

Quick answers to common problems

JIRA 5.x Development Cookbook

This book is your one-stop resource for mastering JIRA extensions and customizations

Jobin Kuruvilla

[PACKT] PUBLISHING enterprise professional expertise distilled

JIRA 5.x Development Cookbook

This book is your one-stop resource for mastering JIRA extensions and customizations

Jobin Kuruvilla

[PACKT] enterprise
professional expertise distilled
PUBLISHING

BIRMINGHAM - MUMBAI

JIRA 5.x Development Cookbook

Copyright © 2013 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: April 2013

Production Reference: 1180413

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-908-6

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

Jobin Kuruvilla

Reviewers

Felix Martineau

Mizan Ali Sayed

Acquisition Editor

Kevin Colaco

Lead Technical Editor

Sruthi Kutty

Technical Editors

Dominic Pereira

Kirti Pujari

Mameet Singh Vasir

Hardik B. Soni

Project Coordinator

Hardik Patel

Proofreaders

Maria Gould

Paul Hindle

Aaron Nash

Indexer

Rekha Nair

Graphics

Abhinash Sahu

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

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.

Jobin is the author of *JIRA Development Cookbook*, *Packt Publishing*, released in 2011, which is a well-received book 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!

Acknowledgement

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 Project Coordinator, Hardik Patel, and Lead Technical Editor, Sruthi Kutty, both of whom went through the pain of making me write another book. Also, thanks to the entire Packt Publishing team for working so diligently to help bring out another high quality product.

It is amazing to work with a team of talented developers and technical geeks. Thank you STORM team, PD&B team, and RAMP team. Your encouragement and support were invaluable to me; you guys rock!

I must also thank the talented JIRA community who are instrumental in helping each other, sharing solutions, being active in the forums, running user groups, and what not? 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!

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.

About the Reviewers

Felix Martineau runs the Atlassian practice at TechSolCom (`http://www.techsolcom.ca`) and has been working as a professional JIRA consultant since 2008. Having started with JIRA in 2007, over the years he has worked with the entire Atlassian portfolio. Felix has a rare blend of technical expertise, people skills, and business acumen. For Felix, the people are always more important than the tools.

I want to thank the people at Packt Publishing for giving me this opportunity to review the book. Jobin Kuruvilla is one of the top people in the JIRA community, so it's an honor for me to be involved.

I would also like to thank my girlfriend Genevieve for her continuous support, as well as Paul Gamache and Jean Perron at TechSolCom, to whom I owe a lot professionally.

Mizan Ali Sayed is a Master in Computer Science from Pune university, India. Mizan currently works as an Atlassian tools specialist and has experience with implementing, customizing, and supporting large scale enterprise JIRA. He is active within the Atlassian forum "Answers" and has published open source add-ons on the Atlassian Marketplace.

I would like to thank my parents and friends for their continuous support and encouragement.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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.



`http://PacktLib.PacktPub.com`

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

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

This book is dedicated to Sarah, Anna, Anu, Juby, Alice, and Kuruvilla. Also, my friends from TKM and JNV Kottayam.

This book would not have been possible without you, one way or the other!

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 7

- Introduction 7
- Setting up the development environment 10
- Creating a skeleton plugin 13
- Adding plugin modules 17
- Deploying a JIRA plugin 19
- Making changes to and redeploying a plugin 23
- Using FastDev for plugin development 24
- Testing and debugging 28

Chapter 2: Understanding the Plugin Framework 31

- Introduction 31
- Architecture explained 34
- Types of plugin modules 39
- Working with the Plugins1 and Plugins2 versions 44
- JIRA system plugins 46
- Stable and core APIs 49
- Modifying Atlassian bundled plugins 50
- Converting plugins from v1 to v2 51
- Adding resources into plugins 54
- Adding web resources into plugins 56
- Building JIRA from source 59
- Adding new webwork actions to JIRA 63
- Extending a webwork action in JIRA 68
- Capturing plugin installation/uninstallation events 71

Chapter 3: Working with Custom Fields 77
Introduction 78
Writing a simple custom field 78
Custom field searchers 83
Dealing with custom fields on an issue 88
Programming custom field options 90
Overriding validation of custom fields 93
Customizing the change log value 94
Migrating from one custom field type to another 96
Making custom fields sortable 99
Displaying custom fields on subtask columns 100
User and date fields 102
Adding custom fields to notification mails 103
Adding help text for a custom field 105
Removing the "none" option from a select field 107
Making the custom field project importable 108
Changing the size of a text area custom field 109

Chapter 4: Programming Workflows 113
Introduction 114
Writing a workflow condition 115
Writing a workflow validator 122
Writing a workflow post function 129
Editing an active workflow 136
Making an issue editable/non-editable based on workflow status 138
Including/excluding resolutions for specific transitions 139
Permissions based on workflow status 140
Internationalization in workflow transitions 142
Obtaining available workflow actions programmatically 144
Programmatically progressing on workflows 146
Obtaining workflow history from the database 148
Reordering workflow actions in JIRA 151
Creating common transitions in workflows 153
Creating global transitions in workflows 158
Jelly escalation 160

Chapter 5: Gadgets and Reporting in JIRA 165
Introduction 166
Writing a JIRA report 167
Reports in an Excel format 174
Data validation in JIRA reports 177
Restricting access to reports 179

Object-configurable parameters for reports 181
Creating a pie chart in JIRA 189
Writing JIRA 4 gadgets 194
Invoking REST services from gadgets 202
Configuring user preferences in gadgets 206
Accessing gadgets outside of JIRA 214

Chapter 6: The Power of JIRA Searching 219
Introduction 219
Writing a JQL function 220
Sanitizing JQL functions 226
Adding a search request view 229
Smart querying using quick search 235
Searching in plugins 239
Parsing JQL queries in plugins 242
Linking directly to search queries 243
Index and de-index programmatically 244
Managing filters programmatically 246
Subscribing to a filter 249

Chapter 7: Programming Issues 253
Introduction 253
Creating an issue from a plugin 254
Creating subtasks on an issue 257
Updating an issue 258
Deleting an issue 259
Adding new issue operations 260
Conditions on issue operations 263
Working with attachments 265
Time tracking and worklog management 267
Working with comments on issues 275
Programming change logs 277
Programming issue links 279
Discarding fields while cloning 282
JavaScript tricks on issue fields 284
Creating issues and comments from an e-mail 287

Chapter 8: Customizing the UI 291
Introduction 292
Changing the basic look and feel 292
Adding new web sections in the UI 295
Adding new web items in the UI 297
Use of decorators 300

Adding conditions for web fragments 305
Creating new Velocity context for web fragments 308
Adding a new drop-down menu on the top navigation bar 310
Dynamic creation of web items 311
Adding new tabs in the View Issue screen 315
Adding new tabs in the Browse Project screen 319
Creating the project tab panel using fragments 321
Adding new tabs in the Browse Version screen 325
Adding new tabs in the Browse Component screen 328
Adding issue link renderers 330
Extending a webwork action to add UI elements 338
Displaying dynamic notifications/warnings on issues 342
Re-ordering issue operations in the View Issue page 346
Re-ordering fields in the View Issue page 348

Chapter 9: Remote Access to JIRA 353
Introduction 354
Writing a Java client for the REST API 354
Creating a SOAP client 356
Writing a Java XML-RPC client 358
Working with issues 360
Working with attachments 364
Remote time tracking 366
Working with comments 368
Remote user and group management 369
Progressing an issue in a workflow 370
Managing versions 371
Managing components 373
Remote administration methods 374
Exposing services and data entities as REST APIs 378
Deploying a SOAP service in JIRA 384
Deploying an XML-RPC service within JIRA 388

Chapter 10: Dealing with the JIRA Database 393
Introduction 394
Extending the JIRA database with a custom schema 395
Accessing database entities from plugins 399
Persisting plugin information in the JIRA database 401
Using Active Objects to store data 405
Accessing the JIRA configuration properties 408
Getting a database connection for JDBC calls 409
Migrating a custom field from one type to another 410

Retrieving issue information from a database 411
Retrieving custom field details from a database 414
Retrieving permissions on issues from a database 416
Retrieving workflow details from a database 419
Updating the issue status in a database 421
Retrieving users and groups from a database 422
Dealing with change history in a database 424

Chapter 11: Useful Recipes 427
Introduction 428
Writing a service in JIRA 428
Adding configurable parameters to a service 430
Writing scheduled tasks in JIRA 433
Writing listeners in JIRA 435
Customizing e-mail content 442
Redirecting to a different page in webwork actions 444
Adding custom behavior for user details 445
Deploying a servlet in JIRA 450
Adding shared parameters to the servlet context 453
Writing a servlet context listener 455
Using filters to intercept queries in JIRA 456
Adding and importing components in JIRA 459
Adding new module types to JIRA 463
Enabling access logs in JIRA 467
Enabling SQL logging in JIRA 470
Overriding JIRA's default components in plugins 471
Internationalization in webwork plugins 474
Sharing common libraries across v2 plugins 476
Operations using direct HTML links 478

Index 483

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 for mastering plugins in JIRA.

Then we will move on to writing custom field plugins to create new field types or custom searchers. We then learn how to program and customize workflows to transform JIRA into a user-friendly system.

We will then look at customizing the various searching aspects of JIRA such as JQL, searching in plugins, managing filters, and so on.

Then the book 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 latter half of the book, you will learn how to customize JIRA by adding new tabs, menus, and web items, and communicate with JIRA using the REST, SOAP, or XML/RPC interfaces, as well as working 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 the Plugin Framework*, covers, in detail, the JIRA architecture, and also looks at the various plugin points. It also looks at how to build JIRA from source and extend or override the 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 the JIRA workflows. It includes writing new conditions, validators, post functions, and so on, and contains related recipes that are useful in extending the workflows.

Chapter 5, *Gadgets and Reporting in JIRA*, covers the reporting capabilities of JIRA. It looks at writing reports, dashboard gadgets, and much more 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 the change logs and issue links, time tracking, and much more.

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 remote capabilities of JIRA—REST, SOAP, and XML/RPC—and the ways of extending them.

Chapter 10, *Dealing with the JIRA Database*, looks at the database architecture of JIRA and covers the major tables in detail. It also covers the different ways to extend the storage and access or modify the data from 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 5.*x*+
- JAVA 1.6+
- Maven 2.*x*
- Atlassian Plugin SDK
- An IDE of your choice. The examples in the book use Eclipse and SQL Developer.

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

If you are a JIRA developer or a project manager who wants to fully exploit the exciting capabilities of JIRA, then this is the perfect book for you.

Conventions

In this book, you will find a number of styles of text 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: " To use this, edit the `settings.xml` file under `M2_HOME/conf/` by modifying the `localRepository` attribute to point to the embedded `repository` folder."

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>
```

Any command-line input or output is written as follows:

```
select id from changegroup where issueid = '10010'
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "That is, you can just import the project using the **File** | **Import** | **Existing Maven Projects** option, and selecting 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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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 all Packt books you have purchased 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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Plugin Development Process

In this chapter, we will cover:

- Setting up the development environment
- Creating a skeleton plugin
- Adding plugin modules
- Deploying a JIRA plugin
- Making changes to 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 tracking** system. 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! Maybe a help desk system, a user story management system, an online approval process, and a lot more. This is in addition to the 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: **plugins**. JIRA has given the power to its users to write plugins 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 through plugins, and that is where we start our cookbook before we move on to the in-depth details.

What is a JIRA plugin?

So, what is a JIRA plugin? JIRA itself is a web application written in Java. That doesn't mean you need to know Java to write a plugin, though in most cases you will need to. You might 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.

A JIRA 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. Using Velocity templates is therefore recommended over JSPs.



You can find more details on writing Velocity templates at `http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html#velocity_template_language_vtl:_an_introduction`.

The **plugin descriptor**, the only mandatory part of a plugin, is an XML file that 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. 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:

- Developing the plugin
- Deploying it into local JIRA
- Testing the plugin functionality
- Making changes and redeploying the plugin if required

Each of these steps 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.

Following is a sample `atlassian-plugin.xml` file 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 from the preceding sample, has details such as description, version, vendor details, and so on.

When a plugin is loaded, all the unique modules in it are also loaded. The plugin classes override the system classes, and so if there is an action that has the same alias name as that of a JIRA action, it is the plugin action class that will be loaded. We will see more about extending actions in the coming chapters.

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 complete 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 that 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 a 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 your 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 our plugin development easier, Atlassian provides the **Atlassian plugin software development kit** (**SDK**). It comes along with Maven and a preconfigured `settings.xml` file 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 only on JIRA.

Getting ready

Following are the prerequisites for running the Atlassian plugin SDK:

- The default port **2990** for the SDK should be available. This is important because different ports are reserved for different Atlassian products.
- JDK should be installed and Java version 1.6.x is required.



The SDK is not yet compatible with Java 1.7.

- Make sure the `JAVA_HOME` environment variable is set properly and the `java -version` command outputs the correct Java version details.

How to do it...

Following are the steps to set up our development environment:

1. Once we have installed Java and the port 2990 is ready, we can download the latest version of Atlassian plugin SDK from `https://developer.atlassian.com/display/DOCS/Getting+Started`.

2. Install the SDK by following the instructions provided on the *Atlassian Developer Documentation* page, depending upon the operating system. Let's call this directory `SDK_HOME`.
3. Add the SDK's `bin` directory into the environment `Path` variable.
4. Create a new environment variable, `M2_HOME`, pointing to the `apache-maven` directory in your `/SDK_HOME/atlassian-plugin-sdk` directory.



SDK version 4.x and above doesn't need this step.

5. A lot of commonly used dependencies are already available in the `repository` folder embedded in the SDK. To use this, edit the `settings.xml` file under `M2_HOME/conf/` by modifying the `localRepository` attribute to point to the embedded `repository` folder. By default, it will use the `USER_HOME/.m2/repository` directory.
6. Install the IDE of your choice. Atlassian recommends Eclipse, IntelliJ IDEA, or NetBeans, as they all support Maven.

With the preceding steps executed properly, we have a development environment for JIRA plugins. The next step is to create a skeleton plugin, import it in to your IDE, and start writing some code! Creating the skeleton plugin, deploying it, and so on, is explained in detail in the forthcoming recipes of this chapter. So, ready, set, go...

There's more...

If you are facing issues while downloading the dependencies using Maven, just read on.

Proxy settings for Maven

If you are executing behind a firewall, make sure you have configured proxy settings in Maven's `settings.xml` file. A 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>
```



To find out more about proxy settings for Maven and other aspects of it, go to `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` environment variable to your local Maven, and update the respective `settings.xml` file with the repository details in the default `settings.xml` file that ships with the Atlassian plugin SDK. Or, you can simply add the following lines to the existing `settings.xml` file:

```
<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, configuring it to use the SDK is an easy job because IDEA integrates Maven out of the box. Just load the project by selecting the `pom.xml` file! See `https://developer.atlassian.com/display/DOCS/Configure+IDEA+to+use+the+SDK` for more 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/display/DOCS/Set+Up+the+Eclipse+IDE+for+Linux` for Linux, or `https://developer.atlassian.com/display/DOCS/Set+Up+the+Eclipse+IDE+for+Windows` for Windows.

If you are using Netbeans, check out `https://developer.atlassian.com/display/DOCS/Configure+NetBeans+to+use+the+SDK` on how to configure it 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's `settings.xml` file 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.

See also

- The *Creating a skeleton plugin* recipe

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

Before beginning this recipe, make sure you have the Atlassian plugin SDK installed.

How to do it...

Following are the steps to create a skeleton plugin:

1. Open a command window and go to the folder where you want to create the plugin.



Make sure you use a directory without any spaces in its name, because there are known issues with the SDK not working in directory names with spaces. See `https://studio.atlassian.com/browse/AMPS-126` for more details.

