning of this chapter, some methods are not supported by **JRJC** yet and this is an example.

In this case, we will see a few important **REST** methods that can be used for managing users and groups, along with the sample request and response data. There are other useful methods, which are not covered in this recipe, and you can find the full list at `https://docs.atlassian.com/jira/REST/latest`.

How to do it…

Let us look at the various user operations exposed by JIRA's REST API.

Creating a User

The following are the REST API method details for creating a user:

Method URL	**/rest/api/2/user**
Method type	POST
Parameters	None

Request data	• **key**: Key of the user (String). • **name**: Name of the user (String). • **password**: Password of the user (String). If not sent, JIRA will automatically generate a password. • **emailAddress**: E-mail of the user (String). • **displayName**: Full name of the user (String). • **notification**: Send notification or not (true/false). • **applicationKeys**: Array of applications for which the user needs access – jira-core,jira-software,jira-servicedesk.
Response code	• **201**: User created successfully • **400**: Request invalid • **401**: User not authenticated • **500**: User not created due to another error • **403**: Calling user does not have permission to create a user

The following is a sample input for creating a user:

```
{
     "name": "jtricks-user",
     "password": "abracadabra",
     "emailAddress": "user@jtricks.com",
     "displayName": "JTricks User",
    "applicationKeys": ["jira-core"]
}
```

After successful creation, you will get a response similar to the following:

```
{
     "self":"http://localhost:8080/jira/rest/api/2
     /user?username=jtricks-user",

     "key":"jtricks-user",
     "name":"jtricks-user",
     "emailAddress":"user@jtricks.com",
     "avatarUrls":{

        "16x16":"http://www.gravatar.com/avatar
        /1e8e9f2cc079fa9142a9c4a222d4064f?d=mm&s=16",

        "24x24":"http://www.gravatar.com/avatar
        /1e8e9f2cc079fa9142a9c4a222d4064f?d=mm&s=24",

        "32x32":"http://www.gravatar.com/avatar
        /1e8e9f2cc079fa9142a9c4a222d4064f?d=mm&s=32",
```

```
            "48x48":http://www.gravatar.com/avatar
            /1e8e9f2cc079fa9142a9c4a222d4064f?d=mm&s=48
        },
        "displayName":"JTricks User",
        "active":true,
        "timeZone":"America/New_York",
        "locale":"en_US",
        "groups":{
            "size":0,
            "items":[]
        },
        "applicationRoles":{
            "size":0,
            "items":[]
        },
        "expand":"groups,applicationRoles"
}
```

Updating a User

Updating a user is very similar:

Method URL	**/rest/api/2/user**
Method type	PUT
Parameters	• **username**: Name of the user • **key**: Key of the user Either **username** or **key** should be sent.
Request data	Same as **Create**
Response code	• **201**: User updated successfully • **400**: Request invalid • **401**: User not authenticated • **403**: Calling user does not have permission to edit a user • **404**: Calling user has permission but user does not exist on JIRA

Calling URL will include the `username` or `key` parameter, shown in the parameter column. For example, the following is the URL to update the user we created earlier:

```
http://localhost:8080/jira/rest/api/2/user?username=jtricks-user
```

As sample input will be:

```
{
 "displayName": "JTricks Test User"
}
```

And the response will be similar to what we saw for create.

Adding a User to an application

The following is the REST API method details for adding a user to an application. An application here refers to JIRA Core, JIRA Software or JIRA Service Desk:

Method URL	**/rest/api/2/user/application**
Method type	POST
Parameters	• **username**: Name of the user • **applicationKey**: Key of the application for which the user needs access, in other words, jira-core,jira-software or jira-servicedesk
Request data	None
Response code	• **201**: User added successfully • **400**: Request invalid • **401**: User not authenticated • **403**: Calling user does not have permission to add a user to the application

Calling url will include the `username` and `applicationKey` parameters, shown in the parameter column. For example, the following is the url to add the user we created earlier to **JIRA Software**:

```
http://localhost:8080/jira/rest/api/2/user/application?username=jtricks-user&ap
plicationKey=jira-software
```

Removing a User from an application

Removing a user from an application is similar:

Method URL	**/rest/api/2/user/application**
Method type	DELETE
Parameters	• **username**: Name of the user • **applicationKey**: Key of the application from which the user needs to be removed, in other words, jira-core,jira-software or jira-servicedesk
Request data	None
Response code	• **204**: User removed successfully • **400**: Request invalid • **401**: User not authenticated • **403**: Calling user does not have permission to remove the user from the application

Calling url will include the `username` and `applicationKey` parameters, shown in the parameter column. For example, the following is the url to remove the user we created earlier from **JIRA Software**:

```
http://localhost:8080/jira/rest/api/2/user/application?username=jtricks-user&applicationKey=jira-software
```

Deleting a User

Deleting a user uses the same resource path as that of creating and updating a user. The method type is DELETE though. The following are the details:

Method URL	**/rest/api/2/user**
Method type	DELETE
Parameters	• **username**: Name of the user • **key**: Key of the user. Either **username** or **key** should be sent.
Request data	None

Response code	• **204**: User deleted successfully • **400**: Request invalid • **401**: User not authenticated • **403**: Calling user does not have permission to delete user • **404**: Calling user has permission but the user does not exist

Calling url will include the `username` or `key` parameter, shown in the parameter column. For example, the following is the url to delete the user we created earlier:

```
http://localhost:8080/jira/rest/api/2/user?username=jtricks-user
```

Creating a Group

The following are the REST API method details for creating a group:

Method URL	**/rest/api/2/group**
Method type	POST
Parameters	None
Request data	• **name**: Name of the group (String)
Response code	• **201**: Group created successfully • **400**: Empty group name or group already exists • **401**: User not authenticated • **403**: Calling user does not have permission to create a group

The following is a sample input for creating a group:

```
{
    "name": "jtricks-group"
}
```

After successful creation, you will get a response similar to the following:

```
{
      "name":"jtricks-group",
      "self":"http://localhost:8080/jira/rest/api/2
       /group?groupname=jtricks-group",
      "users":{
         "size":0,
         "items":[],
         "max-results":50,
         "start-index":0,
         "end-index":0
```

```
        },
        "expand":"users"
    }
```

Adding a user to a Group

The following are the REST API method details for adding a user to an existing group:

Method URL	**/rest/api/2/group/user**
Method type	POST
Parameters	• **groupname**: Name of the group
Request data	• **name**: Name of the user to be added (String)
Response code	• **201**: User added to group successfully • **400**: Empty group name or user already belongs to the group • **401**: User not authenticated • **403**: Calling user does not have administrative permissions • **404**: Group or user not found

Calling url will include the `groupname` parameter, shown in the parameter column. For example, the following is the url to add a user to the group we created earlier:

```
http://localhost:8080/jira/rest/api/2/group/user?groupname=jtricks-group
```

The following is a sample input for adding the user:

```
{
    "name": "jobinkk"
}
```

After successfully adding the user **jobinkk** to **jtricks-group**, you will get a response similar to the following:

```
{
       "name":"jtricks-group",
       "self":"http://localhost:8080/jira/rest/api/2
       /group?groupname=jtricks-group",
       "users":{
          "size":1,
          "items":[],
          "max-results":50,
          "start-index":0,
          "end-index":0
```

```
        },
        "expand":"users"
    }
```

As you can see, the **size** element in the JSON response now shows 1 instead of 0.

Getting users in a Group

The following are the REST API method details for getting all the users in a group:

Method URL	**/rest/api/2/group/member**
Method type	GET
Parameters	• **groupname**: Name of the group. • **includeInactiveUsers**: Return inactive users or not (true/false). • **startAt**: The index of the first user in the group to return (0 based). Used for pagination. • **maxResults**: The maximum number of users to return (max 50, default 50). Only **groupname** is mandatory.
Request data	None
Response code	• **200**: Users returned successfully • **400**: Empty group • **401**: User not authenticated • **403**: Calling user does not have administrative permissions • **404**: Group not found

Calling url will include the `groupname` and can optionally include `includeInactiveUsers`, `startAt` and `maxResults` parameters, shown in the parameter column. For example, the following is the url to get users from the group we created earlier:

```
http://localhost:8080/jira/rest/api/2/group/member?groupname=jtricks-group
```

This request will be similar to the following:

```
http://localhost:8080/jira/rest/api/2/group/member?includeInactiveUsers=false&m
axResults=50&groupname=jtricks-group&startAt=0
```

The following is a sample output for this request:

```
{
    "self":"http://localhost:8080/jira/rest/api/2
     /group/member?includeInactiveUsers=false&maxResults
     =50&groupname=jtricks-group&startAt=0",

     "maxResults":50,
     "startAt":0,
     "total":1,
     "isLast":true,
     "values":[{
         "self":"http://localhost:8080/jira/rest/api/2
         /user?username=jobinkk",
         "name":"jobinkk",
         "key":"jobinkk",
         "emailAddress":"jobinkk@gmail.com",
         "avatarUrls":{

             "48x48":"http://www.gravatar.com/avatar
            /4cbc21c676a9b226009ca828f608897f?d=mm&s=48",

             "24x24":"http://www.gravatar.com/avatar
             /4cbc21c676a9b226009ca828f608897f?d=mm&s=24",

             "16x16":"http://www.gravatar.com/avatar
             /4cbc21c676a9b226009ca828f608897f?d=mm&s=16",

            "32x32":"http://www.gravatar.com/avatar
            /4cbc21c676a9b226009ca828f608897f?d=mm&s=32"
         },
      "displayName":"JobinKuruvilla", "active":true,
      "timeZone":"America/New_York" }]
}
```

As you can see, theresponse contains the list of users in `jtricks-group`, only one user in our example. Even if the response can only have a maximum of 50 users, you can use the `total` element to figure out the total number of users in the group and then use the `startAt` and `maxResults` parameters to query the remaining users.

For example, the next 50 users can be queried using the following url:

```
http://localhost:8080/jira/rest/api/2/group/member?includeInactiveUsers=false&m
axResults=50&groupname=jtricks-group&startAt=50
```

Removing a user from a Group

The following are the REST API method details for removing a user from a group:

Method URL	**/rest/api/2/group/user**
Method type	DELETE
Parameters	• **groupname**: Name of the group • **username**: Name of the user to be removed
Request data	None
Response codes	• **200**: Users removed successfully • **400**: Empty group name • **401**: User not authenticated • **403**: Calling user does not have administrative permissions • **404**: Group or user not found

Calling url will include the `groupname` and `username` parameters, shown in the parameter column. For example, the following is the url to remove the user we added earlier from the `jtricks-group`:

```
http://localhost:8080/jira/rest/api/2/group/user?groupname=jtricks-group&username=jobinkk
```

Deleting a Group

The following are the REST API method details for deleting a group:

Method URL	**/rest/api/2/group**
Method type	DELETE
Parameters	• **groupname**: Name of the group. • **swapGroup**: Optional parameter. This group name will be used to transfer restrictions such as comment and worklog visibility to that group before the original group is deleted.
Request data	None

Response codes	• **200**: Group deleted successfully • **400**: Empty group name • **401**: User not authenticated • **403**: Calling user does not have administrative permissions • **404**: Group or user not found

Calling url will include the `groupname` and can optionally have a `swapGroup` parameter, shown in the parameter column. For example, the following is the url to delete the `jtricks-group`:

```
http://localhost:8080/jira/rest/api/2/group?groupname=jtricks-group
```

If there was content, such as comments and worklogs, restricted to this group, it will be hidden from all users after the group is deleted. In order to avoid this, we can specify the `swapGroup` parameter, as shown here:

```
http://localhost:8080/jira/rest/api/2/group?groupname=jtricks-group&swapGroup=new-group
```

Once the `jtricks-group` is deleted, all the content restricted to that group will now be visible to members of `new-group`.

Progressing an issue in workflow

This is something everyone wants to do when JIRA is integrated with third-party applications. The status of an issue needs to be changed for various use cases and the right way to do this is to progress the issue through its workflow.

Progressing will move the issue to the appropriate statuses and will fire the appropriate post functions and events. In this recipe, we will see how to do this.

Getting ready

As usual, create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

JRJC exposes the `transition` method inside the `IssueRestClient` to progress an issue through its workflow. The following are the steps to do it:

1. Identify the list of transitions available for the issue:

```
Promise<Iterable<Transition>> transitions =
issueClient.getTransitions(browsedIssue.getTransitionsUri());
```

2. Iterate over the list and identify the transition to be performed by name as follows. You can skip this step if you know the ID and use it directly in step 5:

```
private Transition getTransitionByName(Iterable
<Transition> transitions, String transitionName) {
  for (Transition transition : transitions) {
    if (transition.getName().equals(transitionName)) {
     return transition;
    }
  }
  return null;
}
Transition resolveTransition =
getTransitionByName(transitions.get(), "Resolve Issue");
```

3. Populate the list of fields required for the transition. For example, the resolution field required for the "Resolve Issue" transition can be populated as follows:

```
Collection<FieldInput>fieldInputs = Arrays.asList(new
FieldInput("resolution",
ComplexIssueInputFieldValue.with("name", "Fixed")));
```

4. Create an optional comment:

```
Comment comment = Comment.valueOf("Resolving issue using JRJC");
```

5. Create a `TransitionInput` object using the transition ID, fieldInputs, and comment:

```
TransitionInputtransitionInput = new
 TransitionInput(resolveTransition.getId(), fieldInputs, comment);
```

6. Invoke the `transition` method on `IssueRestClient`:

```
issueClient.transition(browsedIssue.getTransitionsUri(),
transitionInput);
```

Thus the issue can be transitioned to a given status.

Searching issues

Searching an issue is another important operation in JIRA. As you would expect, JRJC provides a `SearchRestClient` to do this.

Getting ready

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

Similar to other REST clients, you can retrieve the `SearchRestClient` from `JiraRestClient` using a getter method, as shown below:

```
SearchRestClientsearchClient = jiraRestClient.getSearchClient();
```

`SearchRestClient` uses **JIRA Query Language** (**JQL**) to search for issues in JIRA. If you already know JQL, you can search for the issues using the `searchJql` method, as shown below:

```
Promise<SearchResult> result = searchClient.searchJql("project = DEMO");
```

We can then iterate on the search results and retrieve the field values, as we have seen in the *Working with issues* recipe:

```
SearchResultresultObject = result.get();
Iterable<Issue> issues = resultObject.getIssues();
for (Issue issue : issues) {
    System.out.println("Got issue:"+issue.getKey());
}
```

It is also possible to implement paging by passing `maxResults` and `startAt` parameters, along with the set of field names that we want to retrieve:

```
Set<String> fields = new HashSet<String>();
fields.add("*all");
Promise<SearchResult>
result = searchClient.searchJql("project = DEMO", 1, 0, fields);
```

The above example retrieves the first issue but we can adjust the `maxResults` and `startAt` parameters, as needed. For `fields`, we can specify `*`all for all fields or *navigable (which is the default value, used when null is given) which will cause only navigable fields to be included in the result. To ignore a specific field, you can use "-" before the field's name. Note that the following fields are required: summary, issuetype, created, updated, project, and status.

`SearchRestClient` also exposes a method to retrieve the favorite filters as shown below:

```
Promise<Iterable<Filter>>
filters = searchClient.getFavouriteFilters();
Iterable<Filter>filterList = filters.get();
for (Filter filter : filterList) {
    System.out.println("Got filter:"
    + filter.getName()
    + " with JQL:"
    + filter.getJql());
}
```

It is also possible to get a filter by **id** or **URI** using the `getFilter` method.

From a filter, we can get JQL using the `getJql` method and use it for searching, as we detailed earlier.

Managing versions

We have seen how to add versions as **fix for versions** or **affected versions** on an issue. But how do we create those versions using REST? In this recipe, we will see how to create **versions** in a project and manage them using JRJC!

Getting ready

As usual, create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

A new version can be added into a project as follows:

1. Create a `VersionInput` object with the necessary details:

```
VersionInputversionInput = new VersionInput
("DEMO", "JRJC", "Test", new DateTime(), false, false);
```

2. Use the `createVersion` method on `VersionRestClient`:

```
Promise<Version> version =
jiraRestClient.getVersionRestClient().createVersion(versionInput);
```

Once a **version** is created, you can retrieve it any time using the version URI:

```
Promise<Version> version =
jiraRestClient.getVersionRestClient().getVersion(versionURI);
```

Given a version, you can update it by sending a new `VersionInput` object with updated parameters. For example, if you want to "release" the previously created version, you can do it as follows:

```
jiraRestClient.getVersionRestClient()
.updateVersion(version.get().getSelf(),
new VersionInput("DEMO", "JRJC", "Test", new DateTime(), false, true));
```



Notice that the last parameter, the `isReleased` flag, is changed to **true**.

If you want to move the `version` to a specific position, you can do that as follows:

```
jiraRestClient.getVersionRestClient().moveVersion(version.get().getSelf(),
VersionPosition.LAST);
```

You can move it after another version for which you have the URI:

```
jiraRestClient.getVersionRestClient().moveVersion(version.get().getSelf(),
anotherVersionURI);
```

A version can be deleted using the `removeVersion` method. Here, we can specify a different version, which will be used to replace "Fix for version" in all the issues that have the current version as "Fix for version", and another version, which will be used to replace "Affected version" in all the issues that have the current version as "Affected version".

You can also choose to leave those two parameters as **null** to simply drop the "Fix for version" and "Affected version" values on the impacted issues:

```
jiraRestClient.getVersionRestClient()
.removeVersion(version.get().getSelf(),
moveFixIssuesToVersionUri, moveAffectedIssuesToVersionUri);
```

You can get the list of unresolved issues where this version is available as "Fix for version" as follows:

```
Promise<Integer>
number = jiraRestClient.getVersionRestClient()
.getNumUnresolvedIssues(version.get().getSelf());
```

This covers the most important methods used for managing versions in JIRA.

Managing components

Managing components using JRJC is very similar to managing versions.

Getting ready

Create a JIRA REST client as mentioned in the *Writing Java client for REST API* recipe.

How to do it…

A new component can be added into a project as follows:

1. Create a `ComponentInput` object with the necessary details:

```
ComponentInputcomponentInput = new ComponentInput("JRJC",
 "Test", "jobinkk", AssigneeType.COMPONENT_LEAD);
```

2. Use the `createComponent` method on `ComponentRestClient`:

```
Promise<Component>component = jiraRestClient
.getComponentRestClient()
.createComponent("DEMO", componentInput);
```

Once a component is created, you can retrieve it any time using the component URI:

```
Promise<Component>component=
jiraRestClient.getComponentRestClient().getComponent(componentURI);
```

Given a component, you can update it by sending a new `ComponentInput` object with updated parameters. For example, if you want to change the project lead to "test", you can do it as follows:

```
jiraRestClient.getComponentClient()
.updateComponent(component.get().getSelf(),
new ComponentInput("JRJC", "Test", "test", AssigneeType.COMPONENT_LEAD));
```

A component can be deleted using the `removeComponent` method. Here, we need to specify another component to which all the issues in this component will be moved:

```
jiraRestClient.getComponentClient()
.removeComponent(component.get().getSelf(), moveIssueToComponentUri);
```

You can get the list of issues related to this component as follows:

```
Promise<Integer>
number = jiraRestClient.getComponentClient()
.getComponentRelatedIssuesCount(component.get().getSelf());
```

This covers the most important methods in managing components in JIRA.

Remote administration methods

Before we wind up the various useful methods using remote APIs, we can have a look at the administration methods. In this recipe, we will just concentrate on some methods revolving around the creation of projects and permissions. The remaining methods are an easy read once you have a fair idea of the ones we are discussing in this recipe.

Since JRJC doesn't support all the administration methods, let us look at the required REST method details, along with sample request and response data, as we did in the *Remote user and group management* recipe.

How to do it…

Let us consider a simple scenario to explain some of the administrative REST APIs:

1. Create a simple permission scheme. The permission scheme will grant **Project Admin** permissions to the **jira-administrators** group and will ignore the rest of the permissions.
2. Create a project.
3. Update the project with the new permission scheme, created in step 1.
4. Retrieve the project roles.
5. Add a user to the **Administrators** project role.

Creating a Permission Scheme

The following are the REST API method details for creating a permission scheme in JIRA:

Method URL	**/rest/api/2/permissionscheme**
Method type	POST
Parameters	• expand
Request data	• **name**: Name of the permission scheme (String). • **description**: Description of the permission scheme (String). • **permissions**: List of permissions. This will be an array of objects with the following elements: • **holder**: This is an object with the following elements: • **type**: Type of holder – group, role, and so on. • **parameters**: Value of holder. For example, *jira-administrators* if the type is a *group*. • **permission**: Appropriate project permission, for example, ADMINISTER_PROJECTS.
Response codes	• **201**: Permission scheme created successfully • **401**: User not authenticated • **403**: Calling user does not have permission to create a permission scheme

The following is a sample input for creating a permission scheme:

```
{
     "name": "JTricks permission scheme",
     "description": "Demo Permission scheme",
     "permissions": [{
         "holder": {
             "type": "group",
             "parameter": "jira-administrators"
         },
         "permission": "ADMINISTER_PROJECTS"
     }]
}
```

After successful creation, you will get a response similar to the following:

```
{
     "expand":"permissions,user,group,projectRole,field,all",
     "id":10000,

     "self":"http://localhost:8080/jira/rest/api/2
      /permissionscheme/10000",

     "name":"JTricks permission scheme",
     "description":"Demo Permission scheme"
}
```

Creating a Project

The following are the REST API method details for creating a JIRA project:

Method URL	**/rest/api/2/project**
Method type	POST
Parameters	none

Request data	• **key**: Project key (String) • **name**: Name of the permission scheme (String) • **projectTypeKey**: Type of project, for example, business (String) • **projectTemplateKey**: Project template key, if we need to specify one (String) • **description**: Description of the project (String) • **lead**: Project lead's username (String) • **url**: URL of the project (String) • **assigneeType**: Type of default assignee – PROJECT_LEAD or UNASSIGNED (String) • **avatarId**: ID of the avatar (Integer) • **issueSecurityScheme**: ID of issueSecurityScheme (Integer) • **permissionScheme**: ID of permissionScheme (Integer) • **notificationScheme**: ID of notificationScheme (Integer) • **categoryId**: ID of the project category (Integer)
Response codes	• **201**: Project created successfully • **400**: Request not valid or project could not be created • **401**: User not authenticated • **403**: Calling user does not have permission to create a project

The following is a sample input for creating a permission scheme:

```
{
     "key": "JTRICKS",
     "name": "JTricks Project",
     "projectTypeKey": "business",
     "description": "Example Project description",
     "lead": "jobinkk",
     "url": "http://www.j-tricks.com",
     "assigneeType": "PROJECT_LEAD"
}
```

After successful creation, you will get a response similar to the following:

```
{
     "self":"http://localhost:8080/jira/rest/api/2/project/10200",
     "id":10200,
     "key":"JTRICKS"
}
```

Updating a Project

Updating a project is very similar to creating one. The only difference is that the method type is `PUT` and we need to pass only the data that we need to update:

Method URL	**/rest/api/2/project/${projectIdOrKey}**
Method type	PUT
Parameters	• expand
Request data	Same as Create
Response codes	• **201**: Project updated successfully • **400**: Request not valid or project could not be created • **401**: User not authenticated • **403**: Calling user does not have permission to create a project • **404**: Project does not exist

For example, if we want to update the permission scheme of the project we created above with the permission scheme we created before that, we can send a **PUT** request to the below URL:

```
http://localhost:8080/jira/rest/api/2/project/JTRICKS
```

And the input will be:

```
{
    "permissionScheme": 10000
}
```

Here, 1000 is the id of the permission scheme we created earlier.

Retrieving the project roles

The following are the REST API method details for retrieving all the project roles:

Method URL	**/rest/api/2/project//${projectIdOrKey}/role**
Method type	GET
Parameters	none
Request data	none
Response codes	• **200**: Roles returned successfully • **404**: Project not found or user does not have permissions

The following is a sample url to retrieve the project roles from the project we created earlier:

```
http://localhost:8080/jira/rest/api/2/project/JTRICKS/role
```

And a sample output would be:

```
{
     "Developers":"http://localhost:8080/jira/rest/api/2
     /project/10202/role/10001",

    "Administrators":"http://localhost:8080/jira/rest/api/2
    /project/10202/role/10002",

    "Users":"http://localhost:8080/jira/rest/api/2
    /project/10202/role/10000"
}
```

Add actors to a project role

The following are the REST API method details for adding a user or group (called "role actors") to a project role retrieved in the previous method:

Method URL	**/rest/api/2/project/{$projectIdOrKey}/role/${roleId}**
Method type	POST
Parameters	none
Request data	Object with the actor type (**user** or **group**) and an array of actors (usernames or group names, as appropriate) For example: { "user" : ["jobinkk"] } { "group" : ["jira-developers"] }
Response codes	• **200**: Actor added to role successfully • **404**: Actor could not be added to the role

The following is a sample url to add a user to the "Administrators" role, in the project we created earlier:

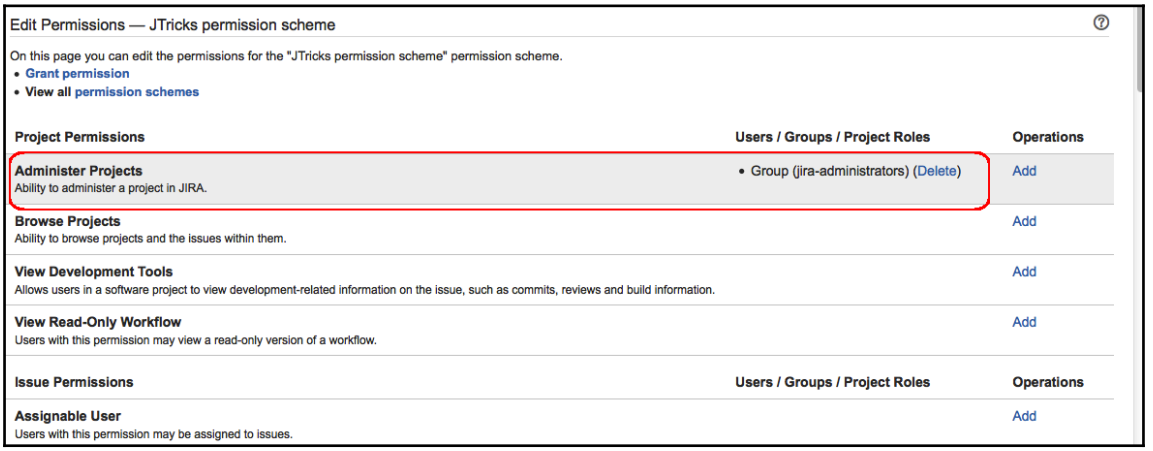
```
http://localhost:8080/jira/rest/api/2/project/JTRICKS/role/10002
```

And the input would be:

```
{
    "user" : ["jobinkk"]
}
```

How it works…

Once the permission scheme is created, it appears in JIRA as follows:



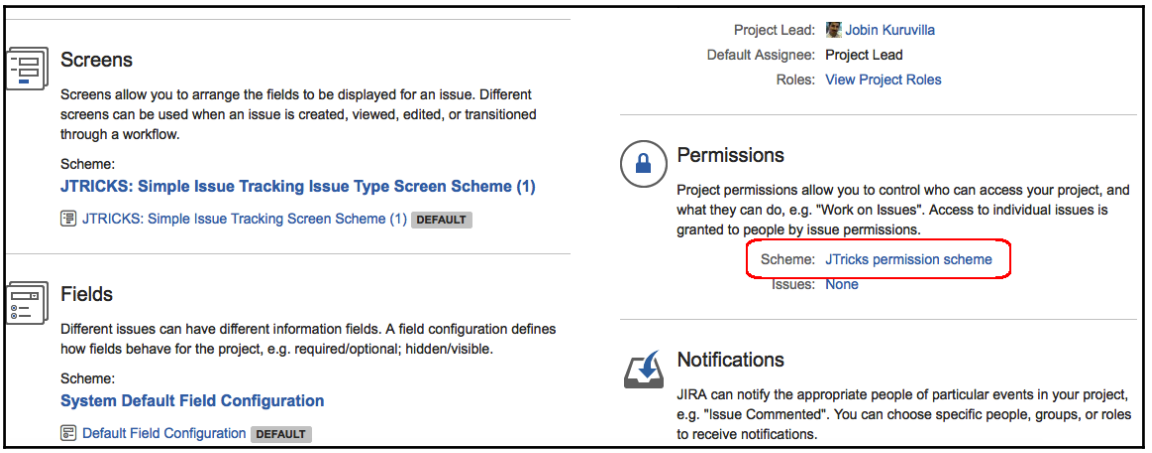
Here, a single permission is added, that is, to administer the project. We can add the rest in a similar fashion.

And the project, after creation, will be shown as follows:



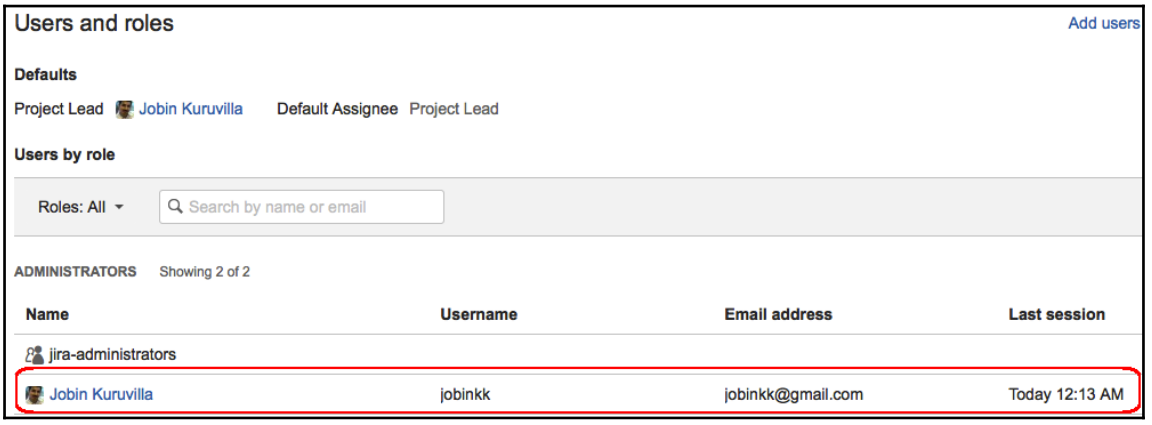
The project is populated with the default JIRA schemes, as we didn't pass any of them during the creation.

After the project is updated using the new permission scheme, it appears as follows:



As you can see, the permission scheme is changed while the other schemes are the default ones.

After adding the actors, the project role screen is reflected with the change, as shown below:



Similarly, we can use the other REST methods to perform different administrative operations.

Exposing services and data entities as REST APIs

So far, we have seen various REST APIs to perform different operations in JIRA. What about operations that are not supported by REST? That little something which prevents you from integrating JIRA with your third-party app? That is where the REST plugin module type comes handy. Using the REST plugin module, services or data can be exposed to the outside world.

In this recipe, we will see how to expose the `getProjectCategories` method we have used as examples in the previous recipes using the REST interface.

Getting ready

Create a skeleton plugin using the Atlassian plugin SDK.

How to do it…

The following is a step-by-step procedure to create a REST plugin to expose the `getProjectCategories` method:

1. Add the Maven dependencies require for REST to the `pom.xml` file. This will be automatically added, if you use the Atlassian plugin SDK to create the module:

```
<dependency>
 <groupId>javax.ws.rs</groupId>
 <artifactId>jsr311-api</artifactId>
 <version>1.1.1</version>
 <scope>provided</scope>
</dependency>

<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>servlet-api</artifactId>
 <version>2.4</version>
 <scope>provided</scope>
</dependency>

<dependency>
 <groupId>javax.xml.bind</groupId>
 <artifactId>jaxb-api</artifactId>
```

```
 <version>2.1</version>
 <scope>provided</scope>
</dependency>

<dependency>
 <groupId>com.atlassian.plugins.rest</groupId>
 <artifactId>atlassian-rest-common</artifactId>
 <version>1.0.2</version>
 <scope>provided</scope>
</dependency>
```

Note that all the dependencies are of the scope **provided**, since they are already available in the JIRA runtime.

2. Add the REST plugin module into `atlassian-plugin.xml`:

```
<rest name="Category Resource"
i18n-name-key="category-resource.name"
key="category-resource" path="/jtricks" version="1.0">
    <description key="category-resource.description">
        The Category Resource Plugin
    </description>
</rest>
```

Here, the **path** and **version** define the full path where the resources will be available after the plugin is deployed. In this case, the full path will become `BASE_URL/rest/jtricks/1.0/`, where **BASE_URL** is the JIRA base URL.

3. Define the data that will be returned to the client. **JAXB** annotations are used to map these objects to XML and JSON formats.In our example, the `getCategories` method should return a list of `Category` objects and hence we need to define a `Categories` object and a `Category` object, the former containing a list of the latter. For both the objects, we should use the annotations.
4. The `Category` object is defined as follows:

```
@XmlRootElement public static class Category {
    @XmlElement private Long id;
    @XmlElement private String name;public Category() {
    }
    public Category(Long id, String name) {
        this.id = id; this.name = name;
    }
}
```



Make sure the annotations are used properly. The `@XmlRootElement` annotation maps a **class** or an **Enum** type to an XML element and is used for the categories in this case. `@XmlElement` maps a **property** or **field** to an XML element. Other annotations available are `@XmlAccessorType` and `@XmlAttribute`, used for controlling whether fields or properties are serialized by default and mapping a property or field to an XML attribute respectively.
The details can be read at:
`http://jaxb.java.net/nonav/jaxb20-pfd/api/javax/xml/bind/annotation/package-summary.html`.
Make sure a public **non-argument** constructor is available so as to render the output properly when accessed via the direct URL. Also, note that only the annotated elements will be exposed via the REST API.