2. Type `atlas-create-jira-plugin` and press *Enter*.
3. Press *1* if you want to create a JIRA 5 plugin or *2* if you want to create a plugin for JIRA 4 or earlier versions.



This step can be avoided if you use the `atlas-create-jira5-plugin` command instead of `atlas-create-jira-plugin`. Note the use of `jira5` instead of `jira`!

4. Enter the `group-id` information when prompted. `group-id` 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. `group-id` will be used to identify your plugin along with `artifact-id`, for example, `com.jtricks.demo`.
5. Enter the `artifact-id` information (the identifier for this artifact) when prompted. Do not use spaces here, for example, `demoplugin`.
6. Enter the `version` information when prompted. The default version is `1.0-SNAPSHOT`. Enter a new version if you want to change it or press *Enter* to keep the default version, for example, `1.0`.
7. Enter the `package` information when prompted. Press *Enter* if the package value is the same as the `group-id` value. If not, enter the new value here and press *Enter*, for example, `com.jtricks.mypackage`.
8. Confirm the selection when prompted. If you want to change any of the entered values, type *N* and press *Enter*.
9. Wait for the **BUILD SUCCESSFUL** message to appear.



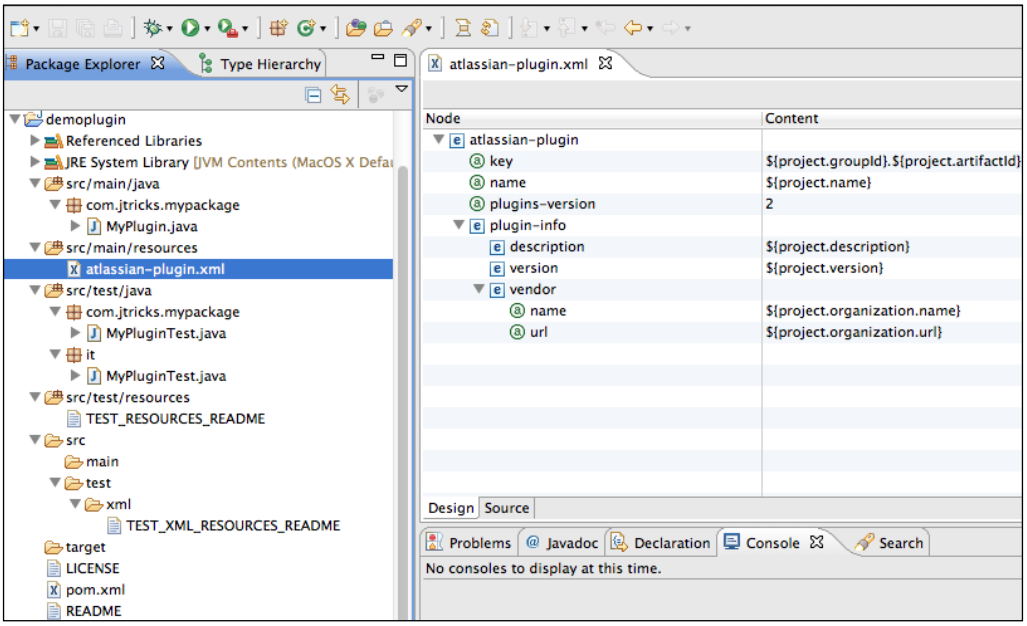
Downloading the example code

You can download the example code files for all Packt books you have purchased 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.

How it works...

A skeleton plugin is nothing but a set of directories and subdirectories, along with a `pom.xml` file (Maven Project Object Model) and some sample Java and XML files in the appropriate folders.

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



As you can see, there is a `pom.xml` file 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 shown in the previous screenshot are populated from the `pom.xml` file. In our case, the plugin key will be populated as `com.jtricks.demo:demoplugin` when the plugin is built.

There are two more folders under the `src/test` tree. The `src/test/resources` folder, which will hold any resources required for unit tests or integration tests, and the `src/test/xml` folder, which can hold the XML data from any other JIRA instance. If the XML is supplied, the SDK will use it to configure the JIRA instance before running the integration tests.

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



Remember, the first Maven run is going to take some time, as it downloads all the dependencies into 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 the next 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 the parameters such as `group-id`, `artifact-id`, and so on, as arguments to the `atlas-create-jira-plugin` command:

```
atlas-create-jira5-plugin -g my_groupID -a my_artefactId
-v my_version -p my_package ---non-interactive
```

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

```
atlas-create-jira5-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 too!

Creating an Eclipse project

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

```
atlas-mvneclipse: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 **File** | **Import** | **Existing Maven Projects** option, and select the relevant project.

See also

- The *Adding plugin modules* recipe
- The *Deploying a JIRA plugin* recipe
- The *Making changes to and redeploying a plugin* recipe

Adding plugin modules

In this recipe, we will look at adding plugin modules into 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...

Following are the steps to add plugin modules into an existing plugin project:

1. Open a command window and go to the plugin project folder where `pom.xml` resides.
2. Type `atlas-create-jira-plugin-module` and press *Enter*. This will show all the available plugin modules as a numbered list, as shown in the following screenshot:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building demoplugin
[INFO]    task-segment: [com.atlassian.maven.plugins:maven-jira-plugin:3.11:create-plugin-module]
[INFO] ------------------------------------------------------------------------
[INFO] [jira:create-plugin-module]
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:  JQL Function
10: Keyboard Shortcut
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):
```

3. 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.
4. Follow the instructions to provide 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*.

5. Once the module is completed, type *Y* or *N* and press *Enter* when prompted, depending on whether you want to add another module or not.
6. Repeat the steps for every module you want to add.
7. Wait for the **BUILD SUCCESSFUL** message to appear when no more modules are there to be added.

How it works...

Similar to the skeleton plugin creation, a set of directories and subdirectories are created during the process of adding plugin modules, 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` file based on our inputs in step 4. A sample plugin descriptor, after adding the Web Item module, is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<atlassian-plugin key="${project.groupId}.${project.artifactId}"
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}"/>
  </plugin-info>
  <resource type="i18n" name="i18n" location="com.jtricks.demo.
demoplugin"/>
  <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. You can also see a `resource` module, which is added automatically the first time a plugin module is created. This will be the **i18n** resource file for the plugin, and more **key-value pairs** will be added into 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, a `demo.properties` file is created under the `src/main/resources/com/jtricks/demo` folder.

A sample property file is shown as follows:

```
my-web-item.label=J-Tricks
my-web-item.name=My Web Item
my-web-item.description=The My Web Item Plugin
```

See also

- The *Creating a skeleton plugin* recipe
- The *Deploying a JIRA plugin* recipe
- The *Making changes to and redeploying a plugin* recipe

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 the 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. The following steps show 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` file resides.

2. Type `atlas-run` and press *Enter*. It is possible to pass more options as arguments to this command. The details regarding this can be found at `https://developer.atlassian.com/display/DOCS/atlas-run`.

   You will see a lot of things happening as Maven downloads all the dependent libraries into your local repository. As usual, it is going to take a lot of time when you run it for the first time.

3. If you are on Windows, and if you see a security alert popping up, click on **Unblock** to allow incoming network connections.
4. When the installation is complete, you will see the following message:

   ```
   [WARNING] [talledLocalContainer] INFO: Server startup in 123558 ms
   [INFO] [talledLocalContainer] Tomcat 6.x started on port [2990]
   [INFO] jira started successfully and available at http://
   localhost:2990/jira
   [INFO] Type CTRL-C to exit
   ```

5. Open `http://localhost:2990/jira` in your browser.
6. Log in using the username as `admin` and password as `admin`.
7. Test your plugin! You can always go to **Administration** | **Plugins** | **Manage Add-ons** to confirm that the plugin is deployed properly.
8. If you already have a local JIRA installed, or if you want to manually install your plugin, all you need to do is to package the plugin JAR and install it via the **Universal Plugin Manager** (**UPM**), as described in detail at `https://confluence.atlassian.com/display/UPM/Installing+Add-ons#InstallingAdd-ons-Installingbyfileupload`. Or, you can copy it across to the `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 packaged plugin into your local repository.

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 tasks:

- Builds your plugin JAR file
- 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's `/target` folder
- Copies the JAR file into 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 two main folders are the `container` folder, which has the Tomcat container setup, and the `jira` folder, which has the JIRA WAR along with the JIRA home setup!

You will find the database (HSQLDB), indexes, backups, and attachments under `/target/jira/home`. You can also see your `jira-webapp` at `/target/container/tomcat6x/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 re-use the data that we have used for testing. To find out how, just read on.

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 this:

- Mention the JIRA version as an argument to `atlas-run`. Make sure you run `atlas-clean` if you already have the latest version deployed.
    1. Run `atlas-clean` (if required).
    2. 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's `pom.xml` file.
    1. Go to your `pom.xml` file.
    2. Modify the `jira.version` property value to the desired version.
    3. Modify the `jira.data.version` property value to a matching version.

    Following 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>
```

Re-using the configurations in each run

Suppose you have added some data onto virtual JIRA; how do you retain it when you clean startup JIRA next time? This is where a new SDK command comes to our rescue. After the `atlas-run` command has finished its execution, that is, after you have pressed *Ctrl + C*, execute the following command:

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

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

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

Modify the path accordingly. This will re-use the configurations the next time you run `atlas-run`.

Troubleshooting

Following are some points to remember:

- Missing a JAR file exception? Make sure the `local-repository` attribute in the `settings.xml` file 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 into the local repository.



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

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

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

  Run `atlas-clean` followed by `atlas-run` after you have added the preceding entry.

Making changes to 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 redeploy it while the JIRA application is still running. Following is how we do it:

1. Keep the JIRA application running in the window where we executed `atlas-run`.
2. Open a new command window and go to the root plugin folder where your `pom.xml` file resides.
3. Run `atlas-cli`.
4. Wait for the **Waiting for commands** message to appear.
5. Run the `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.

There's more...

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

Debugging in Eclipse

Following are the steps to debug your plugin in Eclipse:

1. Use `atlas-debug` instead of `atlas-run`.
2. Once the virtual JIRA is up and running with your plugin deployed in it, go to **Run** | **Debug Configurations in Eclipse**.
3. Create a new remote Java application.
4. Give your application a name, keep the defaults, and give the port number as `5005`. This is the default debug port on which the virtual JIRA runs.
That's it! Happy debugging!

See also

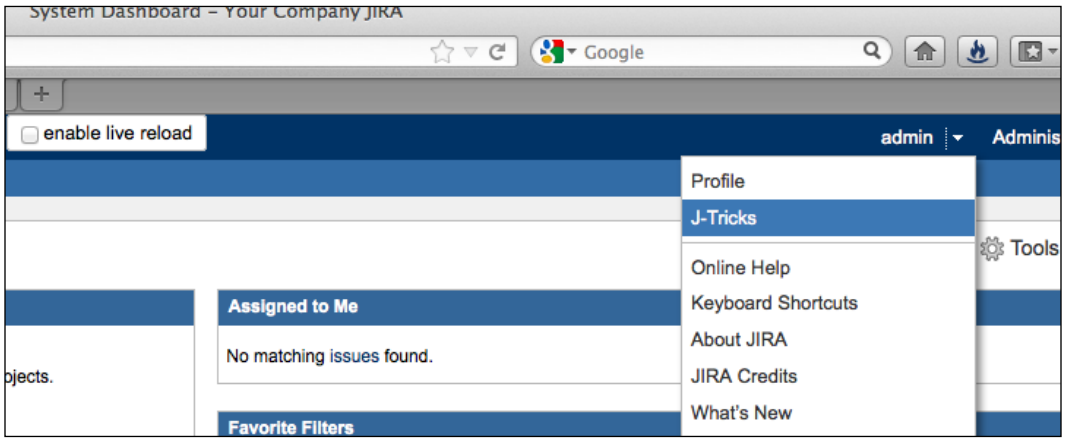
- The *Setting up the development environment* recipe
- The *Creating a skeleton plugin* recipe
- The *Using FastDev for plugin development* recipe

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 discussed 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. If we run this sample Web Item plugin, using `atlas-run`, we can access JIRA at port 2990 as mentioned earlier, and the Web Item will look as highlighted in the following screenshot:



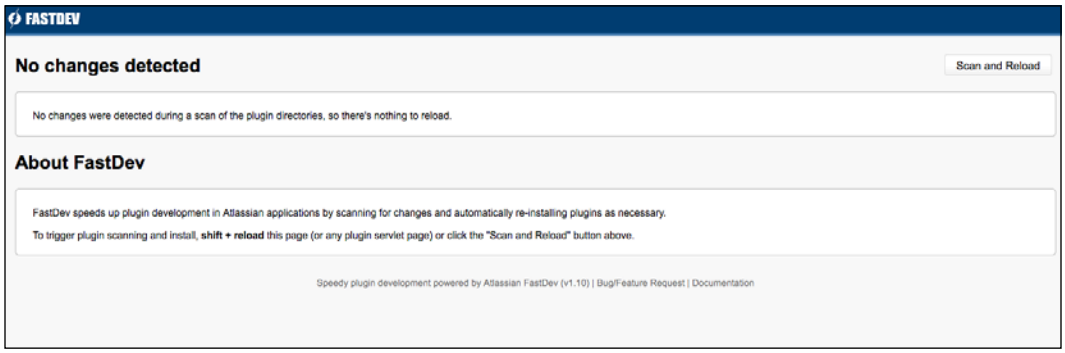
As you can see, the **J-Tricks** link appears along with the **Profile** link under the personal section. But what if you wanted the link at the jira-help section, along with **Online Help**, **About JIRA**, and so on?

How to do it...

Following are the simple steps to make the aforementioned change to the plugin and reload it using FastDev:

1. Access the FastDev servlet on the browser using the following path: `http://localhost:2990/jira/plugins/servlet/fastdev`

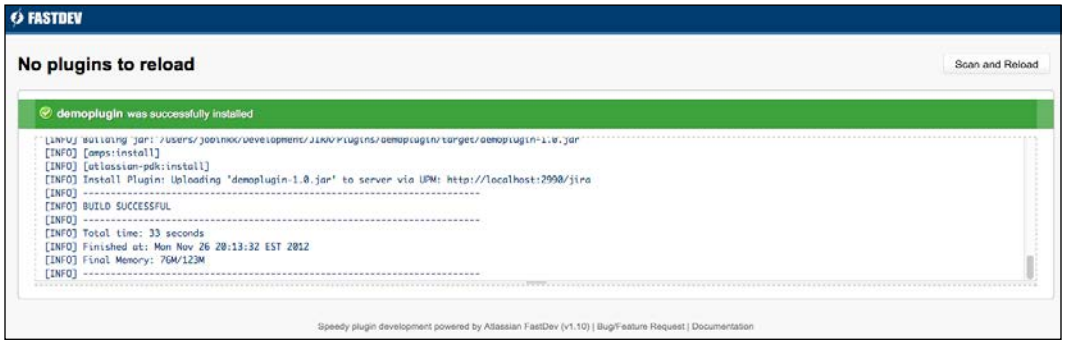
   You will find the servlet as shown in the following screenshot:



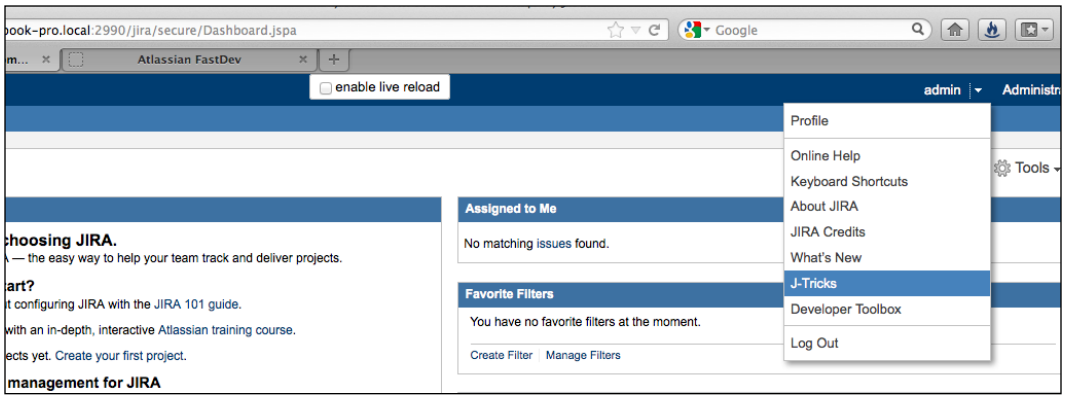
2. Make the change to your plugin. In this example, the change is pretty small. All we do is modify the `atlassian-plugin.xml` file to change the `section` attribute in the `web-item` module from `section="system.user.options/personal"` to `section="system.user.options/jira-help"`.
3. Click on the **Scan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



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



5. Reload the JIRA page to see if the change is effective. In our example, the new screen will be as follows:



As you can see, the **J-Tricks** link has moved to the help section.