5. Define the `Categories` object:

```
@XmlRootElement public static class Categories {
    @XmlElement private List<Category> categories;
    public Categories() {
    }
    public Categories(List<Category> categories) {
         this.categories = categories;
    }
}
```

   The same rules apply here as well.

6. Create the `Resource` class. On the package level or the class level or the method level, we can have `@Path` annotations to define the path where the resource should be available. If it is available on all the levels, the final path will be a cumulative output.
   This means that if you have `@Path("/X")` at package level, `@Path("/Y")` at class level, and `@Path("/Z")` at method level, the resource is accessed at:

```
BASE_URL/rest/jtricks/1.0/X/Y/Z
```

   Different methods can have different paths to differentiate between them. In our example, let us define a `/category` path at class level:

```
packagecom.jtricks;
 ..............
@Path("/category")
public class CategoryResource {
     ...................
```

```
}
```

7. Write the method to return the `Categories` resource:

```
@GET @AnonymousAllowed @Produces({
    MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response getCategories() throws SearchException {
     Collection<ProjectCategory> categories =
     this.projectManager.getAllProjectCategories();

     List<Category>categoryList = new ArrayList<Category>();
     for (ProjectCategory category : categories) {
        categoryList.add(new Category(category.getId(),
        category.getName()));
     }
     Response.ResponseBuilderresponseBuilder =
     Response.ok(new Categories(categoryList));
     return responseBuilder.build();
}
```

As you can see, the method doesn't have a `@Path` annotation and hence will be invoked at the `BASE_URL/rest/jtricks/1.0/category` URL. Here, we normally construct a `Categories` object with a simple bean class and then use the `ResponseBuilder` to create the response.

The `@GET` annotation mentioned earlier denotes that the class method will handle requests for a GET HTTP message.



Other valid annotations include POST, PUT, DELETE, and so on, and can be viewed in detail at `http://jsr311.java.net/nonav/javadoc/javax/ws/rs/package-summary.html`.

`@AnonymousAllowed` indicates that the method can be called without supplying user credentials. `@Produces` specifies the content types the method may return. The method can return any type if this annotation is absent. In our case, the method must return an **XML** or **JSON** object.Two other useful annotations are: `@PathParam` and `@QueryParam`. `@PathParam` maps a method variable to an **element** in the `@Path` whereas `@QueryParam` maps a method variable to a **query** parameter.
The following is how we use each of them:

a. @QueryParam: The following is an example of how `@QueryParam` is used:

```
@GET @AnonymousAllowed @Produces({
```

```
    MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    public Response getCategories(@QueryParam("dummyParam")
    String dummyParam) throws SearchException {
        System.out.println("This is just a dummyParam to
        show how parameters can be passed to REST methods:
        "+dummyParam);
         ..............
        return responseBuilder.build();
}
```

Here, we take a query parameter named `dummyParam`, which can then be used within our method. The resource will then be accessed as follows:

```
BASE_URL/rest/jtricks/1.0/category?dummyParam=xyz
```

In this case, you will see that the **xyz**value is printed into the console.

b. @PathParam: The following is an example of how `@PathParam` is used:

```
@GET
@AnonymousAllowed @Produces({
    MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML
})
@Path("/{id}")
public Response getCategoryFromId(@PathParam("id")
String id) throws SearchException {
    GenericValue category =
    this.projectManager.getProjectCategory(new Long(id));
    Response.ResponseBuilderresponseBuilder =
        Response.ok(new Category(category.getString("id"),
        category.getString("name")));
        return responseBuilder.build();
}
```

Let's say we want to pass the **id** of a category in the path and get the details of that `Category` alone; we can use the `PathParam` object, as shown above. In that case, the URL to this method will be as follows:

```
BASE_URL/rest/jtricks/1.0/category/10010
```

Here, 10010 is the category id passed into the previously described method.



When query parameters are used, the resource will not be cached by a proxy or your browser. So, if you are passing in an id to find some information about some sort of entity, then use a path parameter. This information will then be cached.

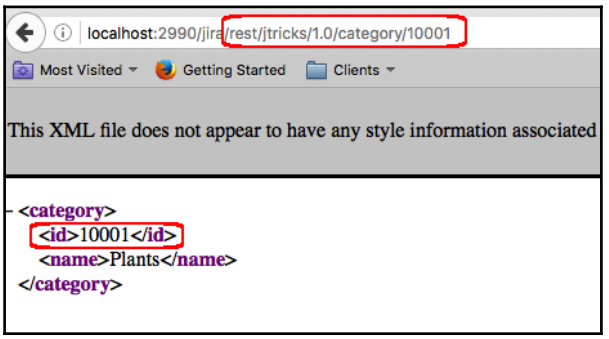
8. Package the plugin and deploy it.

How it works…

If you have deployed the plugin with both the `getCategories()` and `getCategoryFromId()` methods seen earlier, the list of categories can be retrieved at the URL: `BASE_URL/rest/jtricks/1.0/category`, as shown in the following screenshot:



The details of a particular category can be retrieved using the **id** in the `BASE_URL/rest/jtricks/1.0/category/10001` path, for example, as shown in the following screenshot:



Atlassian has published some guidelines at `http://confluence.atlassian.com/display/REST/Atlassian+REST+API+Design+Guidelines+version+1`, which is a very useful read before developing a production version of the REST service plugin. Also check out `http://confluence.atlassian.com/display/REST/REST+API+Developer+Documentation`for more details.

Using the REST API browser

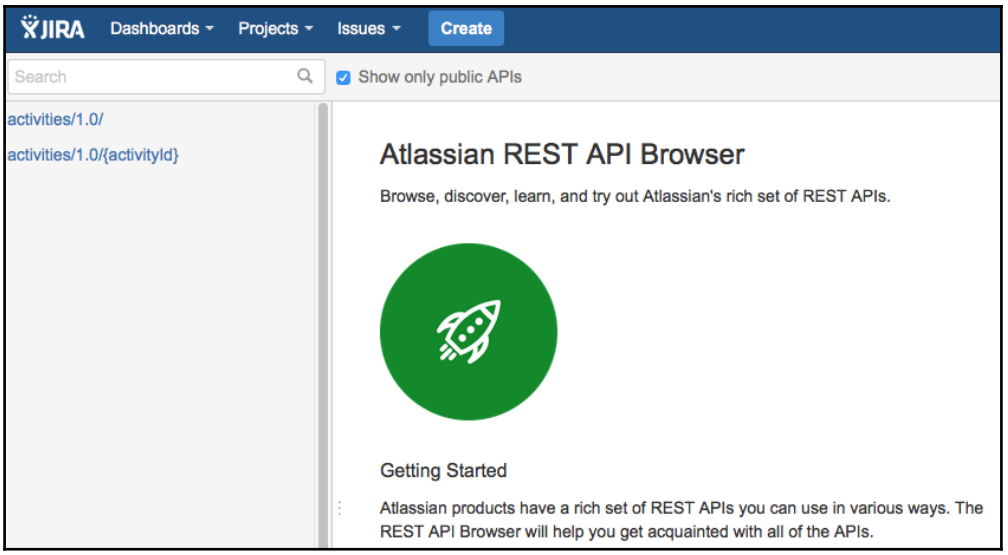
The Atlassian **REST API Browser** (**RAB**) is a tool for discovering the REST APIs and other remote APIs available in Atlassian applications, including JIRA. It shows the available REST resources, the parameters required for them, and so on, and allows us to make a test call using valid input data. The RAB will return the response, as appropriate.

Most importantly, the RAB shows both internal JIRA APIs and external APIs exposed by third-party plugins. This is very useful as most of the methods exposed by third-party plugins and even some of the JIRA internal methods don't have proper documentation.

How to do it…

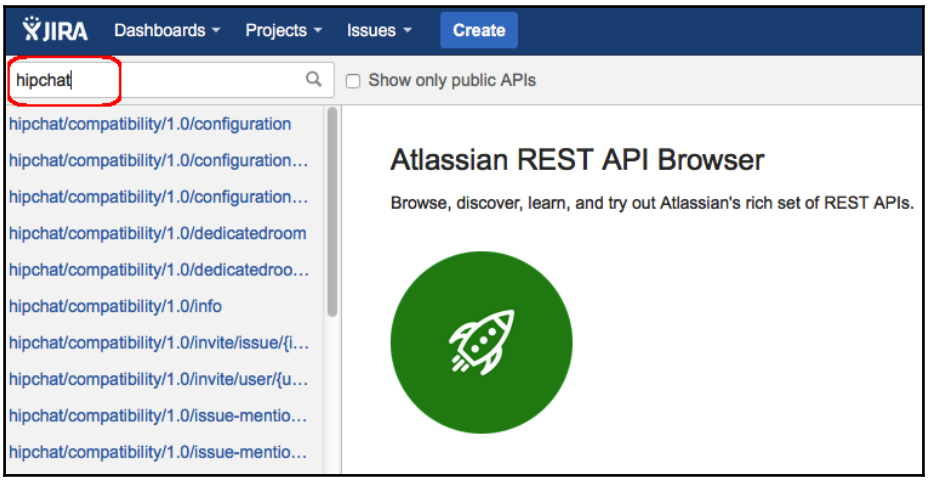
If you are running the Atlassian plugin SDK to develop a plugin, the RAB is already installed in the JIRA instance spun up by the SDK. If not, you can get the plugin from Atlassian Marketplace.

Once the RAB is installed in JIRA, you can navigate to **Administration**>**Add-ons**>**REST API Browser**. Clicking on the **REST API Browser** link will open up the RAB screen as shown below:

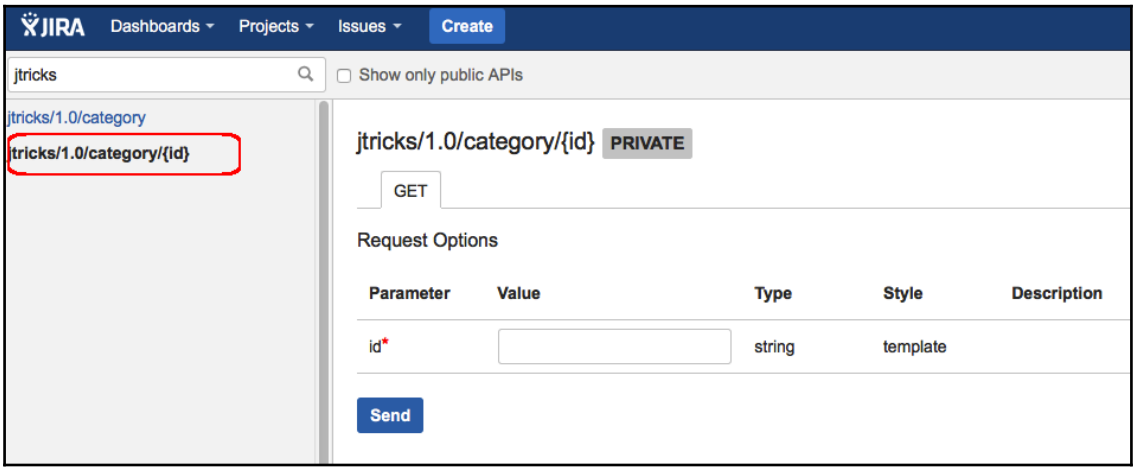


As you can see, only public APIs (APIs marked with `@PublicApi` annotation) are visible by default. When the checkbox to show only public APIs is unchecked, you will find all the available APIs (including the APIs marked with `@ExperimentalApi` annotation or APIs without any annotation) and you can use the search box to narrow down the results.

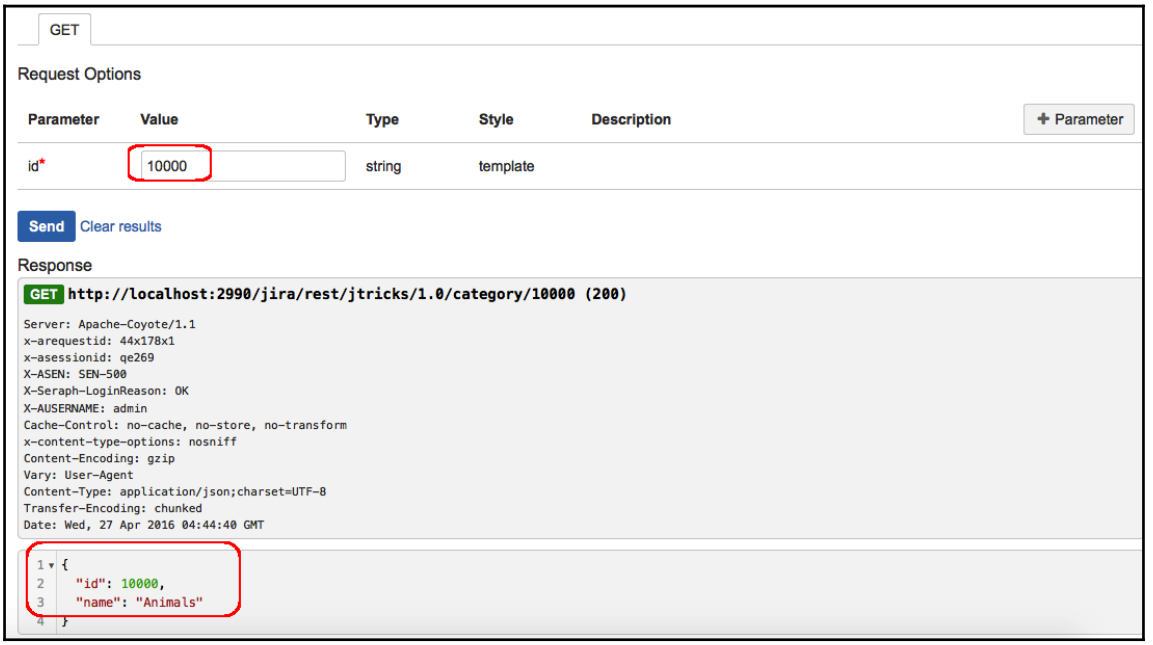
For example, searching for "hipchat" will show only resources with `hipchat` in the path, as shown below:



Once the relevant resource is found, clicking on it will open up the resource details, which include the API URL, HTTP method type, parameter definitions, and so on, as shown here:



We can also provide the required input and execute the method by clicking on the Send button. The RAB will execute the method and will return the response as shown here:



Similarly, we can use the RAB to dig deep into the public and private REST API definitions. More about the RAB can be read at `https://developer.atlassian.com/docs/developer-tools/using-the-rest-api-browser`.

Working with JIRA Webhooks

A webhook is a user-defined callback over HTTP. We can use webhooks to notify third-party applications when an event happens in JIRA. This is a great functionality as it can be used to integrate JIRA with third-party applications without even writing a single line of code.

All we need to do is to configure a webhook, based on the appropriate JIRA event, and then invoke the third-party application's remote API over HTTP.

For example, you can trigger a build in Bamboo or create a pull request in Stash when a JIRA issue moves from one state to another.

How to do it…

The following are the details required for configuring a webhook:

1. **Name**: Name of the webhook.
2. **URL**: URL to send the callback.
3. **Scope**: Scope of the webhook. We can apply it to all issues or to issues based on a JQL query.
4. **Events**: Issue events for which the webhook should be fired.

The following are the steps to configure a webhook in JIRA:

1. Navigate to **Administration** | **System** | **Advanced** | **WebHooks**.
2. Click on **Create a WebHook**.
3. Provide a valid url for the callback. This will be the remote application's url and you can even do variable substitution in this url.
   For example, we can send the JIRA issue id like this:

   ```
   http://j-tricks.example.com/webhook/${issue.key}
   ```

   JIRA will replace the `${issue.key}` with the appropriate issue key. At runtime, the URL that is called will be

`http://j-tricks.example.com/webhook/DEMO-1`, if the event is thrown on DEMO-1. The following is the full list of available variables in JIRA7. Please check the documentation for the current list:

```
${board.id}, ${issue.id}, ${issue.key},
${mergedVersion.id}, ${modifiedUser.key},
${modifiedUser.name}, ${project.id}, ${project.key},
${sprint.id}, ${version.id}
```

4. Provide an optional `description`.
5. Select the events for which the webhook needs to be fired. The following are the supported events:

Issue events	• Created • Updated • Deleted • Worklog changed, if a worklog is updated or deleted
Worklog, comment, user, project events	• Created • Updated • Deleted
Version events	• Created • Updated • Deleted • Released • Unreleased • Moved • Merged
JIRA feature events (in other words, when the feature is enabled or disabled in JIRA)	• Voting • Watching • Allow unassigned issues • Subtasks • Attachments • Issue linking • Time tracking

Sprint events	• Created • Updated • Deleted • Started • Closed
Rapid board events	• Created • Updated • Deleted

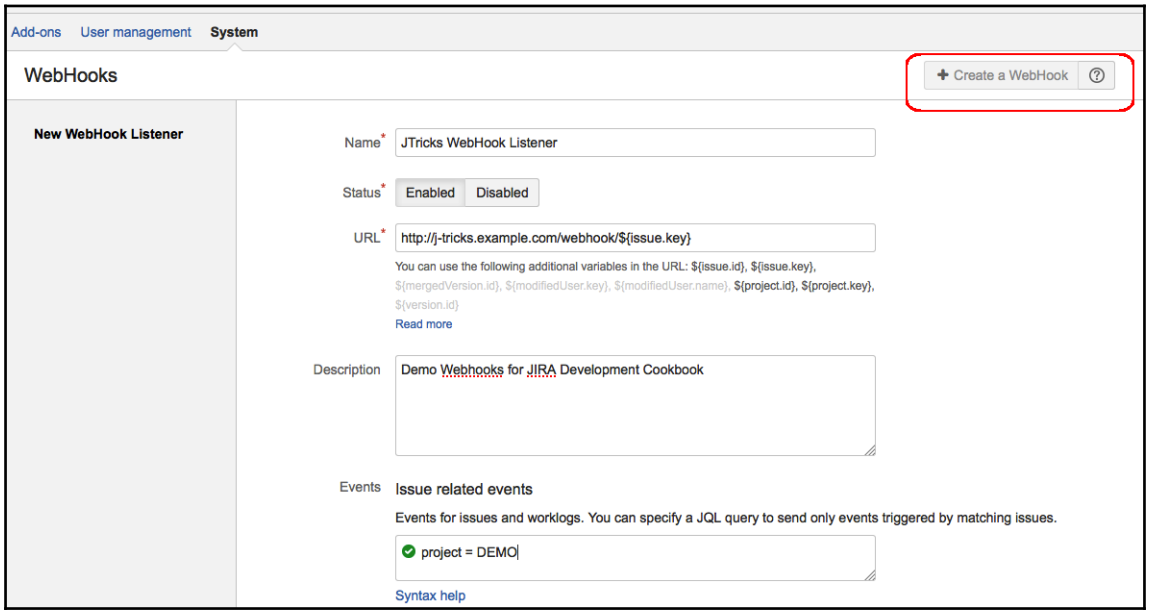
6. Optionally, provide a **JQL** in the box provided above Event selection to restrict the issues for which the events are triggered. This is very useful to set up different webhooks based on issue attributes such as projects, issuetypes or other field values.
7. Select the **Exclude body** option, if you do not need a callback from the third-party URL. By default, the webhook will send the request with a JSON callback when it is triggered.
   Callbacks can contain vital information, including webhook id, timestamp of the call, information about the event, the entity on which the event is fired, and so on. A callback for an issue-related event is structured like this:

```
{
    "timestamp" "event" "user": {
    // User shape - The same shape returned from the JIRA REST API
    when a user is retrieved, but without the active, timezone,
    or groups elements
    },
    "issue": {
    // Issue shape - The same shape returned from the
    JIRA REST API when an issue is retrieved with
    NO expand parameters
    },
    "changelog" : {
    // Changelog shape - The shape of the changelog you would
    retrieve from a JIRA issue, but without the user and timestamp
    },
     "comment" : {
    // Comment shape - The same shape returned from the
    JIRA REST API when a comment is retrieved
    }
}
```

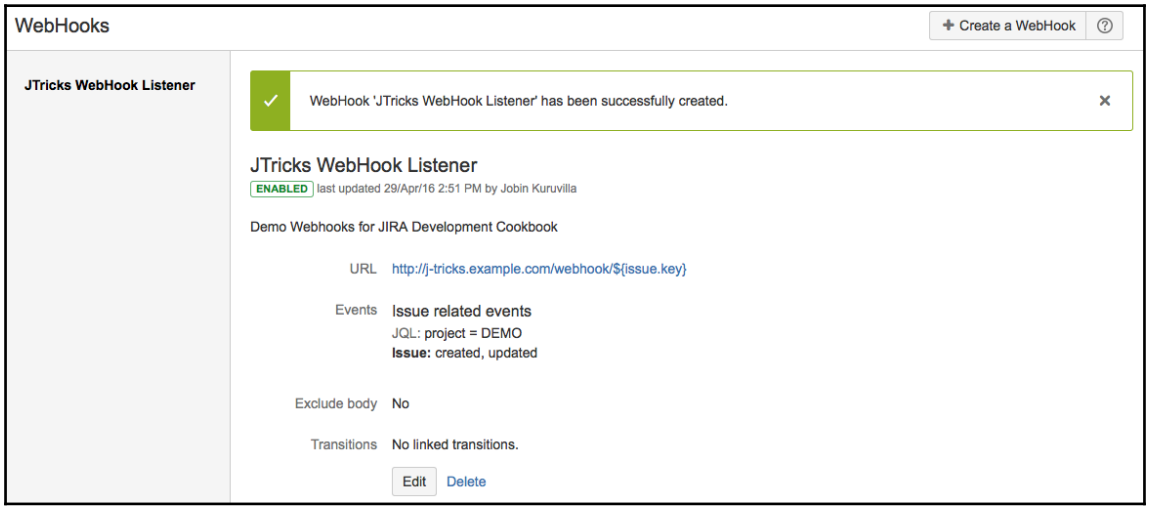
8. Click **Create**.

How it works…

The **Create a WebHook** screen will look like the following:



Once created, you can find the existing webhooks and edit them in the **WebHooks** page, as shown here:



When an event matching the list of events selected in the webhook configuration is fired in JIRA, it calls to the remote application URL. Before making the call, JIRA does the variable substitution in the URL, if needed, and includes the JSON callback data, if not excluded using the **Exclude Body** option.

If the remote application doesn't catch the webhook POST, JIRA will retry three times and will then log an error in the `atlassian-jira.log` file.

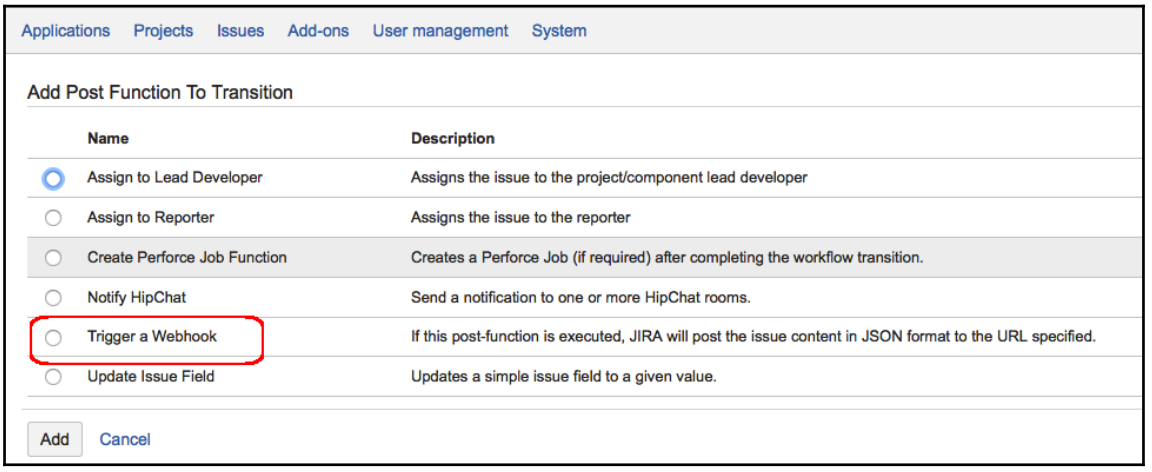
More information on webhooks can be found at:

`https://developer.atlassian.com/jiradev/jira-apis/webhooks`

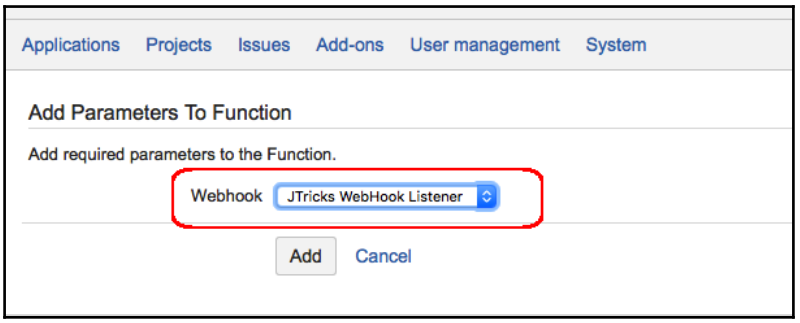
There's more…

It is possible to associate a configured webhook with a workflow post function. The issue transition will then trigger the webhook.

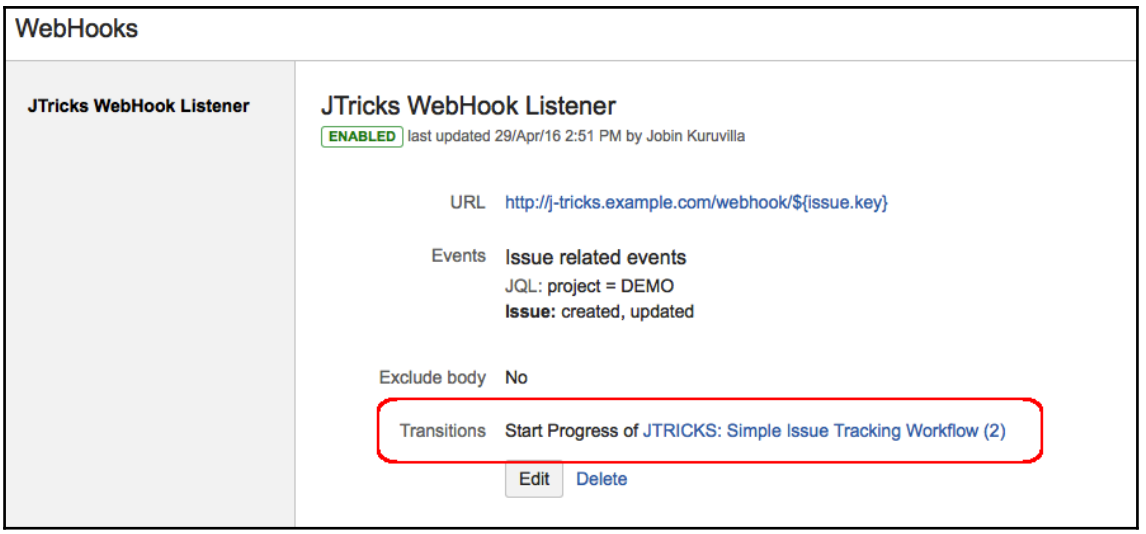
Adding a webhook to the transition is fairly straightforward. We can use the Trigger a **Webhook** post function, as shown here:



We can pick up the appropriate webhook in the next screen, as shown here:



Once a webhook is associated with a workflow, you can find it under the webhook details, as shown here:



If a webhook is associated with the event and is part of a workflow, then the webhook might be triggered twice- once for the event and once for the workflow transition.

Also, you won't be able to delete a webhook, as long as it is associated with a workflow.

10
Dealing with the JIRA Database

In this chapter, we will cover:

- Extending JIRA databases with a custom schema
- Accessing database entities from plugins
- Persisting plugin information in the JIRA database
- Using Active Objects to store data
- Accessing JIRA configuration properties
- Getting a database connection for JDBC calls
- Migrating a custom field from one type to another
- Retrieving issue information from a database
- Retrieving custom field details from a database
- Retrieving permissions on issues from a database
- Retrieving workflow details from a database
- Updating issue status in a database
- Retrieving users and groups from a database
- Dealing with change history in a database

Introduction

We have already seen in `Chapter 2`, *Understanding the Plugin Framework*, that JIRA uses the Ofbiz suite's **Entity Engine** module to deal with database operations. **OfBiz** stands for **Open for Business** and the **Ofbiz Entity Engine** is a set of tools and patterns used to model and manage entity-specific data.

As per the definition from standard entity-relationship modeling concepts of a relational database management system (RDBMS), an entity is a piece of data defined by a set of fields and a set of relations to other entities.

In JIRA, these entities are defined in two files, `entitygroup.xml` and `entitymodel.xml`, both residing in the `WEB-INF/classes/entitydefs` folder. `entitygroup.xml` stores the entity names for a previously defined group. If you look at the file, you will see that the default group in JIRA is named `default` and you will find this defined in the entity configuration file, which we will see in a moment. `entitymodel.xml` holds the actual entity definitions, the details of which we will see in the recipes.

The entity configuration is defined in `entityengine.xml`, residing in the `WEB-INF/classes` folder. It is in this file that the delegator, transaction factory, field types, and so on are defined. We won't be touching this file, as we did in earlier JIRA versions, because the database configuration is now done in `dbconfig.xml` residing in the JIRA**Home** folder.

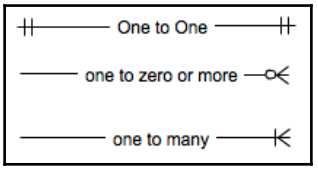
More about connecting to various other databases can be found at:
`https://confluence.atlassian.com/jira/connecting-jira-to-a-database-185729615.html`

Read more about entity modelling concepts at:
`https://cwiki.apache.org/confluence/display/OFBIZ/Entity+Engine+Guide`

In this chapter, we will also see glimpses of the JIRA database architecture, which is also explained in detail at:
`https://developer.atlassian.com/jiradev/jira-platform/jira-architecture/database-schema`



Entity diagrams throughout this chapter use the following legends:



P: Primary Key
F: Foreign Key

Extending the JIRA database with a custom schema

Now that we know that JIRA schema definitions are maintained in `WEB-INF/classes/entitydefs/entitygroup.xml` and `entitymodel.xml`, let us have a look at extending the existing schema definitions. How would you extend the JIRA scheme if you wanted to add one or two custom tables into JIRA? Is it just about creating the new tables in our database? We will see that in this recipe.



For plugins, it is recommended to use **Active Objects** technology to persist data. The JIRA schema itself should be modified only when it is absolutely necessary to have pre-defined tables created in the JIRA database.
If the JIRA schema is modified, care must be taken during upgrades to port those changes to the new version.

How to do it…

JIRA uses the schema definitions entered in the `WEB-INF/classes/entitydefs/entitygroup.xml` and `entitymodel.xml` files. It makes use of these files not only to validate and create the schema but also during the import and export of the JIRA data backup. JIRA also uses these entity definitions to read and write to a database, using **OFBizDelegator** (`https://docs.atlassian.com/jira/latest/com/atlassian/jira/ofbiz/OfBizDelegator.html`), details of which we will see in the upcoming recipes.

The following are the steps to add a new table into the JIRA schema. Let us assume we are adding a table to hold the details of an employee:

1. Identify an entity name for the table. This could be the same as the table name or different from it. This name will be used in the XML backups and also by **OFBizDelegator** to read or write the data.
   In our example, let us choose `Employee` as the entity name.
2. Modify the `WEB-INF/classes/entitydefs/entitygroup.xml` file to include the new entity group definition:

```
<entity-group group="default" entity="Employee"/>
```

Here, the `group` attribute refers to the group name the delegator is associated with. You can find it in the `WEB-INF/classes/entityengine.xml` file, as shown in the following code snippet:

```
<delegator name="default" entity-model-reader="main"
  entity-group-reader="main">
  <group-map group-name="default" datasource-name="defaultDS"/>
</delegator>
```

The `entity` attribute holds the **name** of the entity.

3. Modify the `WEB-INF/classes/entitydefs/entitymodel.xml` file to include the new entity definition:

```
<entity entity-name="Employee" table-name="employee"
package-name="">
  <field name="id" type="numeric"/>
  <field name="name" type="long-varchar"/>
  <field name="address" col-name="empaddress"
   type="long-varchar"/>
  <field name="company" type="long-varchar"/>
  <prim-key field="id"/>
  <index name="emp_entity_name">
  <index-field name="name"/>
  </index>
</entity>
```

Here, the `entity-name` attribute holds the name of the **entity** we have used in the previous step. The `table-name` attribute holds the name of the **table**; it is optional and will be derived from `entity-name`, if not present. `package-name` can be used if you want to organize and structure the entities' definitions into different packages.

The entity element contains one `field` element for each column in the table that needs to be created. The `field` element has a `name` attribute that holds the name of the field. If the column name of the field is different, the `col-name` attribute can be used, as in the case with employee address. If `col-name` is missing, the name of the field is used. The next important attribute is `type`. In our example, `id` is numeric whereas `name` and `address` are `long-varchar`.

These types of definitions of a field are mapped to the appropriate column type for each database type. The `field-type` mappings are stored under `WEB-INF/classes/entitydefs/` and are declared in `entityengine.xml`, as shown in the following code snippet:

```
<field-type name="h2" loader="maincp"
location="entitydefs/fieldtype-h2.xml"/>
```

If you look inside `fieldtype-h2.xml`, you will notice that `numeric` is mapped to **BIGINT** and `long-varchar` is mapped to **VARCHAR**. You can find out the various mappings and even the related Java data type from the same file.

The `prim-key` element is used to define the **primarykey** constraint for the table, as shown previously. In our case, `id` is the primary key. It is mandatory to name the primary key as **id** for all the new tables we are creating.

The `index` element creates a DB index for the field specified for that table. We can specify the index name and the group of the fields that needs to be indexed underneath it.

You can also define the relationship between entities using the element relation as shown in the following code snippet:

```
  <relation type="one"  rel-entity-name="Company">
        <key-map field-name="company" rel-field-name="id"/>
 </relation>
```

Here, we are adding a relationship between the **Employee** entity and the **Company** entity by saying that an employee can have only one company. In the preceding case, **Employee** should have a field company that points to the `id` field of a company's record. In other words, the `company` field in an employee's record will be the foreign key to the company's record.

More details on entity definitions can be found at `https://cwiki.apache.org/confluence/display/OFBIZ/Entity+Engine+Guide#EntityEngineGuide-EntityModeling`.

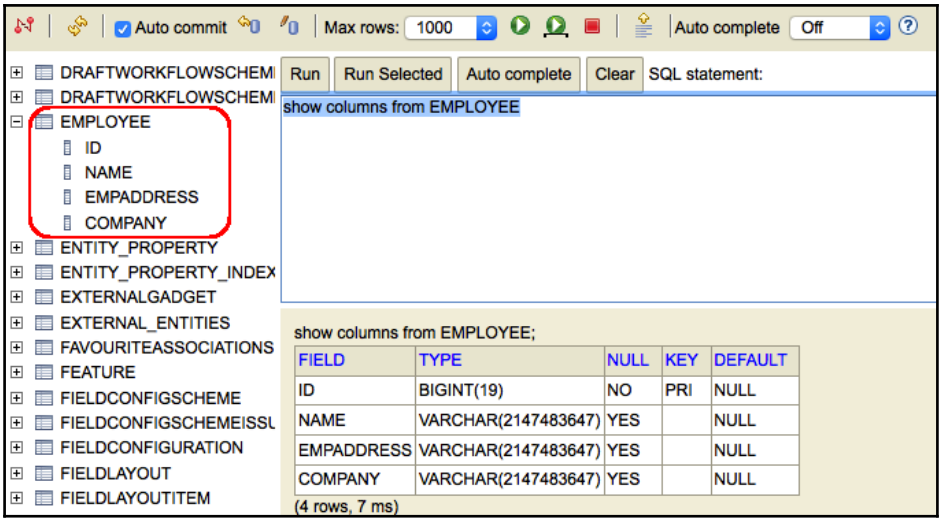
4. Restart JIRA after the changes have been made.

How works…

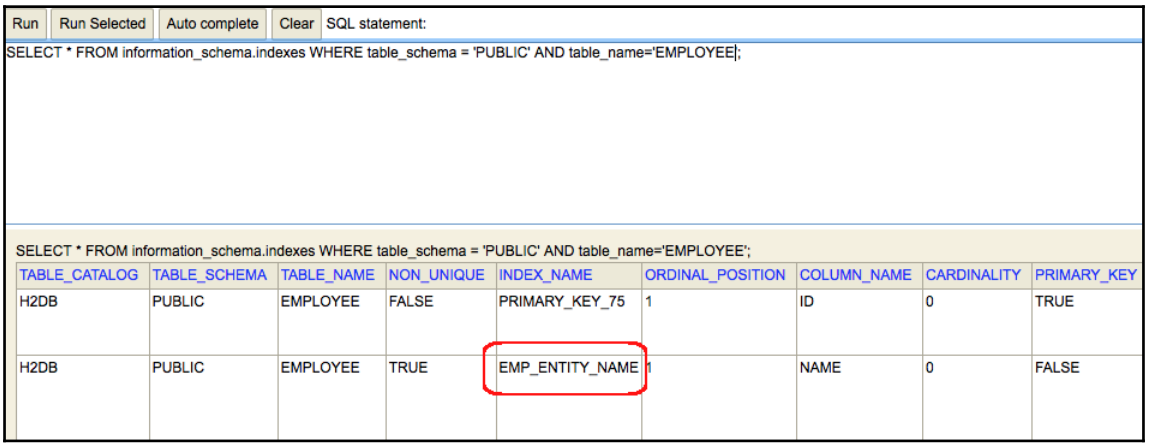
When JIRA is restarted with the previous changes, you will notice that a warning message appear in the logs during startup, as shown in the following screenshot:

```
[c.a.jira.startup.JiraHomeStartupCheck] The jira.home directory '/Users/jobinkk/Softwares/JIRA/7.0.9' is va

[c.a.j.config.database.SystemDatabaseConfigurationLoader] Reading database configuration from /Users/jobink

[c.a.jira.startup.JiraStartupLogger] Running JIRA startup checks.
[c.a.jira.startup.JiraStartupLogger] JIRA pre-database startup checks completed successfully.
[o.o.c.entity.jdbc.DatabaseUtil] Database Product Name is H2
[o.o.c.entity.jdbc.DatabaseUtil] Database Product Version is 1.4.185 (2015-01-16)
[o.o.c.entity.jdbc.DatabaseUtil] Database Driver Name is H2 JDBC Driver
[o.o.c.entity.jdbc.DatabaseUtil] Database Driver Version is 1.4.185 (2015-01-16)
[o.o.c.entity.jdbc.DatabaseUtil] Entity "Employee" has no table in the database
[o.o.c.entity.jdbc.DatabaseUtil] Created table "PUBLIC.employee"
[o.o.c.entity.jdbc.DatabaseUtil] Created declared indices for entity "Employee"
[o.o.c.entity.jdbc.DatabaseUtil] Database Product Name is H2
[o.o.c.entity.jdbc.DatabaseUtil] Database Product Version is 1.4.185 (2015-01-16)
[o.o.c.entity.jdbc.DatabaseUtil] Database Driver Name is H2 JDBC Driver
[o.o.c.entity.jdbc.DatabaseUtil] Database Driver Version is 1.4.185 (2015-01-16)
[c.a.j.config.database.DatabaseConfigurationManagerImpl] Now running Database Checklist Launcher
[c.a.jira.startup.DatabaseChecklistLauncher] JIRA database startup checks completed successfully.
[c.a.j.config.database.DatabaseConfigurationManagerImpl] Now running Post database-configuration launchers
[c.a.j.config.database.SystemDatabaseConfigurationLoader] Reading database configuration from /Users/jobink
```

Once JIRA recognizes that there is no table corresponding to the new entity name employee in the database, it will create one, as shown in the following screenshot:



Even the **index** information is stored, as highlighted in the following screenshot:



If you want to add a new column to an existing table, you can add an additional field definition to the entity and, on restarting JIRA, the table will be updated to include the column.

You will notice an error message in the JIRA logs if the database has a table, or a column in the table, that doesn't have a valid entity or field definition in the `entitymodel.xml` file.



Care must be taken to update the `entitygroup.xml` and `entitymodel.xml` files when JIRA is upgraded or else the changes will be lost.

Accessing database entities from plugins

We have seen how various entities in the JIRA database are defined and how we can introduce new entities. In this recipe, we will see how we can read and write data from the database using these entity definitions.

How to do it…

JIRA exposes the `OfBizDelegator` (`https://docs.atlassian.com/jira/latest/com/atlassian/jira/ofbiz/OfBizDelegator.html`) component, which is a wrapper around `org.ofbiz.core.entity.DelegatorInterface`, to communicate with its database using the Ofbiz layer.

You can get hold of an instance of `OfBizDelegator` by injecting it in the constructor or from `ComponentAccessor`, as follows:

```
OfBizDelegator delegator = ComponentAccessor.getOfBizDelegator();
```

Reading from a database

We can read from the database using the various methods exposed via the previous delegator class. For example, all the records in the **Employee** table we defined in the previous recipe can be read as:

```
List<GenericValue> employees = delegator.findAll("Employee");
```

Here, the `findAll` method takes the **entity name** (not the table name) and returns a list of the `GenericValue` objects, each representing a row in the table. The individual fields can be read from the object using the **name** of the field (not col-name), as follows:

```
Long id = employees.get(0).getLong("id"); String name =
employees.get(0).getString("name");
```

The data type, to which the field should be converted, can be found from the `field-type` mapping XML like we saw in the previous recipe.

We can read data from a database, when certain conditions are satisfied, using the `findByAnd` method:

```
List<GenericValue> employees = delegator.findByAnd("Employee",
MapBuilder.build("company", "J-Tricks"));
```

This will return all the records where the company name is **J-Tricks**. You can enforce more complex conditions using the `findByCondition` method and select only the interested fields, as follows:

```
List<String> fieldList = new ArrayList<String>(); fieldList.add("id");
fieldList.add("name"); List<GenericValue> employees =
this.delegator.findByCondition("Employee", new EntityExpr("id",
EntityOperator.LESS_THAN, "15000"), fieldList);
```

Here, we find all employee records with an **id** less than 15000 and we retrieve only the *id* and *name* of the employees.

The `findListIteratorByCondition` method can be used to add more options such as the `orderBy` clause, the where conditions, and the having conditions, as follows:

```
EntityFindOptions entityFindOptions = new EntityFindOptions();
entityFindOptions.scrollInsensitive();
entityFindOptions.setResultSetConcurrency(ResultSet.CONCUR_READ_ONLY);
entityFindOptions.setDistinct(true); OfBizListIterator iterator =
this.delegator.findListIteratorByCondition("Employee", new EntityExpr("id",
EntityOperator.LESS_THAN, "15000"), null, UtilMisc.toList("name"),
UtilMisc.toList("name"), entityFindOptions); List<GenericValue> employees =
iterator.getCompleteList(); iterator.close();
```

Here, we search for all records with an **id** less than 15000. We don't have a `having` condition in this case and, so, we will leave it `null`. The next two arguments specify that only the **name** field needs to be selected and the records should be ordered by the **name** field. The last argument specifies the `EntityFindOptions` class.

Here, we define `EntityFindOptions` with `scrollInsensitive()` which is same as using both `setSpecifyTypeAndConcur(true)` and `setResultSetType(TYPE_SCROLL_INSENSITIVE)`. If `specifyTypeAndConcur` is **true**, the following two parameters will be used to specify `resultSetType` and `resultSetConcurrency`. If false, the default values of the JDBC driver will be used. In our case, `specifyTypeAndConcur` is true and, hence, `resultSetType` is taken as `TYPE_SCROLL_INSENSITIVE` and `resultSetConcurrency` is taken as `CONCUR_READ_ONLY`. More about this and the possible values can be found at `http://download.oracle.com/javase/tutorial/jdbc/basics/retrieving.html`.

As you can see, we have also set the `distinct` option to be true. Apparently, this is the only way to make a distinct selection using Entity Engine. You will find more information about this in the Entity Engine cookbook at `http://www.opensourcestrategies.com/ofbiz/ofbiz_entity_cookbook.txt`.



Don't forget to close the `iterator`, as shown in the previous code snippet.

Writing a new record

Creating a new record in a table using `OfBizDelegator` is pretty easy, as shown in the following code snippet:

```
Map<String, Object>
employeeDetails = new HashMap<String, Object>();
employeeDetails.put("name", "Some Guy1");
employeeDetails.put("address", "Some Address1");
employeeDetails.put("company", "J-Tricks");
GenericValue employee =
this.delegator.createValue("Employee", employeeDetails);
```



Make sure you don't provide the **id**, as it is automatically generated. Also, the missing fields in the map will be set to null. Data for all the mandatory fields should be provided so as to avoid errors.

Updating a record

Writing a record is done by retrieving the record, modifying the values, and using the `store()` method. Once the record is retrieved, as we saw earlier in this recipe, we can modify the fields as shown below:

```
employee.setString("name","New Name");
employee.store();
```

More useful methods can be found in the Java docs at `https://docs.atlassian.com/jira/latest/com/atlassian/jira/ofbiz/OfBizDelegator.html`.

Persisting plugin information in the JIRA database

While developing plugins, we may come across many scenarios where we need to store specific information about the plugins, be it configuration details or metadata for entities. How can we do this without creating a custom schema and going through the pain of editing entity definitions? In this recipe, we will learn how we can make use of JIRA's existing framework to store information specific to the plugins we develop.

JIRA uses the Open symphony `PropertySet` framework to store properties in the database. These properties are a set of key/value pairs and are stored against any entity that the user wants. The key of the property is always a `String` value; the value can be: `String`, `Long`, `Date`, `Boolean`, or `Double`. We have already seen how JIRA uses it in `Chapter 2`, *Understanding the Plugin Framework*. In this recipe, we will see how we can use `PropertySet` to store our custom data.

How to do it…

Suppose that we need to store a **Boolean** value in the database as part of our plugin's configuration and read it later; here are the steps to follow to do it:

1. Get an instance of `PropertySet`, using `PropertiesManager`:

```
PropertySet propertySet =
ComponentAccessor.getComponent(PropertiesManager.class)
.getPropertySet();
```

   You can also inject the `PropertiesManager` class into the constructor.

2. Persist the **Boolean** property using the `setBoolean` method:

```
propertySet.setBoolean("jtricks.custom.key1", new Boolean(true));
```

   Similarly, **String**, **Long**, **Double**, and **Date** values can be stored using the respective methods.

3. The property that is stored can be retrieved at any point, as follows:

```
Boolean key = propertySet.getBoolean("jtricks.custom.key1");
```

However, how do we store a more complex structure, such as a property, to an existing entity? Let us say we want to store the address of a user. JIRA stores the user information against the entity **User**, as follows:

1. Retrieve the **id** of the **User** entity we are going to store the address against. For example, if there is a user *jobinkk*, we can find the **id** of the user from the **User** entity that corresponds to the `cwd_user` table in JIRA.

   Let us assume that the **id** is 10000.

2. Get an instance of `PropertySet`, using `PropertySetManager`, by passing the details of the entity we got:

```
Map<String, Object> entityDetails = new HashMap<String, Object>();
entityDetails.put("delegator.name", "default");
entityDetails.put("entityName", "User");
entityDetails.put("entityId", new Long(10000));
PropertySet userProperties = PropertySetManager.getInstance("ofbiz",
entityDetails);
```

3. Here, we create a map with the entity **name**, that is, `User`, and the **id** of the user, that is, 10000. We also pass the **delegator** name as defined in the `entityengine.xml` file, under the `WEB-INF/classes` folder, which is `default` in this case. We then retrieve the `PropertySet` instance from `PropertySetManager`, using `ofbiz` as the key.
4. The values can be set as before, depending on the type of the field. In this case, we will have more than one key for state, country, and so on:

```
userProperties.setString("state", "Kerala");
userProperties.setString("country", "India");
```

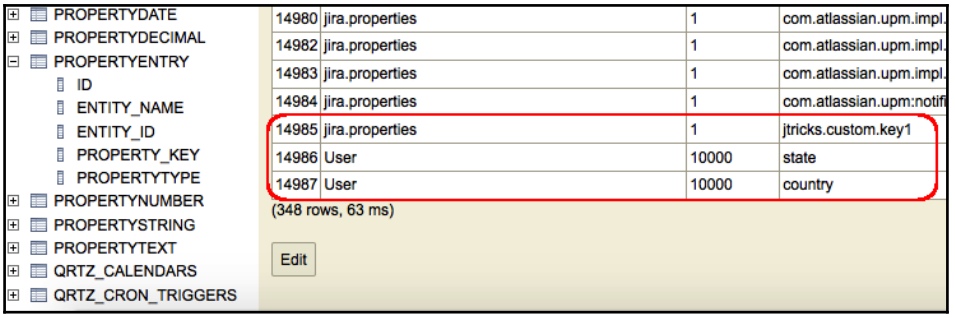
5. This will then be stored in the appropriate tables.
6. We can retrieve these values later by creating the PropertySet instance in a similar manner and using the getter methods:

```
System.out.println("Address:" + userProperties.getString("state") +
 "," + userProperties.getString("country"));
```

How it works…

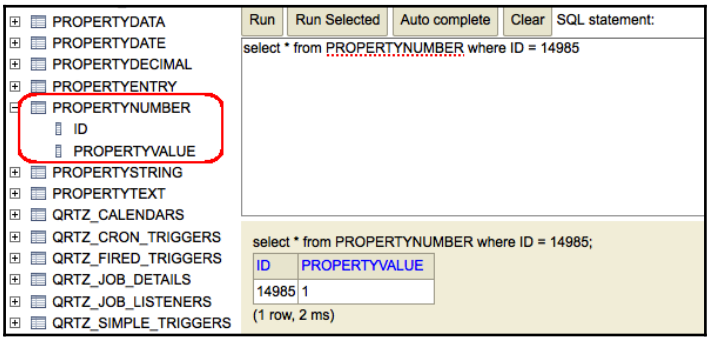
When a property is set using `PropertySet`, instantiated from `PropertiesManager` as we did in the case of the **Boolean** value, it gets stored in the `propertyentry` table with the `ENTITY_NAME` value as `jira.properties` and `ENTITY_ID` as **1**. It will also have a unique **id**, which will then be used to store the value in the `propertynumber`, `propertystring`, `propertytext`, or `propertydate` table, depending on the data type we used.

In our case, the `propertyentry` table is populated with values, as shown in the following screenshot:



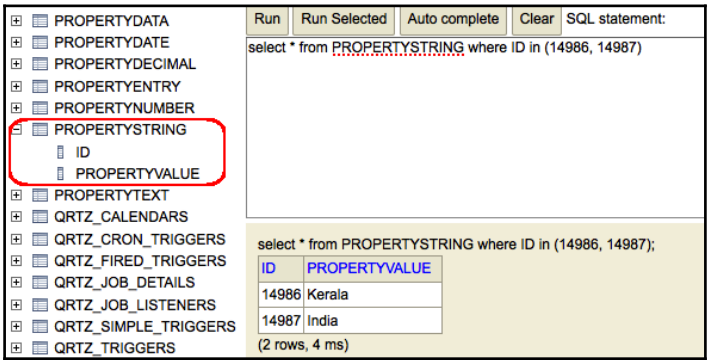
The first one is the **Boolean** property we added whereas the second and third are the user properties.

Boolean values get stored as numbers (0 or 1) and hence, the `propertyentry` table stores the `propertytype` value as 1, which denotes a number value. There is a corresponding entry in the `propertynumber` table, with an ID of 14985, for the Boolean property, as shown in the following screenshot:



In our example, the **Boolean** is set to true and hence, the `propertynumber` table stores the value 1. If set to false, it will store 0.

In the case of address, the entity is `User` and it has an **entityId** value of 10000. We have seen two rows with **id**s 14986 and 14987, each with `propertytype` as 5, which denotes **String** values. Because they are **String** values, they are stored in the `propertystring` table, as shown in the following screenshot:



Hopefully, this gives a fair idea about how we can store attributes against an existing entity record.

The good thing about the usage of `PropertySet` is that we don't need to create an extra scheme or entity definition and these properties are exported in the backup XML when JIRA data is exported. So, all configurations stored like this will be retained when the data is imported back into another JIRA instance.

Using Active Objects to store data

Active Objects represent a technology used by JIRA to allow per-plugin storage. This gives the plugin developers a real protected database where they can store the data belonging to their plugin and which other plugins won't be able to access. In this recipe, we will see how we can store an address entity in the database using Active Objects.

You can read more about Active Objects at:
`http://java.net/projects/activeobjects/pages/Home`

Getting ready…

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it…

In order to understand it better, let us look at the simple address entity example that we used in the previous recipe. This will also help in an easy comparison with `PropertySet`, if desired.

Following are the steps to use Active Objects in the plugin:

1. Include the Active Objects dependency in `pom.xml`. Add the appropriate ao version, which you can find from the Active Objects JAR bundled in your JIRA:

```
<dependency>
    <groupId>com.atlassian.activeobjects</groupId>
    <artifactId>activeobjects-plugin</artifactId>
    <version>${ao.version}</version>
    <scope>provided</scope>
</dependency>
```

2. Add the Active Objects plugin module to the Atlassian plugin descriptor:

```
<ao key="ao-module">
    <description>The configuration of the Active Objects
      service</description>
    <entity>com.jtricks.entity.AddressEntity</entity>
</ao>
```

   As you can see, the module has a unique key and it points to an entity we are going to de ne later, `AddressEntity` in this case.

3. Include a `component-import` plugin to register `ActiveObjects` as a component in `atlassian-plugin.xml`:

```
<component-import key="ao" name="Active Objects components"
interface="com.atlassian.activeobjects.external.ActiveObjects">
   <description>Access to the Active Objects service</description>
</component-import>
```

   Note that this step is not required if you are using the **Atlassian Spring Scanner**. Instead, you can use the `@ComponentImport` annotation, while injecting `ActiveObjects` in the constructor.

4. Define the entity to be used for data storage. The entity should be an interface and should extend the `net.java.ao.Entity` interface. All we need to do in this entity interface is to de ne getter and setter methods for the data that we need to store for this entity.

   For example, we need to store the name, city, and country as part of the address entity. In this case, the `AddressEntity` interface will look like the following:

```
public interface AddressEntity extends Entity {
    public String getName();
    public void setName(String name);
    public String getState();
    public void setState(String state);
    public String getCountry();
    public void setCountry(String country);
}
```

   By doing this, we have set up the entity to facilitate the storage of all the three attributes.

We can now create, modify, or delete the data using the `ActiveObjects` component. The component can be instantiated by injecting it into the constructor:

```
private ActiveObjects ao;
@Inject
public ManageActiveObjects(@ComponentImport ActiveObjects ao) {
     this.ao = ao;
}
```

A new row can be added to the database using the following piece of code:

```
AddressEntity addressEntity = ao.create(AddressEntity.class);
addressEntity.setName(name);
addressEntity.setState(state);
addressEntity.setCountry(country);
addressEntity.save();
```

Details can be read either using the ID, which is the primary key, or by querying the data using a `net.java.ao.Query` object. Using the ID is as simple as is shown in the following code line:

```
AddressEntity addressEntity = ao.get(AddressEntity.class, id);
```

The Query object can be used as follows:

```
AddressEntity[] addressEntities = ao.find(AddressEntity.class,
Query.select().where("name = ?", name));
for (AddressEntity addressEntity : addressEntities) {
    System.out.println("Name:"+addressEntity.getName()+",
    State:"+addressEntity.getState()+",
    Country:"+addressEntity.getCountry());
}
```

Here, we are querying for all records with a given name.

Once you get hold of an entity by either means, we can edit the contents simply by using the setter method:

```
addressEntity.setState(newState);
addressEntity.save();
```

Deleting is even simpler!

```
ao.delete(addressEntity);
```

How it works…

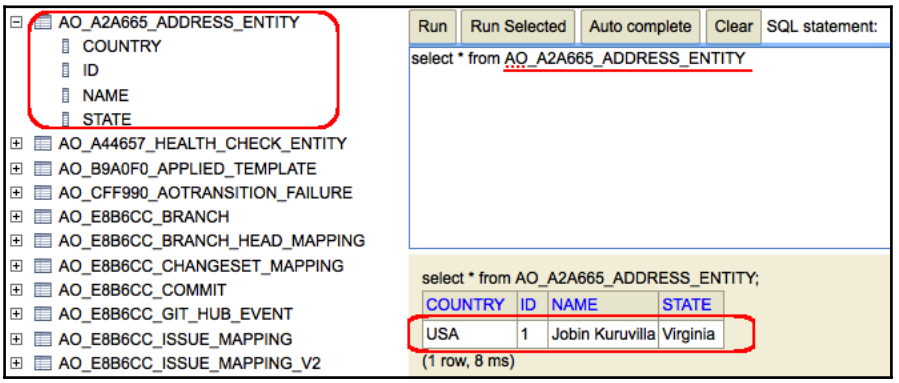
Behind the scenes, separate tables are created in the JIRA database for every entity that we add. The Active Objects service interacts with these tables to do the work.

If you look at the database, a table of the name `AO_{SOME_HEX}_MY_OBJECT` is created for every entity named `MyObject` belonging to a plugin with the key `com.example.ao.myplugin`, where:

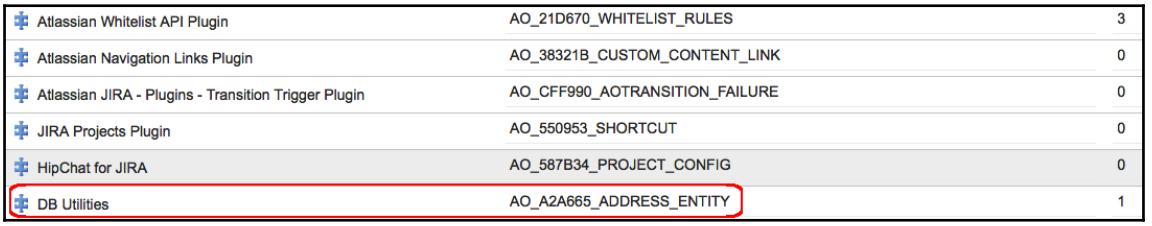
- `AO` is a common prefix.
- `SOME_HEX` is the set of the first six characters of the hexadecimal value of the hash of the plugin key `com.example.ao.myplugin`.
- `MY_OBJECT` is the upper-case translation of the entity class name `MyObject`.

For every attribute with the getter method, `getSomeAttribute`, defined in the entity interface, a column is created in the table with the name `SOME_ATTRIBUTE` using the Java Beans naming convention-separating the two words by an underscore and keeping them both in upper case.

In our `AddressEntity` example, we have the following table, `ao_a2a665_address_entity`, as follows:



If you navigate to Administration | System | Advanced | Plugin Data Storage, you can find out all the tables created using **Active Objects**, as shown below:



As you can see, the table created using our example plugin is listed along with the tables created by other standard JIRA plugins.

Lots more about Active Objects can be read at:
`https://developer.atlassian.com/docs/atlassian-platform-common-components/active-objects`

Accessing the JIRA configuration properties

We have seen how to use `PropertySet` to store details of plugins in the previous recipes. In this recipe, we will see how we can access the JIRA configuration properties using `PropertySet`.

How to do it…

There are lots of global configuration settings in JIRA that are configured using administration menus. More on the various options can be read at `http://confluence.atlassian.com/display/JIRA/Configuring+Global+Settings`. Where does JIRA store this information and how do we access it?

All these configuration properties, such as settings under General Configuration, Base URL, Attachments Path, License Info, and more, are stored in the `PropertySet` tables we saw earlier. They are stored against a virtual entity, `jira.properties`. This is the same virtual entity that is used when `PropertySet` is retrieved using `PropertiesManager`, as we saw while persisting plugin information.

Here, all the property key entries are stored in the `propertyentry` table, with `jira.properties` as the entity name and entity id as **1**. The `propertytype` value for each property varies, depending on what is stored against it. For example, `jira.option.allowattachments` is a flag and hence is stored in the `propertynumber` table, with a value of either or **1**. In this case, `propertytype` is 1, denoting the number value. `jira.path.index`, on the other hand, stores a **String** that holds the index path and will have **5** as `propertytype`. Here the value is stored in the `propertystring` table.

All the properties can be accessed using the following SQL command:

```
SELECT * FROM propertyentry WHERE ENTITY_NAME='jira.properties';
```

If you want to see only **String** properties and their values, you can get it using the following command:

```
SELECT PROPERTY_KEY, propertyvalue
      FROM propertyentry pe, propertystring ps
      WHERE pe.id=ps.id AND pe.ENTITY_NAME='jira.properties' AND
propertytype='5';
```

If you want to search for a specific property, you can do that using the following command:

```
SELECT PROPERTY_KEY, propertyvalue
     FROM propertyentry pe, propertynumber pn
     WHERE pe.id=pn.id AND pe.ENTITY_NAME='jira.properties' AND
pe.PROPERTY_KEY='jira.option.allowattachments';
```



Note that the appropriate property table should be used, `propertynumber` in this case!

The same things can be achieved in a plugin, as follows:

1. Retrieve the `PropertySet` object using `PropertiesManager`:

```
PropetySet propertySet =
ComponentAccessor.getComponent
(PropertiesManager.class).getPropertySet();
```

2. All property keys can be retrieved as follows:

```
Collection<String> keys = propertySet.getKeys();
```

3. Similarly, all the properties of a specific type can be accessed as:

```
Collection<String> stringKeys = propertySet.getKeys(5);
```

4. The value of a particular key can be accessed as follows:

```
String attachmentHome =
propertySet.getString("jira.path.attachments");
boolean attachmentsAllowed =
propertySet.getBoolean("jira.option. allowattachments");
```

Getting a database connection for JDBC calls

It is not always feasible to use `OfBizDelegator` to get all the details that we need. What if we need to execute a complex query in the database via JDBC? In this recipe, we will see how we can retrieve the database connection that is defined in `entityengine.xml`.

How to do it…

The database connection lookup in JIRA is pretty simple and can be done in a single line in JIRA 7.x+. Just do the following:

```
Connection conn = new DefaultOfBizConnectionFactory().getConnection();
```

Simple, isn't it?

`DataSourceInfo` can be accessed as follows:

```
DatasourceInfo datasourceInfo = new
```

```
DefaultOfBizConnectionFactory().getDatasourceInfo();
```

Over to you to write the **JDBC** calls wisely!

Migrating a custom field from one type to another

Custom fields in JIRA are of different types-text fields, select lists, number fields, and so on. We might come across scenarios where we need to change the type of a field but without losing all the data we have entered until then! Is possible to do that? It is, to a certain extent. In this recipe, we will see how to do it.

The type of a field can only be changed via the database, as the UI doesn't support that. And it won't be possible with all the field types. For example, it isn't possible to convert a text field to a number field because all the values that the field already has may not be numbers. However, the reverse is possible, because all number values can be treated as text values. Similarly, you can convert a select field to a text field but you cannot convert a multi-select field to a text field because a multi-select has multiple values, each with a separate row in the `customfieldvalue` table.

So, the first step is to identify whether the conversion is feasible, by looking at the source and target types. If it is feasible, we can go on and modify the type, as described in this recipe.

How to do it…

The following steps outline how to modify the type of custom field, if the source and target types satisfy the criteria we discussed earlier:

1. Stop the JIRA instance. Updating the DB while JIRA is running might cause data corruption. And in some cases, the change won't be reflected because of cached objects in memory.
2. Connect to the JIRA DB as the JIRA user.
3. Modify the custom field key in the `customfield` table by executing the SQL script as shown:

```
UPDATE customfield
 SET customfieldtypekey =
 'com.atlassian.jira.plugin.system.customfieldtypes:textfield'
 WHERE cfname = 'Old Number Value';
```

Here, the type of the custom field named '**Old Number Value**' is changed to a text field. Make sure that the custom field name is unique; if not, use a custom field **id** in the where condition.

4. Modify the searcher key similarly, with an appropriate searcher. In the previous case, we need to modify the searcher value to a text searcher, as shown:

```
UPDATE customfield
    SET customfieldsearcherkey =
   'com.atlassian.jira.plugin.system.customfieldtypes:textsearcher'
    WHERE cfname = 'Old Number Value';
```

5. Commit the changes and disconnect.
6. Start JIRA.
7. Do a complete re-indexing of the JIRA instance by going to **Administration** | **System** | **Advanced** | **Indexing**.

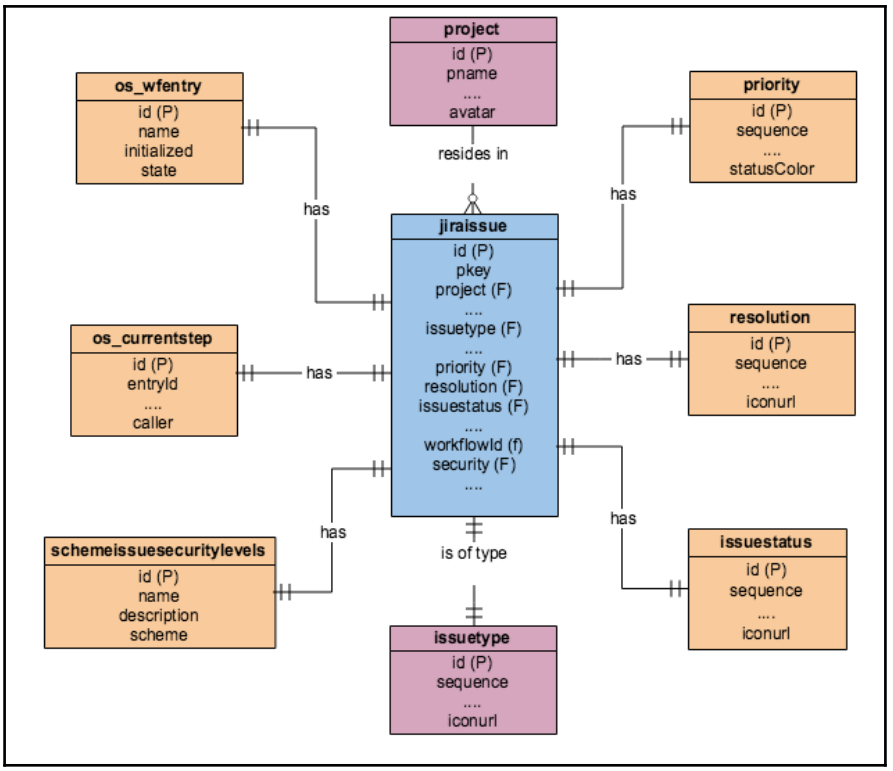
The custom field should now be modified to a **text** field from the old **number** field. Add or update values and search them to verify the change.