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

How it works...

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



More information on FastDev can be found at `https://developer.atlassian.com/display/DOCS/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. You will find the details in the Atlassian documentation (see the previous note box), but the most useful ones are mentioned later.

Adding ignored files

While looking for changes, FastDev ignores certain files such as `.js` or `.css` files that don't need a reinstall of the plugin. The following table shows 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, shown as follows:

```
<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/LICENSE.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` properties, shown as follows:

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

See also

- The *Setting up the development environment* recipe
- The *Creating a skeleton plugin* recipe
- The *Making changes to and redeploying a plugin* recipe

Testing and debugging

In the world of **test-driven development** (**TDD**), writing tests is a part and parcel of the development process. I don't want to bore you with why testing is important! Let us just assume that this holds true for a JIRA plugin development as well. In this recipe, we will see the various commands for running unit tests and integration tests in JIRA plugins.



If you are wondering what exactly TDD is, just go to `http://en.wikipedia.org/wiki/Test-driven_development`.

Getting ready

Make sure you have the plugin development environment set up and the skeleton plugin created! 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/your_package/` and `src/test/ java/it` folders. Once you have done this, it is time to write some tests and run those tests to make sure things work as expected!

How to do it...

Without further delay, lets perform the following steps:

1. The first step is to write some tests! We recommend you use 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 onto your `pom.xml` file.
2. Let us now make a huge assumption that you have written a few tests! 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.
3. If you are running the integration tests, the command to use is:

   ```
   atlas-integration-test
   ```

   Or, the Maven command to use is:

   ```
   atlas-mvn clean integration-test
   ```
4. Once we are onto the stage of running tests, we will see them failing at times. Here comes the need for *debugging*. Checkout the `*.txt` and `*.xml` files created under `target/ surefire-reports/`, which have all the required information on the various tests that are executed.
5. 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.
6. You can also use the Maven options directly to skip the unit/integrations tests or both together:
   - `-Dmaven.test.skip=true`: Skips both unit and integration tests
   - `-Dmaven.test.unit.skip=true`: Skips unit tests
   - `-Dmaven.test.it.skip=true`: 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` command 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!

But 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 else will be treated as unit tests!

There's more...

It is also possible to use custom data for integration tests and to test it against different JIRA versions.

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. Save it under the `src/test/xml/` directory.
3. Provide this path as the value for the `jira.xml.data.location` property in the `localtest.properties` file under `src/main/resources`.

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

Testing against different versions 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. Just like before, make sure you run `atlas-clean` before executing the tests if you had tested it against another version earlier.

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

- The *Setting up the development environment* recipe
- The *Deploying a JIRA plugin* recipe

2
Understanding the Plugin Framework

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

- Modifying Atlassian bundled plugins
- Converting plugins from v1 to v2
- Adding resources into plugins
- Adding web resources into plugins
- Building JIRA from source
- Adding new webwork actions to JIRA
- Extending a webwork action in JIRA
- 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 look at 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/display/JIRADEV/JIRA+Architectural+Overview`. We will redraw the diagram a little bit to explain it in a brief but useful way.

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.

Webwork

Webwork is nothing but a Java web application development framework. The following is a quick overview of webwork as described 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."*

Note that JIRA uses webwork1 and not 2. 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 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/`.

OSUser

OSUser is OpenSymphony's user and group management framework. It is designed to provide a simple-to-use API for user management. JIRA uses OSUser framework in versions prior to JIRA 4.3.

Embedded Crowd

JIRA 4.3+ uses Crowd, which is Atlassian's Identity management and Single Sign-on tool, for user management. JIRA embeds a subset of Crowd's core modules, which provides the user with 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 **Object Relational Mapping** (**ORM**) 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 have also discussed about Active Objects in *Chapter 10*, *Dealing with 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 modeling concepts of Relational Database Management Systems:

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

Read more about Entity Modeling and Entity-Relation concepts at `http://ofbiz.apache.org/docs/entity.html`.

Apache Lucene

The following is a simple definition of Apache Lucene that you can find in its documentation:

> *Apache Lucene(TM) 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 4 introduces a powerful gadget framework. Atlassian has gone OpenSocial with 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 between gadgets.

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

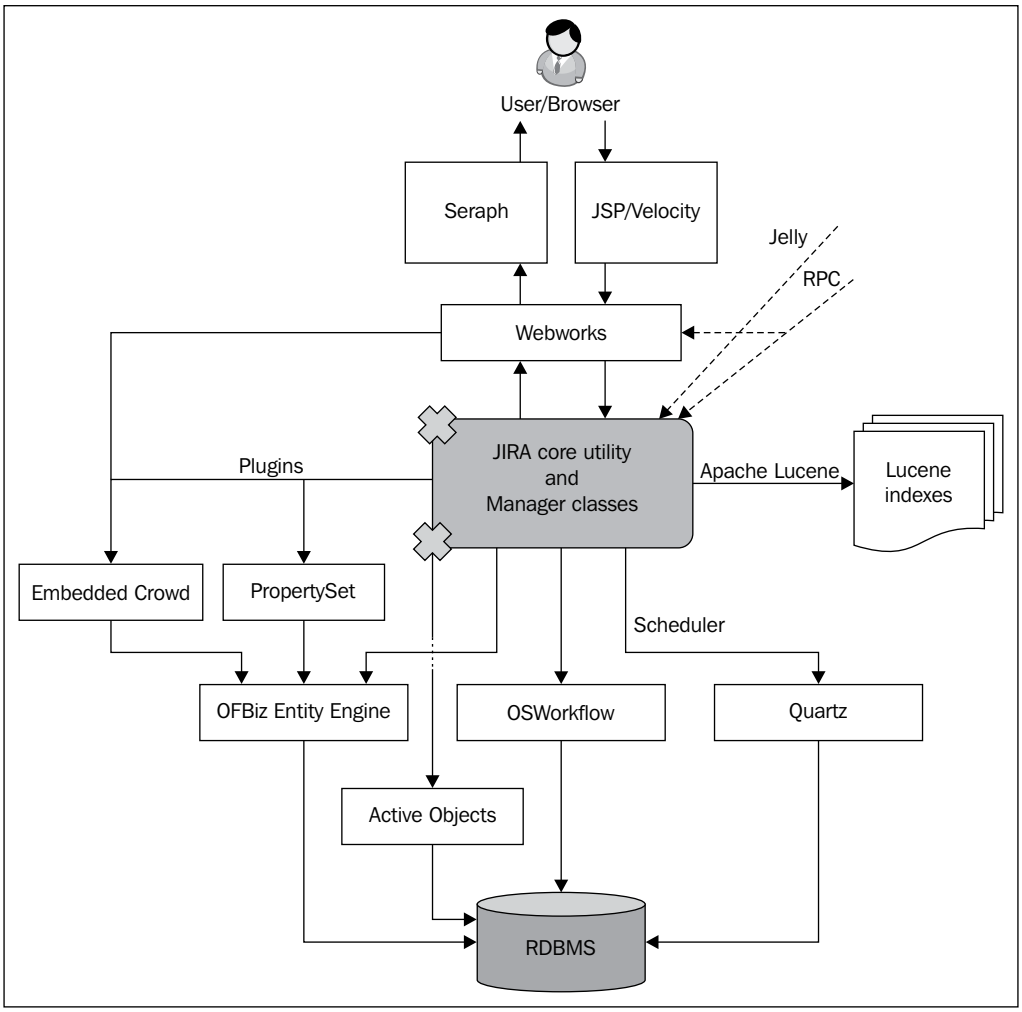
Quartz

Quartz is an open source job scheduling service. It can be used to create jobs that can be scheduled within any Java EE and SE applications. The tasks are defined as standard Java components and scheduler includes many enterprise-class features, such as JTA transactions and clustering.

Read more at `http://www.quartz-scheduler.org/`.

Architecture explained

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 (or a long stare, whichever you are more comfortable with!) 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 becomes the heart of JIRA. It also interacts with a lot of third-party components, which we saw earlier, to deliver powerful functionalities such as workflows, permissions, user management, and searching.

As with any other web application, let us start with the incoming requests. Users interact with JIRA using web browsers, but there are other ways to interact with JIRA, such as using the **Jelly scripts** or by making remote calls using **REST/SOAP/XML-RPC**.

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 them with a specific user. It supports various authentication mechanisms such as HTTP Basic Authentication, form-based authentication, and it even looks up already stored credentials in a user session when implemented with SSO (Single Sign On).

However, Seraph doesn't do any user management itself. It delegates this to the Embedded Crowd framework (OSUser in JIRA versions prior to 4.3). One additional thing that Seraph does in JIRA is to intercept URLs starting with `/admin/` and allow users access only if they have the "Global Admin" permission.

Coming back to authentication and other user management functions, it is OSUser that does the work for JIRA in versions prior to 4.3. It does the following activities:

- **User management**: Creates/updates/deletes users and stores the details in the JIRA database. It also stores user preferences.
- **Group management**: Creates/updates/deletes groups and stores the details in the JIRA database. It also manages group memberships.
- **Authentication**: Password matching.

From JIRA 4.3 onward, user management in JIRA is done using Crowd. **Crowd** is a single sign-on and identity management system from Atlassian, which is now embedded in JIRA 4.3+. 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`.

The following are the core Crowd functionalities:

- User, Group, and Membership management—stores all details in JIRA DB
- Authentication—password matching
- Allows connection to external user directories such as LDAP, AD, and Crowd.
- 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 such as User, Group, Project, and Issue. 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 by the OSUser framework like 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 new technology called Active Objects, which can be used to store plugin data. This is explained in detail in *Chapter 10*, *Dealing with the JIRA Database*.

Presentation

The presentation layer in JIRA is built using JSPs and Velocity templates. Another experimental component available in JIRA 5 for the 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 that 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 `entitmodel.xml` and `entitygroup.xml` files residing in the `WEB-INF/classes/entitydefs` folder. The entity configuration details goes in to `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/display/JIRA/Connecting+JIRA+to+a+Database`.

Workflows

Workflows are one of the most important features in JIRA. They provide us with a highly configurable workflow engine that 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, or post functions. We can even write plugins to add more of these, in addition to the ones that ship with 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 changes.

Scheduled jobs

JIRA uses the **Quartz** API to schedule jobs. These jobs, including the subscriptions to the filters and the custom ones we add, are stored in the JIRA database, and are executed by the Quartz job scheduling service.

JIRA's built-in scheduled job details can be found at `scheduler-config.xml`. It is possible to schedule new events in JIRA using the SAL services implementation. As Atlassian puts it:

> *The Shared Access Layer, or SAL inshort, provides a consistent, cohesive API to common plugin tasks, regardless of the Atlassian application into which your plugin is deployed.*

More information on scheduling events in JIRA using SAL can be found at `https://developer.atlassian.com/display/DOCS/Plugin+Tutorial+-+Scheduling+Events+via+SAL`.

Plugins

Last but not least, plugins 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 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 in the coming chapters, and we will cover how to customize them by writing plugins. Off you go!

Types of plugin modules

Let us briefly see the different types of plugin modules supported in JIRA5. 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.

Let us group them based on functionality instead of seeing them all together!

Reporting

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 in to JIRA.

Workflows

Module type	Description
`workflow-condition`	Adds new workflow conditions to the JIRA workflow. It can then be used to limit the workflow actions to users, based on pre-defined conditions.
`workflow-validator`	Adds new workflow validations to the JIRA workflow. Validations can be used to prevent certain workflow actions when certain 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

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 creating custom logic. See also `customfield-searcher`.

Searching

Module type	Description
`customfield-searcher`	Adds new field searchers in 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.

Links and tabs

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 earlier or the existing JIRA web sections.
`project-tabpanel`	Adds new tabs to the **Browse Project** screen. We can define what has to appear in the tab.
`component-tabpanel`	Adds new tabs to the **Browse Component** screen. As above, we can define what appears in the tab.
`version-tabpanel`	Adds new tabs to the **Browse Version** screen. Same as above.
`issue-tabpanel`	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 renderers for issue links to JIRA. This is useful for Remote Issue Links.

Remote invocation

Module type	Description
`rest`	Creates new REST APIs for JIRA to expose more services and data entities.
`rpc-soap`	Publishes new SOAP end-points for JIRA. It is deployed as a new SOAP service and exposes a new WSDL with the operations we have published in the plugin.
`rpc-xmlrpc`	Same as above. Exposes XML-RPC endpoints, instead of SOAP, within JIRA.

Actions and components

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

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, this 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 into JIRA. The order and position in the application's filter chain can be specified.

Module type	Description
`user-format`	Adds custom behaviors for user details. This is used to enhance the user profile.
`keyboard-shortcut`	Available only from 4.1.x. This defines new keyboard shortcuts for JIRA. You can also override the existing shortcuts from JIRA 4.2.x!
`ao`	Allows the use of the Active Objects service to persist plugin data.
`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="${project.groupId}.${project.artifactId}"
name="${project.artifactId}" plugins-version="2">
  <plugin-info>
    <description>${project.description}</description>
    <version>${project.version}</version>
    <vendor name="${project.organization.name}"
url="${project.organization.url}" />
  </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 part. 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}` 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 plugin menu under administration.

- `plugins-version`: This is different from the `version` attribute. `plugins-version` defines whether the plugin is version 1 or 2. `plugins-version="2"` defines the plugin as a version 2 plugin. Remove the entire attribute to make it a version 1 plugin.
- `state`: This is an optional element to define the plugin as disabled, by default. Add `state="disabled"` under the `atlassian-plugin` element.

- **plugin-info element**: 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 that 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="5.0.2" max="5.1"/>` will be supported from 5.0.2 to 5.1.



Remember, this is only for information's sake. The plugin might still work fine in JIRA 5.2!

  - `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 the 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, `<paramname="configure.url">/secure/JTricksConfigAction.jspa</param>` defines the configuration URL for our demo plugin.
  - `bundle-instructions`: Here we define the OSGi bundle instructions that will be used by the Maven Bundle Plugin while generating the OSGi bundle.



More about this can be read under aQute's bnd tool at `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 section where the actual plugin modules (which we saw a bit earlier and will see in detail later in this book) will appear.

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

Working with the Plugins1 and Plugins2 versions

Let us also quickly see how to deal with the **Plugins1** and **Plugins2** versions.

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

The key motive behind version2 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.

The following table shows the key differences at the development stage of plugin development for both the versions:

	Plugins1	Plugins2
Version	No `plugins-version` element in `atlassian-plugin.xml`.	Include the `plugins-version` element in the `atlassian-plugin.xml` file 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 the 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 as `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 the *Managing complex dependencies* row in this table.
Dependency injection	Done by Picocontainer 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.

	Plugins1	Plugins2
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** therefore allows you to define complex dependencies between plugins, eliminating the classpath contradictions and the upgradation of plugins.

The following table shows the key differences at the installation stage of plugin development for both the versions:

Plugins1	Plugins2
Plugins must be on the application classpath. Hence deploy it under the `WEB-INF/lib` folder.	Plugins 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/`. `jira-home` is declared in the `jira-application.properties` file under `WEB-INF/classes`.

Right, we now know how the two plugin versions work. Maybe, it is time to see the plugins that JIRA comes with!

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. This 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 information in `*.xml` files placed under `WEB-INF/classes`. 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-contentlinkresolvers-plugin.xml`	**System content link resolvers**: Resolves parsed content links into link objects. ▸ Attachment link resolver ▸ Anchor link resolver ▸ JIRA issue link resolver ▸ User profile link resolver
`system-customfieldtypes-plugin.xml`	**JIRA system custom fields**: All the out-of-the-box custom fields in JIRA and the searcher associations. Examples are: ▸ Text field ▸ Text area ▸ User picker ▸ Select
`system-footer-plugin.xml`	This plugin renders the content of the footer in JIRA.
`system-issueoperations-plugin.xml`	**System issue operations**: Renders the issue operations using `web-items` grouped using `web-sections`. Examples are: ▸ Edit issue ▸ Assign issue ▸ Log work
`system-issuetabpanels-plugin.xml`	**System issue tab panels**: Renders the various tabs on the **View Issue** page. Examples are: ▸ All tab panel ▸ Comment tab panel ▸ Work log tab panel ▸ Change history tab panel ▸ CVS tab panel
`system-issueviews-plugin.xml`	Renders the single issue view and the various search request views. Examples are: ▸ **Single issue views**: XML, Word, Printable ▸ **Search views**: XML, RSS, RSS (comments), Printable, Word, Full content, Excel (all fields), Excel (current fields), Charts

System plugin XML	Functionality
`system-jql-function-plugin.xml`	Built-in JQL functions.
`system-keyboard-shortcuts-plugin.xml`	Built-in keyboard shortcuts.
`system-macros-plugin.xml`	JIRA's base system macros.
`system-project-plugin.xml`	System project panels: Renders the Browse Project, Browse Version, and Browse Component panels.
`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-renderercomponentfactories-plugin.xml`	Renderer component factories plugin: Instantiates renderer components using the plugin system. Examples: Macro renderer, Link renderer, URL renderer, and so on.
`system-renderers-plugin.xml`	Built-in system renderers: ▸ 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-webresources-plugin.xml`	System web resources: Includes static resources such as JavaScript files, style sheets, and so on.
`system-webwork1-plugin.xml`	System webwork plugin: Can be used to add custom webwork actions, which can also be done using plugins.
`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.



Care must be taken to upgrade the changes during the time of a JIRA upgrade.

So that was a pretty lengthy introduction to the JIRA architecture! Let us quickly move on to the recipes in this chapter. Time to code!!

Stable and core APIs

So, we are in to coding. If you are familiar with JIRA plugin development or have been using them prior to JIRA 5, one thing you might have noticed is that there are quite a few versions of every plugin out there! There are not many plugin versions that work on all JIRA versions. You might also know the reason behind this: JIRA has been evolving a lot in terms of its APIs.

With JIRA 5, Atlassian has introduced **stable** and **core** APIs, and with great interest, the development community read the following 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 JIRA 5, is now replaced by two different dependencies: `jira-api` and `jira-core`. `jira-api` (aka the stable API) is a set of classes or interfaces that 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 one's.

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