You can use the same approach for other custom field types, as long as they are compatible. Make sure to use the appropriate `customfieldtypekey` and `customfieldsearcherkey`, depending on the target custom field type.

Retrieving issue information from a database

Information about an issue is scattered around in multiple tables in the JIRA database. However, a good starting point is the `jiraissue` table, which is where the issue record is stored. It has foreign keys referencing other tables and, at the same time, the issue **id** is referenced in a few other tables.

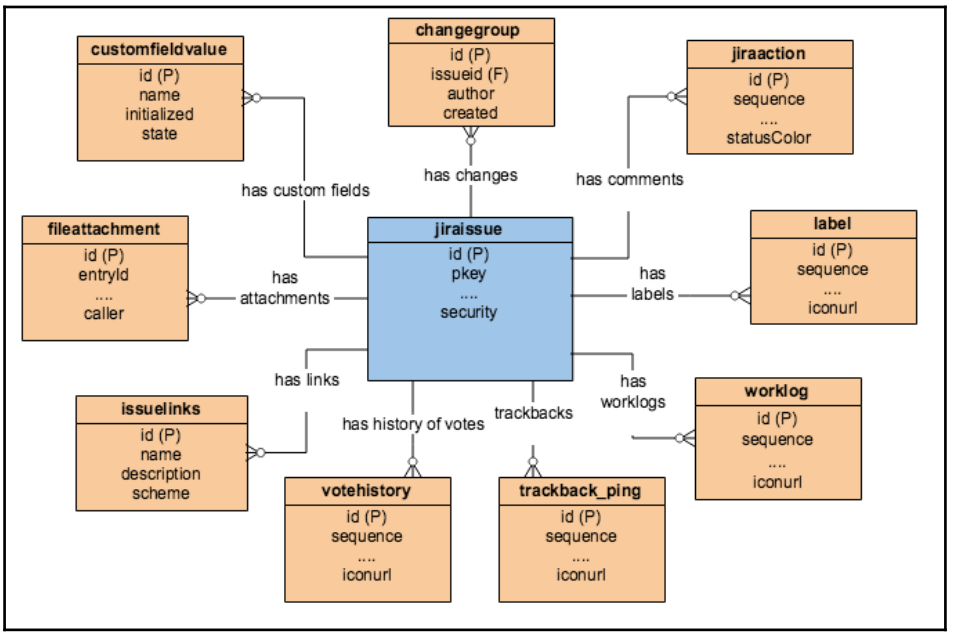
The following diagram captures the important tables that the `jiraissue` table has a parent relationship with. Depending on the JIRA version, there might be slight variations but this is probably a good starting point:



As you can see, critical information about an issue, such as the **project**, **issue type**, **status**, **priority**, **resolution**, **security level**, and **workflow**, is all stored in the respective tables but is referenced from the `jiraissue` table, using a foreign key. The foreign key points to the **id** of the other tables in all cases, but there are no foreign key constraints enforced on any of these tables.

From JIRA 6.1, JIRA supports **project** renaming and hence the `pkey` column doesn't include the JIRA issue key anymore. Instead, `issuenum` holds the numeric part of a JIRA issue key and the project key has to be found from the `project` table, using the `project` column in the `jiraissue` table. We will see an example in this recipe.

The following diagram shows the tables that the `jiraissue` table has a child relationship with:



Here, the tables `customfieldvalue`, `changegroup`, `jiraaction`, `label`, `worklog`, `fileattachment`, `issuelink`, `trackback_ping`, and so on have a foreign key with the name `issueid` or `issue` (or source or destination) pointing to the relevant issue's **id**.

In this recipe, we will learn how to access some of the issue's information with the help of the previous diagrams.

How to do it…

When there is a parent-child relationship between tables, we can do a **join** operation to get most of the information we are looking for. For example, all the issues along with their project names can be retrieved by the following query:

```
SELECT ji.id, ji.issuenum, pr.pname
     FROM jiraissue ji INNER JOIN  project pr
     ON ji.project = pr.id;
```

Here we do an inner join on the condition that the project's `id` is the same as the `project` column value in the `jiraissue` table.

The results will look like the following:

ID	ISSUENUM	PNAME
10000	1	DEMO Project
10001	2	DEMO Project

As you probably noticed, the above query returns only the numeric part of the JIRA issue key. We can use the SQL functions or operands, depending on the database server you are using, to combine the `issuenum` column with `pkey` column in the `project` table. For example, we can use the `concat` function in the **H2** database, as shown below:

```
SELECT ji.id, concat(pr.pkey, '-',ji.issuenum) as issuekey, pr.pname
     FROM jiraissue ji INNER JOIN  project pr
     ON ji.project = pr.id;
```

The results will now look like the following:

ID	ISSUEKEY	PNAME
10000	DEMO-1	DEMO Project
10001	DEMO-2	DEMO Project

Similarly, all the comments on an issue can be retrieved by the following query:

```
SELECT ji.id, concat(pr.pkey, '-',ji.issuenum) as issuekey, ja.actionbody,
ja.created, ja.author
     FROM jiraissue ji
     LEFT JOIN jiraaction ja ON ji.id = ja.issueid
     INNER JOIN project pr ON ji.project = pr.id;
```

And, the results will look like the following:

ID	ISSUEKEY	ACTIONBODY	CREATED	AUTHOR
10000	DEMO-1	Sample Comment 1	2015-10-27 22:43:14.569	jobinkk
10000	DEMO-1	Sample Comment 2	2015-10-29 15:43:14.569	testuser

In the example, we retrieve the comments on issues with their **author** and **created date**. The same approach can be used with all tables in the previous diagrams.

There's more…

Accessing **version** and **component** information on an issue is slightly different. Even though you see the `fixfor` and `component` columns in the `jiraissue` table, they are not used anymore!

Each issue can have multiple versions or components and hence there is a join table between the `jiraissue` and `version/component` tables, called `nodeassociation`. `source_node_entity` will be the `issue` and the `source_node_id` represents the issue **id**. The `sink_node_entity` value will be **Component** or **Versions** in this case, and `sink_node_id` will hold the **id** of the respective component or version.

There is a third column, `association_type`, which will be **IssueFixVersion**, **IssueVersion**, or **IssueComponent** for the **fix for versions**, **affected versions**, or **components** respectively.

We can access the components of an issue as follows:

```
SELECT concat(pr.pkey, '-',ji.issuenum) as issuekey, comp.cname
     FROM nodeassociation na, component comp, jiraissue ji, project pr
     WHERE comp.id = na.sink_node_id
     AND ji.id = na.source_node_id
     AND na.association_type = 'IssueComponent'
     AND pr.pkey='DEMO'
     AND ji.issuenum='123';
```

Here, **DEMO-123** is the issue. We can also retrieve the affected versions and fix versions in a similar fashion.

Retrieving custom field details from a database

In the previous recipe, we saw how to retrieve the standard fields of an issue from the database. In this recipe, we will see how to retrieve the custom field details of an issue.

All the custom fields in JIRA are stored in the `customfield` table, as we have seen while modifying the custom field types. Some of these custom fields, such as **select** fields and **multi-select** fields, can have different options configured and they can be found in the `customfieldoption` table.

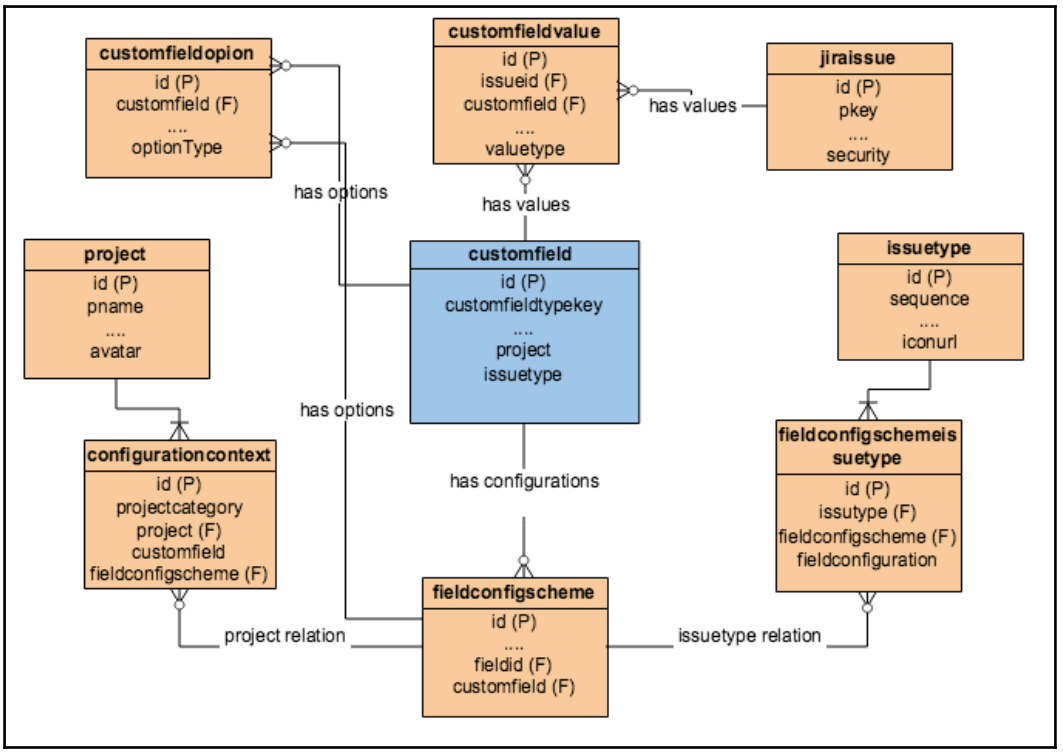
For each custom field, there can be a set of contexts configured. These contexts specify the projects or a list of issue types the field is associated with. For each such context, an entry is made in the `fieldconfigscheme` table with a unique **id**. For each `fieldconfigscheme`, there will be entries in the `configurationcontext` and `fieldconfigschemeissuetype` tables, `configurationcontext` holding the **projects** the field is associated with in the relevant context, and `fieldconfigschemeissuetype` holding the **issue types** the field is associated with!

For fields such as `select` and `multi-select`, there can be different options configured for different contexts and this can be found from the `customfieldoption` table, using the `customfildconfig` column, which points to the respective row in the `fieldconfigscheme` table.

There must always be a record in `configurationcontext` and `fieldconfigschemeissuetype` for each configuration scheme. If the scheme isn't restricted to any projects or issue types, the `project` and `issuetype` columns of the respective tables should be **NULL**.

For individual issues, the value(s) of the custom fields are stored in the `customfieldvalue` table with a reference to the `jiraissue` and `customfield` tables. For multi-value fields, such as **multi-select** and multiple **checkboxes**, there will be multiple entries in the `customfieldvalue` table.

We can capture this information in a simple diagram:



How to do it…

Once a custom field is added, the details of the field can be retrieved from the `customfield` table with this simple query:

```
SELECT * FROM customfield WHERE cfname = 'CF Name';
```

If it is a field with multiple options, such as the **select** field, the options can be retrieved using a simple **join**, as shown in the following command line:

```
SELECT cf.id, cf.cfname, cfo.customvalue
     FROM customfield cf INNER JOIN customfieldoption cfo
     ON cf.id = cfo.customfield
     WHERE cf.cfname = 'CF Name';
```

The various field configurations can be retrieved from the `fieldconfigscheme` table, as follows:

```
SELECT * FROM fieldconfigscheme
     WHERE fieldid = 'customfield_12345';
```

Here, **12345** is the unique **id** for the custom field.

The projects associated with a custom field can be retrieved as follows:

```
SELECT project.pname
     FROM configurationcontext INNER JOIN project
     ON configurationcontext.project = project.id
     WHERE fieldconfigscheme
     IN (SELECT id FROM fieldconfigscheme WHERE fieldid =
     'customfield_12345');
```

When the project is **NULL**, the field is *global* and hence available for all projects!

Similarly, the issue types associated with the field can be retrieved as follows:

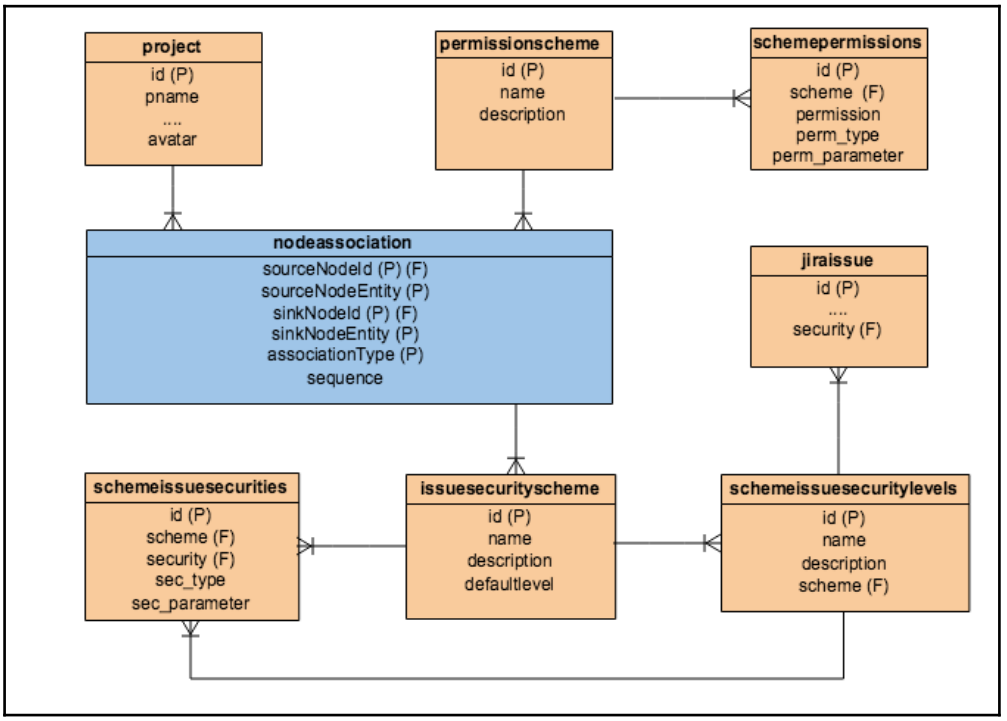
```
SELECT issuetype.pname
     FROM fieldconfigschemeissuetype INNER JOIN issuetype
     ON fieldconfigschemeissuetype.issuetype = issuetype.id
     WHERE fieldconfigscheme IN (SELECT id FROM fieldconfigscheme
     WHERE fieldid = 'customfield_12345');
```

Retrieving permissions on issues from a database

JIRA is quite powerful in enforcing permissions on issues. There are quite a lot of configuration options in controlling *who* can do *what*. All these revolve around two different schemes in JIRA, the **Permission scheme** and the **Issue Security scheme**.

The **Permission scheme** enforces *project-level* security, whereas the **Issue Security scheme** enforces *issue-level* security. It is possible for you to grant access to view issues in a project and yet hide some of those issues from the user. However, the reverse is not possible, that is, one cannot grant access to certain selected issues when the user originally didn't have access to view the issues in the project.

The various tables involved in storing permission information in the JIRA database, along with the relations between them, can be depicted as follows:



As you can see here, both the Permission schemes and Issue Security schemes are related to a project via the `nodeassociation` table. Here, `SOURCE_NODE_ENTITY` is the **project** and the corresponding `SOURCE_NODE_ID` holds the **id** of the project. `SINK_NODE_ENTITY` is the **Permission scheme** or **Issue Security scheme** depending on the scheme type. `SINK_NODE_ID` will point to the appropriate scheme. The `ASSOCIATION_TYPE` is `ProjectScheme`, in both cases.

For each of the Permission schemes, there are multiple permissions predefined, such as administer project, browse project, and create issues. For each of these permissions, the `perm_type` and `perm_parameter` columns hold the type of the entity and its value that has the relevant permission. For example, the `perm_type` column could be **group**, **user**, **project role**, and so on, and `perm_parameter` will be the group name, username, or the project role, respectively. Multiple permission types can be granted a single permission.

Similarly, the Issue Security scheme holds a number of security levels that are stored in the `schemeissuesecuritylevels` table. Each of these security levels can have different entities in them, which are also defined using the type and parameter values; in this case, the column names are `sec_type` and `sec_parameter`.

The Permission scheme is enforced on an issue based on the project it resides in, whereas the security scheme is enforced by looking at the security level the issue is assigned. The security column in the `jiraissue` table holds this information.

Let us see how we can retrieve some of this information from an issue, based on the previous diagram.

How to do it…

It is fairly easy to find out the Permission scheme associated with a project with the help of the `nodeassociation` table, as shown below:

```
SELECT pr.pname, ps.name
     FROM nodeassociation na, project pr, permissionscheme ps
     WHERE pr.id = na.source_node_id
     AND  ps.id = na.sink_node_id
     AND na.association_type = 'ProjectScheme'
     AND na.source_node_entity = 'Project'
     AND na.sink_node_entity = 'PermissionScheme';
```

Similarly, the Issue Security scheme can be retrieved as follows:

```
SELECT pr.pname, iss.name
     FROM nodeassociation na, project pr, issuesecurityscheme iss
     WHERE pr.id = na.source_node_id
     AND iss.id = na.sink_node_id
     AND na.association_type = 'ProjectScheme'
     AND na.source_node_entity = 'Project'
     AND na.sink_node_entity = 'IssueSecurityScheme';
```

The permissions parameters associated with a specific permission in a permission scheme, with an **id** value 10000, can be easily retrieved as follows:

```
SELECT sp.perm_type, sp.perm_parameter
     FROM schemepermissions sp INNER JOIN permissionscheme ps
     ON sp.scheme = ps.id
     WHERE ps.id = 10000 AND sp.permission = 23
```

Here, `sp.permission = 23` denotes the **PROJECT_ADMIN** permission. The different permission types can be found in the `com.atlassian.jira.security.Permissions` class. Here, `perm_type` denotes whether the permission is granted to a group, user, or role; `perm_parameter` holds the name of the respective group, user, or role.

Similarly, queries can be written to retrieve information on the issue security schemes. For example, the security levels and the security type and parameters for each level in an Issue Security scheme can be retrieved as follows:

```
SELECT iss.name, sisl.name, sis.sec_type, sis.sec_parameter
     FROM issuesecurityscheme iss , schemeissuesecurities sis,
     schemeissuesecuritylevels sisl
     WHERE sis.scheme = iss.id AND sisl.scheme=iss.id;
```

Writing more complex queries is outside the scope of the book but, hopefully, the previous schema diagram and the sample SQL scripts provide enough information to start with!

Retrieving workflow details from a database

Other major information that people normally look for in the database is about workflows. What is the current status of an issue? How does one find out which workflow an issue is associated with? Where is the workflow XML stored in the database? In this recipe, we will take a quick tour of the tables related to workflows.

JIRA workflows, as we have seen in the previous chapters, have **statuses**, **steps**, and **transitions**. There is always a one-to-one mapping between status and step and they are always kept in sync. Then, there are transitions which will move the issue from one step to another and, hence, from one status to another.

The workflows themselves are stored as XML files in the `jiraworkflows` table. JIRA processes these XMLs using the `OSWorkflow` APIs to retrieve the necessary information for each transition, step, and so on. Any draft workflows are stored in the `jiradraftworkflows` table.

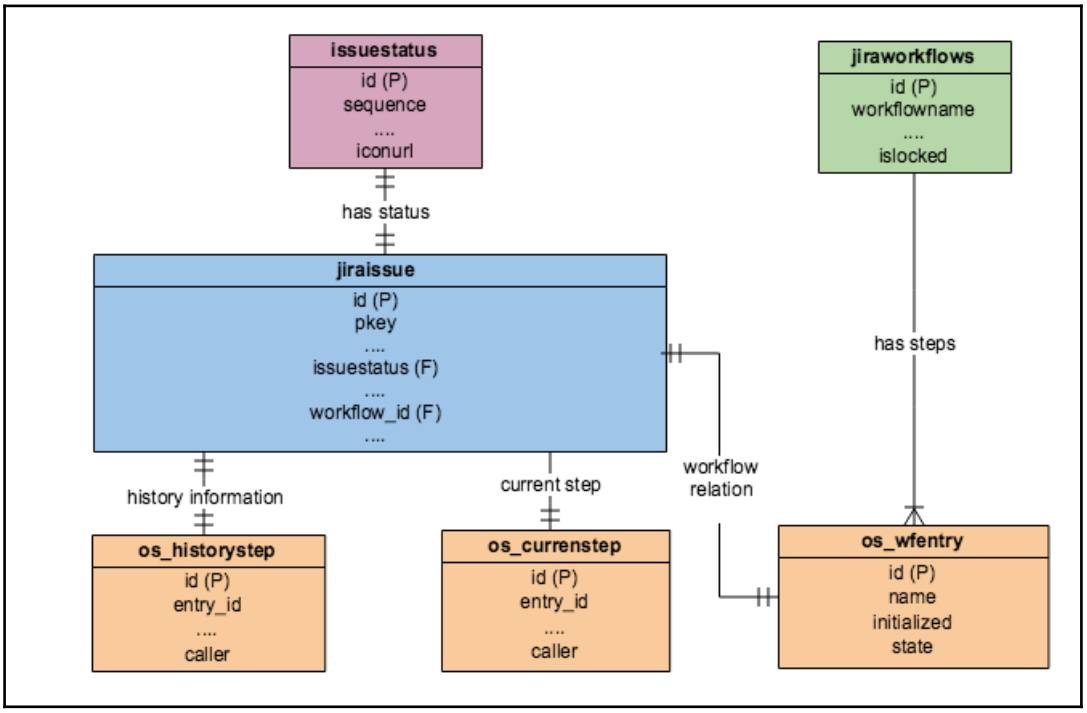
The `jiraissue` table holds the **id** of its current status and the status details are stored in the `issuestatus` table. We can use the status **id** in the `jiraissue` table to retrieve the corresponding details from the `issuestatus` table.

`jiraissue` also has another column, `workflow_id`, which points to the workflow the issue is associated with and the current step in the workflow the issue is in. The first bit of information, that is, the workflow an issue is associated with, is stored in the `os_wfentry` table. Here, the `workflow_id` column will point to the `entry.id` column of the `os_wfentry` table. The second bit of information, that is, the current step associated with an issue, is stored in the `os_currentstep` table. Here, the `workflow_id` column points to the `entry_id` column in the `os_currentstep` table.

So, for every issue, there is an entry in the `os_wfentry` and `os_currentstep` tables. And the relations are as follows: `jiraissue.WORKFLOW_ID == OS_WFENTRY.ID` and `jiraissue.WORKFLOW_ID == OS_CURRENTSTEP.ENTRY_ID`.

There is another table, `os_historystep`, which holds all the history information of the steps an issue has gone through. Here, again, the `workflow_id` column points to the `entry_id` column in the `os_historystep` table. From this table, we can retrieve information on how long an issue remained in a particular step or status.

The following schema diagram captures the important relations:



How to do it…

The status of an issue with **id**`10000` can be retrieved by a simple query, as shown in the following command line:

```
SELECT istat.pname
     FROM issuestatus istat, jiraissue ji
     WHERE istat.id=ji.issuestatus AND ji.id = '10000';
```

The details of the workflow associated with the issue can be retrieved as follows:

```
SELECT * FROM os_wfentry
     WHERE id=(SELECT workflow_id FROM jiraissue
     WHERE id='10000');
```

You can retrieve the *workflow XML* for the issue using the following query:

```
SELECT ji.pkey, wf.descriptor
     FROM jiraissue ji, jiraworkflows wf, os_wfentry osw
     WHERE ji.workflow_id = osw.id
     AND osw.name = wf.workflowname
     AND ji.id='10000';
```

The current step associated with the issue can be retrieved as follows:

```
SELECT * FROM os_currentstep
     WHERE entry_id = (SELECT workflow_id FROM jiraissue
     WHERE id = '10000');
```

The history of workflow status (step) changes can be retrieved from the `os_historystep` table, as shown in the following command line:

```
SELECT * FROM os_historystep
     WHERE entry_id = (SELECT workflow_id FROM jiraissue
     WHERE id = '10000');
```

Updating the issue status in a database

In this recipe, we will see how to update the status of an issue in the JIRA database.

Getting ready

Go through the previous recipe to understand the workflow-related tables in JIRA.

How to do it…

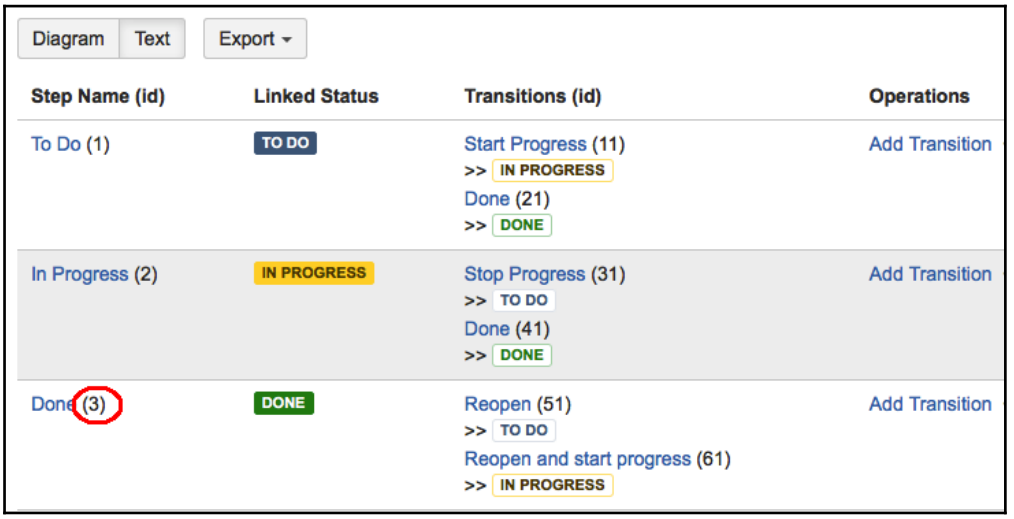
The following are the steps to update the status of an issue in JIRA database:

1. Stop the JIRA server.
2. Connect to the JIRA database.
3. Update the `issuestatus` field in the `jiraissue` table with the status you need:

```
UPDATE jiraissue SET issuestatus = (SELECT id FROM issuestatus
WHERE pname = 'Done') WHERE id = 10000;
```

   Here, 10000 is the **id** of the issue to be updated.

   Modify the `step_id` column in the `os_currentstep` table with the step **id** linked to the status you used in the previous step. `step_id` can be found in the workflow XML alongside the step name within brackets, as shown in the following screenshot:



4. As you can see, the status **DONE** in the above workflow is linked to the **Done** step with an ID value **3**. Now, the `step_id` column can be updated as follows:

```
UPDATE os_currentstep
    SET step_id = 3
    WHERE entry_id = (SELECT workflow_id FROM jiraissue
    WHERE id = '10000');
```

   Here, we modify `step_id` in `os_currentstep` where the `entry_id` column is the same as `workflow_id` in the `jiraissue` table.

   This is very important as the **step** and **status** should always be in sync. Updating the **status** alone will change it on the issue but will prevent further workflow actions on it.

5. Add entries in the `os_historystep` field if you want to keep track of the status changes. This is entirely optional. Leaving it out won't cause any issues except that the records won't be available for reporting at a later stage.
6. Update the `os_currentstep_prev` and `os_historystep_prev` tables accordingly. These tables hold the **id** of the previous record. This is again optional.
7. Commit the changes and start JIRA.
8. Do a full re-index by going to **Administration** | **System** | **Advanced** | **Indexing**. This step is required to refresh the search results.

Retrieving users and groups from a database

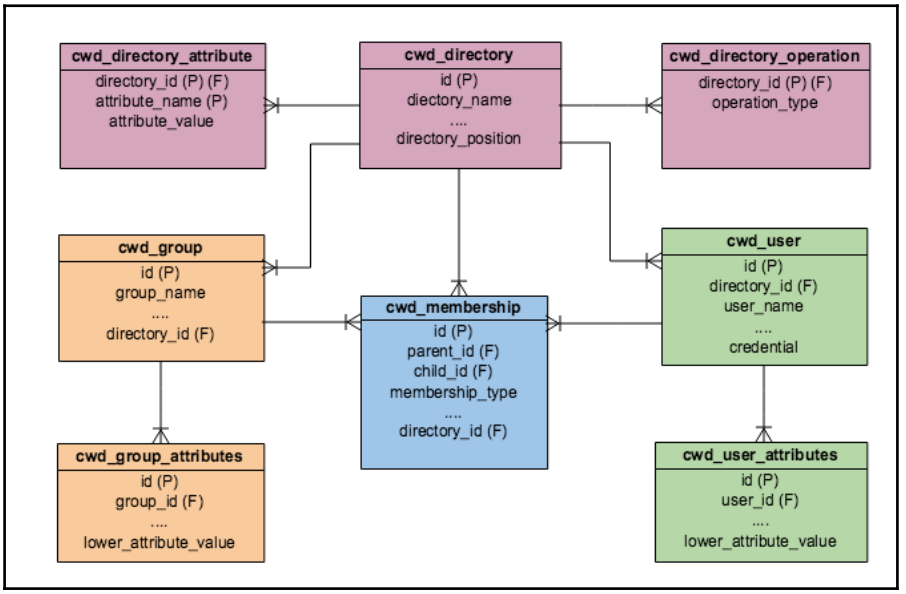
We can find all the information about JIRA users and their groups from the database by running a few simple SQL queries. In this recipe, we will see the various tables involved.

JIRA uses **Embedded Crowd** as its user management framework. Here, the users are stored in the `cwd_user` table, groups are in the `cwd_group` table and the membership details are in the `cwd_membership` table. It is possible to have the **group-user** membership or the **group-group** membership, the latter for nested groups, and this information is also stored in the `cwd_membership` table. **User** attributes are stored in the `cwd_user_attributes` table and **Group** attributes are stored in the `cwd_group_attributes` table.

JIRA also have the concept of user directories. A JIRA instance can have multiple directories and different directories can have the same name in them. The directory details are stored in the `cwd_directory` table and its attributes in the `cwd_directory_attribute` table. There are references in the `cwd_user` table and the `cwd_group` table, both with the name `directory_id`, and pointing to the appropriate directory **id**. The `cwd_directory_operation` table stores the available operations on a directory, based on user permissions.

When there are multiple users with the same name in different directories, JIRA will only recognize the user in the directory with the highest priority. The priority is stored in the `directory_ position` column.

The ER diagram for user/group relations in JIRA can be drawn as follows:



There is another table, `userassociation`, that holds the information about watching an issue and voting on an issue. In this table, the `source_name` column holds the unique username and `sink_node_id` holds the **id** of the issue. `sink_node_entity` has the value **Issue** and `association_type` has the value **WatchIssue** or **VoteIssue**, depending on the operation.

How to do it…

With the simple layout of the table structure, it is quite easy to list the users, groups, or their relationships by directly accessing the database. For example, we can find all the users in a group by simply running the following command:

```
SELECT child_name
     FROM cwd_membership
     WHERE parent_name='jira-administrators'
     AND membership_type = 'GROUP_USER'
     AND directory_id = 1;
```

Here, we consider the **directory** as well, because we can have the same users and groups in different directories.

Properties like full name and e-mail are part of the `cwd_user` table. But there can be other attributes such as **last login time** and **invalid password attempts** that are stored in the `cwd_user_attributes` table. We can access those as shown in the following command:

```
SELECT attribute_name, attribute_value
     FROM cwd_user_attributes
     WHERE user_id = (SELECT id FROM cwd_user
     WHERE user_name = 'someguy' AND directory_id =1);
```

Users watching an issue can be retrieved as follows:

```
SELECT source_name
     FROM userassociation
     WHERE association_type = 'WatchIssue'
     AND sink_node_entity = 'Issue'
     AND sink_node_id = (SELECT id FROM jiraissue WHERE id='10000');
```

Similarly, all the issues watched by a user can be retrieved as follows:

```
SELECT concat(pr.pkey, '-',ji.issuenum) as issuekey
     FROM jiraissue ji
     INNER JOIN userassociation ua ON ua.sink_node_id =  ji.id
     INNER JOIN project pr ON ji.project = pr.id
     WHERE ua.association_type = 'WatchIssue'
     AND ua.sink_node_entity = 'Issue' AND ua.source_name = 'someguy';
```

It is the same for votes, except that the association type will be **VoteIssue**.

Dealing with change history in a database

Before we wind up this chapter, let us touch on change history tables as well. Change histories on issues hold important information on what was changed and when. It is sometimes very useful for reporting and, sometimes, we find ourselves manually adding change histories in the database to keep a record of the changes we made via SQL-such as updating the status of an issue via SQL as we saw earlier in this chapter.

A set of changes happening on an issue at a single point of time is grouped together to form a **change group**. There is an entry for each such change group in the `changegroup` table, with the information about the issue on which the change is made, the user who made the change, and the time at which the changes happened.

Then, there is an entry for each of those individual changes in the `changeitem` table, all pointing to the respective `changegroup`. The `changeitem` table holds information on what was actually changed-the **old value** and the **new value**. There can be both numerical and textual representation in some cases, such as **status**, where there is a human-readable text (status name) as well as a unique **id** (status id). They are stored in `oldvalue` & `oldstring`, and `newvalue` & `newstring` respectively.

How to do it…

Let us have a look at both retrieving change histories and adding them. For a given issue, we can find out all the changes that happened on it using a simple join, as follows:

```
SELECT  cg.author, cg.created, ci.oldvalue, ci.oldstring, ci.newvalue,
ci.newstring
      FROM changegroup cg INNER JOIN changeitem ci
      ON cg.id = ci.groupid
      WHERE cg.issueid = (SELECT id FROM jiraissue WHERE id = '10000')
      ORDER BY cg.created;
```

Here, 10000 is the **id** of the issue. It is quite easy to modify this query to filter out changes made by a user or during a particular period.

Now, let us quickly have a look at adding a new change on an issue via the database. The following are the steps:

1. Stop the JIRA server.
2. Connect to the JIRA database.

3. Identify the `seq_id` value for the `ChangeGroup` table, which holds the next **id** value for the `changegroup` table:

```
SELECT seq_id
    FROM sequence_value_item
    WHERE seq_name = 'ChangeGroup';
```

4. Create an entry in the `changegroup` table, with the correct **id** of the issue, author name, and created time:

```
INSERT INTO changegroup
    VALUES (12345,10000,'jobinkk','2016-01-15');
```

   Here **12345** is the sequence id value retrieved in step 3 and **10000** is the issue id.

5. Identify the `seq_id` value for `ChangeItem` table, which holds the next **id** value for `changeitem` table:

```
SELECT seq_id
    FROM sequence_value_item
    WHERE seq_name = 'ChangeItem';
```

6. Insert a change item for the preceding change group we created in step 4. Let us consider the status change we made in the earlier recipe:

```
INSERT INTO changeitem
 VALUES (110700, 12345, 'jira','status','1',' Open','10001','Done');
```

   Here, **110700** is the sequence id value retrieved in step 5 and the `groupid` value is the **id** entered into the `changegroup` table in step 4 (**12345**). The third column holds the field type, which could be `jira` or `custom`. For all the standard JIRA fields such as **summary** and **status**, the field type is `jira`. For custom fields, we use the field type `custom`.

   For fields such as **status**, there is a textual representation (the name) and there is a unique **id**. Hence, both the `oldvalue` and `oldstring` columns are populated. The same is the case with the `newvalue` and `newstring` columns. For fields such as **summary**, only the `oldstring` and `newstring` columns need to be populated.

7. Update the `sequence_value_item` table to hold a higher value in the seq_id column for the ChangeGroup and ChangeItem entities. In our example, we can give a value of **12346** for `ChangeGroup` and **110701** for `ChangeItem`. Ofbiz normally allocates **id**s in batches of 10, so `seq_id` is the next available **id**, rounded up to the nearest value of 10, though adding 1 should be enough:

```
UPDATE sequence_value_item SET seq_id = 12346
    WHERE seq_name = 'ChangeGroup';
UPDATE sequence_value_item SET seq_id = 110701
    WHERE seq_name = 'ChangeItem';
```



This step is required whenever a row is inserted into any of the JIRA tables. The `seq_id` value in the `sequence_value_item` table should be updated for the entity where the new row is added. The new sequence value should be at least one more than the `max(id)` value of the entity.

8. Commit the changes and start JIRA.
9. Re-index the JIRA instance by going to **Administration** | **System** | **Advanced** | **Indexing**.

With this, a new change history record is added into the JIRA issue and you can view it under the **History** tab on the issue page.

11
Useful Recipes

In this chapter, we will cover:

- Writing a service in JIRA
- Adding configurable parameters to a service
- Writing scheduled tasks in JIRA
- Writing listeners in JIRA
- Customizing e-mail content
- Redirecting to a different page in webwork actions
- Adding custom behavior for user details
- Deploying a servlet in JIRA
- Adding shared parameters to Servlet Context
- Writing a ServletContextListener
- Using filters to intercept queries in JIRA
- Adding and importing components in JIRA
- Adding new module types to JIRA
- Enabling access logs in JIRA
- Enabling SQL logging in JIRA
- Internationalization in webwork plugins
- Sharing common libraries across v2 plugins
- Operations via direct HTML links
- Implementing Marketplace licensing in plugins

Introduction

So far, we have grouped recipes under common themes as different chapters in this book. We have explored all the important themes but we are still left with some useful recipes and a handful of plugin modules that have not yet been covered in previous chapters.

In this chapter, we will look at some of those powerful plugin points, as well as useful tricks in JIRA that have not been covered in earlier chapters. Not all of these recipes are related, but they are all useful in their own way.

Writing a service in JIRA

A service that runs at regular intervals is a much-wanted feature in any web application. It is even more desirable if it can be managed with user-configured parameters and without a reboot, and so on. JIRA offers a mechanism to add new services that run at regular intervals after every start-up. It lets us do things related to JIRA and things independent of it. It lets us integrate with third-party applications. It lets us do wonders!

There are built-in services in JIRA; Export Service, POP Service, Mail Service, and so on, to name a few. In this recipe, we are going to see how we can add a custom service to JIRA.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK.

How to do it…

As opposed to the other JIRA plug-in modules, services are not defined inside the plugin descriptor. Instead, they use a configuration XML. They are still plugins but without any specific definitions in the `atlassian-plugin.xml`, and they have classes, files, and configuration XML related to the service residing inside.

Perform the following steps to write a simple service that just prints something onto the server console.

1. Write the configuration XML. This is the most important part of a service. The following is a simple configuration XML:

```
<someservice id="jtricksserviceid">
    <description>My New Service</description>
    <properties></properties>
</someservice>
```

   This is a simple configuration XML that doesn't take any properties. It has a root element and a unique **id**, both of which can be custom names of your choice. The root element we have is `someservice` and the **id** is `jtricksserviceid`. The **description**, as the name suggests, is just a short description of the service. The properties tag holds the different **properties** you want to associate with the service. The user, while configuring the service, will enter these properties. We will see more on that later.

2. Put the XML file in a folder under `src/main/resources`. In our example, we have put it under `com/jtricks/services`.
3. Create the service class. The class can be put under any package structure as it will be referenced with the fully qualified name when it is added in JIRA. The class should extend `AbstractService`, which implements `JiraService`:

```
public class JTricksService extends AbstractService {
 ...
}
```

4. Implement the mandatory methods in the service class. The following are the only ones that you need to implement:

```
@Override
public void run() {
   System.out.println("Running the JTricks service!!");
}
@Override
public ObjectConfiguration getObjectConfiguration() throws
ObjectConfigurationException {
    return getObjectConfiguration("MYNEWSERVICE",
    "com/jtricks/services/myjtricksservice.xml", null);
}
```

Here, `run` is the key method that is executed when the service runs at the scheduled time.
The other key mandatory method is `getObjectConfiguration`. We get the configurations from the XML we wrote earlier (in *Step1*) in this method. All we need to do here is to call the parent class' `getObjectConfiguration` method by passing three arguments. The first argument is a unique **id** (which need not be same as the ID in the XML file). This **id** is used as a key while saving configurations internally. The second one is the **path** to the configuration XML file we wrote earlier, and the third argument is a **Map,** using which you can add user parameters to the object configuration.



The third argument is mostly `null` in the case of services as these user parameters are not used anywhere. It is meaningful in other places in JIRA, such as portlets, though not in the case of services.

5. Install the plugin using Universal Plugin Manager.
   Now the service is ready. We can go to **Administration** | **System** | **Advanced** | **Services** and add the new service with the appropriate schedule. While adding the service, we need to use the fully qualified name of the service class. More about registering a service can be found at `http://confluence.atlassian.com/display/JIRA/Services#Services-RegisteringaService`, but it is outside the scope of the book.



It is also possible to just create a JAR file with the service classes, files, and configuration XML, but without `atlassian-plugin.xml`, and drop it into the `<jira-application-dir>/WEB-INF/lib` directory. If you do this, you will have to restart JIRA after dropping the JAR file. Also, you will have to copy the files over after every upgrade. Hence, a plugin is recommended over this approach.

How it works…

As soon as the service is added, it will start running in the background at the scheduled interval. In our example, the service will print to output logs, as shown below.

```
2016-05-24 22:13:46,413 http-nio-8080-exec-7 WARN jobinkk 1333x7740x1 x4swnz fe80:0:0:0:0:0:0:1%1 /secure/admi
n/EditService!default.jspa [w.view.taglib.IteratorTag] Value is null! Returning an empty set.
2016-05-24 22:29:44,729 UpmAsynchronousTaskManager:thread-3 INFO jobinkk 1237x6505x1 g9pujl 0:0:0:0:0:0:0:1 /r
est/plugins/1.0/ [c.a.plugin.loaders.ScanningPluginLoader] No plugins found to be installed
2016-05-24 22:29:44,806 UpmAsynchronousTaskManager:thread-3 INFO jobinkk 1237x6505x1 g9pujl 0:0:0:0:0:0:0:1 /r
est/plugins/1.0/ [c.a.plugin.util.WaitUntil] Plugins that have yet to be enabled: (1): [com.jtricks.jtricks-ut
ilities], 60 seconds remaining
2016-05-24 22:31:40,597 http-nio-8080-exec-5 WARN jobinkk 1351x7848x1 1crdg3f fe80:0:0:0:0:0:0:1%1 /secure/adm
in/EditService!default.jspa [w.view.taglib.IteratorTag] Value is null! Returning an empty set.
Running the JTricks service!!
Running the JTricks service!!
Running the JTricks service!!
Running the JTricks service!!
Running the JTricks service!!
```

See also

- The *Adding configurable parameters to a service* recipe in this chapter

Adding configurable parameters to a service

For a simple service, such as the one we just wrote, there is only one parameter that can be configured – the schedule at which the service runs! But what if we need to add more parameters? Let's say we want to add the *tutorial name* in the service, which can be changed later if needed.

How to do it…

The following are the steps required:

1. Modify the service configuration XML to include the configurable properties:

```
<someservice id="jtricksserviceid">
    <description>My New Service</description>
    <properties>
        <property>
            <key>Tutorial</key>
            <name>The tutorial you like</name>
            <type>string</type>
        </property>
    </properties>
</someservice>
```

Here, we have added a string **property** with the key: *Tutorial*.

2. Override the`init` method in the service class to retrieve the new property:

```
@Override
public void init(PropertySet props) throws
ObjectConfigurationException {
    super.init(props);
    if (hasProperty(TUTORIAL)) {
        tutorial = getProperty(TUTORIAL);
    } else {
    tutorial = "I don't like tutorials!";
    }
}
```

   Here, we retrieved the property **Tutorial** from the `PropertySet` in the `init` method.

3. Use the property as appropriately in the run() method. Here, let us just print the tutorial name:

```
@Override public void run() {
    System.out.println("Running the JTricks service!! Tutorial? "
     + tutorial);
}
```

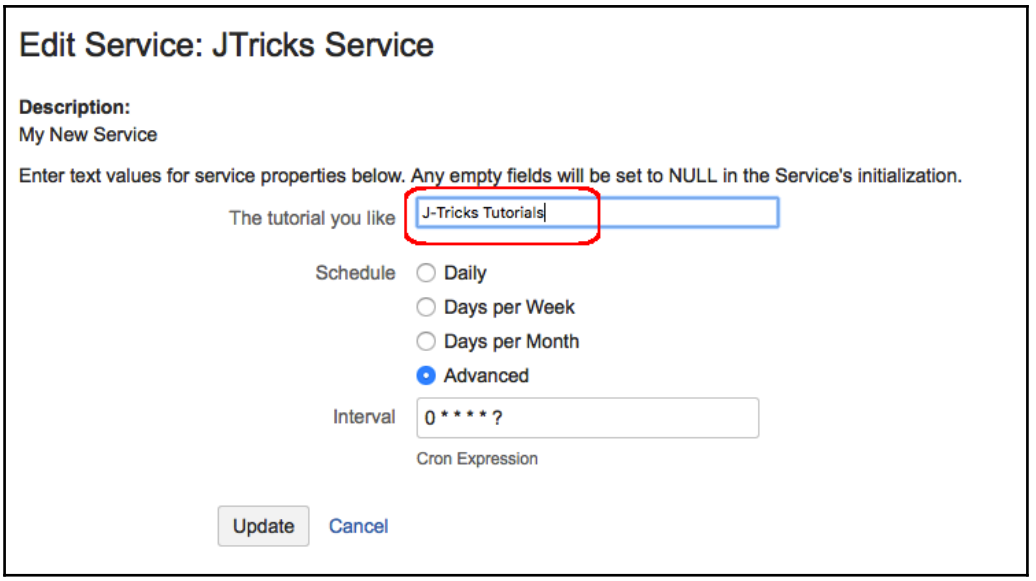
How it works…

The `init` method will be called whenever the service is configured or re-configured. The property values we entered on the JIRA Admin UI are retrieved in this method for use in the `run` method.

We can also optionally override the `destroy` method to do anything we want before the service is removed!

Once the service is deployed and added in the UI, it prints **Running the JTricks service Tutorial? I don't like tutorials!,** as the **tutorial** property is not configured yet.

We can set the property by editing the service at **Administration** | **System** | **Advanced** | **Services**, as shown here:



After adding the property value, the service will pick it up from the next run:

```
2016-05-24 22:49:40,823 http-nio-8080-exec-6 INFO jobinkk 1369x211x1 vwb92c 0:0:0:0:0:0:0:1 /secure/admin/Edit
Service!default.jspa [c.a.j.web.tags.TextTag] An empty i18n key was provided in /secure/admin/views/services/e
ditservice.jsp
2016-05-24 22:49:40,841 http-nio-8080-exec-6 WARN jobinkk 1369x211x1 vwb92c 0:0:0:0:0:0:0:1 /secure/admin/Edit
Service!default.jspa [w.view.taglib.IteratorTag] Value is null! Returning an empty set.
Running the JTricks service!! Tutorial? I don't like tutorials!
Running the JTricks service!! Tutorial? I don't like tutorials!
2016-05-24 22:51:52,277 http-nio-8080-exec-2 INFO jobinkk 1371x228x1 vwb92c fe80:0:0:0:0:0:0:1%1 /secure/admin
/EditService!default.jspa [c.a.j.web.tags.TextTag] An empty i18n key was provided in /secure/admin/views/servi
ces/editservice.jsp
2016-05-24 22:51:52,285 http-nio-8080-exec-2 WARN jobinkk 1371x228x1 vwb92c fe80:0:0:0:0:0:0:1%1 /secure/admin
/EditService!default.jspa [w.view.taglib.IteratorTag] Value is null! Returning an empty set.
Running the JTricks service!! Tutorial? I don't like tutorials!
Running the JTricks service!! Tutorial? J-Tricks Tutorials
Running the JTricks service!! Tutorial? J-Tricks Tutorials
Running the JTricks service!! Tutorial? J-Tricks Tutorials
```

See also

- The *Writing a service in JIRA* recipe in this chapter

Writing scheduled tasks in JIRA

Have you ever thought of running scheduled tasks within JIRA? While services can run as scheduled tasks in JIRA 7, there might be scenarios where we want to do everything in code. We might want to remove the extra step of registering a service and the possibility of an admin deleting/editing/rescheduling the service.

Atlassian Scheduler API and the **SchedulerService** give us an easy way to write scheduled jobs in JIRA. In this recipe, we will write a simple scheduled task and see how to automatically register them.

Just like services, these scheduled tasks can also be part of a v2 plugin and can be uploaded using UPM.

How to do it…

Let us write a simple scheduled task that prints a line in the console. Perform the following steps:

1. Write a java class that implements the `com.atlassian.scheduler.JobRunner` interface.
2. Register the new class as a **component** and make it an **OSGI** service using the `ExportAsService` and `Named` annotations, as shown here. This is equivalent to defining a `component` with `public="true"` when **Atlassian Spring Scanner** is not used.

```
@ExportAsService({ JTricksJob.class })
@Named("jtricksJob")
public class JTricksJob implements JobRunner {
    ......
}
```

3. Implement the `runJob` method.

```
@Override
public JobRunnerResponse runJob(JobRunnerRequest request) {
    System.out.println("Running JTricksJob at " +
    request.getStartTime());
    return JobRunnerResponse.success();
}
```

Here, we have kept it very simple by printing a line and returning success. We can use this method to do complex operations and can return `success`, `failed`, or `aborted` statuses, as required. The `JobRunnerRequest` also contains the configuration details set at the time of registering the job. We can retrieve those configurations and use them in the `runJob` method, if needed.

That is all it takes to define a scheduled job. Now we can schedule this job from a custom plugin component we already have or we can automatically register it when the plugin has installed.

To automatically register and unregister the job, we can listen on plugin installation and un-installation events, as we saw in `Chapter 2`, *Understanding the Plugin Framework*. Following are the steps required:

1. Modify the plugin `pom.xml` to include the spring dependencies, as shown here:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>2.5.6</version>
    <scope>provided</scope>
</dependency>
```



Make sure the project is rebuilt/refreshed with the new dependencies.

2. Modify the `JTricksJob` class to implement the `InitializingBean` and `DisposableBean` interfaces:

```
public class JTricksJob implements JobRunner, InitializingBean,
DisposableBean
{
    .....
}
```

3. Implement the `afterPropertiesSet` method. This method is invoked when the plugin is installed or enabled. We can register the scheduled job we wrote earlier using `SchedulerService`.

There are a few things that need to be defined to register the job:

a. Define a unique for the job runner. This will be used by `SchedulerService` while registering the class:

```
private static final JobRunnerKey JOB_RUNNER_KEY =
JobRunnerKey.of(JTricksJob.class.getName());
```

b. Define a run interval:

```
private static final long EVERY_MINUTE =
TimeUnit.MINUTES.toMillis(1);
```

This could be different based on your requirements.

c. Define a job **config** using the required options:

```
final JobConfig jobConfig =
JobConfig.forJobRunnerKey(JOB_RUNNER_KEY).withRunMode
(RunMode.RUN_LOCALLY).withSchedule(Schedule.forInterval
(EVERY_MINUTE, null));
```

d. Define a unique ID to schedule the job using the above job config:

```
private static final JobId JOB_ID =
JobId.of(JTricksJob.class.getName());
```

e. Inject `SchedulerService` in the class, if not already done:

```
private final SchedulerService scheduler;
@Inject
public JTricksJob(@ComponentImport SchedulerService scheduler) {
    this.scheduler = scheduler;
}
```

It can also be retrieved using `ComponentAccessor`.

f. Register the job runner using the **key** defined at in step 3a.

```
scheduler.registerJobRunner(JOB_RUNNER_KEY, this);
```

g. Schedule the job using the **id** defined in step 3d and **jobConfig** defined in step 3c.

```
scheduler.scheduleJob(JOB_ID, jobConfig);
```

The full method will look like the following:

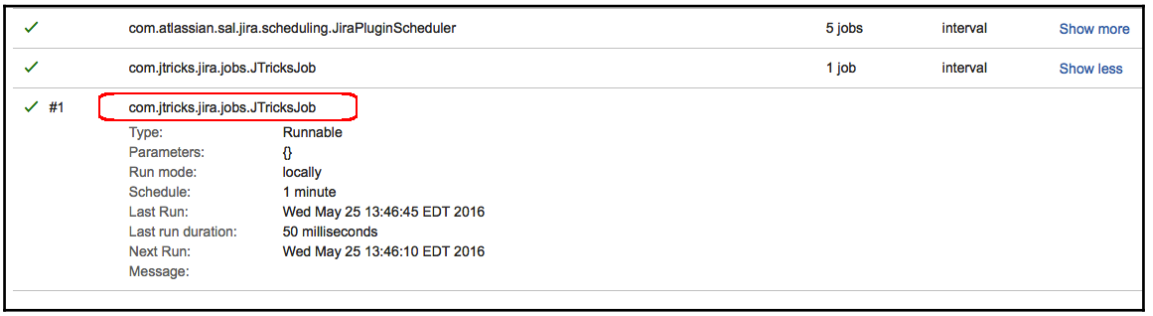
```
@Override
public void afterPropertiesSet() throws Exception {
    scheduler.registerJobRunner(JOB_RUNNER_KEY, this);
    final JobConfig jobConfig =
    JobConfig.forJobRunnerKey(JOB_RUNNER_KEY).withRunMode
    (RunMode.RUN_LOCALLY).withSchedule(Schedule.forInterval
    (EVERY_MINUTE, null));
    try {
        scheduler.scheduleJob(JOB_ID, jobConfig);
    } catch (SchedulerServiceException e) {
        e.printStackTrace();
    }
}
```

4. Implement the `destroy` method to **unschedule** the job if needed.

With this, the job will automatically register whenever the plugin is installed or enabled.

How it works…

Once the plugin is installed, the scheduler is automatically registered, and you can find the details at **Administration** | **System** | **Troubleshooting and Support** | **Scheduler details**, as shown here:



The job will run at the scheduled intervals. In our example, it will print the message to the console, as shown here:

```
2016-05-25 00:03:30,740 UpmAsynchronousTaskManager:thread-1 DEBUG jobinkk 1432x31x3 cs89qh fe80:0:0:0:0:0:0:1%1
/rest/plugins/1.0/installed-marketplace [c.a.activeobjects.osgi.ActiveObjectsServiceFactory] onPluginDisabledEve
nt removing delegate for [com.jtricks.jtricks-utilities]
2016-05-25 00:03:30,763 UpmAsynchronousTaskManager:thread-1 INFO jobinkk 1432x31x3 cs89qh fe80:0:0:0:0:0:0:1%1 /
rest/plugins/1.0/installed-marketplace [c.a.plugin.loaders.ScanningPluginLoader] Removed plugin 'com.jtricks.jtr
icks-utilities'
2016-05-25 00:03:30,807 UpmAsynchronousTaskManager:thread-1 INFO jobinkk 1432x31x3 cs89qh fe80:0:0:0:0:0:0:1%1 /
rest/plugins/1.0/installed-marketplace [c.a.plugin.util.WaitUntil] Plugins that have yet to be enabled: (1): [co
m.jtricks.jtricks-utilities], 60 seconds remaining
Starting...
Running JTricksJob at Wed May 25 00:03:30 EDT 2016
Running JTricksJob at Wed May 25 00:04:30 EDT 2016
Running JTricksJob at Wed May 25 00:05:30 EDT 2016
Running JTricksJob at Wed May 25 00:06:30 EDT 2016
```

See also

- The *Capturing plugin installation/uninstallation events* recipe in `Chapter 2`, *Understanding the Plugin Framework*

Writing listeners in JIRA

Listeners are very powerful features in JIRA. JIRA has a mechanism for throwing events whenever something happens on an issue, such as creating an issue, updating an issue, progressing on the workflows, and so on. Using listeners, we can capture these events and do special things based on our requirements.

Listeners are implemented using the `atlassian-event` library. All we have to do is to register the listener class as a **component** and then use the `EventListener` annotation to listen on the events.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK.

How to do it…

Let us write a simple listener that prints a different message to the console based on different issue events. Following are the steps required.

1. Modify the plugin `pom.xml` to include the spring dependencies, as shown here:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>2.5.6</version>
    <scope>provided</scope>
</dependency>
```



Make sure the project is rebuilt/refreshed with the new dependencies.

2. Create a Java class that implements the `InitializingBean` and `DisposableBean` interfaces:

```
public class JTricksListener implements InitializingBean,
DisposableBean {
.....
}
```

   This will enable us to listen on plugin installation and uninstallation events and we can register/un-register the listener on such events, as we did with the job scheduler in the previous recipe.

3. Register the new class as a **component** and make it an **OSGI** service using the `ExportAsService` and `Named` annotations, as shown below. This is equivalent to defining a `component` with `public="true"` when **Atlassian Spring Scanner** is not used:

```
@ExportAsService({JTricksListener.class })
@Named("jtricksListener")
public class JTricksListenerimplements InitializingBean,
DisposableBean {
    ......
}
```

4. Inject `EventPublisher` in the class, if not already done:

```
private final EventPublisher eventPublisher;
@Inject
public JTricksListener(@ComponentImport EventPublisher
eventPublisher) {
    this.eventPublisher= eventPublisher;
}
```

We will be using the event publisher to register/un-register the listener.

5. Implement the `afterPropertiesSet` method. This method is invoked when the plugin is installed or enabled:

```
@Override
public void afterPropertiesSet() throws Exception {
    eventPublisher.register(this);
}
```

As you can see, we are registering the listener class in this method.

6. Implement the `destroy` method and un-register the listener in that method:

```
@Override
public void destroy() throws Exception {
    eventPublisher.unregister(this);
}
```

7. Write a method with the `EventPublisher` annotation to capture issue events. This event can take the `IssueEvent` object as an argument and use it to figure out the event type, as shown here:

```
@EventListener
public void onIssueEvent(IssueEvent issueEvent) {
    Long eventTypeId = issueEvent.getEventTypeId();
    Issue issue = issueEvent.getIssue();
    if (eventTypeId.equals(EventType.ISSUE_CREATED_ID))
    {
        System.out.println("Created issue:" + issue.getKey());
    } else if (eventTypeId.equals(EventType.ISSUE_RESOLVED_ID))
    {
        System.out.println("Resolved issue:" + issue.getKey());
    } else
    {
        System.out.println("Event:" + issueEvent.getEventTypeId() +
        " thrown on issue:" + issue.getKey());}}
```

As you can see, we are comparing the event type ID with known events and then printing different messages based on them.

8. Package and deploy the plugin.

How it works…

As soon as the plugin is deployed, the listener is automatically registered and it will start picking up new issue events, as shown below.

If the listener was not registered using `EventPublisher` in the lifecycle events, you need to manually add it under **Administration** | **System** | **Advanced** | **Listeners**. See `https://confluence.atlassian.com/adminjiraserver071/listeners-802592235.html#Listeners-Registeringalistener` for more details:

```
Starting...
Running JTricksJob at Wed May 25 00:15:14 EDT 2016
2016-05-25 00:15:31,931 StreamsCompletionService::thread-1 DEBUG jobinkk 15x293x4 4m8c9i fe80
:0:0:0:0:0:0:1%1 /plugins/servlet/streams [c.a.activeobjects.osgi.ActiveObjectsServiceFactory
] getService bundle [com.atlassian.streams.thirdparty-plugin]
25-May-2016 00:15:32.346 WARNING [http-nio-8080-exec-22] com.sun.jersey.spi.container.servlet
.WebComponent.filterFormParameters A servlet request, to the URI http://localhost:8080/rest/a
ctivity-stream/1.0/preferences?_=1464149731704, contains form parameters in the request body
but the request body has been consumed by the servlet or a servlet filter accessing the reque
st parameters. Only resource methods using @FormParam will work as expected. Resource methods
 consuming the request body by other means will not work as expected.
Created issue:DEMO-8
Event:13 thrown on issue:DEMO-8
Event:13 thrown on issue:DEMO-8
Running JTricksJob at Wed May 25 00:16:14 EDT 2016
```

See also

- The *Capturing plugin installation/un-installation events* recipe in `Chapter 2`, *Understanding Plugin Framework*

Customizing e-mail content

We have already seen how JIRA throws various events when something happens and how we can handle these events to do various things. One example includes sending e-mail notifications to users based on the notification schemes that are set up in JIRA. But what if we don't like the default content of JIRA notifications? What if we just want a different wording or maybe even need to amend the e-mail content?

In this recipe, we will see how we can customize e-mail content that is sent as a notification when an event is thrown in JIRA.

How to do it…

JIRA has a set of e-mail templates written using velocity and rendered when a notification is sent. For each event, a template is configured within JIRA and that template is used when the event is thrown. We can either create new templates and edit the events to use these new templates or modify the existing templates and leave the events as they are!

In both cases, the steps are pretty similar and are as follows:

1. Identify the event for which the notification needs to be changed. The event could be an existing JIRA event such as **Issue Created** and **Issue Updated** or a custom event that the JIRA administrator has created.
2. Find the template mapped to the event. For each event, be it **system-based** or **custom**, there is a template associated with it. We cannot change the templates associated with a system event. For example, an **Issue Updated** event is associated with an **Issue Updated** template. We can, however, choose any template for the custom events we have added.
3. The e-mail template mapping for the chosen template can be found at `atlassian-jira/WEB-INF/classes/email-template-id-mappings.xml`. In this file, we can find many templates associated with each event. For example, the **Issue Updated** event has the following entry:

```
<templatemappings>
    ...
    <templatemapping id="2">
        <name>Issue Updated</name>
        <template>issueupdated.vm</template>
        <templatetype>issueevent</templatetype>
    </templatemapping>
    ...
</templatemappings>
```

   Here, we can add new mappings if we are adding new templates:

```
<templatemappings>
    ...
    <templatemapping id="10001">
        <name>Issue Approved</name>
        <template>issueapproved.vm</template>
        <templatetype>issueevent</templatetype>
```

```
        </templatemapping>
    </templatemappings>
```

Make sure the **id** we use here is unique in the file.

4. Identify the template to be edited, if we are customizing an existing template, or add a new template with the name mentioned in the `email-template-id-mappings.xml` file.Email templates are stored under two different locations within JIRA, one for**HTML** e-mails and another for **Text** e-mails. The templates for those can be found at `WEB-INF/classes/templates/email/html` and `WEB-INF/classes/templates/email/text` respectively. In addition to these, the **subject** of the e-mail can be found under `WEB-INF/classes/templates/email/subject`.



Note that the name of the template is the same in all three places. In our example, the name of the template being edited is issueupdated.vm, and hence if we need to only modify the subject, we just need to modify the `WEB-INF/classes/templates/email/subject/issueupdated.vm` file. Similarly, HTML or text content can be edited at `WEB-INF/classes/templates/email/html/issueupdated.vm` or `WEB-INF/classes/templates/email/text/issueupdated.vm` respectively.

If we are adding the template, `issueapproved.vm` in our case, we need to create three templates, one each for subject, HTML body, and text body, all with the same name and placed in the respective folders.

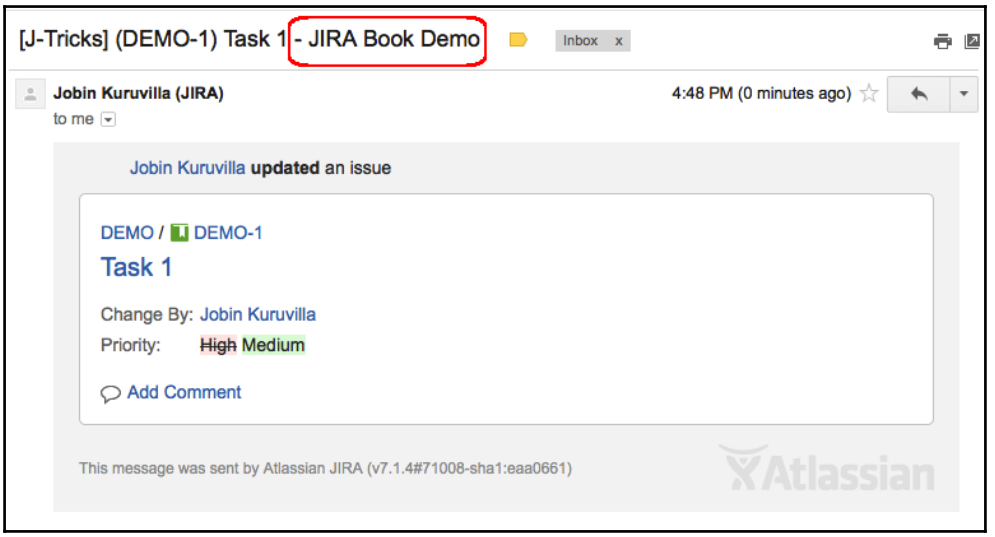
5. Restart JIRA after editing the templates appropriately.

How it works…

Let us assume that we updated the `issueupdated.vm` file under the `WEB-INF/classes/templates/email/subject` folder to modify the subject of the **Issue Updated** event notifications. The updated file looks like the following:

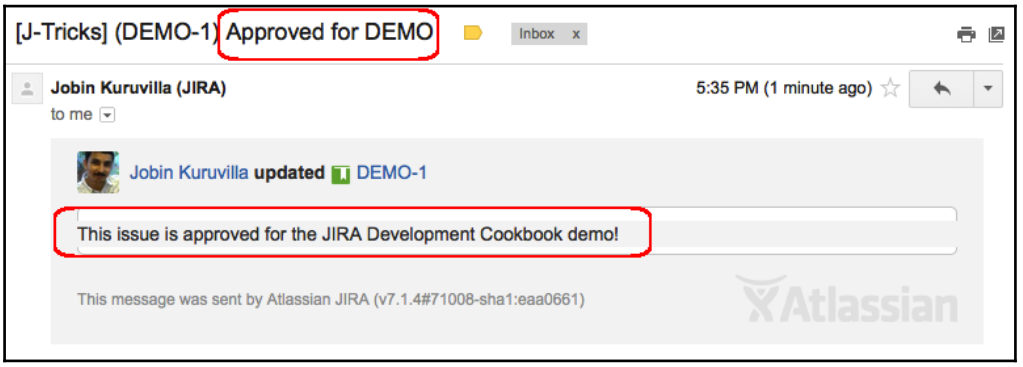
```
#disable_html_escaping()
($issue.key) $issue.summary - JIRA Book Demo
```

The resulting notification will look like the following:



On the other hand, if we added a new template and restarted JIRA, we can associate it with the custom events we have created. When the notification is sent, JIRA will use the updated or newly added templates to render the e-mail content.

Let us assume that we created the new `issueapproved.vm` file under the respective folders. The following screenshot shows how a sample notification will look as follows:



As you can see, we have used a custom **subject** line for custom **body** text. We can, of course, use different styles to make it prettier, but the process is the same.

Redirecting to a different page in webwork actions

This recipe covers a very simple concept in JIRA web actions. While writing plugins, we often come across scenarios where we need to navigate to a new page, such as a Dashboard, to browse a new project, or view another issue after the action has executed. `JiraWebActionSupport` provides a simple method for doing this, which we will see in this recipe.