```
<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 the following:

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

The old `atlassian-jira` dependency will still work as it creates a "shaded" Maven module that combines both `jira-core` and `jira-api` dependencies. It is, however, not advised to use `atlassian-jira` as you might be bringing in dependencies that you do not need.



Before jumping into the next recipe, have a look at the JIRA API policy that you can find at `https://developer.atlassian.com/display/JIRADEV/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. There is no better way to showcase the plugin architecture, I must admit! But that does make the life of high end users who want to modify those functionalities a bit difficult.

Let me take a once simple scenario such as "I want to display the description before the 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's it! But now, the **View Issue** screen rendering is done by `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` file—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 the `atlassian-bundled-plugins.zip` file under the `JIRA_Install_Directory/atlassian-jira/WEB-INF/classes/` 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.zip` file directly! Following are the steps to do this, and these steps are applicable to modifying any Atlassian bundled plugins:

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

3. Zip all the JARs back with the same name: `atlassian-bundled-plugins.zip`.
4. Restart JIRA.

And we are done! Steps 1 to 3 can be done easily if you have a utility such as 7-Zip in Windows.

For the example we considered, all we need to do is to modify the `atlassian-plugin.xml` file inside the `jira-view-issue-plugin-5.x.jar` file. The different modules on the **View Issue** page such as the **Details** module, **Description** module, and so on are `web-panel` plugin modules, and its 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 `atlassian-bundled-plugins.zip` file during startup. Now that we have the modified ZIP file, its contents will overwrite what is in that 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 moving to JIRA 4.x+ from JIRA 3.13.x or earlier versions, one of the important differences is the introduction of v2 plugins. While designing the upgrade to JIRA 4.x+, 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 version 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 for JIRA. To check if the components are available, use dependency injection. The plugin will fail on 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. Following are the steps for conversion:

1. Add the `plugins-version="2"` attribute in the `atlassian-plugin.xml` file. 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 it is! Once added, the plugin descriptor looks like the following code snippet:

   ```
   <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` file earlier in this chapter, or simply by adding the appropriate entries into the manifest file in your JAR.



This is an optional step that 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 it 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"`.

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" />
```

5. You can also optionally add advanced Spring configurations by adding **Spring Dynamic Modules** (**Spring DM** configuration files of the format `*.xml`) under the `META-INF/spring/` directory. The Spring DM loader will then load these files. The details of how to do this 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 upon the startup of JIRA 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. JIRA then checks for the presence of an explicit `atlassian-plugin-spring.xml` file. 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, JIRA 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 Plugin Manager.

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

See also

- The *Deploying a JIRA 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 to 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`	The 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 slash (`/`).
`namePattern`	The pattern to use when loading a directory resource.
`type`	The type of the resource. Examples are: ▸ `download`: For resources such as CSS, JavaScript, images, and so on ▸ `velocity`: For Velocity files
`location`	The 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 slash (`/`) is required.

Name	Description
`property` (`key/value`)	This is used to add properties as key-value pairs to the resource. It is added as a child tag to resources. For example: `<property key="content-type" value="text/css"/>`
`param` (`name/value`)	This is used to add name/value pairs. It is added as a child tag to resources. For 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 looks 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:

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

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

2. 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
   ```

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

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

   In this case, `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/display/JIRADEV/Downloadable+Plugin+Resources`.

Adding web resources into plugins

The web resources plugin module, like the resource module we just saw, allows for the defining of 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`	This indicates whether the plugin module should be disabled by default or not. The default value `enabled`. `value="disabled"` will keep the plugin disabled at startup.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	The human-readable name of the web resource.
`system`	This indicates whether the plugin module is a system plugin module. This property is available only to non-OSGi plugins. The default value is `false`.

The following are the key elements supported:

Name	Description
`description`	The description of the module, i18n keys supported.
`resource`	All the resources to be added as web resources. (See *the Adding resources into plugins* recipe.)
`dependency`	This is used to define dependency on the other `web-resource` modules. The dependency should be defined as `pluginKey:web-resourceModuleKey`, for example: `<dependency>com.jtricks.demoplugin:demoResource</dependency>`
`context`	This defines the context where the web resource is available.
`condition`	This adds conditions to define whether the resource should be loaded or not. (See the *Adding conditions to web fragments* recipe.)
`transformation`	This makes a particular transformer available to the web resource. (See the *Developing a web resource transformer* recipe.)

We can define the `web-resource` module by populating the attributes and elements appropriately. An example would look as follows:

```
<atlassian-plugin  name="Demo Plugin"
key="com.jtricks.demoplugin" plugins-version="2">
  <plugin-info>
    <description>Demo Plugin for web-resources</description>
    <vendor name="J Tricks" url="http://www.j-tricks.com"/>
    <version>1.0</version>
  </plugin-info>

  <web-resource key="demoresource" name="Demo">
    <resource type="download" name="demoscript.js"
location="includes/js/demoscript.js" />
    <resource type="download" name="demoscript1.js"
location="includes/js/demoscript1.js" />
  </web-resource>
</atlassian-plugin>
```

How it works...

When a `web-resource` module 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, shown as follows:

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

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`: Available everywhere except administration screens
- `atl.admin`: Available on administration screens
- `atl.userprofile`: Available on user profile screens
- `atl.popup`: Available on browser pop-up windows



JIRA supports a lot more contexts, and the full list can be found at `https://developer.atlassian.com/display/JIRADEV/Web+Resource+Plugin+Module`.

You can have multiple contexts added, as follows:

```
<web-resource key="demoresource" name="Demo">
  <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>
```

You can then load these resources into the pages by adding the following line in your page:

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

Turning off batch mode

As mentioned earlier, the resources are loaded in one 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:

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

See also

- The *Adding resources into plugins* recipe

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. You can see it, modify it, break it... err? In this recipe we will modify it, because you have the license to do it!

Getting ready

Following are some of the prerequisites prior to building JIRA from the source:

- A valid JIRA license to get access to the source code.
- An environment with JDK 1.5 or higher for JIRA 4.2 and lower versions. JDK 1.6.x is required for JIRA 4.3 and above versions.
- You will need both Maven 1 and Maven 2 if you are building versions prior to JIRA 4.3. Download Maven version 1.0.x and 2.1.x from `http://maven.apache.org/`. JIRA 4.3 and above versions only need Maven 2.1.0.



You need both Maven 1 and Maven 2 for versions prior to JIRA 4.3, because Maven 1 is required to build the JIRA source and Maven 2 is required to build plugins for JIRA. JIRA has bundled plugins that need to be built along with JIRA, and so Maven 2 is also a must.

Maven 2.1.0 and above is required for the plugin development process.

How to do it...

Let us see the steps to build JIRA WAR from the source for versions prior to JIRA 4.3:

1. Configure Maven 1.0.x:
    a. Extract the Maven 1.0.x version downloaded earlier to a directory, which we will now refer to as `MAVEN_INSTALL_DIR`.
    b. Download an Atlassian patched version of Ant JAR from `http://confluence.atlassian.com/download/attachments/185729661/ant-optional-1.5.3-1.jar?version=1&modificationDate=1276644963420`.
    c. Copy it to `MAVEN_INSTALL_DIR/maven-1.0/lib`.

   d. Set the `MAVEN_HOME` environment variable, which will be `MAVEN_INSTALL_DIR/maven-1.0`.
   e. Add Maven's `bin` directory to the path variable.

2. Configure Maven 2.1.x. If you have already set up your development environment using Atlassian plugin SDK, you can skip this test as it comes along with a bundled Maven 2.x:
   a. Install Maven 2.1.x as per the instructions at `http://maven.apache.org/download.html`.
   b. Configure `settings.xml` by following the example `settings.xml` file provided by Atlassian at `https://developer.atlassian.com/display/DOCS/Example+settings.xml`.
3. Download the JIRA source ZIP file from `https://my.atlassian.com/download/source/jira`.
4. Extract the JIRA source to a directory, which we will call `JIRA_DIR`.
5. Go to the `jira` subdirectory, that is, `JIRA_DIR/jira`.
6. Run the following command to create an open WAR file:

   ```
   maven war:webapp
   ```

   If you want to create a closed WAR file, execute the following command:

   ```
   maven war:war
   ```



See `http://maven.apache.org/maven-1.x/plugins/war/goals.html` for more Maven WAR goals.

7. Confirm that the WAR file is created properly.

The following are the steps to create the WAR file on JIRA 4.3 and higher versions:

1. Configure Maven 2.1.0.
2. Download and install the required third-party libraries, as these libraries are not available in the public Maven repositories:
   a. Download the correct version of the JAR files, as mentioned in the following table:

`activation`	`javax.activation:activation`	1.0.2
`jms`	`javax.jms:jms`	1.1
`jmxri`	`com.sun.jmx:jmxri`	1.2.1
`jmxtools`	`com.sun.jdmk:jmxtools`	1.2.1
`jndi`	`jndi:jndi`	1.2.1

jta	Jta:jta	1.0.1B
mail	javax.mail:mail	1.3.2
ojdbc6	com.oracle:ojdbc6	11.2.0.2.0

b. Install them to the local Maven repository using the Maven install command, shown as follows:

```
mvninstall:install-file -DgroupId=javax.activation
-DartifactId=activation -Dversion=1.0.2 -Dpackaging=jar
-Dfile=activation-1.0.2.jar

mvninstall:install-file -DgroupId=javax.jms -DartifactId=jms
-Dversion=1.1 -Dpackaging=jar -Dfile=jms-1.1.jar

mvninstall:install-file -DgroupId=com.sun.jmx
-DartifactId=jmxri -Dversion=1.2.1 -Dpackaging=jar
-Dfile=jmxri.jar

mvninstall:install-file -DgroupId=com.sun.jdmk
-DartifactId=jmxtools -Dversion=1.2.1 -Dpackaging=jar
-Dfile=jmxtools.jar

mvninstall:install-file -DgroupId=jndi -DartifactId=jndi
-Dversion=1.2.1 -Dpackaging=jar -Dfile=jndi.jar

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

mvninstall:install-file -DgroupId=javax.mail
-DartifactId=mail -Dversion=1.3.2 -Dpackaging=jar
-Dfile=mail.jar

mvninstall:install-file -DgroupId= com.oracle:ojdbc6
-DartifactId=ojdbc6 -Dversion=11.2.0.2.0 -Dpackaging=jar
-Dfile=ojdbc6.jar
```

3. Extract the JIRA source archive to a local directory, which we will call `JIRA_DIR`.
4. Navigate to the extracted subdirectory with the name `atlassian-jira-X.Y-source` where `X.Y` is the version.
5. Run `build.bat` if on Windows, or `build.sh` if on Linux or Mac.
6. Confirm that the WAR file is created properly under the `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 ships with the `project.xml` or `pom.xml` files if in version 4.3 and above, called the Project Object Model (POM), which is used by Maven to build the WAR file.

You will be able to find the JIRA dependencies inside the `project.xml` or `pom.xml` files. Maven will first build the dependencies and then build the JIRA WAR file using them.

The only key thing here is to set up Maven correctly. There are a couple of issues normally observed while building JIRA WAR, both related to Maven. Maybe it is worth touching upon them before we move ahead:

- An error can occur while downloading dependencies due to the `java.net.ConnectException: Connection timed out: connect` exception.

  If you encounter this, make sure that the Maven proxy settings are configured properly. If they are already configured and still you are getting the error, try disabling your antivirus!

- A `failed to resolve artifact...` error can occur. Building JIRA 4.0 fails to download `javaxjms jar`. In such cases, download the JAR manually and install them into the local repository using `mvninstall`.

  ```
  mvninstall:install-file -Dfile=<path-to-file>
  -DgroupId=<group-id> -DartifactId=<artifact-id>
  -Dversion=<version> -Dpackaging=<packaging>
  ```



If using version 4.3 and above, refer to step 2 in the recipe where the relevant `mvninstall` commands are given.

Once the WAR file is created, deploy it into a supported application server, and enjoy the power of JIRA!

There's more...

Along with the JIRA source, we have access to the source code of some of the JIRA dependencies from Atlassian. You might want to build them separately if you ever want to modify their behavior.

Building JIRA dependencies

Similar to JIRA, the dependent projects also use Maven. But it uses Maven 1 in some cases and Maven 2 in others.

You can determine whether the dependency uses Maven 1 or Maven 2 by checking its POM by looking in the root directory. If the file is named `project.xml`, it uses Maven 1, and if the file is named `pom.xml`, it uses Maven2. Simple, right?

Use the following command to generate the JAR file for a dependency if it uses Maven 1:

```
maven jar
```

For dependencies with Maven 2, use the following:

```
mvn package
```

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 that are supported:

Name	Description
`key`	A unique key within the plugin. It will be used as the identifier for the plugin.
`class`	This will be `java.lang.Object` as the real logic will reside in the action, Class.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	The human-readable name of the webwork action.

The following are the key elements supported:

Name	Description
`description`	The 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`	The 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.

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

Name	Description
`view`	This delegates the user to the appropriate view based on the output of the action. This element has a `name` attribute that maps to the return value of the action class.

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="demoaction" name="JTricks Demo Action" class="java.
lang.Object">
  <actions>
    <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>
    </action>
  </actions>
</webwork1>
```

How to do it...

Let us now aim at creating a sample webwork action. For the following 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.

In order to do this, perform the following steps:

1. Add the new webwork action module into your `atlassian-plugin.xml` file. Let us say we add the same aforementioned snippet.
2. Create the `demoaction` action class under the `com.jtricks` package. The class must extend `com.atlassian.jira.action.JiraActionSupport`.

3. Identify the parameters that you need to receive from the user. Create private variables for them with the name exactly similar to that of the related HTML tag.

   In our example, we need to take a user input. Let us say it is the name of the user. The HTML code in the `input` view (in our case, `/templates/input.vm`) will be as follows:

   ```
   Name:  <input type="text" name="userName">
   ```

   So, we need to create a string variable of the name `userName` in our action class.

4. Create setter methods for the variables that are used to get values from the `input` view.

   In our example, we retrieve the `userName` name from the `input` view and process it in the action class. So we need to create a setter method for that will look like the following code snippet:

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

5. Identify the parameter that needs to be printed in the output page. In our case, we will print `modifiedName` in the output page.

6. Create getter methods for the parameters to be printed. Velocity or JSPs will invoke the getter methods to retrieve the value from the action class. For our example, we have 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 it. It is entirely up to the plugin developer to determine which methods are to be overridden. It totally depends on the logic of the plugin.

   The three main methods of interest are the following. You can however 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 our 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 the following 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 the following method to redirect the user to the input page:

  ```
  return "input";
  ```

8. Create the Velocity template for the `input` view. The `input` view, in our example, uses the template `/templates/input.vm`. Add the HTML code of the input text within a form that will invoke the action `DemoAction`:

   ```
   <h2>My Input Form</h2><br><br>
   <form method="POST" action="$requestContext.baseUrl/secure/
   DemoAction.jspa">
     Name:  <input type="text" name="userName"><br><br>
     <input type="submit">
   </form>
   ```

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

9. Create the `success` view to print the `modifiedName` in `/templates/joy.vm`:

   ```
   <h2>Output</h2><br><br>
   $modifiedName
   ```

10. Create the `error` view in `/templates/error.vm`:

    ```
    <h2>Oh No, Error!</h2><br><br>
    Please specify a user name!
    ```

11. Package the plugin and deploy it.
12. Point your browser to `${jira_base_url}/secure/DemoAction!default.jspa`. Enter a name and submit the form to see it in action!



The example given in this recipe is just for the sake of understanding how the webwork action works.

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:

1. When `${jira_base_url}/secure/DemoAction!default.jspa` is invoked, the plugin framework looks for the `DemoAction` action registered in the `atlassian-plugin.xml` file and identifies the command and view associated with it.
2. Here, the `default` command is invoked, and therefore the `doDefault` method in the action class is executed.
3. The `doDefault` method returns the `view` name as `input`.
4. The `input` view is resolved as `input.vm`, which presents the form to the user.
5. 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 `input` (current) view is shown. You can print the error messages appropriately on the `input` view, if there are any. See the `webwork1` documentation for details.

6. In `doExecute`, the input string `userName` is modified and assigned to `modifiedName`. Then `success` is returned. Here, if there are any errors (like the `userName` is empty), you can handle it to return the `error` view instead of `success`.
7. The `success` view is resolved as `joy.vm` where the `modifiedName` is printed. `$modifiedName` will invoke the `getModifiedName()` method to print the modified name.

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

In this way, we can write complex actions in JIRA that can be used to customize a lot of the 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 such as `doCommand()` in your action 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 a JIRA plugin* recipe in *Chapter 1*, *Plugin Development Process*

Extending a webwork action in JIRA

There are so many user stories for this one! How do you override some of the JIRA built-in actions? How do you do some additional stuff in the JIRA built-in action? Like doing some crazy things immediately after logging work on a ticket, or doing some innovative validations on some of those actions.

Extending the existing JIRA action is an answer to all these questions. Let us see in detail how to do that.

How to do it...

Extending a JIRA action is done with the help of the webwork plugin module. Most of it is very similar to writing new webwork actions.

Let us take the case of the `AddProject` action. What should we do if we need to extend this action? Say, to do some additional validation during the creation of a project?

Let us consider an example where we want to prevent users from creating project keys using reserved words.

The following are the steps to do this:

1. Identify the action to be overridden by looking up the `actions.xml` file under `WEB-INF/classes` in your JIRA installation directory.

   In our case, `AddProject` is the action class that does the creation of the work log on an issue:

   ```
   <action name="project.AddProject" alias="AddProject" roles-
   required="admin">
     <view name="error">
   /secure/admin/views/addproject.jsp</view>
     <view name="input">
   /secure/admin/views/addproject.jsp</view>
   </action>
   ```

   This snippet defines the action class and the related views that are using JSP files.

2. Determine whether we need to override the action or just modify the JSP files. In our example, let us do some extra validation.

3. Add the webwork plugin module in the `atlassian-plugin.xml` file:

   ```
   <webwork1 key="jtricks-add-project" name="JTricks Create Project">
     <actions>
       <action name="com.jtricks.MyAddProject" alias="AddProject"
   roles-required="admin">
       <view name="error">
   /secure/admin/views/addproject.jsp</view>
       <view name="input">
   /secure/admin/views/addproject.jsp</view>
       </action>
     </actions>
   </webwork1>
   ```

   Note the change in action class name. We can also change the JSP files if that is needed. Most importantly, the `alias` name should be exactly the same as the action `alias` name in the `actions.xml` file. In this case, the `alias` name is `AddProject`.

4. Create the action class `com.jtricks.MyAddProject`. This can be any name.

5. We can do the full action class implementation in `MyAddProject`. However, in most cases, you might just need to override some methods of the existing action class, as in our example. If so, just extend the original action class shown as follows:

```
public class MyAddProject extends AddProject{
```

6. Add the appropriate constructor to carry out a dependency injection and to call the superclass constructor. Eclipse, or the IDE you use, will usually prompt this. If you need any other Manager classes to add your extra logic, inject them as well in the constructor.
7. Override the methods you want. In our example, we need to do extra validations. Let us see how to add a validation to check if the key is a reserved word or not:

```
@Override
protected void doValidation() {
  super.doValidation();
  if (TextUtils.stringSet(getKey()) && Arrays.
asList(reservedKeys).contains(getKey())) {
    addErrorMessage(getKey() + " is a reserved word. Please use a
different key");
  }
}
```

   Here we check if the key is a valid string and is not in the `reservedKeys` array. Let us assume `reservedKeys` is a constant array, shown as follows:

```
private static final String[] reservedKeys = new String[] { "YES",
"NO", "STUPID" };
```

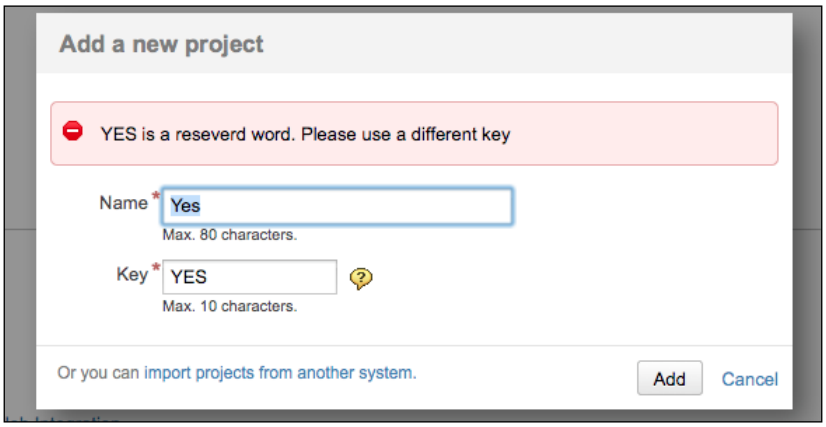
8. Package the plugin and deploy it.
9. Create a project with one of the reserved words and see how JIRA behaves!

How it works...

The key aspect of extending an existing action is to use the same alias name in your webwork plugin module. JIRA registers all the actions in an `actions.xml` file and overwrites them with the actions in plugins, if the same alias name is found.

In this case, JIRA registers the class `com.jtricks.MyAddProject` for the `AddProject` action instead of the original `project.AddProject` class.

If you try to create a project with the reserved word as the key, after the plugin is deployed, you will see an error as shown in the following screenshot:





You cannot override an action in another plugin, even if you use the same alias name. Some of the actions in JIRA might be in the bundled plugin, and so you will not be able to override them using this technique. Also, look out for any third-party plugins, which might override the same action!

See also

- The *Adding new webwork actions to JIRA* recipe

Capturing plugin installation/uninstallation events

We have seen plugins that are great in terms of functionality but come with a big list of configuration steps. Its much like a treadmill; a great asset but very hard to assemble!

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

How to do it...

It is simple. Really! Let us look at creating a custom field automatically during a plugin installation and deleting it while uninstalling. The same logic applies as for the enabling/disabling of the plugin. All you need to do is two simple steps:

- Write an event listener. We will use the `atlassian-event` library here.
- Implement the Spring interfaces `InitializingBean` and `DisposableBean` to capture the plugin lifecycle events.

Writing an event listener in JIRA is quite easy. Just perform the following steps:

1. Import the `EventPublisher` instance used to register events. You can do this by adding the following component in `atlassian-plugin.xml`:

   ```
   <component-import key="eventPublisher" interface="com.atlassian.
   event.api.EventPublisher"/>
   ```

2. Now, instantiate our listener class as follows:

   ```
   <component key="eventListener" >
     <description>A Listener for plugin lifecycle events</
   description>
   </component>
   ```

   So, what's the deal with a listener here? Basically, Atlassian plugins are implemented as Spring dynamic modules and the `atlassian-plugin.xml` file is transformed into a Spring XML bean configuration before it is loaded by JIRA. As our listener is registered as a `component` module, 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 (here `PluginListener`) 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.

   Our listener will look like the following code snippet:

   ```
   public class PluginListener implements InitializingBean,
   DisposableBean {

     @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
    }
  }
```

That's it!

How it works...

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

Let us create a text custom field during the plugin enabling/installation and associate it with a screen. We will also remove this when the plugin is disabled or uninstalled. The code is an extension to the previous one and is self-explanatory. The following is how it goes:

```
public class PluginListener implements InitializingBean,
DisposableBean {

  private static final String TEST_TEXT_CF = "Test Text CF";
  private final CustomFieldManagercustomFieldManager;
  private final FieldScreenManagerfieldScreenManager;

  public PluginListener(CustomFieldManagercustomFieldManager,
FieldScreenManagerfieldScreenManager) {
    this.customFieldManager = customFieldManager;
    this.fieldScreenManager = fieldScreenManager;
  }

  @Override
  public void destroy() throws Exception {
    //Get the already installed custom field by name
    CustomFieldcField = this.customFieldManager.
getCustomFieldObjectByName(TEST_TEXT_CF);
    //Remove if not null
    if (cField != null) {
      this.customFieldManager.removeCustomField(cField);
    }
  }

  @Override
  public void afterPropertiesSet() throws Exception {
```

```
    //Create a list of issue types for which the custom field needs to
be available
    List<GenericValue>issueTypes = new ArrayList<GenericValue>();
    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
    CustomFieldcField = 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
    FieldScreendefaultScreen = fieldScreenManager.
getFieldScreen(FieldScreen.DEFAULT_SCREEN_ID);
    if (!defaultScreen.containsField(cField.getId())) {
      FieldScreenTabfirstTab = defaultScreen.getTab(0);
      firstTab.addFieldScreenLayoutItem(cField.getId());
    }
  }
}
```

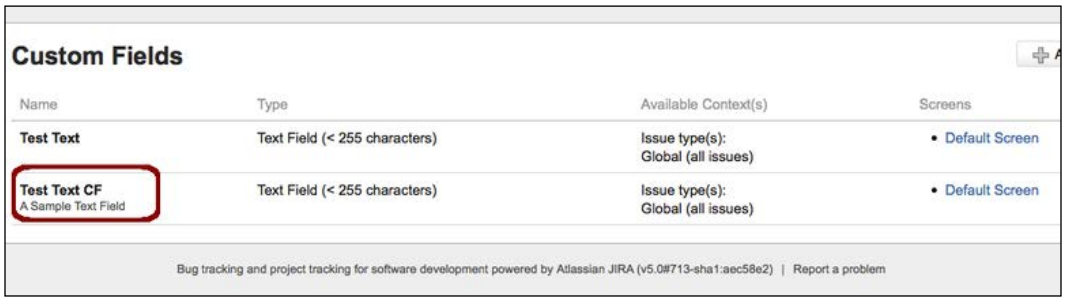
As mentioned earlier, when the plugin is installed or enabled, the `afterPropertiesSet()` method from `InitializingBean` is invoked. In our example, we create a custom field, create a context for it, and add it into a screen.

Similarly, when the plugin is uninstalled or disabled, the `destroy()` method from `DisposableBean` is invoked. In our example, we delete the custom field in that method.



It is not wise to delete a custom field during uninstallation as it will also delete all the values associated with it, and this is not an irreversible action. Somebody might accidently disable a plugin! The example is just to show the power of such a simple code.

Following is a screenshot of how the field will look once the plugin is installed:



Try uninstalling, installing it back, disabling, enabling it back, and so on to see the custom field disappearing and appearing back!

See also

- The *Deploying a JIRA 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 the change log value
- Migrating from one custom field type to another
- Making custom fields sortable
- Displaying custom fields on subtask columns
- 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 when 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.

JIRA comes with a group of predefined custom field types. These include types such as 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 supported:

Name	Description
`key`	This should be unique within the plugin
`class`	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

The following are the key elements supported:

Name	Description
`description`	Description of the custom field type.
`resource type="velocity"`	Velocity templates for the custom field views.

Name	Description
`valid-searcher`	From JIRA 5.2, you can define a searcher in the custom field definition itself. This element is most useful when we have a new custom field type that wants to make use of JIRA's core searchers. It has two attributes: ▸ `package`: The key of the Atlassian plugin where the custom field searcher resides ▸ `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 edited the issue the last time. It is simple in functionality and enough to explain the basic concepts.

The following are the major steps to complete:

1. Modify the `atlassian-plugin.xml` file to include the `customfield-type` module. Make sure the appropriate class names and views are added.