How to do it…

What if we want to navigate to the Dashboard instead of rendering a **success** view when an action is executed? What if we can't directly link it from the JSP page or the velocity template because we want to perform something in the action class beforehand?

All you need to do here is to return `getRedirect(URL)` in the action class's `doExecute` method (or the appropriate method)! This method will redirect to the specified location after the action method has successfully finished. If there are any errors, it will go to the error page, as the `getRedirect()` method returns `Action.ERROR` in that case.

You can force redirect to the URL even if there are errors by using `forceRedirect(URL)` instead of the `getRedirect()` method. It doesn't clear the return URL and will always go to the redirect URL.

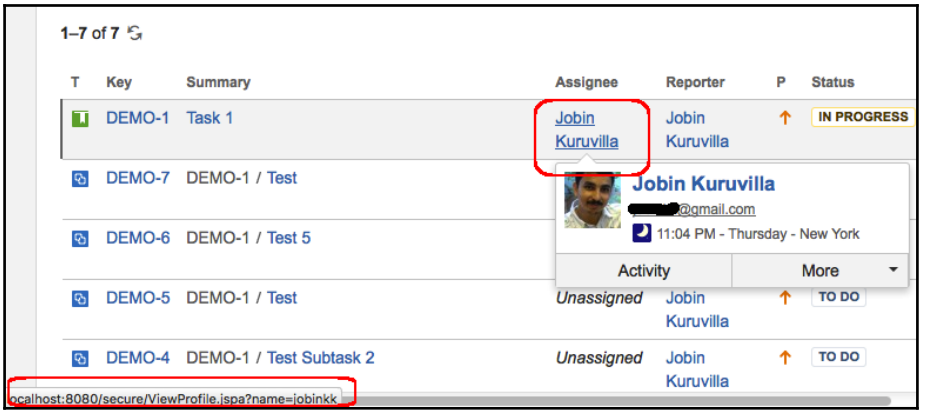
For example, if we need to return to the Dashboard after **SUCCESS**, we can do so as follows:

```
@Override public String doExecute() throws Exception {
    System.out.println("Action invoked. Doing something important before
    redirecting to Dashboard!");
   return getRedirect("/secure/Dashboard.jspa");
}
```

Replacing `getRedirect` with `forceRedirect` will take the user to the Dashboard irrespective of the result.

Adding custom behavior for user details

In JIRA, you can see that user details are formatted with the full name and a link to the user's profile within the application. For example, when the issues are displayed in the issue navigator, the assignee and reporter are displayed as follows:



You can see that the link points to the profile page.

But what if we want to change how user details are displayed? Say, we want to display the user's avatar alongside their name, or that we want to display their usernames with an external link, such as a link to the user's Twitter profile.

JIRA provides the **User Format plugin** module to serve this purpose. Using this module, we can define different formats in which the user will be displayed, and we can use them within the existing JIRA display or within our custom plugins.

Getting ready

Create a skeleton plugin using the **Atlassian Plugin SDK**.

How to do it…

In this recipe, let us try to create a new user profile that will display the **username** (instead of the full name) with a link to their **Twitter** profile to add some spice! The following are the steps to do it:

1. Add the `user-profile` module to the `atlassian-plugin.xml` file:

```
<user-format key="twitter-format" name="Twitter User Format"
  class="com.jtricks.jira.user.TwitterUserFormat" system="true">
    <description>User name linking to twitter</description>
    <type>twitterLink</type>
    <resource type="velocity" name="view"
      location="templates/user/twitterLink.vm"/>
    </user-format>
```

As with other plugin modules, the user profile module also has a unique **key**. It then points to the **class** that will be used by the user formatter, `TwitterUserFormat`in this case.



The type element holds the unique profile type name that will be used while formatting the user. The following types exist in JIRA by default with effect from version 7.1: **profileLink**,**profileLinkWithAvatar**, **avatarWithHover**, **fullName**, **fullNameHover**, **avatarFullNameHover**, **userName**, **profileLinkSearcher**, **profileLinkExternal**, **profileLinkActionHeader**, and **fullProfile**.

The **resource** element points to the velocity template to be used for rendering the view, `twitterLink.vm` in this case.

2. Create the formatter class in the previous step. The class should implement the `UserFormat` interface:

```
@Named("TwitterUserFormat")
public class TwitterUserFormat implements UserFormat {
    private final VelocityTemplatingEngine templatingEngine;
    @Inject
    public TwitterUserFormat(@ComponentImport
    VelocityTemplatingEngine
      templatingEngine) {
        this.templatingEngine = templatingEngine;
    }
    ........
}
```

Here, we inject `VelocityTemplatingEngine` into the class to render the velocity template, as shown in the next step.

3. Implement the required methods. We will have to implement the two overridden **format** methods.

   a. The first method takes a **username** and **id,** where the **username** is the name of the user, which can also be *null*, and the **id** is an extra argument that can be used to pass extra context to the renderer. Ideally, an implementation might include this **id** in the rendered output such that it can be used for test assertions. An example of how the **id** is used can be found by displaying the assignee in the column view (`/WEB-INF/classes/templates/jira/issue/field/assignee-columnview.vm`) where the **id** is **assignee**.

   We are not going to use **id** in the example, and the method is implemented as follows:

```
public String format(String username, String id) {
    final Map<String, Object>
    params = getInitialParams(username, id);
    return templatingEngine.render(file
      ("templates/user/twitterLink.vm"))
      .applying(params).asHtml();
}
```

   Where `getInitialParams` just populates the `params` map with the **username** as shown:

```
private Map<String, Object>
getInitialParams(final String username, final String id) {
    final Map<String, Object>
    params = MapBuilder.<String, Object>
    newBuilder().add("username", username).toMutableMap();
    return params;
}
```

   We can populate the map with as many things as needed if we want to render the user details in some other way!

   b. The second method takes **username**, **id**, and a **map** pre-populated with extra values to add more to the context! The method is implemented as follows:

```
public String format(String username, String id,
Map<String, Object> params) {
  final Map<String, Object>
  velocityParams = getInitialParams(username, id);
  velocityParams.putAll(params);
```

```
        return templatingEngine.render(file
       ("templates/user/twitterLink.vm"))
       .applying(velocityParams).asHtml();
    }
```

The only difference is that the extra context is also populated into the `params` map.

In both cases, the `VelocityTemplatingEngine` renders the velocity template defined in the `view` template.

4. Write the velocity template that uses the context populated in `params` map in the previous step to display the user information:

```
#disable_html_escaping()
#if ($username)
  #set ($quote = '"')
  #set($author = "<a id=${quote}assignee_$username)${quote}
     class=${quote}user-hover${quote}
     rel=${quote}$username${quote}
     href=${quote}https://twitter.com
     /${username}${quote}>$username</a>")
  #else #set($author = $i18n.getText('common.words.anonymous'))
  #end ${author}
```

In our example, we just display the username as it is with a link, `https://twitter.com/${username}`, that will point to the **twitter** account with that username. Note that the `quote` variable is assigned a **double quotation mark** inside a **single quotation mark**. Here, single quotation marks are the velocity syntax and double quotation marks are the value. It is used to construct the URL where name, class, href value, and so on are placed between quotes!

Don't forget to handle a scenario where the user is *null*. In our case, we just display the name as **Anonymous** when the user is null.

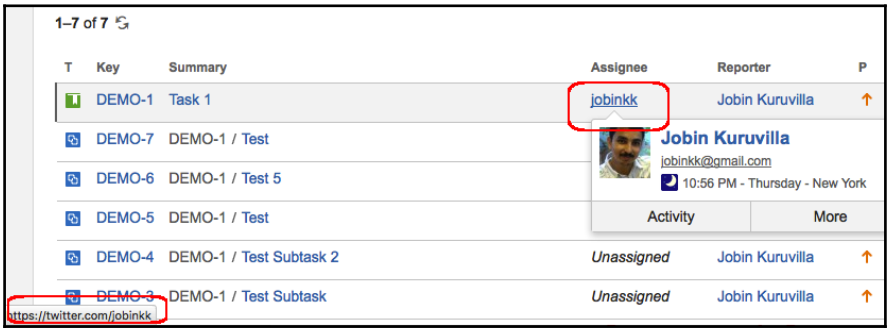
- Package the plugin and deploy it.

How it works…

Once the plugin is deployed, the new user profile, `twitterLink` in this case, can be used in various places in JIRA where appropriate. For example, the `/WEB-INF/classes/templates/jira/issue/field/assignee-columnview.vm` file can be modified to include a `twitterLink` profile instead of the default `profileLink` as follows:

```
#disable_html_escaping()
    #if ($assigneeUserkey)
        #if ($displayParams && $displayParams.nolink)
            $userformat.formatUserkey($assigneeUserkey, 'fullName',
            'assignee')
        #else <span class="tinylink">
           $userformat.formatUserkey($assigneeUserkey,'twitterLink',
           'assignee')
           </span>
        #end
   #else
        <em>$i18n.getText('common.concepts.unassigned')</em>
    #end
```

When you do that, the **assignee** column in issue navigator will appear as follows with a link to the user's Twitter account:



We can also use the new profile in the plugins to render user details just by invoking `formatUser`, as follows:

```
$userformat.formatUser($username, 'twitterLink', 'some_id')
```

Or:

```
$userformat.formatUser($username, 'twitterLink', 'some_id',
$someMapWithExtraContext)
```

Deploying a servlet in JIRA

We all know how useful a servlet is! JIRA provides an easy way to deploy a JAVA servlet with the help of the **Servlet plugin** module. In this recipe, we will see how to write a simple servlet and access it in JIRA.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it…

Following are the steps required to deploy a JAVA servlet in JIRA:

1. Include the servlet plugin module in the `atlassian-plugin.xml file`. The **Servlet plugin** module allows the following set of attributes:

   a. class: This is the servlet Java class and it must be a subclass of `javax.servlet.http.HttpServlet`. This attribute is mandatory.

   b. disabled: This indicates whether the plugin module should be disabled or enabled by default. By default, the module is enabled.

   c. i18n-name-key: This is the localization key for the human-readable name of the plugin module.

   d. key: This represents the unique key for the plugin module. This attribute is mandatory.

   e. name: This is the human-readable name of the servlet.

   f. system: This indicates whether this plugin module is a system plugin module or not. Only available for non-OSGi plugins.

   The following child elements are supported:

   a. description: The description of the plugin module.

   b. init-param: Initialization parameters for the servlet, specified using `param-name` and `param-value` sub-elements, just as in `web.xml`. This element and its child elements may be repeated.

c. resource: Resources for this plugin module. This element may be repeated.

d. url-pattern: The pattern of the URL to match. This element is mandatory and may be repeated.

In our example, let us use only the mandatory fields and some example `init-params`, as shown:

```
  <servlet name="Test Servlet" key="jtricksServlet"
class="com.jtricks.JTricksServlet">
      <description>Test Servlet</description>
      <url-pattern>/myWebsite</url-pattern>
      <init-param>
          <param-name>siteName</param-name>
          <param-value>Atlassian</param-value>
      </init-param>
      <init-param>
          <param-name>siteAddress</param-name>
          <param-value>http://www.atlassian.com/</param-value>
      </init-param>
  </servlet>
```

Here, `JTricksServlet` is the servlet class, whereas `/myWebsite` is the URL pattern. We are also passing a couple of **init-params**: `siteName` and `siteAddress`:

2. Create a servlet class. The class must extend `javax.servlet.http.HttpServlet`.

```
public class JTricksServlet extends HttpServlet {
    ...
}
```

3. Implement the necessary methods.

a. We can retrieve the **init-params** in the `init` method, as shown next:

```
@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    authenticationContext =
        ComponentAccessor.getJiraAuthenticationContext();
    siteName = config.getInitParameter("siteName");
    siteAddress = config.getInitParameter("siteAddress");
}
```

The `init` method is invoked every time the servlet is initialized, and this happens when the servlet is first accessed. The `init` method is also invoked when the servlet is first accessed after the plugin module is disabled and re-enabled.

As you can see, the **init-params** we defined in the servlet plugin module can be accessed here from `ServletConfig`. Here, we also initialize the `JiraAuthenticationContext` so that we can use it to retrieve the logged-in user details in the servlet. Similarly, we can initialize any JIRA components here.

b. Implement the `doGet()` and/or `doPost()` methods to implement what needs to be done. For the example, we will just use the **init-params** to create a simple HTML page and print a line to the console.

```
@Override
protected void doGet(HttpServletRequest req,
HttpServletResponse resp)
throws ServletException, IOException {
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    User user = authenticationContext.getLoggedInUser();
    out.println("Welcome " + (user != null ? user.getDisplayName()
    : "Anonymous"));
    out.println("<br>Invoking the servlet...");
    out.println("<br>My Website : <a href="" + siteAddress + "">"
    + siteName + "</a>");
    doSomething();
    out.println("<br>Done!");
}
private void doSomething() {
   System.out.println("Invoked servlet at " + (new Date()));
}
```

`authenticationContext` retrieves the current username, as mentioned earlier.

4. Package the plugin and deploy it.

How it works…

Once deployed, the servlet will be accessible at the URL: `http://yourserver/jira/plugins/servlet/${urlPattern}`. In our case, the URL is `http://yourserver/jira/plugins/servlet/myWebsite`.

When the servlet is accessed at `/plugins/servlet/myWebsite`, the output is as shown in the following screenshot:



Adding shared parameters to Servlet Context

In the previous recipe, we saw how to deploy a servlet and how to make use of the **init-params**. What if we have a set of servlets, servlet filters, or context listeners that makes use of the same parameters? Do we really need to initialize them in all the plugin modules?

In this recipe, we will see how we can use the Servlet Context Parameter plugin module to share parameters across servlets, filters, and listeners.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it…

All we need to do is to define the shared parameters to add a `servlet-context-param` module for each shared parameter in the `atlassian-plugin.xml file`.

For example, a parameter with the key `sharedText` can be defined as follows:

```
<servlet-context-param key="jtricksContext">
    <description>Shares this param!</description>
    <param-name>sharedText</param-name>
     <param-value>This is a shared Text</param-value>
</servlet-context-param>
```



Make sure the module has a unique key. Here, the parameter name is `sharedText` and it has a value of **ThisisasharedText**. Once the plugin is packaged and deployed, the parameter `sharedText` is available across servlets, filters, and listeners.

In a servlet, we can access the parameter in the `init` method as follows:

```
@Override
public void init(ServletConfig config) throws ServletException {
   super.init(config);
   String sharedText =
config.getServletContext().getInitParameter("sharedText");
}
```

How it works…

Once the shared text has been retrieved, we can use it anywhere, for example while constructing the HTML:

```
out.println("<br>Shared Text:"+sharedText);
```

The servlet will now print that as well, as shown in the following screenshot:



Writing a Servlet Context Listener

We have seen how to write servlets. How about writing a context listener for them? This will come in handy if you want to integrate with frameworks that use context listeners for initialization.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it…

Following are the steps required to write a simple context listener:

1. Include the `servlet-context-listener` module in the `atlassian-plugin.xml file`:

```
<servlet-context-listener name="Test Servlet Listener"
key="jtricksServletListener"
class="com.jtricks.JTricksServletListener">
    <description>Listener for Test Servlet</description>
</servlet-context-listener>
```

   Here, we have a unique module **key** and a **class** that is the servlet context listener's Java class.

2. Write the servlet context listener's class. The class must be `javax.servlet.ServletContextListener`:

```
public class JTricksServletListener implements
ServletContextListener{
    ...
}
```

3. Implement the context listener methods as appropriate. For example, we just print some statements to the console:

```
public void contextDestroyed(ServletContextEvent event) {
    System.out.println("Test Servlet Context is destroyed!");
}
public void contextInitialized(ServletContextEvent event) {
    System.out.println("Test Servlet Context is initialized!");
}
```

   The details of the context that is initialized or destroyed can be found from the `ServletContextEvent` object.

4. Package the plugin and deploy it.

How it works…

The `contextInitialized` method is not invoked at application startup. Instead, it is invoked the first time a servlet or filter in the plugin is accessed after each time it is enabled.

```
2016-05-27 13:50:02,548 http-nio-8080-exec-20 INFO jobinkk 830x530x1 m392ok 0:0:0:0:0:0:0:1 /rest/plugins/1.0/com.jt
ricks.jtricks-utilities-key [c.a.plugin.manager.DefaultPluginManager] Disabling com.jtricks.jtricks-utilities
Filter destroyed!
Stopping...
Test Servlet Context is destroyed!
2016-05-27 13:50:02,611 http-nio-8080-exec-20 DEBUG jobinkk 830x530x1 m392ok 0:0:0:0:0:0:0:1 /rest/plugins/1.0/com.j
tricks.jtricks-utilities-key [c.a.activeobjects.osgi.ActiveObjectsServiceFactory] onPluginDisabledEvent removing del
egate for [com.jtricks.jtricks-utilities]
Starting...
Running JTricksJob at Fri May 27 13:50:06 EDT 2016
Test Servlet Context is initialized!
Initiating the filter:JTricks Filter
Intercepted in filter, request by user:Jobin Kuruvilla from IP 0:0:0:0:0:0:0:1 at Fri May 27 13:50:14 EDT 2016. Acce
ssed URL:/plugins/servlet/myWebsite
Invoked servlet at Fri May 27 13:50:14 EDT 2016
```

Similarly, the `contextDestroyed` method is invoked every time the plugin module containing a servlet or filter is disabled.

Using filters to intercept queries in JIRA

Servlet filters provide a powerful mechanism to intercept queries and perform clever actions such as profiling, monitoring, content generation, and so on. They work exactly like any normal Java servlet filter and JIRA provides the **ServletFilterPluginModule** to add them using plugins. In this recipe, we will learn about how to use filters to intercept certain queries to JIRA and how we can utilize them!

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it…

As with other servlet plugin modules, a `servlet-filter` plugin module also has a unique **key** and a **class** associated with it. The **name** attribute holds the human-readable name of the filter and **weight** indicates the order in which the filter will be placed in the filter chain. The higher the **weight**, the lower the filter's position.

There is another important attribute **location** that denotes the position of the filter in the application's filter chain. There are four possible values for the location:

- **after-encoding**: Very top of the filter chain in the application, but after any filters that ensure the integrity of the request
- **before-login**: Before the filter that logs in the user
- **before-decoration**: Before the filter that performs the sitemesh decoration of the response
- **before-dispatch**: At the end of the filter chain, before any servlet or filter that handles the request by default

The **weight** attribute is used in conjunction with a location. If two filters have the same location, they are ordered based on the weight attribute.

init-param, as usual, takes the initialisation parameters for the filter.

url-pattern defines the pattern of the URL to match. This element can be repeated and the filter will be invoked for all the URLs matching any of the patterns specified. Unlike a servlet URL, the **url-pattern** here matches `${baseUrl}/${url-pattern}`. The pattern can use wild chars `*` or `?`, with the former matching zero or many characters, including directory slashes, and the latter matching zero or one character.

dispatcher is another element that determines when the filter is invoked. You can include multiple dispatcher elements with the values **REQUEST**, **INCLUDE**, **FORWARD**, or **ERROR**. If not present, the filter will be invoked in all cases.

Let us try to intercept the servlet we wrote earlier, which has the URL in the format `${baseUrl}/*/myWebsite`, and log it. The following is the step-by-step procedure required to write a filter and implement the given logic:

1. Add the **ServletFilter** plugin module to `atlassian-plugin.xml`:

```
<servlet-filter name="Browse Issue Filter"
key="jtricksServletFilter" class="com.jtricks.JTricksServletFilter"
location="before-dispatch" weight="200">
    <description>Filter for Browse Issue</description>
    <url-pattern>/*/myWebsite</url-pattern>
    <init-param>
        <param-name>filterName</param-name>
        <param-value>JTricks Filter</param-value>
    </init-param>
</servlet-filter>
```

Here, `JTricksServletFilter` is the filter class, and we have added the filter before dispatch. In our example, **url-pattern** is `/*/myWebsite`, which is the URL to access the servlet we created in the earlier recipe. We can use different URL patterns as required in our context.

2. Create the `Filter` class. The class should implement `javax.servlet.Filter`:

```
public class JTricksServletFilter implements Filter {
    ...
}
```

3. Implement the appropriate filter methods:

```
public void destroy() {
    System.out.println("Filter destroyed!");
}
public void doFilter(ServletRequest req, ServletResponse res,
FilterChain chain) throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    // Get the IP address of client machine.
    String ipAddress = request.getRemoteAddr();
// Log the user details, IP address , current timestamp and URL.
    System.out.println("Intercepted in filter, request by user:" +
    authenticationContext.getLoggedInUser().getDisplayName() +
    " from IP " + ipAddress + " at " + new Date().toString() +
    ". Accessed URL:"+request.getRequestURI());
    chain.doFilter(req, res);
}
public void init(FilterConfig config) throws ServletException {
    System.out.println("Initiating the filter:
    "+config.getInitParameter("filterName"));
    authenticationContext =
    ComponentAccessor.getJiraAuthenticationContext();
}
```

Here, the `init` method is invoked when the filter is initialized, that is, the first time it is accessed after the plugin is enabled. In this method, we can retrieve the **init-param** instances or parameters defined using the **Servlet Context Parameter** plugin module.

The `destroy` method is invoked whenever a filter is destroyed.

The `doFilter` is the method that is invoked every time the URL matches the **url-pattern**. Here, we are just printing the IP address and user details, requesting the servlet, and logging the time, but we can do many other things, such as logging, using the data for profiling or monitoring, and so on and so forth.

4. Package the plugin and deploy it.

How it works…

Whenever the URL in JIRA matches the **url-pattern**, the respective filter is invoked. This can be a huge help when you want to do specific things when a particular operation in JIRA is performed, or if you want to monitor who is doing what and when, or something else based on a specific URL.

```
2016-05-27 13:45:43,345 UpmAsynchronousTaskManager:thread-1 DEBUG jobinkk 817x135x3 wmud9l fe80:0:0:0:0:0:0:1%1 /res
t/plugins/1.0/installed-marketplace [c.a.activeobjects.osgi.ActiveObjectsServiceFactory] onPluginDisabledEvent remov
ing delegate for [com.jtricks.jtricks-utilities]
2016-05-27 13:45:43,351 UpmAsynchronousTaskManager:thread-1 INFO jobinkk 817x135x3 wmud9l fe80:0:0:0:0:0:0:1%1 /rest
/plugins/1.0/installed-marketplace [c.a.plugin.loaders.ScanningPluginLoader] Removed plugin 'com.jtricks.jtricks-uti
lities'
2016-05-27 13:45:43,387 UpmAsynchronousTaskManager:thread-1 INFO jobinkk 817x135x3 wmud9l fe80:0:0:0:0:0:0:1%1 /rest
/plugins/1.0/installed-marketplace [c.a.plugin.util.WaitUntil] Plugins that have yet to be enabled: (1): [com.jtrick
s.jtricks-utilities], 60 seconds remaining
Starting...
Running JTricksJob at Fri May 27 13:45:43 EDT 2016
Initiating the filter:JTricks Filter
Intercepted in filter, request by user:Jobin Kuruvilla from IP [redacted] at Fri May 27 13:45:55 EDT 2016. Acce
ssed URL:/plugins/servlet/myWebsite
Invoked servlet at Fri May 27 13:45:55 EDT 2016
Intercepted in filter, request by user:Jobin Kuruvilla from IP [redacted] at Fri May 27 13:46:21 EDT 2016. Acce
ssed URL:/plugins/servlet/myWebsite
Invoked servlet at Fri May 27 13:46:21 EDT 2016
```

With our code in the example, the details are printed, as shown in the previous screenshot, whenever the `/myWebsite` servlet is invoked.

Adding and importing components in JIRA

JIRA has a component system that has many **Service** classes and **Manager** classes that are registered in `PicoContainer` and available for use by core classes and plugins alike. At times, however, we will have to add custom components to that component system, which can then be used with other modules in the same plugin or shared by other plugins.

In this recipe, we will see how we can add a new component in JIRA and how we can consume that from within the same or a separate plugin.

Getting ready

Create a Skeleton plugin using Atlassian Plugin SDK.

Also, take a look at `https://bitbucket.org/atlassian/atlassian-spring-scanner` if you are using the **Atlassian Spring Scanner** to get an understanding of the various annotations used in this recipe.

How to do it…

First, let us see how we can define a component and use it in different modules within the same plugin. In our example, we will define a sample component and use the methods exposed by it in a simple webwork action. Following are the steps required:

1. Create an interface with the required method definitions. The component will expose these methods when used elsewhere:

```
package com.jtricks.jira.component;
public interface MyPrivateComponent {
    public void doSomething();
}
```

2. Create the implementation class and implement the methods:

```
public class MyPrivateComponentImp implements MyPrivateComponent {
    @Override public void doSomething() {
      System.out.println("Do something inside the private
      component!");
    }
}
```

   In the implementation class, we can inject the JIRA components, as usual, and use them for various things. Here, we have kept it simple!

3. Declare the class as a component. If you are using the **Atlassian Spring Scanner** in the plugin, you can do it by simply adding the `@Component` annotation, as shown here:

```
@Component
public class MyPrivateComponentImp implements MyPrivateComponent {
    @Override
    public void doSomething() {
      System.out.println("Do something inside the private
component!");
```

```
        }
    }
```

If you are not using the **Atlassian Spring Scanner**, you can do the same thing by adding the `component` module in the `atlassian-plugin.xml` file, as shown here:

```
<component key="myPrivateComponent" name="My Private Component"
class="com.jtricks.jira.component. MyPrivateComponentImp">
   <interface>
     com.jtricks.jira.component. MyPrivateComponent
   </interface>
</component>
```

Here, the `component` module has a unique **key** and a **class** attribute that points to the Implementation class. The element **interface** points to the component interface that we created in *step 1*.

Our **component** is now ready and available to use within other plugin modules. For example, we can use this component in a webwork action class, as shown here:

```
public class RedirectAction extends JiraWebActionSupport {
   private final MyPrivateComponent myPrivateComponent;
   public RedirectAction(MyPrivateComponent myPrivateComponent) {
       super();
       this.myPrivateComponent = myPrivateComponent;
   }
   @Override
   protected String doExecute() throws Exception {
       System.out.println("Action invoked.
       Doing something important before redirecting to Dashboard!");
       this.myPrivateComponent.doSomething();
       return getRedirect("/secure/Dashboard.jspa");
   }
}
```

Here, the component is injected in the constructor of the action class, as you would normally do for a standard JIRA component.

Exposing components to other plugins

When we create components as discussed earlier, they remain private and are available only within the plugin, even though we can expose these components to other plugins.

Depending on whether the plugin is using **Atlassian Spring Scanner** or not, the component is registered as public in different ways.

If **Atlassian Spring Scanner** is used, we can register the component as public using the `@ExportAsService` annotation, as shown here:

```
package com.jtricks.jira.component;
import org.springframework.stereotype.Component;
import
com.atlassian.plugin.spring.scanner.annotation.export.ExportAsService;
@ExportAsService
@Component
public class MyPublicComponentImp implements MyPublicComponent {
    @Override
    public void doSomething() {
        System.out.println("Do something inside the public component!");
    }
}
```

`ExportAsService` works in different ways:

- If you provide one or more types (typically interfaces) as the annotation value, your component is exported as those types
- If you provide no value for the annotation and your component implements any interface, it's exported as ALL of those interfaces
- If you provide no value for the annotation and your component implements no interface, it's exported as its own class

In addition to adding the annotation, we need to export the component interface using the `Export-Package` section in the `maven-jira-plugin` definition in the plugin `pom.xml`, as shown here:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.atlassian.maven.plugins</groupId>
      <artifactId>maven-jira-plugin</artifactId>
        ....
        <configuration>
        <productVersion>${jira.version}</productVersion>
        <instructions>
          <Atlassian-Plugin-Key>
            ${atlassian.plugin.key}
          </Atlassian-Plugin-Key>
          <!-- Add package to export here -->
          <Export-Package>
              com.jtricks.jira.component,
          </Export-Package>
          ....
```

```
            <Spring-Context>*</Spring-Context>
          </instructions>
        </configuration>
      </plugin>
      ....
    </plugins>
  </build>
```

If **Atlassian Spring Scanner** is not used, we can register the component as public using the component plugin module, by adding the `public` attribute as `true`, as shown here:

```
<component key="myPublicComponent" name="My Public Component"
class="com.jtricks.jira.component.MyPublicComponentImpl" public="true">
     <interface>com.jtricks.jira.component.MyPublicComponent</interface>
</component>
```

With that, the component is now ready and available to other plugins.

Importing public components

If we are using **Atlassian Spring Scanner** in the plugin, importing a component is done with the help of the `@ComponentImport` annotation. If the public component is in the same plugin, you do not need the annotation.

Also, don't forget to use the `@Inject` annotation if the component is not injected in the default constructor.

The following snippet shows examples of injecting a public component from the same plugin, without any annotations, and another from the standard JIRA framework, using the `@ComponentImport` annotation:

```
@Named
public class RedirectAction extends JiraWebActionSupport {
    private final JiraAuthenticationContext authenticationContext;
    private final MyPublicComponent myPublicComponent;
    @Inject
    public RedirectAction( @ComponentImport JiraAuthenticationContext
    authenticationContext, MyPublicComponent myPublicComponent) {
      super();
      this.authenticationContext = authenticationContext;
      this.myPublicComponent = myPublicComponent;
    }
    @Override
    protected String doExecute() throws Exception {
      System.out.println("Action invoked. Doing something important before
      redirecting to Dashboard!");
```

```
        this.myPublicComponent.doSomething();
        ApplicationUser loggedInUser =
        this.authenticationContext.getLoggedInUser();
        System.out.println("Current User:" + (loggedInUser ==
        null ? "Anonymous" :
        loggedInUser.getDisplayName()));
        return getRedirect("/secure/Dashboard.jspa");
    }
}
```

On the other hand, if we are not using **Atlassian Spring Scanner** in the plugin, we will have to first import the component using the `component-import` plugin module. The module is entered in `atlassian-plugin.xml`, as follows:

```
<component-import key="myPublicComponent">
    <interface>com.jtricks.jira.component.MyPublicComponent</interface>
</component-import>
```

Now the component is available as if it were created within the plugin itself. The webwork class will look exactly the same as it did while injecting a private component.

How it works…

When a component is installed, it generates the `atlassian-plugins-spring.xml` **Spring Framework** configuration file, transforming **Component** plugin modules into **Spring** bean definitions. The generated file is stored in a temporary plugin JAR file and installed into the framework. If the `public` attribute is set to `true`, the component will be turned into an **OSGi** service under the hood, using Spring dynamic modules to manage its lifecycle.

Component imports also generate the `atlassian-plugins-spring.xml` **Spring Framework** configuration file and transform the **Import Plugin Module** to **OSGi** service references using Spring dynamic modules. The imported component will have its bean name set to the component import key.

In both cases, it is possible to write our own Spring configuration file, stored under the folder `META-INF/spring` in the plugin JAR.

If you are using **Atlassian Spring Scanner**, it has two aspects. At the build time, the annotations in the code are read by the SDK and are written into some index files stored in the JAR's `META-INF` directory. At run time, these index files are scanned to create **Spring** components and **OSGi** services.

The use of those index files removes the need of scanning the plugin's bytecode at runtime to `transform` it into an OSGi bundle, and hence the plugin is faster to load. Such plugins are called "**transformerless**" plugins and are recommended over the traditional P2 plugins.



More details about **Atlassian Spring** Scanner can be found at `https://bitbucket.org/atlassian/atlassian-spring-scanner`. The **Component** Plugin module and **Component Import** plugin module can be found in the Atlassian documentation at `https://developer.atlassian.com/docs/getting-started/plugin-modules/component-plugin-module`and `https://developer.atlassian.com/docs/getting-started/plugin-modules/component-import-plugin-module` respectively.

Adding new module types to JIRA

So far, we have seen a lot of useful plugin module types in JIRA; custom field module types, webwork module types, servlet module types, and so on. But is it possible to add a custom module type in JIRA, one that can then be used to create different modules?

JIRA provides the **Module Type** plugin module, using which we can add new module types dynamically to the plugin framework. In this recipe, we will see how we can add such a new plugin module type and use it to create different modules of that type.

Getting ready

Create a Skeleton plugin using Atlassian Plugin SDK.

How to do it…

Let us consider the same example Atlassian has used in its online documentation, that is, to create a new dictionary plugin module that can then be used to feed a dictionary service used by other plugins or modules. In this recipe, we will be using the **Atlassian Sping Scanner,** and hence the code will be slightly different.

Following are the steps required to define a new plugin module type:

1. Create an interface that can be used in the `ModuleDescriptor` class. This interface will have all the methods needed for the new module. For example, in the dictionary, we need a method to retrieve the definition of a given text, and hence we can define the interface as follows:

```
public interface Dictionary {
    String getDefinition(String text);
}
```

   The new modules of this particular type will ultimately implement this interface.

2. Create the module descriptor class.

   a. The class must extend the `AbstractModuleDescriptor` class and should use the interface we created as the generic type:

```
public class DictionaryModuleDescriptor extends
AbstractModuleDescriptor<Dictionary> {
    ...
}
```

   b. Inject the `ModuleFactory` component in the constructor using the `@ComponentImport` annotation:

```
@Named public class DictionaryModuleDescriptor extends
AbstractModuleDescriptor<Dictionary> {
    @Inject
    public DictionaryModuleDescriptor(@ComponentImport
    ModuleFactory moduleFactory) {
        super(moduleFactory);
    }
}
```

   c. Implement the `getModule` method to create the module:

```
public Dictionary getModule() {
    return moduleFactory.createModule(moduleClassName, this);
}
```

   Here, we have used `ModuleFactory` to create a module of this type.

d. Define the attributes and elements that will be used in the new module type. For a dictionary, we need at least one attribute, that is, the **language**, to differentiate the various dictionary modules. Create a variable to retrieve the `language` attribute:

```
private String language;public String getLanguage() {
    return language;
}
```

e. Use the `init` method to retrieve the attribute from the module definition. Let us assume that the attribute name is `lang`. We will retrieve the attribute value and set it on the `language` variable defined in the previous step:

```
@Override
public void init(Plugin plugin, Element element) throws
PluginParseException {
    super.init(plugin, element);
    language = element.attributeValue("lang");
}
```

The entire descriptor class will look like the following.