   For our example, the modified `atlassian-plugin.xml` file will look as follows:

```
<customfield-type key="readonly-user" name="Read Only User CF"
class="com.jtricks.ReadOnlyUserCF">
  <description>Read Only User CF Description</description>
  <resource type="velocity" name="view" location="templates/com/
jtricks/view-readonly-user.vm" />
  <resource type="velocity" name="column-view"
location="templates/com/jtricks/view-readonly-user.vm" />
  <resource type="velocity" name="xml" location="templates/com/
jtricks/view-readonly-user.vm" />
  <resource type="velocity" name="edit" 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.ReadOnlyUserCF`. Now, our field type is nothing but a text field in essence, so it makes sense to override the already existing `GenericTextCFType`. This replaces the `TextCFType` class we used to use in JIRA 4.

The class will look as follows:

```
public class ReadOnlyUserCF extends GenericTextCFType {

  protected ReadOnlyUserCF(CustomFieldValuePersist
er customFieldValuePersister, GenericConfigManager
genericConfigManager, JiraAuthenticationContext authContext) {
    super(customFieldValuePersister, genericConfigManager);
    this.authContext = authContext;
}
```

As you can see, the class extends the `GenericTextCFType` class. We perform a **constructor injection** to call the superclass 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, `JiraAuthenticationContext` is injected, in addition to the components required by the superclass, as we use it in our implementation. As you can see, `authContext` is an argument that is injected and is assigned to a class variable with the same name for using it later in the various methods.

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 extend the `GenericTextCFType` class, and so we only need to override the selected methods.

   The only method that we override here is the `getVelocityParameters` method, where we populate the Velocity parameters with additional values. In this case, we add the current user's name. We will later use these parameters in the Velocity context to generate the views. The same method is used in creating different views in different scenarios, that is, create, edit, and so on. The following is the code snippet:

   ```
   @Override
   public Map<String, Object> getVelocityParameters(Issue issue,
   CustomField field, FieldLayoutItem fieldLayoutItem) {
   ```

```
  Map<String, Object> params = super.getVelocityParameters(issue,
field, fieldLayoutItem);
  params.put("currentUser", authContext.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` template file is shared across `view`, `column-view`, and `xml` resources. 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
   ```

   Here we display the existing custom field value of the issue.



This code uses Velocity syntax, the details of which can be found at `http://velocity.apache.org/engine/devel/developer-guide.html`.

6. 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:

   ```
   #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)
   ```

   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.

7. Package the plugin and deploy it!



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

8. Once the plugin is installed, it is available under **Administration** | **Issue Fields** | **Custom Fields**.

How it works...

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

Create a new custom field of the type we just created 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 in the appropriate places.

In our example, whenever an issue is edited, the name of the user who edited it is stored in the custom field.



More details on adding a custom field can be found at `http://confluence.atlassian.com/display/JIRA/Adding+a+Custom+Field`.

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 such variable among them and the full list can be found at `https://developer.atlassian.com/display/JIRADEV/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* recipe in *Chapter 1*, *Plugin Development Process*
- The *Deploying a JIRA 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—that is, 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 a text searcher. A number field can be searched with a number searcher or a 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! Yeah, you know what I mean.

JIRA has defined the text searcher for its system custom fields as follows:

```
<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 are 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!

The following are the key attributes supported by the `customfield-searcher` module:

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

The following are the key elements supported by the `customfield-searcher` module:

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 ▸ `key`: The module key for the custom field type

Let's look in detail at 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.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` class 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="readonly-
user"/>
</customfield-searcher>
```

   Here we use `com.atlassian.jira.issue.customfields.searchers.TextSearcher`, which 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 our 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 by clicking on **View and Hide** (**Search** from JIRA 4.3) on 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` file, which is `com.jtricks` in our case:

   ```
   <atlassian-plugin key="com.jtricks" name="J-Tricks Customfields
   Plugin" plugins-version="2">
   ```

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

5. Package the plugin and deploy it.

If you are using JIRA 5.2+, and if you are going to use a JIRA core searcher in the read-only user custom field we created earlier, the steps we just discussed can be replaced by one simple step:

1. Identify the searcher to use and add it using the `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 code snippet:

   ```
   <customfield-type key="readonly-user" name="Read Only User CF"
   class="com.jtricks.ReadOnlyUserCF">
     <description>Read Only User CF Description</description>
     <resource type="velocity" name="view" location="templates/com/
   jtricks/view-readonly-user.vm" />
     <resource type="velocity" name="column-view"
   location="templates/com/jtricks/view-readonly-user.vm" />
     <resource type="velocity" name="edit" 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.

How it works...

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

For any existing custom fields, the searcher can be defined or modified using the edit operation. Once the searcher is changed, a reindexing 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 the `valid-searcher` element with 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.



Note that the `valid-searcher` module is available only from JIRA 5.2.

Once the searcher is defined against the custom field, you can see it appearing in the issue navigator *when the correct context is selected*. That 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 sometimes fail, 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 v2 plugins, as they were in v1 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, 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 need to be manipulated, be it on a listener, a 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. Complete the following 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:
   - Inject the Manager class in the constructor of your plugin class implementation.
   - 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 custom field name:

   ```
   CustomField customField = customFieldManager.getCustomFieldObjectB
   yName(demoFieldName);
   ```

   Or, retrieve the `customField` object using the custom field ID:

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

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.

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

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

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

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

7. Update the custom field value for the issue using `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, continue with the following steps.

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

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

9. Add the new custom field value into the input parameter's map:

```
issueInputParameters.addCustomFieldValue(customField.
getIdAsLong(), "New Value with event");
```

10. Update the issue using `IssueService`:

```
User 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!");
  }
}
```

This will throw an issue update event and all the handlers will be able to pick it up.

Note that the value passed to the custom field, via the `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`.

How it works...

The following are the activities taking place at 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 activities also take 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

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 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 there is a validation 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? We'll look at that in this recipe.

Getting ready

Create a multi-valued custom field, say X, in your JIRA instance. Add a few options onto the field X.

How to do it...

To deal with custom field options, Atlassian has written a Manager class named `OptionsManager`. Perform the following steps to configure the options 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:
   - Inject the `Manager` class in the constructor
   - Directly get an instance from the `ComponentManager` class shown as follows:

     ```
     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.getConf
   igSchemesForField(customField);
   ```

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

3. Retrieve the field configuration from the scheme identified in the previous step:

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



If you have the `issue` object, you can replace steps 2 and 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();
```

   `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 multi-level select.

   If you need to add new options to the list, it is again `OptionsManager` who comes to the rescue, as you can see in the following steps.

5. Create the new option:

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

   The first parameter is `config` that we saw earlier. The second parameter is the parent option ID, used in case of a multi-level 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.

6. Add the new option to the list of options and update!

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

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 Javadocs.

See also

- The *Writing a simple custom field* recipe

Overriding 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 multi-valued 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, which don't come from its preconfigured options? Normally, you would add these 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 where you can suppress the validation in your custom field.

Getting ready

Create your custom field as we have seen 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, not just for suppressing validation. If you are writing a custom field type from scratch, you will be implementing the `CustomFieldType` interface. You will then need to implement the preceding method and can do the same thing.

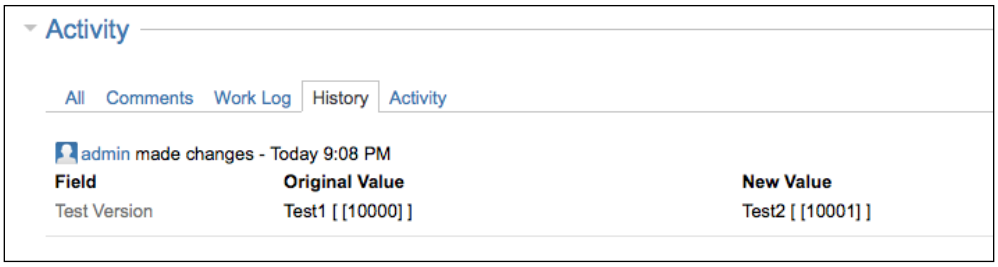
And if you are interested and have access to the JIRA source code, go and have a look at how the validation is done in some existing custom field types.

See also

- The *Writing a simple custom field* recipe

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:



In the preceding screenshot, **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 have seen in the previous screenshot, the change log values for both old and new values are displayed in the following format:

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

The string value and ID value are both 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 `change log string` and another for `change log id`—which need to be modified. The method definitions in the interface are as follows:

```
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 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, `getChangelogValue` returns an empty string and `getChangelogString` returns the truncated string. The overridden methods are shown as follows:

```
@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 for 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 made (created), and the issue 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 old and new values, 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" title="Parent" 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 fields `oldvalue` and `newvalue` are populated using the `getChangelogValue` method. Similarly, the fields `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. Even with SQL, there are some restrictions on doing this.

The first and foremost 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 `1234a` stored in one of the issues, it can't be stored as a number field as it is not a valid number. The same applies to all the field types.

Let's look at 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`. Execute the following steps:

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 and 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.

4. Commit the changes.
5. Disconnect from the database.
6. Start JIRA.
7. Reindex JIRA by going to **Administration** | **System** | **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 VARCHAR2 (255) in the `stringvalue` column. So, when you convert, we need to update the custom field ID, read the VARCHAR2 (255) value from the `stringvalue` column, and store it in the `textvalue` column as a CLOB. And 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 255 characters.

You can find the data type for the various custom fields by looking at the `getDatabaseType` method. For a text field, 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 aforementioned cases and update the `CustomFieldValue` table accordingly.

See also

- The *Retrieving custom field details from a 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 as follows:

   ```
   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, as in the following 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.

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`, and `GenericValueComparator` are some re-usable comparators that ships 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
- The *Making the custom field project importable* recipe

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 extra fields, don't we? Let us see how to do it and especially how to add custom fields.

How to do it...

In short, you need to modify the `jira.table.cols.subtasks` property. The following steps show you how to do it prior to JIRA 4.4. We can add a custom field in the example shown here.

1. Stop JIRA.
2. Navigate to the `WEB-INF/classes` folder and modify the `jira.table.cols.subtasks` property in the `jira-application.properties` file:

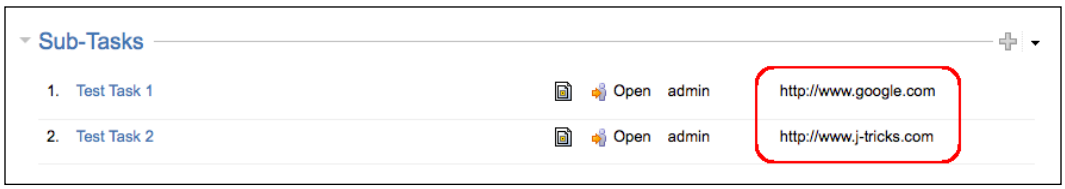
   ```
   jira.table.cols.subtasks = issuetype, status, assignee,
   customfield_10140, progress
   ```

3. Add the extra fields you want to add along with the existing fields such as status, assignee, and so on. For a custom field to be added in the columns, add `customfield_xxxxxx` where `xxxxx` is the unique numeric ID of the custom field. You can find this unique ID from the database or from the URL when you hover over any of the operations on the custom field (for example, **Edit**).
4. Start JIRA.
5. From JIRA 4.4, this property is available under **Administration** | **General Configuration** | **Advanced**. There is no need to modify the property file and to restart JIRA.

How it works...

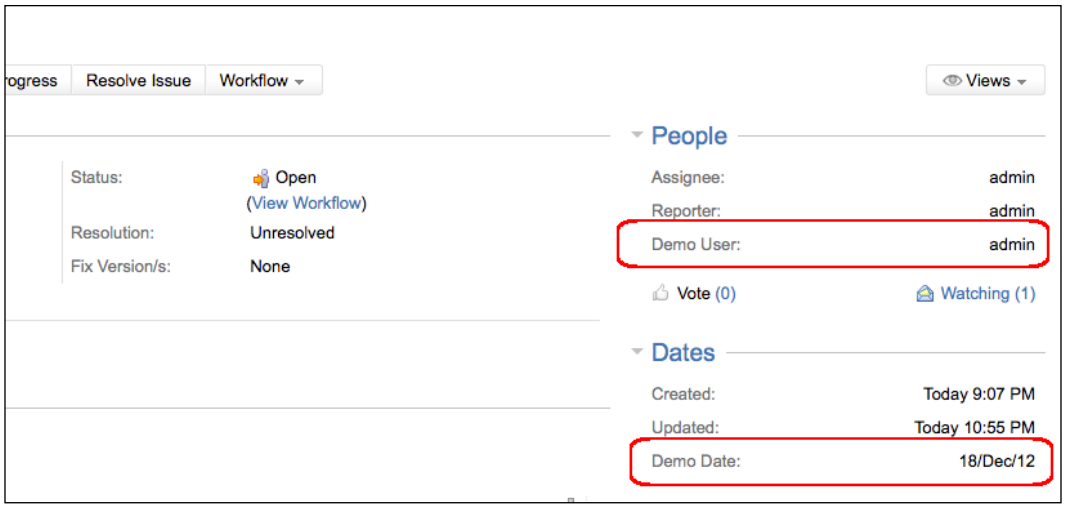
JIRA renders the subtask columns on the **View Issue** page by looking at the preceding property. While adding the standard subtask fields is useful, adding custom fields can be extremely helpful sometimes.

In our example, we have added `customfield_10140`, where `10140` is the numeric ID for the custom field. It stores the URL associated with the task, as shown in the following screenshot:



User and date fields

If you have upgraded from a pre 4.1 to a post 4.1 version, you must have noticed the new **View Issue** page. People have different opinions on the new UI usability, but one thing that gets everyone's vote is how the date and user fields are arranged in the UI. You will see that they have their own sections, as shown in the following screenshot:



So how do our fields appear in that section?

How to do it...

When you write your new date fields or user fields, all you need to do to make them appear in the correct section 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 in there automatically.

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 `ViewIssue` class, but the actual check is done in the `util` class: `com.atlassian.jira.issue.fields.util.FieldPredicates`.

See also

- The *Writing a simple custom field* recipe

Adding custom fields to notification 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 mails is a cakewalk, as you'll 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 mail when an issue is updated. Perform the following 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` file 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.getCustomFieldObjec
   t("customfield_10010")))
   ```

   In both the cases, `10010` is the numeric ID of the custom field that we discussed earlier.

If there are a lot of custom fields that need to be added to the templates, it makes sense to add a macro. 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 in that file with the required HTML tags. An example in JIRA 5.1 is as follows:

```
#macro (printcf $cfId)
#if ($issue.getCustomFieldValue($customFieldManager.
getCustomFieldObject($cfId)))
  #set ($cfVal = $issue.getCustomFieldValue($customFieldManager.
getCustomFieldObject($cfId)))
  <tr valign="top">
    <td style="color:${textColour};
font-family:${textFontFamily};font-size:${textSize};padding:0 10px
10px 0;white-space:nowrap;">
      <strong style=
"font-weight:normal;color:${textSubtleColour};">
$customFieldManager.getCustomFieldObject($cfId).getName():
</strong>
    </td>
    <td style="color:${textColour};
font-family:${textFontFamily};font-size:${textSize};padding:0 0 10px
0;width:100%;">$cfVal</td>
    </tr>
  #end
#end
```

You can add custom fields to the appropriate templates just by calling this macro, something like the following instruction:

```
#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.

How it works...

The e-mail notifications are rendered using Velocity templates. JIRA already has a lot of objects in the Velocity context, including `customFieldManager` and `issue` objects that we have just used. In this case, we use the customFieldManager object to retrieve information about the custom field and then we use the issue object to retrieve its value from the issue.



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/display/JIRADEV/Velocity+Context+for+Email+Templates`.

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 also give them some help right there on the screen, next to the field.

Let us see the various options to do this.

Getting ready

Make sure you have the custom field for which the help needs to be displayed and configured properly.

How to do it...

There are various ways to provide help. Let's just look at the most widely used ways:

1. Link to a help page. This is just common sense. Just link to a documentation about the field, hosted somewhere. We can do this easily by adding a few hyperlinks in the description of the custom field. We just need to re-use some of the JIRA styles to make sure the help appears consistent across the system.

   The hyperlinks again can be added in two ways:

   - Open the help document in a new window. Here we just link to an external page, which opens in a new window.

     ```
     My Demo Field <a class="localHelp"
     href="http://www.j-tricks.com" title="Get My Help" target="_
     blank">
     <img src="/jira/images/icons/help_blue.gif"</a>
     ```

`My Demo Field` here is the custom field name. As you can see, we use the image that ships along with JIRA for the consistency we discussed earlier. One thing we need to note here is the URL of the (`image/jira/images/icons/help_blue.gif`). In this case, we assume that `/jira` is the context path for this instance. If there is no context path, just use `/images/icons/help_blue.gif` or replace `/jira` using the context path of your instance.

Also notice the CSS class `localHelp`, which is again used for consistency across help texts. Modify the help URL and title as per your needs.

- Open the help document as a pop up and shift focus. Here we open the help document as a pop up instead of opening a new window. The focus is passed to the new window.

```
My Demo Field <a class="localHelp" onclick="var child =
window.open('http://www.google.com', 'myHelp', 'width=600,
height=500, resizable, scrollbars=yes'); child.focus();
return false;"><img src="/jira/images/icons/help_blue.gif"
title="Get My Help "></a>
```

Again, the image and CSS files remain the same. Here we can specify the width, height, and so on, of the pop-up window as shown in the preceding code. Everything else remains the same!

2. 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` element and toggle the visibility as the user clicks on the help image.

   Put the following code snippet 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
<a class="localHelp" href="#" onclick=" AJS.$('#mdfFieldHelp').
toggle();">
<img src="/jira/images/icons/help_blue.gif"></a>
<div id="mdfFieldHelp" style="display:none">
  Inline help for my demo field!
</div>
```

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 a 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 wanting to force the users to select a valid value.

How to do it...

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.

For any system custom field, you can find the associated classes and their Velocity templates from the file `system-customfieldtypes-plugin.xml` in the `WEB-INF/classes` folder.

In our case, we can find the following code snippet related to select field:

```
<customfield-type key="select" name="Select List"
  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="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"/>
</customfield-type>
```

As evident from the preceding snippet, the `edit` template for the select field is `templates/plugins/fields/edit/edit-select.vm`. This 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
  <option value="">$i18n.getText("common.words.none")</option>
#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 # symbol at the beginning of the line.
3. Restart JIRA and then 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

Making the custom field project importable

Up to JIRA 3.13, individual projects can be imported from an existing JIRA backup file. More information on this can be found at `https://confluence.atlassian.com/display/JIRA/Restoring+a+Project+from+Backup`.

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 fields project importable, or in simple terms, inform JIRA that our fields are okay to be imported!

How to do it...

All we need to do to tag our custom field project as importable is implement the following interface:

```
com.atlassian.jira.imports.project.customfield.
ProjectImportableCustomField.
```

You will have to then implement the following method:

```
ProjectCustomFieldImporter getProjectImporter();
```

Implementations already exist for the `ProjectCustomFieldImporter` class like the `SelectCustomFieldImporter` class, which we can re-use. It is in this class that we check whether the value getting imported is valid 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

Changing the size of a text area custom field

As we have discussed earlier, 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 for JIRA users to increase the size of the field either globally or for a particular custom field.

We will have a look at how to achieve this in the recipe.

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="Free Text Field (unlimited
text)"
    ........................................
    <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 requirements.

If we needed to increase the width (number of columns) to 50 and height (number of rows) to 8, the `cols` and `rows` properties would need to be updated to `50` and `8`, respectively. The template would then look like the following code snippet:

```
#customControlHeader ($action $customField.id $customField.name
$fieldLayoutItem.required $displayParameters $auiparams)
#if ($!customField.isRenderable() && $rendererDescriptor)
  ## setup some additional parameters
  $!rendererParams.put("rows", "8")
  $!rendererParams.put("cols", "50")
  $!rendererParams.put("wrap", "virtual")
  ## let the renderer display the edit component
  $rendererDescriptor.getEditVM($!value, $!issue.key,
$!fieldLayoutItem.rendererType, $!customField.id, $!customField.name,
$rendererParams, false)
#else
  <textarea class="textarea long-field" cols="50" id="$customField.
id" name="$customField.id" rows="8">$textutils.htmlEncode($!value)</
textarea>
#end
#customControlFooter ($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:

```
#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")
  $!rendererParams.put("wrap", "virtual")

  ## let the renderer display the edit component
  $rendererDescriptor.getEditVM($!value, $!issue.key,
$!fieldLayoutItem.rendererType, $!customField.id, $!customField.name,
$rendererParams, false)

#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 auto-loading* is enabled, as we discussed in the previous recipe.

See also

- The *Removing the "none" option from a select field* recipe

4
Programming Workflows

In this chapter, we will cover:

- Writing a workflow condition
- Writing a workflow validator
- Writing a workflow post function
- Editing an active workflow
- Making an issue editable/non-editable based on workflow status
- Including/excluding resolutions for specific transitions
- Permissions based on workflow status
- Internationalization in workflow transitions
- Obtaining available workflow actions programmatically
- Programmatically progressing on workflows
- Obtaining workflow history from the database
- Reordering workflow actions in JIRA
- Creating common transitions in workflows
- Creating global transitions in workflows
- Jelly escalation

Introduction

Workflows are one standout feature that help users to transform JIRA into a user-friendly system. They help users to define a lifecycle for the issues, the purpose for which they are using JIRA, and so on. As the Atlassian documentation (`http://confluence.atlassian.com/display/JIRA/Configuring+Workflow`) says:

> *A JIRA workflow is the set of steps and transitions that 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 three different plugin modules to add extra functionalities into its workflow, which we will see in detail through this chapter.

To make things easier, JIRA ships with a default workflow. We can't modify the default workflow, but can copy it into a new workflow and amend it to suit our needs. Before we go into the development aspect of a workflow, it makes sense to understand the various components of a workflow.

The two most important components of a JIRA workflow are **steps** and **transitions**. 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/display/JIRA/Defining+%27Status%27+Field+Values`), 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.

The following 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

These three 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 have a workflow screen. A transition executes in the following sequence:

1. **Conditions**: A workflow action is visible only if the condition is satisfied.
2. **Transition screens**: A workflow screen is presented to the user, if configured.
3. **Validators**: User inputs are validated.
4. **Post functions**: Post functions are executed on successful validation and status change.

JIRA ships with a few built-in conditions, validators, and post functions. There are plugins out there, which also provide a wide variety of useful workflow elements. And if you still don't find the one you are looking for, JIRA lets us write them as plugins. We will see how to do it 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 `http://confluence.atlassian.com/display/JIRA/Configuring+Workflow`.

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 installations and how there is a need to restrict the actions either to a set of people, roles, and so on, or based on some criteria (for example, the field is not empty), writing workflow conditions is inevitable.

Workflow conditions are created with the help of the `workflow-condition` module. The following are the key attributes supported in this module:

Name	Description
`key`	This should be unique within the plugin.
`class`	The 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`	The human-readable name of the workflow condition.

The following are the key elements supported in this module:

Name	Description
`description`	Description of the workflow condition.
`condition-class`	The class to determine whether the user can see the workflow transition. It must implement the `com.opensymphony.workflow.Condition` class. This element is recommended to extend the `com.atlassian.jira.workflow.condition.AbstractJiraCondition` class.
`resource type="velocity"`	Velocity templates for the workflow condition views.



Visit `https://developer.atlassian.com/display/JIRADEV/Workflow+Plugin+Modules#WorkflowPluginModules-Conditions` for more details on the `workflow-condition` module

Getting ready

As usual, create a skeleton plugin. 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 while performing the workflow action, but the inputs in defining the condition.

   The condition factory class, `RoleConditionFactory` in this case, extends the `AbstractWorkflowPluginFactory` interface, which implements the `WorkflowPluginFactory` interface. There are three abstract methods that we should implement: `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_NAME = "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:

- `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 `getProjectRoles` method merely returns all the project roles and the collection of roles is then put into the Velocity parameters with the key `ROLES`.
- `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 back 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 `descriptor` is a condition descriptor or not, and then extract the role as shown in the preceding code snippet.

- `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. Hence 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:

   - `edit-roleCondition.vm`: This template displays all project roles and highlights the already selected one in the edit mode. In the input mode, the same template is re-used, but the selected role will be null and hence a null check is done:

```
<tr bgcolor="#ffffff">
  <td align="right" valign="top" bgcolor="#fffff0">
    <span class="label">Project Role:</span>
  </td>
  <td bgcolor="#ffffff" 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>
```

- `view-roleCondition.vm`: The template 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) throws WorkflowException {
  Issue issue = getIssue(transientVars);
  User user = getCaller(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 `getCaller` method. In the preceding method, `projectRoleManager` is injected in the constructor, as we have seen earlier.

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 EasyMap.build(ROLE, extractSingleParam(conditionParams,
ROLE));
  }
  // Create a 'hard coded' parameter
    return EasyMap.build();
}
```

The method here builds a map of the key/value pair, where the 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!

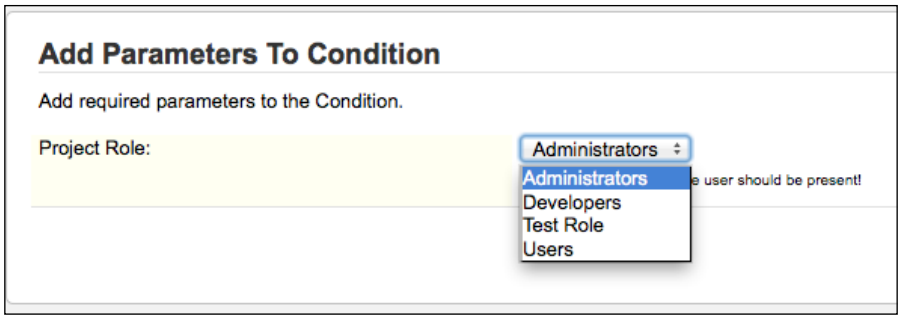
5. All that is left now is to populate the `atlassian-plugin.xml` file with the aforementioned components. We use the `workflow-condition` module and it looks like the following block of code:

```
<workflow-condition key="role-condition" name="Role Based
Condition" class="com.jtricks.RoleConditionFactory">
  <description>Role Based Workflow Condition</description>
  <condition-class>
    com.jtricks.RoleCondition
  </condition-class>
  <resource type="velocity" name="view" location="templates/com/
jtricks/view-roleCondition.vm"/>
  <resource type="velocity" name="input-parameters"
location="templates/com/jtricks/edit-roleCondition.vm"/>
  <resource type="velocity" name="edit-parameters"
location="templates/com/jtricks/edit-roleCondition.vm"/>
</workflow-condition>
```

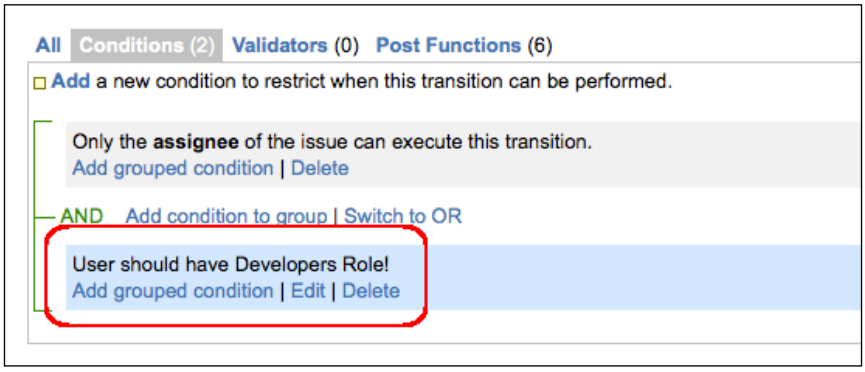
6. Package the plugin and deploy it!

How it works...

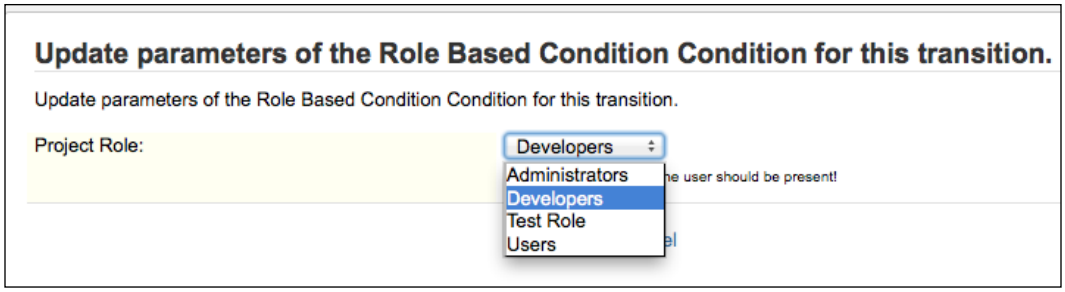
After the plugin is deployed, we need to modify the workflow to include the condition. The following screenshot is how the condition looks when it is added initially. This, as you now know, is rendered using the input template.



After the condition is added (that is, after selecting the **Developers** 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 edit template, as shown in the following screenshot:



Note that the **Developers** role is already selected.

After the workflow is configured, when the user goes to an issue, he/she will be presented with the transition only if he/she is 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 a JIRA 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 supported in this module:

Name	Description
`key`	This should be unique within the plugin.
`class`	The class to provide contexts for rendered Velocity templates. It must 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`	The human-readable name of the workflow validator.

The following are the key elements supported in this module:

Name	Description
`description`	The description of the workflow validator.
`validator-class`	The class which does the validation. It must implement `com.opensymphony.workflow.Validator`.
`resource type="velocity"`	The velocity templates for the workflow validator views.



Visit `http://confluence.atlassian.com/display/JIRADEV/Workflow+Plugin+Modules#WorkflowPluginModules-Validators` for more details on the `workflow-validator` module.

Getting ready

As usual, create a skeleton plugin. 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 using the following steps:

1. First, let's 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 input while performing the workflow action, but the inputs 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 velocityParams,
AbstractDescriptor descriptor) {
  velocityParams.put(FIELD_NAME, getFieldName(descriptor));
  velocityParams.put(FIELDS, getCFFields());
}

@Override
protected void getVelocityParamsForInput(Map velocityParams) {
  velocityParams.put(FIELDS, getCFFields());
}

@Override
protected void getVelocityParamsForView(Map 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:

- `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 `getCFFields` method returns all the custom fields and the collection of fields are then put into the Velocity parameters with the key `FIELDS`.
- `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 back 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
ValidatorDescriptor.');
  }

  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 `descriptor` is a validator descriptor or not and then extract the field as shown in the preceding code 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. Hence 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 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 field selected for the edit scenario. Let us look at the templates:

   - `edit-fieldValidator.vm`: This template 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 bgcolor="#ffffff">
  <td align="right" valign="top" bgcolor="#fffff0">
    <span class="label">Custom Fields :</span>
  </td>
  <td bgcolor="#ffffff" 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>
```

   - `view-fieldValidator.vm`: This template 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, WorkflowException {
  Issue issue = (Issue) transientVars.get("issue");
  String field = (String) args.get(FIELD_NAME);