```
package com.jtricks.jira.module;
import javax.inject.Inject;
import javax.inject.Named;
import org.dom4j.Element;
import com.atlassian.plugin.Plugin;
import com.atlassian.plugin.PluginParseException;
import com.atlassian.plugin.descriptors
.AbstractModuleDescriptor;
import com.atlassian.plugin.module.ModuleFactory;
import
com.atlassian.plugin.spring.scanner.annotation.imports.ComponentImport;
@Named
public class DictionaryModuleDescriptor extends
AbstractModuleDescriptor<Dictionary> {
    private String language;
    @Inject
    public DictionaryModuleDescriptor(@ComponentImport
    ModuleFactory moduleFactory) {
        super(moduleFactory);
    }
    @Override public void init(Plugin plugin,
    Element element) throws
    PluginParseException {
        super.init(plugin, element);
```

```
        language = element.attributeValue("lang");
    }
    @Override public Dictionary getModule() {
        return moduleFactory.createModule(moduleClassName, this);
    }
    public String getLanguage() {
        return language;
    }
}
```

2. Define a `ModuleType` factory class. This class will extend the `SingleModuleDescriptorFactory` class and will use the module descriptor defined in the earlier step as the generic type.We will also use the `@ModuleType` annotation in this class, as shown here:

```
package com.jtricks.jira.module;
import javax.inject.Inject;
import org.springframework.stereotype.Component;
import com.atlassian.plugin.hostcontainer.HostContainer;
import com.atlassian.plugin.osgi.external
.ListableModuleDescriptorFactory;
import com.atlassian.plugin.osgi.external
.SingleModuleDescriptorFactory;
import com.atlassian.plugin.spring.scanner
.annotation.export.ModuleType;
@ModuleType(ListableModuleDescriptorFactory.class)
@Component public class DictionaryModuleTypeFactory extends
SingleModuleDescriptorFactory<DictionaryModuleDescriptor> {
    @Inject public DictionaryModuleTypeFactory(HostContainer
    hostContainer) {
        super(hostContainer, "dictionary",
        DictionaryModuleDescriptor.class);
    }
}
```

   As you can see, the factory class also injects `HostContainer` in the constructor and uses it to invoke the super class constructor.

   In addition to the `HostContainer`, it also passes a **key**, `dictionary` in our example, and the **module descriptor** class we created in step 2.

   The **key** will be used for creating module definitions using this new module type. We will see that in the next section of this recipe.

3. Add the `com.atlassian.plugin.osgi.bridge.external` package to the `<Import-Package>` section in the `maven-jira-plugin` definition in the plugin `pom.xml`, as shown here:

```
<build>
    <plugins>
        <plugin>
            <groupId>com.atlassian.maven.plugins</groupId>
            <artifactId>maven-jira-plugin</artifactId>
             ....
            <configuration>
               <productVersion>${jira.version}</productVersion>
               <instructions>
                 <Atlassian-Plugin-Key>
                   ${atlassian.plugin.key}
                 </Atlassian-Plugin-Key>
                 ....
                 <!-- Add package import here -->
                 <Import-Package>
                   org.springframework.osgi.*;
                   resolution:="optional",
                    org.eclipse.gemini.blueprint.*;
                   resolution:="optional",
                   com.atlassian.plugin.osgi.bridge.external, *
                 </Import-Package>
                   ....
                 <Spring-Context>*</Spring-Context>
               </instructions>
            </configuration>
        </plugin>
        ....
    </plugins>
</build>
```

With that, the new plugin module is now ready. We can now write new modules of the type `dictionary`.

Creating modules using the new module type

The new module types will be as simple as the following:

```
<dictionary key="myUSEnglishDictionary" lang="us-english"
class="com.jtricks.dictionary.USDictionary" />
<dictionary key="myUKEnglishDictionary" lang="uk-english"
class="com.jtricks.dictionary.UKDictionary" />
```

The root element is the same as the module type's **key** used in the `DictionaryModuleTypeFactory` class, `dictionary` in this case. Each has its own unique **key** and has the **lang** attribute we defined earlier. Each has a **class** that will implement the `Dictionary` interface appropriately. For example:

```
public class USDictionary implements Dictionary {
    public String getDefinition(String text) {
        if (text.equals("JIRA")){
            return "JIRA in San Fransisco!";
        } else {
            return "What are you asking?
            We in the US don't know anything other than JIRA!!";
        }
    }
}
```

and

```
public class UKDictionary implements Dictionary {
    public String getDefinition(String text) {
        if (text.equals("JIRA")){ return "JIRA in London!";
        } else {
                return "What are you asking?
                We in the UK don't know anything other than JIRA!!";
        }
    }
}
```

Using the new modules

Once the new modules are defined, `myUSEnglishDictionary` and `myUKEnglishDictionary` in our example, we can use them in other plugin modules. For example, if we want to use them in a servlet module to find the definition of JIRA in both dictionaries, this can be done using the following steps:

1. Get all the enabled modules that use the `dictionary` module descriptor:

```
List<DictionaryModuleDescriptor> dictionaryModuleDescriptors =
pluginAccessor.getEnabledModuleDescriptorsByClass
(DictionaryModuleDescriptor.class);
```

Here, `pluginAccessor` can be retrieved as follows:

```
PluginAccessor pluginAccessor =
ComponentAccessor.getPluginAccessor();
```

It can also be used to retrieve all the enabled modules that use the given module descriptor class as shown in the code.

2. For each `DictionaryModuleDescriptor`, the `getLanguage()` method will retrieve the value of the `lang` attribute and `getModule()` will retrieve the respective `Dictionary` implementation class.

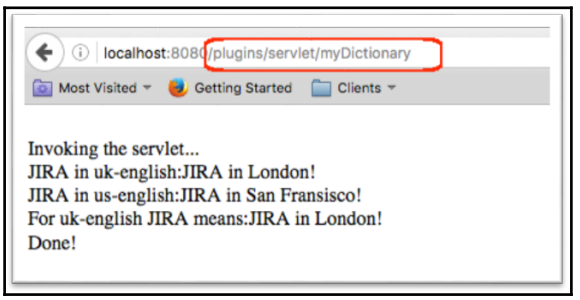
For example, the JIRA definition for `uk-english` can be retrieved as follows:

```
    private String getJIRADescription(String key) {
      // To get all the enabled modules of this module descriptor
      List<DictionaryModuleDescriptor> dictionaryModuleDescriptors =
      pluginAccessor.getEnabledModuleDescriptorsByClass
      (DictionaryModuleDescriptor.class);
      for (DictionaryModuleDescriptor dictionaryModuleDescriptor :
      dictionaryModuleDescriptors){
        if (dictionaryModuleDescriptor.getLanguage().equals(key))
        {
          return
dictionaryModuleDescriptor.getModule().getDefinition("JIRA");
        }
      }
      return "Not Found";
    }
```

Here, the **key** that is passed will be `uk-english`.

How it works…

If we use a servlet to display all the definitions of the word JIRA in all the dictionaries deployed, **US** and **UK** in our case, the result will be as follows:



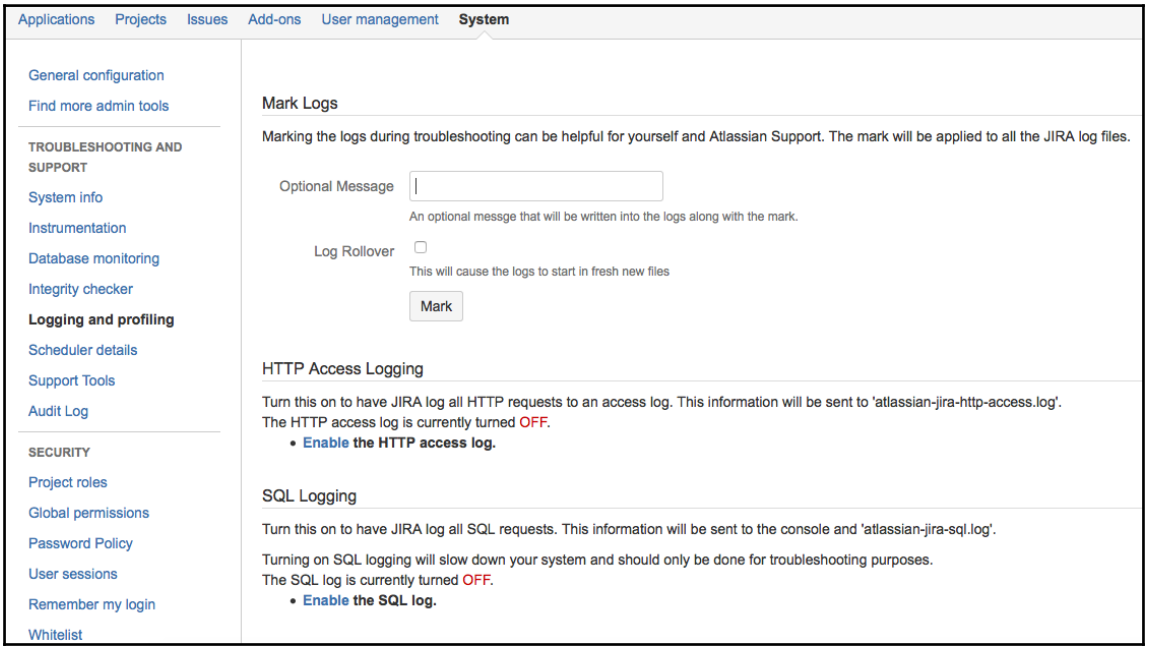
Enabling access logs in JIRA

Access logs are a good way of finding out who is doing what in your JIRA instance. In this recipe, we will see how we can turn on access logging in JIRA.

How to do it…

The list of users who are currently accessing JIRA can be found under the **Administration** | **System** | **Security** | **User sessions** menu. But if you need more detailed information about who is doing what, **access logging** is the way to go.

Enabling access logs can be done via the administration screen by going to **Administration** | **System** | **Troubleshooting and Support** | **Logging & Profiling,** as shown in the following screenshot:



We can turn on **HTTP** access logs using the **Enable** option, as shown in the above screenshot. There is an additional option to mark these access logs with an optional message and to send those logs to a new file. This is very useful for detecting logs when we are troubleshooting an issue.

These logs are disabled by default and, if enabled via the UI, they will be disabled again on the next restart.

In order to enable them permanently, we can switch them ON in the `log4j.properties` file, residing under the `WEB-INF/classes` folder under the section **Accesslogs,** as shown next:

```
################################################## # Access logs
##################################################

log4j.logger.com.atlassian.jira.web.filters.accesslog.AccessLogFilter = OFF, httpaccesslog
log4j.additivity.com.atlassian.jira.web.filters.accesslog.AccessLogFilter = false

log4j.logger.com.atlassian.jira.web.filters.accesslog.AccessLogFilterIncludeImages = OFF,
httpaccesslog
log4j.additivity.com.atlassian.jira.web.filters.accesslog.AccessLogFilterIncludeImages = false

log4j.logger.com.atlassian.jira.web.filters.accesslog.AccessLogFilterDump = OFF,
httpdumplog log4j.additivity.com.atlassian.jira.web.filters.accesslog.AccessLogFilterDump
= false
```

How it works…

Once turned ON, the **HTTP** access logs will be written to `atlassian-jira-http-access.log` and the **HTTP** dump logs to the `atlassian-jira-http-dump.log` file, with both files residing under the `JIRA_HOME/log` folder.

You can find detailed information in the access logs, similar to the following:

```
0:0:0:0:0:0:0:1 o1385x165x1 jobinkk [28/May/2016:23:05:55 -0400] "GET
http://localhost:8080/rest/plugins/1.0/notifications/jobinkk HTTP/1.1" 200
6950 0.0070 "http://localhost:8080/secure/admin/ViewLogging.jspa"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:46.0) Gecko/20100101
Firefox/46.0" "1gb8r0v" 0:0:0:0:0:0:0:1 i1385x166x1 jobinkk
[28/May/2016:23:05:56 -0400] "POST
http://localhost:8080/rest/analytics/1.0/publish/bulk HTTP/1.1" - - -
"http://localhost:8080/secure/admin/ViewLogging.jspa" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.11; rv:46.0) Gecko/20100101 Firefox/46.0"
"1gb8r0v" 0:0:0:0:0:0:0:1 o1385x166x1 jobinkk [28/May/2016:23:05:56 -0400]
"POST http://localhost:8080/rest/analytics/1.0/publish/bulk HTTP/1.1" 200 0
0.0030 "http://localhost:8080/secure/admin/ViewLogging.jspa" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.11; rv:46.0) Gecko/20100101 Firefox/46.0"
"1gb8r0v" 0:0:0:0:0:0:0:1 i1385x167x1 jobinkk [28/May/2016:23:05:57 -0400]
"GET
```

```
http://localhost:8080/secure/admin/ViewLogging!disableHttpAccessLog.jspa
HTTP/1.1" - - - "http://localhost:8080/secure/admin/ViewLogging.jspa"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:46.0) Gecko/20100101
Firefox/46.0" "1gb8r0v"
```

It is also possible to change the individual log file's name or path in `log4j.properties` by modifying the appropriate properties. For example, the **HTTP** access log file can be written to `/var/log/http-access.log` by modifying the `log4j.appender.httpaccesslog.File` property as follows:

```
log4j.appender.httpaccesslog.File=/var/log/http-access.log
```

Enabling SQL logging in JIRA

Similar to access logs, another useful type of logging, especially when debugging an issue, is SQL logging. In this recipe, we will see how to turn on SQL logging.

How to do it…

Similar to access logs, SQL logging is also turned ON from the user interface at **Administration** | **System** | **Troubleshooting and Support** | **Logging & Profiling**.

Just as with access logs, the changes are temporary and will revert back in the next restart. Permanent changes can be made in the `WEB-INF/classes/log4j.properties` file, although this is not recommended because SQL access logs can take up a large amount of space and may adversely affect performance.

In such an event, the logging entry to be modified is as follows:

```
################################################### # SQL logs
################################################### # # Beware of turning
this log level on. At INFO level it will log every SQL statement # and at
DEBUG level it will also log the calling stack trace. Turning this on will
DEGRADE your # JIRA database throughput. #
log4j.logger.com.atlassian.jira.ofbiz.LoggingSQLInterceptor = OFF, sqllog
log4j.additivity.com.atlassian.jira.ofbiz.LoggingSQLInterceptor = false
log4j.logger.com.atlassian.jira.security.xsrf.XsrfVulnerabilityDetectionSQL
Interceptor = OFF, xsrflog
log4j.additivity.com.atlassian.jira.security.xsrf.XsrfVulnerabilityDetectio
nSQLInterceptor = false
```

The latter logs the SQL queries executed for **Xsrf** vulnerability detection.

How it works…

Once turned **ON**, the SQL logs will be written to the `atlassian-jira-sql.log` file under the `JIRA_HOME/log` folder.

You can find details about the numerous SQLs executed as follows:

```
2016-05-28 23:25:53,320 http-nio-8080-exec-2 jobinkk 1405x200x1 m5q59x
/secure/admin/ViewLogging.jspa 0ms "SELECT ID, filename, contenttype,
avatartype, owner, systemavatar FROM PUBLIC.avatar WHERE filename='Avatar-
default.svg' AND avatartype='user' AND systemavatar='1'" 2016-05-28
23:25:53,320 http-nio-8080-exec-2 jobinkk 1405x200x1 m5q59x
/secure/admin/ViewLogging.jspa 0ms Connection returned. borrowed : 0
```

As in the case of access logs, the SQL log file path can be changed by modifying the `log4j.appender.sqllog.File` property as follows:

```
log4j.appender.sqllog.File=/var/log/sql.log
```

Internationalization in webwork plugins

We saw in earlier chapters how to write webwork plugins to create new or extended JIRA actions. In this recipe, we will see how we can personalize the messages in these plugins using **internationalization** and **localization**.

As Wikipedia puts it:

> *"Internationalization and localization are means of adapting computer software to different languages, regional differences and technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text."*

The terms internationalization and localization are abbreviated to **i18n,** where **18** stands for the number of letters between the first **i** and the last **n** in internationalization!

How to do it…

Internationalization in a webwork plugin is achieved with the help of a resource bundle defined in the `atlassian-plugin.xml file`. Following are the steps required to enable it in a JIRA webwork plugin:

1. Create a `resource` definition in the `atlassian-plugin.xml` file.

```
<resource type="i18n" name="i18n" location="jtricks-utilities"/>
```

   Here, the type of the resource is `i18n`, and we have defined the location as `jtricks-utilities`. The `location` is the location of the property file, ending with the file name, and this will be used for adding the key/value pairs required for the translation. This location is the relative path to the `src/main/resources` folder, and we should specify the full path in this attribute if the property file is added in a different folder.

2. Add the key/value pair of properties that needs to be used in action as follows:

```
welcome.demo.heading=English
welcome.demo.message=Welcome to the demo
```

   Here, `welcome.demo.heading` is the **key** that will be used and will be the same across all language property files. The value here, "English", will be used for the default locale but will have equivalent translations in the other language property files.

3. Create property files in the same folder for other required languages in the following format: `${fileName}_${languageCode}_${countryCode}.properties`. For example, if we need to personalize the above action for **French** users, the following will be the property filename:

```
jtricks-utilities _fr_FR.properties
```

4. Add the property `welcome.demo.heading` in each of the property files with the appropriate translation as the values. For example, a property with the value "`English`" in the default property file will have the value "`Français`" in French!

   In our example, following is how the `jtricks-utilities _fr_FR.properties` will look.

```
welcome.demo.heading=Français welcome.demo.message=Bienvenue sur la
demo
```

   As you can see, the key is the same but the values are the respective translations.

With that, the plugin is ready for translation and we can use the i18n keys in the action class and velocity templates.

In velocity templates, using 18n is really easy. All you have to do is use the `i18n` variable, which is already available in the velocity context. This variable is of type `com.atlassian.jira.util.I18nHelper` class and exposes the `getText` method, which can be used as follows:

```
$i18n.getText("welcome.demo.heading")
```

In the action class, we can retrieve the same `I18nHelper` class using the `getI18nHelper()` method, but let us also see how it can be injected in the constructor, just in case you want to do the same in other plugin modules.

```
public class TranslateAction extends JiraWebActionSupport {
     private final I18nHelper i18nHelper;
     public TranslateAction(I18nHelper i18nHelper) {
         super();
         this.i18nHelper = i18nHelper;
     }
 }
```

Once the helper class is initialized, it can be used anywhere in the action class. For example, the translated value can be printed as follows:

```
System.out.println("Translated text:" +
this.i18nHelper.getText("welcome.demo.message"));
```

You can also retrieve the `I18nHelper` for a specific locale using the `I18nHelper.BeanFactory` class.

```
I18nHelper frenchI18nHelper =i18nBeanFactory.getInstance(Locale.FRANCE);
```

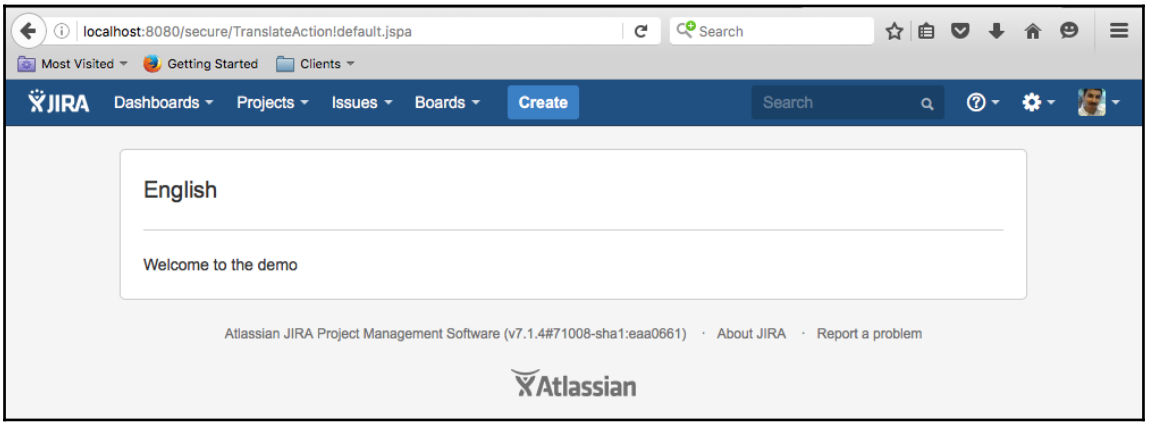
How it works…

Let us say we have used the above `i18n` resources in our webwork action class. A sample velocity template used in the `input` **view** will look like the following.

```
<html>
   <head> <meta name="decorator" content="atl.general"/>
     <title>$i18n.getText("welcome.demo.heading")</title>
   </head>
   <body class="aui-page-focused aui-page-size-large">
     <div id="page">
       <section id="content" role="main">
         <div class="aui-page-panel">
           <div class="aui-page-panel-inner">
             <section class="aui-page-panel-content">
```

```
                <h2>$i18n.getText("welcome.demo.heading")</h2>
                <p>$i18n.getText("welcome.demo.message")</p>
              </section>
            </div>
          </div>
        </section>
      </div>
    </body>
  </html>
```

When the action is invoked, a user with the **English** locale will see the following:



The same page is shown to a user with a **French** locale, as shown below.



Sharing common libraries across v2 plugins

We have already explored creating both v1 and v2 plugins throughout this book. One major difference between v1 and v2 plugins is that v1 plugins have access to all the libraries and classes available in the application class path, whereas v2 plugins can't access them.

For example, v1 plugins can access some common utility classes by dropping the JAR file with those classes in the `WEB-INF/lib` file or adding those classes under `WEB-INF/classes`. However, that won't work with v2 plugins as they need the JAR files embedded with them under `META-INF/lib` or the classes embedded in them. How will we handle this scenario when there is a utility class that we need to share across a few v2 plugins? Should we embed the class in all the plugins? The answer is no, and in this recipe, we will see how we can share those utility classes across v2 plugins by creating an **OSGi** bundle.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK.

How to do it…

Let us assume we have a **Number utility** class that does summation and multiplication of integer numbers. What should we do if we want to make this class available in all the v2 plugins? The following are the steps required:

1. Create the Utility class under the correct package:

```
package com.jtricks.utilities;
public class NumberUtility {
    public static int add(int x, int y) {
      return x + y;
    }
}
```

2. Export the packages that need to be shared so that they are visible to other v2 plugins. This step is very important.
   This is done in the `instructions` section in the `maven-jira-plugin` definition in the plugin `pom.xml`. This element allows various bundle instructions and the following are the two elements we need.

a. Export-Package: To export selected packages from the plugin to be shared across other plugins.

b. Import-Package: To import only selected packages into a plugin. By default, it imports all the exported packages from other plugins.

In our example, the package is `com.jtricks.utilities`, and we can export it using the `Export-Package` section, as shown here:

```
<build>
 <plugins>
   <plugin>
     <groupId>com.atlassian.maven.plugins</groupId>
        <artifactId>maven-jira-plugin</artifactId>
         ....
          <configuration>
          <productVersion>${jira.version}</productVersion>
          <instructions>
            <Atlassian-Plugin-Key>
              ${atlassian.plugin.key}
            </Atlassian-Plugin-Key>
            <!-- Add package to export here -->
            <Export-Package>
              com.jtricks.utilities,
            </Export-Package>
              ...
            <Spring-Context>*</Spring-Context>
          </instructions>
        </configuration>
      </plugin>
      ....
    </plugins>
  </build>
```

3. Package the plugin and deploy it as a v2 plugin.

With that, the plugin is set up to expose our utility class to other v2 plugins.

It is also possible to export only selected versions and not export certain packages. More details on the usage of **bundle instructions** can be found at `http://www.aqute.biz/Bnd/Bnd`.

When developing other plugins, we need to add the above plugin as a dependency in the `pom.xml` with **scope** as `provided`.

```
<dependency>
   <groupId>com.jtricks</groupId>
```

```
      <artifactId>number-utility</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
   </dependency>
```

Optionally, we can use the `Import-Package` element in the **bundle instructions** to import the package we exported earlier. By default, it will be imported anyway, and hence this step can be omitted. However, it can be useful for when you want to import only selected packages or make the import mandatory, and so on. Again, the details can be found at `http://www.aqute.biz/Bnd/Bnd`.

Now, the method `add` can be invoked as if the class is within the same plugin. For example, a webwork action class can use the method as follows:

```
package com.jtricks.jira.webwork;
 import com.atlassian.jira.web.action.JiraWebActionSupport;
 import com.jtricks.utilities.NumberUtility;
 public class MathAction extends JiraWebActionSupport {
     private int sum;
     @Override
     public String doDefault() throws Exception {
         sum = NumberUtility.add(2, 3);
         return super.doDefault();
     }
     public int getSum() {
         return sum;
     }
 }
```

Operations using direct HTML links

How about a little tip on how we can do powerful operations in JIRA with a simple click on a link, from your e-mail, from a web form, or from within JIRA itself?

Almost all actions can be encoded into a single URL, provided we have the right parameters to invoke those actions. Make no mistake; it has its own disadvantages because, in some cases, this will override all the pre-processing, validations, and so forth.

The URL that performs the action is constructed in the following manner:

${baseUrl}/secure/${action}?${arguments}

Where `baseUrl` is the JIRA base URL, `action` is the webwork action to be executed, and `arguments` are the URL-encoded arguments needed for the action. The arguments are constructed as key value pairs separated by `&`. Each key value pair will be of the form `key=value` and must comply with HTML link syntax; that is, all characters must be escaped. Let us explore this in detail.

How to do it…

Let us consider a simple example to start with: creating issues. Creating an issue has four stages.

1. Going to the initial create screen.
2. Selecting the **project** and **issuetype** and clicking on **Next.**
3. Entering all the details on the issue.
4. Clicking on **Submit** once the relevant details have been added.

We can execute each of these in a single step provided we know the details in advance. For this example, let us take http://localhost:8080/ as the base URL for the JIRA instance.

1. Going to the initial create issue screen can be done via the URL:`http://localhost:8080/secure/CreateIssue!default.jspa`



Note that the recent **project** and **issuetype** are pre-selected when you access this link because that is the default JIRA behavior.

2. But what if we want to pre-select some other **project**? All we need to do is add the parameters **pid** in the URL, as follows:`http://localhost:8080/secure/CreateIssue!default.jspa?pid=10100`



If we need to go to the second step directly by selecting **project** and **issuetype**, just add the **issuetype** parameter into the URL separated by `&`.
`http://localhost:8080/secure/CreateIssue.jspa?pid=10100&issuetype=10200`

3. If we need to pre-populate **create issue dialogue** in one click, enter all the details in the URL as shown, with the action name as `CreateIssueDetails!init.jspa`.

```
http://localhost:8080/secure/CreateIssueDetails!init.jspa?pid=10100&issuetype=10200&priority=1&summary=Emergency+Bug&reporter=jobinkk
```



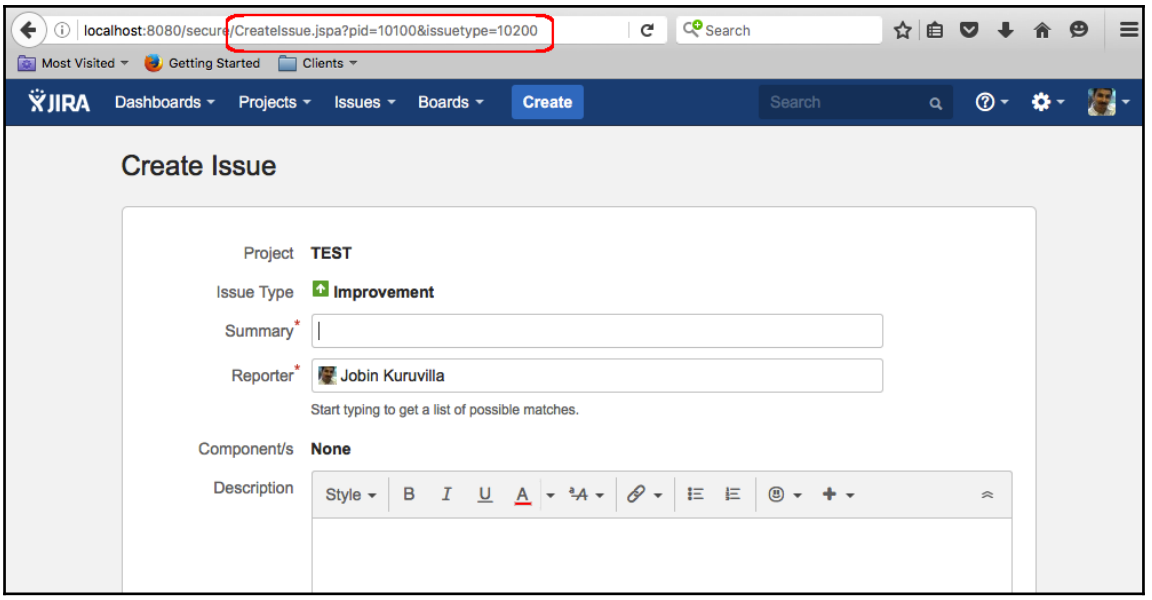
Note that all mandatory fields should be populated to avoid validation errors. The above example also shows how the URL is encoded to comply with HTML syntax by replacing the space in the summary with a +; **Emergency Bug** is written as **Emergency+Bug,** which can also be written as **Emergency%20Bug**.

4. If we want to create the issue in one click with the details above, use the `CreateIssueDetails` action instead of the `CreateIssueDetails!init` action.
   `http://localhost:8080/secure/CreateIssueDetails.jspa?pid=10100&issuetype=10200&priority=1&summary=Emergency+Bug&reporter=jobinkk`
   In this case, though, you might come across the XSRF Security Token Missing error due to **Form Token Handling** in JIRA. You can read more about that at `https://developer.atlassian.com/jiradev/jira-platform/jira-architecture/authentication/form-token-handling`. In that case, pass a valid **XSRFtoken** along with the other parameters.

Hopefully, that gives you an idea about how various operations can be executed via direct links. Make sure the user is logged in or **Anonymous** issue creation is turned on when the above links are clicked.

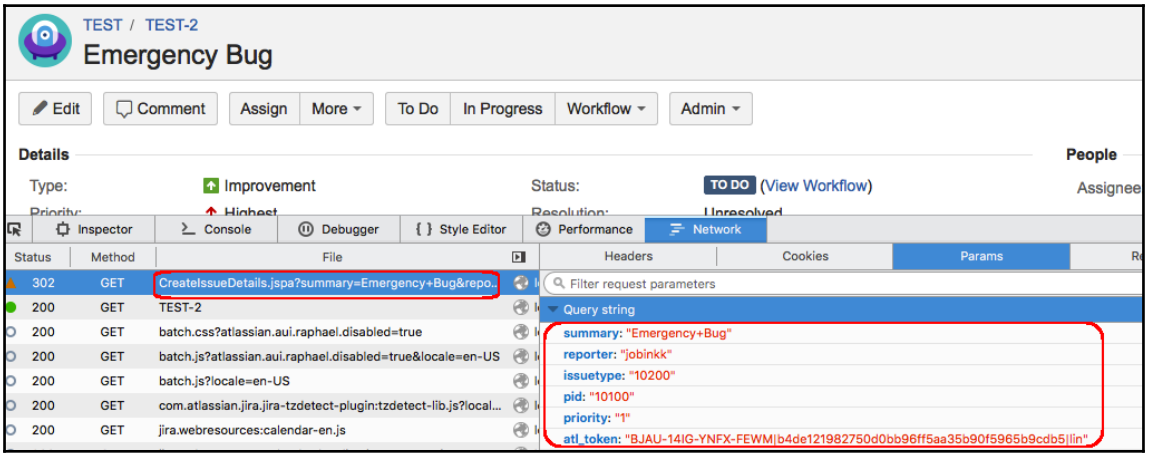
How do we find out what **action** class is involved or what **parameters** are to be passed?

You can find out easily from the browser URL if the request uses the **GET** method. Create an issue with **project** and **issuetype** selected (see case 2 earlier), as shown in the following example:



When the request is **POST,** as in *case 4*, we can find out the **action** name from the URL. However, the **parameters** are worked out from what is posted when the action is executed. There are multiple ways to do this, and an easy way out of it will be to use the browser capabilities.

For example, using the **Developer Tools** in **Mozilla Firefox** will get the parameters posted when an action is executed, as shown below:



Here, we can see the parameters **pid**, **issuetype**, **priority**, **summary,** and **reporter** being submitted in the **Params** section. It also includes the *XSRF token*, mentioned earlier, for which the param name is **atl_token**. We can see the action name in the **URL**. Once you get the action name and the list of parameters, you can use them in the URL with the appropriate values separated by `&`, as we saw in *case 4*.

This technique opens up a lot of possibilities. For example, we can easily automate the submission of URLs we have constructed using command-line tools such as **wget** or **curl**. Read more about these at `https://confluence.atlassian.com/jirakb/creating-issues-via-direct-html-links-159474.html`.

Implementing Marketplace licensing in plugins

Now that we have seen a lot of recipes on customizing JIRA using various plugin modules, it is time to commercialize them. It makes sense to wind up this book by giving you a few tips on how to add **licensing** in a JIRA plugin, right?

By now, you should be familiar with UPM and its ability to install/un-install and enable/disable plugins in a JIRA instance. When you purchase commercial plugins from the Marketplace, you can also manage licenses for these plugins using UPM. More about this at `https://confluence.atlassian.com/display/UPM/Managing+purchased+add-ons`.

In this recipe, we will see how to add licensing support in our custom plugin. Once this is done, users will be able to make use of our plugin functionality, but only after installing a valid license.

Implementing Marketplace licensing in the plugin is only one step in commercializing a plugin. More details about the whole process can be found at `https://developer.atlassian.com/market/developing-for-the-marketplace`, but we will concentrate on the development side in this recipe.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK.

How to do it…

Following are the steps required to add licensing support in a plugin.

1. Add the dependencies required for Marketplace licensing in the `pom.xml file`.

```
<!-- Licensing API dependencies -->
  <dependency>
    <groupId>com.atlassian.upm</groupId>
    <artifactId>licensing-api</artifactId>
    <version>2.18.4</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.atlassian.upm</groupId>
    <artifactId>upm-api</artifactId>
    <version>2.18.4</version>
    <scope>provided</scope>
  </dependency>
```

   At the time of writing this recipe, 2.18.4 is the latest version of these dependencies. You should modify the version attribute to include the latest version available at the time of writing your plugin.



   Make sure you rebuild your project after adding these dependencies.

2. Add the `atlassian-licensing-enabled` parameter in the `plugin-info` section in the `atlassian-plugin.xml` file.

```
<plugin-info>
  <description>${project.description}</description>
    ....
  <param name="atlassian-licensing-enabled">true</param>
</plugin-info>
```



   UPM uses the `atlassian-licensing-enabled` parameter to figure out whether a plugin's Marketplace license enabled or not.

3. Modify the plugin code to check for a valid license. Depending on the plugin logic, you might have to do it in multiple places, and it might make sense to write a utility class or manager class to check for a valid license.
   In our example, we will show you how to check for a valid license in a simple webwork action.

   a. Get the `PluginLicenseManager` instance. You can inject this in the action constructor, as usually done for other JIRA components.

```
@Named
public class LicensingDemoAction extends
JiraWebActionSupport {
   private final PluginLicenseManager pluginLicenseManager;
   @Inject
   public LicensingDemoAction(@ComponentImport
   PluginLicenseManager pluginLicenseManager) {
     this.pluginLicenseManager = pluginLicenseManager;
   }
 }
```

   b. Check if a license (valid or invalid) is installed in the current JIRA instance using the `isDefined` method on the license returned by `PluginLicenseManager`.

```
// Check if there is a license installed if
   (pluginLicenseManager.getLicense().isDefined()) {
     ...
    }
```

   If `isDefined` returns `false`, there is no license installed.

   c. If there is a license installed, that is, the value true is returned by `isDefined`, we can get the license as follows:

```
PluginLicense pluginLicense =
this.pluginLicenseManager.getLicense().get();
```

   d. Check if the license has an error. This can be done by invoking the `isDefined` method on the `LicenseError` returned by `PluginLicense`.

```
if (pluginLicense.getError().isDefined()) { .... }
```

   If `isDefined` returns `false`, there is no error and the license is a valid one. We can proceed with the plugin functionality.

e. If there is an error, we can retrieve the error, as follows:

```
            String licenseErrorName =
pluginLicense.getError().get().name();
```



You might want to show this error to the user, prompting them to install a valid license.

The following code shows how the webwork action performs these checks in our example:

```
@Named
public class LicensingDemoAction extends JiraWebActionSupport {
  private final PluginLicenseManager pluginLicenseManager;
  @Inject
  public LicensingDemoAction(@ComponentImport
  PluginLicenseManager pluginLicenseManager) {
    this.pluginLicenseManager = pluginLicenseManager;
  }
  @Override
  public String doDefault() throws Exception {
    try {
      // Check if there is a license installed if
      (pluginLicenseManager.getLicense().isDefined()) {
        PluginLicense pluginLicense =
        this.pluginLicenseManager.getLicense().get();
        // Check if the installed license has an error if
        (pluginLicense.getError().isDefined()) {
          // An invalid license installed.
         //(e.g. expired or user count mismatch)
         licensed = false; licenseError =
         pluginLicense.getError().get().name();
        } else {
        // A valid license installed. licensed = true;
        }
      } else {
          // No license installed
          licensed = false;
          licenseError = "No license installed";
      }
    } catch (Exception e) {
      // Any unexpected license related error licensed = false;
      licenseError = "Error retrieving license:" + e.getMessage();
      e.printStackTrace(); } return super.doDefault();
    }
    public boolean isLicensed() {
```

```
    return licensed;
  }
  public String getLicenseError() {
    return licenseError;
  }
}
```

As you can see, we are setting a **Boolean** variable licensed to `true` if there is a valid license. If not, we are setting an appropriate error message in a **String** variable named `licenseError`.

For the sake of this example, we can send an appropriate message to the user in the view template, as shown below.

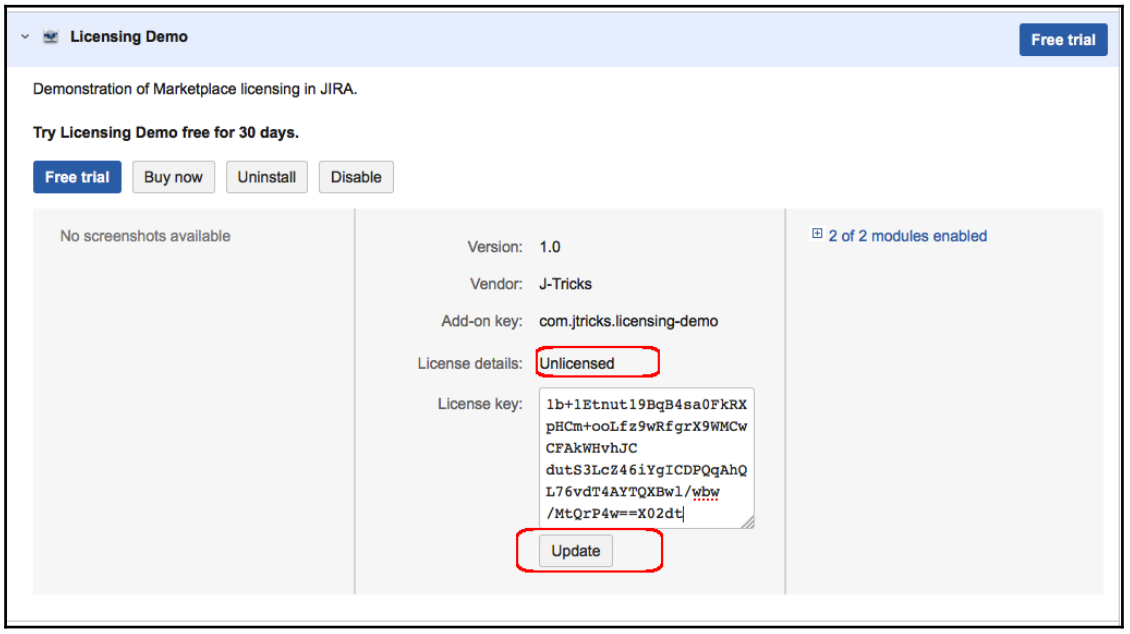
```
<html>
  <head>
    <meta name="decorator" content="atl.general"/>
    <title>Sharing code</title>
  </head>
  <body class="aui-page-focused aui-page-size-large">
    <div id="page">
      <section id="content" role="main">
        <div class="aui-page-panel">
          <div class="aui-page-panel-inner">
            <section class="aui-page-panel-content">
              #if ($licensed) <div class="aui-message
              aui-message-success">
              <p class="title">
                <strong>Valid License</strong>
              </p>
              <p>The plugin license is valid.</p>
            </div>
            #else
            <div class="aui-message aui-message-error">
              <p class="title">
                 <strong>Invalid License</strong>
              </p>
              <p>$!licenseError</p>
            </div>
             #end
           </section>
         </div>
       </div>
     </section>
    </div>
  </body>
</html>
```

Here, we are using the two variables defined in the action class to send a message back to the user.

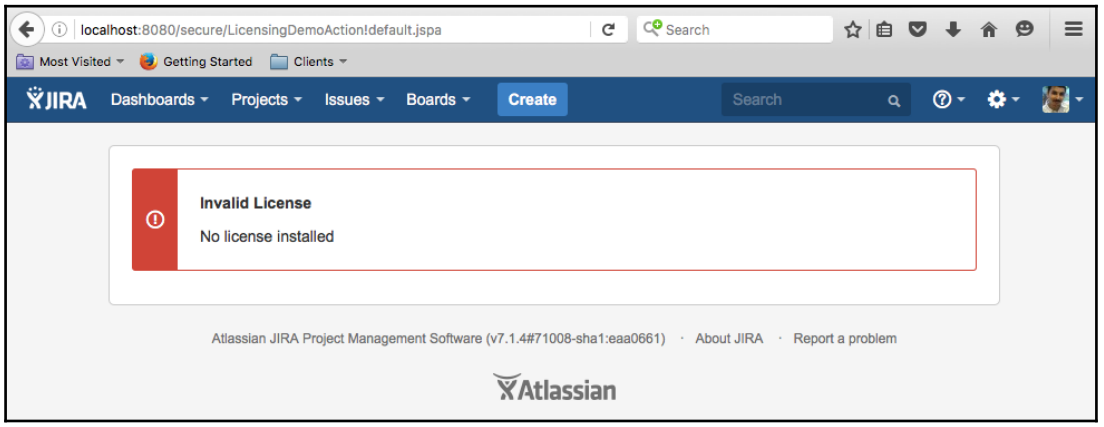
With that, our simple webwork plugin is ready with Marketplace licensing and the relevant license checks.

How it works…

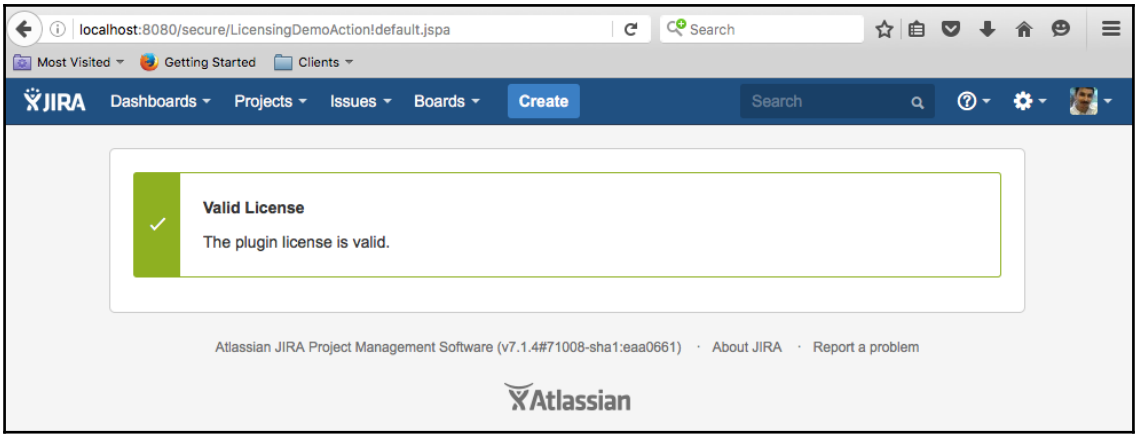
Once the plugin is installed, UPM will look for a valid license, and you can enter it under the plugin, as shown below.



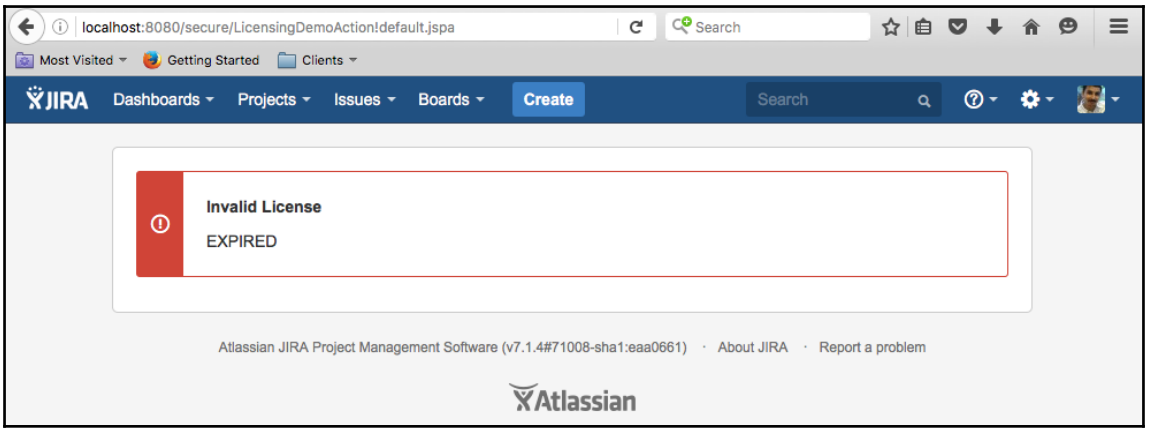
If you try to access the webwork action without entering a license, you will see the following message.



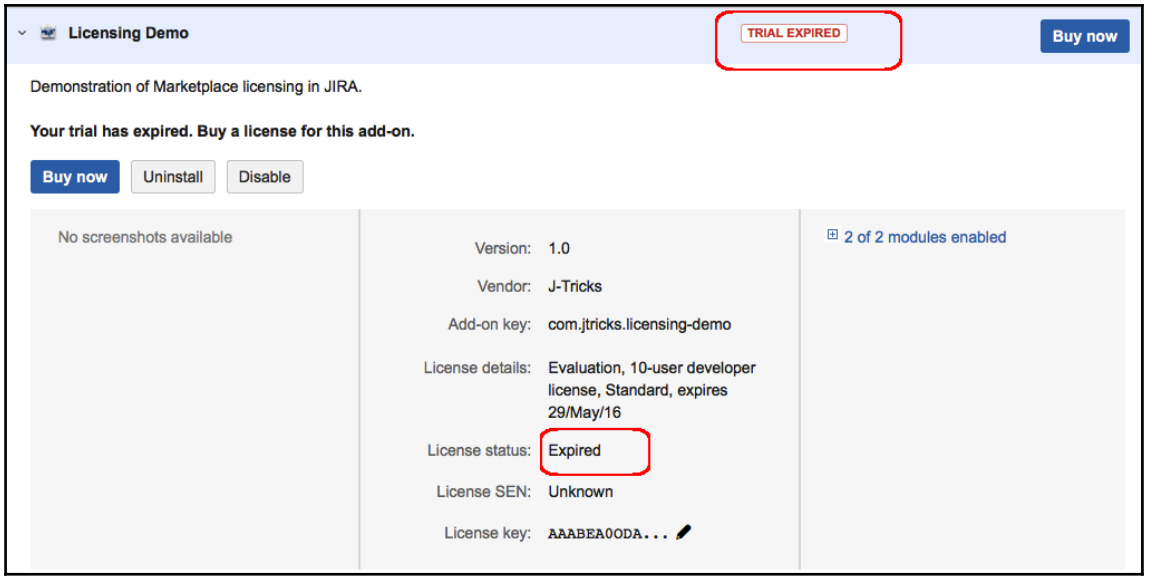
Once a valid license is installed, the user will see the following:



If the license is expired, you will see the following message.



After a license has expired, UPM will also warn administrators about the expired license, as shown here:



It is possible to obtain sample licenses at `https://developer.atlassian.com/market/add-on-licensing-for-developers/timebomb-licenses-for-testing` for testing various scenarios.

Hopefully, that gives you enough information on implementing Marketplace licensing so that you can use various methods, along with complex business logic, to make awesome commercial plugins.

And, as mentioned earlier, take a look at `https://developer.atlassian.com/market/developing-for-the-marketplace` to add the remaining bells and whistles!

Index

A

access logs, JIRA
  enabling 529
Active Objects 444
  about 42
  reference link 38, 455
  using, for data storage 455
active workflow
  editing 155
add-ons 43
admin permission 337
AdminUI section 326
advanced searching
  reference link 236
AJS (Atlassian JavaScipt) 379
annotations
  reference link 430
Apache Lucene
  reference link 39
Apache Tomcat 7.x 67
appearance, JIRA
  colors 324
  date and time formats 325
  favicon 324
  gadget colors 325
  logo 324
  modifying 324
  Site Title 324
Application Links
  reference link 230
application type 378
aQute bnd tool
  reference link 49
Atlassian bundled plugins
  modifying 56
Atlassian Connect
  reference link 9
Atlassian Gadget JavaScript Framework
  reference link 39
Atlassian JIRA 8
Atlassian Marketplace
  reference link 11
Atlassian Plugin SDK 56, 362
Atlassian plugin software development kit (SDK)
  about 11
  reference link 12
Atlassian Scheduler 43
Atlassian Spring Scanner 456
  about 82, 86
  reference link 92, 94, 145, 191, 517
Atlassian User Interface (AUI)
  reference link 74, 356
atlassian-plugin 98
Atlassian
  guidelines, reference links 433
AttachmentManager
  reference link 299
attachments
  AttachmentInput object, using 398
  browsing 399
  creating 297
  deleting 298
  file, using 399
  input stream, using 397
  new filename, using 398, 399
  reading, on issue 298
  reference link 297
  working with 296, 397
AUI Messages
  reference link 383

B

BIGINT 446
BND tool
  reference link 59
Browse permissions 245
Browse Project screen
  `new tabs, adding 352
built-in reports
  reference link 185
Bundle Dependency System 51
bundle instructions
  reference link 537

C

cascading selects
  reference link 209
change log value
  customizing 109
change logs
  programming 310
checkboxes 468
class visibilities
  reference link 309
color property
  density attribute 270
  name attributes 270
comments, on issues
  creating 307
  deleting 310
  restricting, to project role or group 308
  updating 309
  working with 307
comments
  creating, from e-mail 319
  working with 402
common libraries
  sharing, across v2 plugins 536, 538
components, JIRA
  adding 516
  exposing, to other plugins 518
  importing 516
  public components, importing 520
components
  managing 418
condition element 327
conditions element 327
conditions
  adding, for web fragments 337
configurable parameters
  adding, to service 487
configuration settings
  reference link 460
Confluence 12
content-disposition parameter 196
context-provider element 327
Crowd
  about 12
  reference link 38
CrowdService
  reference link 41
custom behavior
  adding, for user details 502, 503, 505
custom field
  adding, to notification e-mails 120
  developing, as date field 118
  developing, as user field 118
  displaying, on subtask columns 117
  help text, adding 123
  making project importable 127
  making sortable 115
  migrating, from one type to another 462
  migrating, to another custom field type 112
  options, programming 105
  searchers, creating 95
  text area size, changing 128
  type, changing 114
  validation, overriding 107
  working with, on issue 101
  writing 89
custom mail handler
  reference link 322
custom schema
  used, for extending JIRA database 444

D

database architecture, JIRA
  reference link 443
database entities
  accessing, from plugins 448

new record, writing 451
reading from 449, 450
record, updating 451
database products
reference link 42
database
custom field details, retrieving 467
details, retrieving 468, 470
groups, retrieving 477
history change, dealing with 480, 481
issue information, retrieving 463, 465, 467
issue on permissions, retrieving 471, 473
issue status, updating 475
issue, updating 476, 477
permission on issues, retrieving 470
users, retrieving 477
workflow details, retrieving 473, 474
debugging
about 33
steps 34
decorators
reference link 336
using 332
description 485
description module 387
descriptor 9, 350
development environment
IDEs configuration, for using SDK 14
local Maven, using 14
Maven download errors, troubleshooting 15
proxy setting, for Maven 13
setting up 11, 12, 13
direct HTML links
used, for operations 538, 542
doDefault method 72
doExecute method 72
doValidation method 72
downloadable plugin resources
reference link 211
dynamic notifications/warnings, on issue
displaying 379
dynamic-height feature 216

E

e-mail content
customizing 497, 500
Edit Issue Permission 159
elements
description 349
label 349
order 349
resource 349
sortable 349
Embedded Crowd 477
entity 445
entity definition
reference link 446
Entity Engine
reference link 450
entity modeling
reference link 443
entity-relation (ER) 39
Enum type 429
extract alias 273
extract, key
alias 272
path attribute 271
type attribute 272

F

FastDev
admin credentials, changing 32
ignored files, adding 31
reference link 31
using, for plugin development 28, 29, 31
field 236
field types
reference link 224
filter subscriptions
Cron Expression 278
Email On Empty 279
Group Name 279
reference link 279
filter
creating 275
deleting 276
managing programmatically 274

retrieving 276
sharing 277
special characters 278
subscribing 277
updating 276
used, for intercepting queries in JIRA 513, 515
working 281
Form Token Handling
reference link 540
frameworks, JIRA add-ons
Atlassian Connect 9
Plugins2 9
function 236
function class 238

G

Gadget JavaScript Framework 39
Gadget XML, specifications
Gadget characteristics 210
gadget content 210
reference link 213
required features 210
screenshot and thumbnail location 210
user preferences 210
gadgets
about 185
accessing, outside of JIRA 230, 231, 232, 233
authentication, reference link 221
user preferences, configuring 222, 224, 225, 227, 228, 229
getContent() method 253
group
creating 408
deleting 412
remote management 403
user, adding to 409
user, removing 412
users, searching 410

H

handlers
reference link 320, 322
HTML e-mails 499

I

i18n-name-key 373
Index Document Configuration module
about 271
entity-key attribute 271
key attribute 271
init-params 510
internationalization
in webwork plugins 532
IRA REST Java Client (JRJC)
reference link 391
issue entity properties
color property 270
searching 269
searching on 270, 272
working 272
issue link renderer 372
issue link renderers
adding 368
issue link
retrieving, on issue 315
issue links
deleting 314
destination issue 313
inward 313
outward 313
programming 313
reference link 313
source issue 313
Issue Navigator 246
issue operations
adding 290
conditions 294
re-ordering, in View Issue page 383
issue permissions, based on workflow status
restricting 157
Issue Security scheme 470
Issue Tab Panel 348
issue
about 283
browsing 396
comments, working with 307
creating 393, 394, 395
creating, from e-mail 319

creating, from plugin 283, 284
creating, with IssueManager 286
de-indexing 268
deleting 289
indexing 268
making, editable with workflow properties 159
making, non-editable with workflow properties 159
progressing, in workflow 413
searching 415
subtasks, creating 286
updating 288, 396
working with 393
IssueInputParameters
reference link 171
IssueOperations module
reference link 294
IssueService
reference link 104, 171
IssueTab PanelPluginModule 348
Item Plugin Module
reference link 290
itemGroups 363

J

J-Tricks 450
Java client
writing, for REST API 390, 392
Java Docs
reference link 268, 281
Javadocs
reference link 394
JavaScript tricks
applying, on issue fields 315
JAXB annotations 428
JDBC calls
database connection, obtaining 461
JIRA add-on 9
JIRA API policy
reference link 56
JIRA architecture
about 37, 40
authentication and user management 41
database 42
plugins 43
presentation 42
property management 42
reference link 37
scheduled jobs 43
searching 43
third party components 37
worflows 43
JIRA configuration properties
accessing 459, 460
JIRA Core 8
JIRA database
extending, with custom schema 444
plugin information, persisting 452
JIRA documentation
reference link 252
JIRA fields
assignee 259
date fields 259
issue key 258
IssueType 259
priority 259
project 259
reporter 259
resolution 260
versions 260
JIRA gadgets
working 215
writing 211, 212, 214
wrting 210
JIRA gagets
web resources, reference link 218
JIRA Home folder 443
JIRA plugin
Atlassian MarketplacE 11
Atlassian Marketplace 11
data, reusing in each run 26
deploying 22, 23, 24, 25
development process 10
specific version, using 25
troubleshooting 11, 26
JIRA Query Language (JQL) 415
JIRA Query Language (JQL) function
advanced searching, reference link 261
reference link 237
sanitizing 244

writing 237, 239, 241
JIRA Query Language (JQL) query
parsing, in plugins 264, 265
JIRA Query Language (JQL)
about 235
working 243
JIRA report
access, restricting to 200
data, validating 197
object configurable parameters 201
working 192
writing 185, 187, 188
JIRA REST API
reference link 390
JIRA REST Java Client (JRJC) 390
JIRA Service Desk 8
JIRA Software 8, 406
reference link 407
JIRA source code
download link 67
JIRA System plugins 52
JIRA Utility and Manager Classes 41
JIRA webhooks
about 436
JIRA workflow
reference link 133
step 132
JIRA.Messages
reference link 383
JIRA
about 9
Administration screen 331
advanced searching 236
basic searching 236
building, from source 66
feature 323
features 235
integration with development tools, reference link 162
listeners, writing 494
new webwork actions, adding 69
scheduled tasks, writing 490
service, writing 484, 485, 486
simple searching 236
single class path, creating 68
stable and core APIs 55
JiraDataTypes
reference link 238
jobinkk 409
jQuery AJAX options
reference link 220
JSPs 42
jtricks-group
about 409
reference link 412, 413
JUnit 33

K

key attribute
property-key 271
key/value parameters 328
key
about 186, 238
extract 271
keywords 236

L

label 327
listeners
writing, in JIRA 494, 496
location 327
location attribute 328

M

Marketplace licensing
implementing, in plugins 542, 547
Maven Bundle plugin 49
Maven repository
reference link 391
Maven
reference link 13
MessageFormat syntax
reference link 328
metadata
using 332
Mockito 33
module types
adding, in JIRA 522, 525
used, for creating modules 526
using 527

N

new drop-down menu
  adding, on top navigation bar 342
new Issue screen
  issue tab, loading asynchronously 352
new web sections
  adding, in UI 326
  reference link 326
non-sectioned 330
none option
  removing, from select field 124

O

object configrable parameters, for JIRA reports
  about 201, 206
  cascadingselect 205
  checkbox 205
  date 204
  filterpick 205
  filterprojectpicker 205
  hidden 204
  long 202
  multiselect 203
  select 202
  string 202
  text 204
  types 201
  user 204
  working 207, 209
object relational mapping (ORM) 38
OfBiz entity engine
  about 39
  reference link 39
OfBizDelegator
  reference link 449
OFBizDelegator
  reference link 444
Open for Business (OfBiz) 442
OpenSymphony documentation 37
Operations Plugin Module
  reference link 290
operations, IssueIndexingService
  deIndex(Issue issue) 268
  reIndex(Issue issue) 268
  reIndexAll() 268
operator 236
OPERATOR
  reference link 270
OSGI manifest entries 59
OSGI platform
  reference link 50
OSWorkflow 43, 132, 178
  conditions 132
  post functions 132
  triggers 132
  validators 132

P

Package javax.xml.bind.annotation
  reference link 429
param 327, 328
param element
  reference link 62
Permission scheme 470
Permissions
  reference link 158
plugin descriptor
  atlassian-lugin element 48
  plugin modules 50
  plugin-info 49
plugin development
  FastDev, using 28, 29
plugin information
  persisting, in JIRA database 452
plugin installation events
  capturing 81
plugin key 10
plugin module types
  remote invocation 46
  actions and components 46
  atlassian-plugin.xml 48
  custom fields 45
  links and tabs 45
  other types 47
  reporting 44
  searching 45
  workflow 44
plugin modules
  adding 19, 22

types 44
plugin
changes, making 27
debugging, in Eclipse 27
issue, creating 283, 284
issues, searching 261, 262, 263
redeploying 27
PluginManager 60
Plugins1 version
Pugins2 version, differences 50
Plugins2 9
plugins
database entities 448
JIRA Query Language (JQL) query, parsing 264
PluginScheduler 43
PowerMock 33
primarykey constraint 446
priority values
blocker 259
critical 259
major 259
minor 259
trivial 259
project dependencies
reference link 391
project roles
retrieving, reference link 424
Project Tab Panel class 354
project-centric view
new links, adding 355
new panels, adding 357
sub-navigation, adding 361
ProjectCustomFieldImporter implementations
reference link 128
property 429

Q

query parameter 430
Query Search
used, for smart querying 258, 260
query
linking, from template 265
Quick Search
used, for smart querying 258, 259

R

Relational Database Management Systems (RDBMS) 39
remote administration method
about 419
actors, adding to project role 424
JIRA project, creating 421
Permission Scheme, creating 420
project roles, retrieving 423
remote time tracking 400
report plugin module
attributes 186
elements 186
reporting, JIRA
gadgets 185
normal JIRA reports 185
reports
about 185
modification, for including Excel reports 194, 195, 197
reference link 197
resolutions, based on workflow transitions
selecting 160
resource element 327
resources
adding, into plugins 60
REST 390
REST API Browser (RAB)
using 433
REST API
about 390
data entities, exposing as 427
Java client, writing 390, 392
services, exposing as 427
REST methods
reference link 218, 403
REST services
invoking, from gadgets 217, 219

S

Scan and Reload button 31
scheduled tasks
writing, in JIRA 490, 493
Scheduler API 490

search link
  creating 266
Search Request View Plugin module 246
search request view
  adding 246
  searching 249
  Single Issue Views, used for rendering search views 252
  working 252
SearchResults
  reference link 263
sectioned locations 330
seraph
  reference link 38
service, registering
  reference link 486
service
  writing, in JIRA 484
Servlet Context Listener
  writing 511
Servlet Context
  shared parameters,a dding 510
servlet, JIRA
  deploying 507, 509
Shared Access Layer (SAL)
  reference link 40
SimpleDateFormat 325
Single Issue Views
  Single Issue Views, rendering to search views 257
single sign-on (SSO) 38
skeleton plugin
  creating 15, 16
  Eclipse project, creating 18
  stepping to 18
  working 16, 18
Smart Search 260
SOAP 390
spring scanner maven plugin 83
SQL logging
  enabling 531
string 402
Struts 37
sub-navigation
  adding, in project-centric vie 362
  adding, in project-centric view 361
subtasks
  creating, on issue 286
summary 186
swapGroup parameter
  reference link 413
synchronous-like 402

T

Test Driven Development (TDD)
  reference link 33
Test Text CF 86
TEST-125 259
testing
  about 33
  against different version of JIRA/Tomcat 35
  custom data for integration/functional tests, using 35
  steps 33, 34
text/plain content type 297
third-party components, JIRA
  about 37
  Active Objects 38
  Apache Lucene 39
  Atlassian Gadget JavaScript Framework 39
  Crowd 38
  OfBiz entity engine 39
  OSWorkflow 39
  PropertySet 38
  Seraph 38
  Shared Access Layer (SAL) 40
  Webwork 37
time tracking
  about 299
  reference link 300
  remaining estimate, auto adjusting 300
tooltip element 327
top navigation bar
  drop-down menu, adding 342
transformerless 522
tricks-group
  reference link 410
Twitter 370
Twitter4j
  reference link 369

Twitter
  reference link 369
twitterlink-fina 377

U

UI
  new web items, adding 329
  new web section, adding 326
Universal Plugin Manager (UPM) 24
user preferences
  about 214
  configuring, in gadgets 222, 224, 225, 227, 228, 229
user
  adding, to application 406
  creating 403
  deleting 407
  deleting, URL 408
  reference link 406, 409
  remote management 403
  removing, from application 407
  updating 405

V

value 236
value generator class 189
VARCHAR 446
velocity context for e-mail templates
  reference link 122
velocity context
  creating, for web fragments 340
velocity templates
  changes, reloading without restart 127
velocity
  writing, reference link 9
version1 plugin
  converting, to version2 plugin 58
versions
  creating 416
  managing 416
View Issue page 352
  fields, re-ordering 386
  Issue Operations, re-ordering 383
View Issue screen
  tabs, adding 348
ViewIssue page 348
views
  body 248
  footer 248
  header 248

W

web fragments
  conditions, adding 337
  velocity context, creating 340
web item
  condition/conditions 330
  context-provide 330
  link 330
  param 330
  resource 330
web resource contexts
  reference link 363
web resource plugin module
  reference link 65
web resources
  adding, into plugins 62
  batch mode, turning off 65
  contexts 65
web section
  creating 330
web-panel module
  reference link 358
WebSection 326
webwork actions
  adding, to JIRA 69
  different page, redirecting to 501
  form token, handling 78
  new commands, adding 77
  XSRF token, providing in HTML links 80
webwork plugins
  about 332
  internationalization 532
webwork
  reference link 38
weight 327
weight attribute 385, 387
workflow actions
  actions ID's given name, obtaining 170
  obtaining, programmatically 168

reordering, in JIRA 175
workflow conditions
key attributes, reference link 134
writing 133
workflow post function
reference link 148
writing 147
workflow statuses
internationalization 166
workflow triggers
adding 162
reference link 165
user mapping, from development tools 165
workflow validator
reference link 141
writing 140
workflow
about 131
common transitions, creating 177
global transitions, creating 180
history, obtaining from database 172
issue, progressing 413
modifying, in JIRA database 156
progression, programmatically 170
transition 132
worklog management
about 299
remaining estimate, adjusting by value 303
remaining estimate, retaining 301
work, logging 301
work, logging with remaining estimate 302
worklogs
remaining estimate, auto adjusting 306
deleting 306
deleting, with new remaining estimate 306
remaining estimate, retaining 306
updating 305

X

XML/RPC 390
XSRF token
obtaining, programmatically 81
opting out 81
providing, in HTML link 80