  CustomField customField = customFieldManager.getCustomFieldObjec
tByName(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!");
    }
  }
}
```

The `transientVars` map contains the variables that are given as inputs to the workflow `initialize` or `doAction` methods. For example, `issue`, on which the validation is done, can be retrieved from the `transientVarsmap` map as shown in the preceding code snippet. The `args` map contains the properties we have set, for example, field name in our case. `customFieldManager` is injected in the constructor as usual.

4. All that is left now is to populate the `atlassian-plugin.xml` file with these components. We use the `workflow-validator` module, and it looks like the following block of code:

```
<workflow-validator key="field-validator" name="Field Validator"
class="com.jtricks.FieldValidatorFactory">
  <description>Field Not Empty Workflow Validator</description>

  <validator-class>
    com.jtricks.FieldValidator
  </validator-class>

  <resource type="velocity" name="view" location="templates/com/
jtricks/view-fieldValidator.vm"/>
  <resource type="velocity" name="input-parameters"
location="templates/com/jtricks/edit-fieldValidator.vm"/>
  <resource type="velocity" name="edit-parameters"
location="templates/com/jtricks/edit-fieldValidator.vm"/>
</workflow-validator>
```

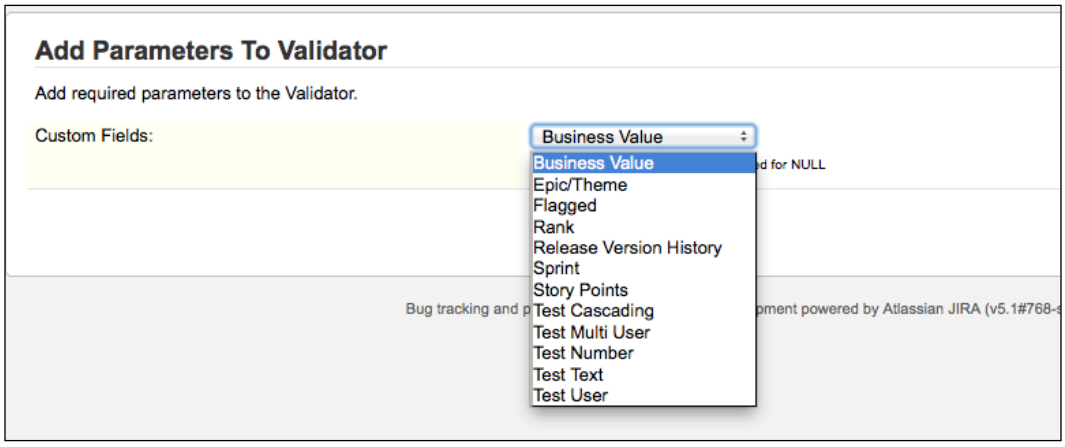
5. Package the plugin and deploy it!



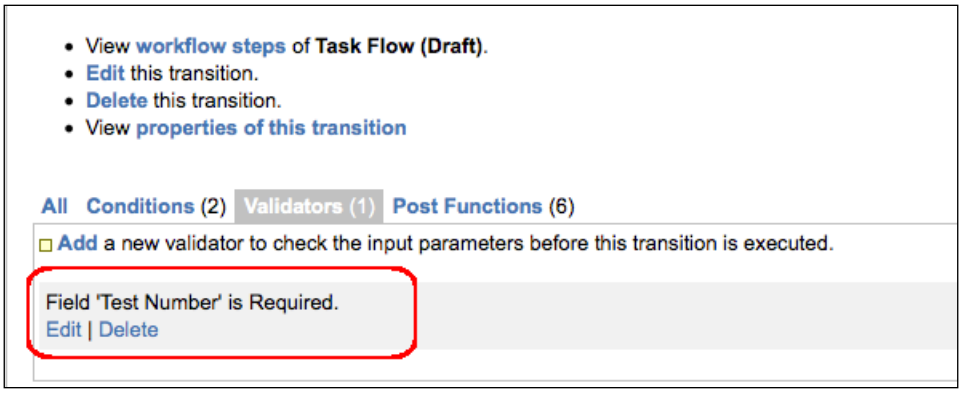
Note that we have stored the custom field name instead of the ID in the workflow, unlike what we did in the workflow condition. However, it is safe to use the ID because administrators can rename the roles, which would then need changes in the workflows.

How it works...

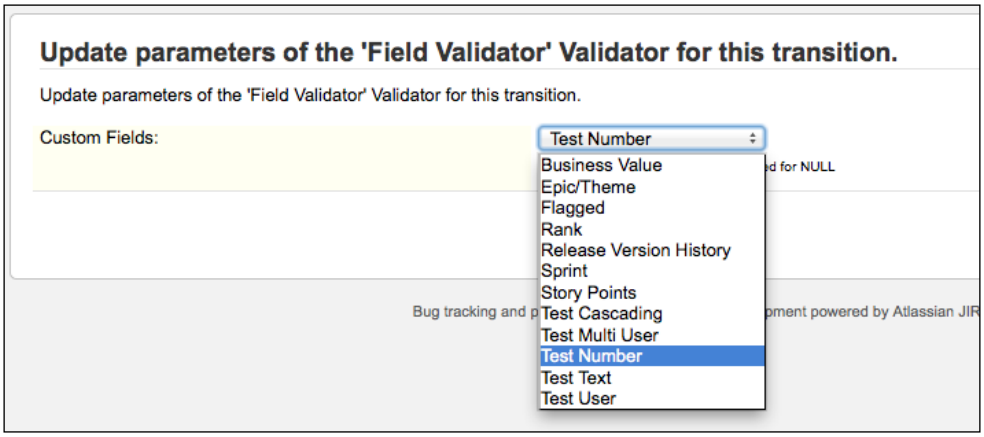
After the plugin is deployed, we need to modify the workflow to include the validator. The following screenshot is how the validator looks when it is added initially. This, as you now know, is rendered using the input template:



After the validator is added (after selecting the **Test Number** field), it is rendered using the view template and looks as follows:



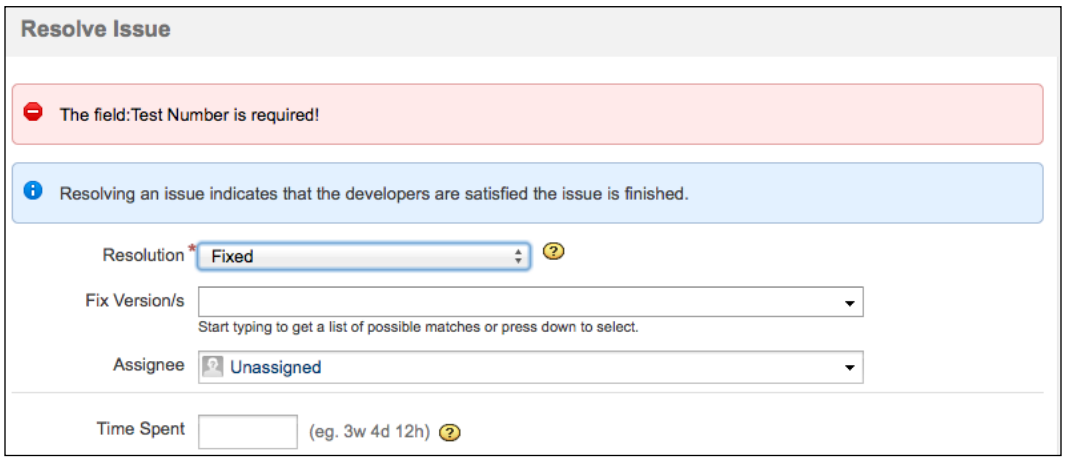
If you try to edit it, the screen will be rendered using the edit template, as shown in the following screenshot:



Note that the **Test Number** field is already selected.

After the workflow is configured, when the user goes to an issue and tries to progress it, the validator will check if the **Test Number** 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 a JIRA 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 supported in this module:

Name	Description
`key`	This should be unique within the plugin.
`class`	The 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 an input, or `com.atlassian.jira.plugin.workflow.WorkflowPluginFunctionFactory` if it needs an input.
`i18n-name-key`	The localization key for the human-readable name of the plugin module.
`name`	The human-readable name of the workflow function.

The following are the key elements supported in this module:

Name	Description
`description`	Description of the workflow function.
`function-class`	The class which does the validation. It must implement `com.opensymphony.workflow.FunctionProvider`. This element 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`): The element specifies if this function can be reordered within the list of functions associated with a transition. The position within the list determines when the function actually executes.
- `unique` (`true/false`): The element specifies if this function is unique, that is, if it is possible to add multiple instances of this post function on a single transition.
- `deletable` (`true/false`): The element specifies if this function can be removed from a transition.



See `http://confluence.atlassian.com/display/JIRADEV/Workflow+Plugin+Modules#WorkflowPluginModules-Functions` for more details on the `workflow-function` module, including the properties `addable`, `weight`, and `default`.

Getting ready

As usual, create a skeleton plugin. 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...

Let us assume we have a user custom field and we want to set the current user or a specified username 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 custom user field:

1. First, let's 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 configured username. 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 input while performing the workflow action, but the inputs in defining the post function.

   The function factory class, `SetUserCFFunctionFactory` in this case, extends the `AbstractWorkflowPluginFactory` interface and implements the `WorkflowPluginFunctionFactory` interface. Just like conditions, there are three abstract methods that we should implement: `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 velocityParams,
   AbstractDescriptor descriptor) {
     velocityParams.put(USER_NAME, getUserName(descriptor));
   ```

```
}

@Override
protected void getVelocityParamsForInput(Map velocityParams) {
  velocityParams.put(USER_NAME, CURRENT_USER); }

@Override
protected void getVelocityParamsForView(Map 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 `descriptor` is a validator descriptor or not, and then extract the field as shown in the preceding code 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. Hence 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:

   - `edit-userCFFunction.vm`: This template displays all custom fields and highlights the already selected one in the edit mode:

```
<tr bgcolor="#ffffff">
  <td align="right" valign="top" bgcolor="#fffff0">
    <span class="label">User Name :</span>
  </td>
  <td bgcolor="#ffffff" 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>
```

   - `view-userCFFunction.vm`: This template displays the selected field:

```
#if ($user)
  The 'Test User' CF will be set with value : $user!
#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);
  User 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 = userUtil.getUser(userName);
    }
  } else {
    // Set the current user here!
    user = authContext.getLoggedInUser();
  }
  // Now set the user value to the custom field
  CustomField userField = customFieldManager.getCustomFieldObjectB
yName("Test User");
  if (userField != null) {
    try {
      setUserValue(issue, user, userField);
    } catch (FieldLayoutStorageException e) {
      System.out.println("Error while setting the user Field");
    }
  }
}
```

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, User 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);
}
```

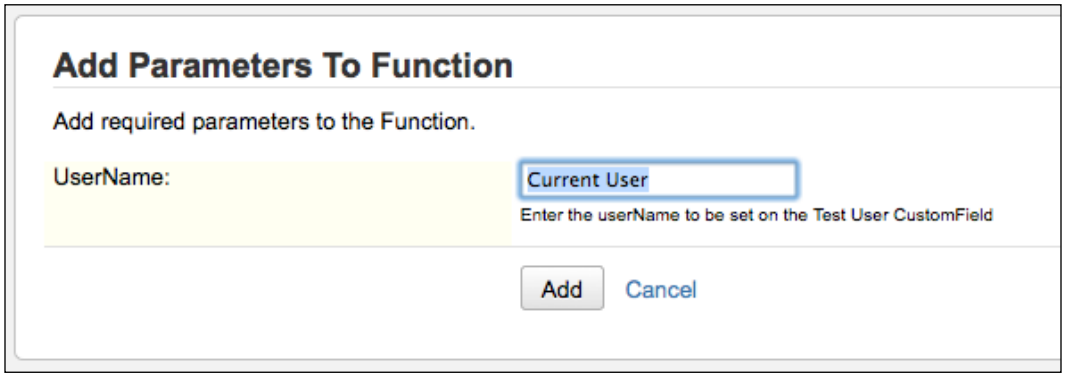
4. All that is left now is to populate the `atlassian-plugin.xml` file with these components. We use the `workflow-condition` module and it looks like the following block of code:

```
<workflow-function key="set-usercf" name="Set User CF Post
Function" class="com.jtricks.SetUserCFFunctionFactory">
  <description>Set Defined User or Current User</description>
  <function-class>
    com.jtricks.SetUserCFFunction
  </function-class>
  <orderable>true</orderable>
  <unique>false</unique>
  <deletable>true</deletable>

  <resource type="velocity" name="view" location="templates/com/
jtricks/view-userCFFunction.vm"/>
  <resource type="velocity" name="input-parameters"
location="templates/com/jtricks/edit-userCFFunction.vm"/>
  <resource type="velocity" name="edit-parameters"
location="templates/com/jtricks/edit-userCFFunction.vm"/>

</workflow-function>
```

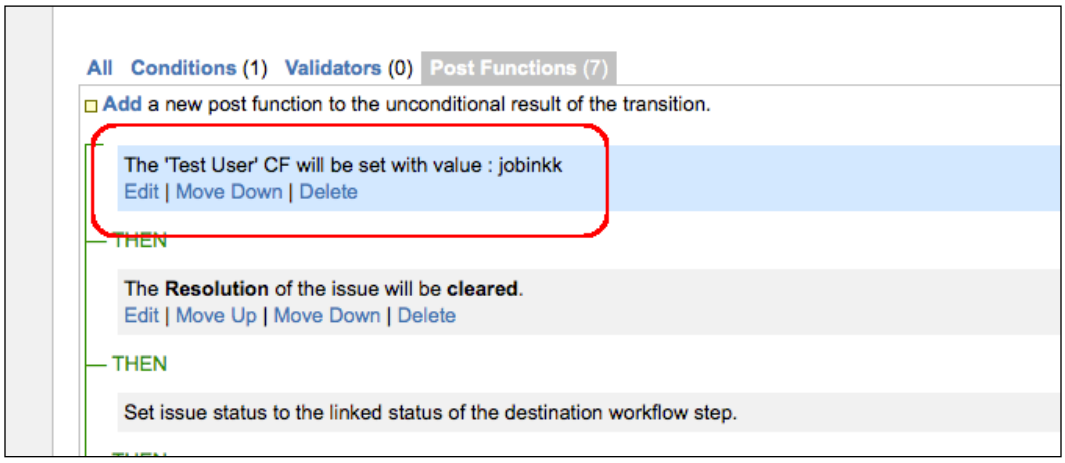
5. Package the plugin and deploy it!

How it works...

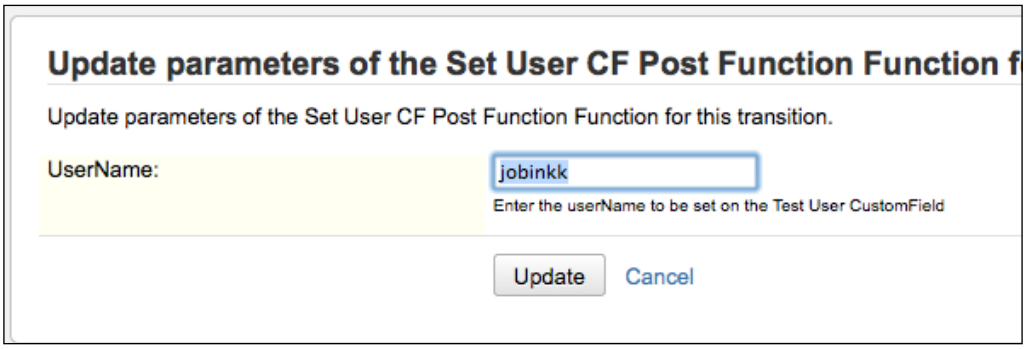
After the plugin is deployed, we need to modify the workflow to include the function. Clicking on our post function takes us to the configuration page, shown in the following screenshot. This, as you now know, is rendered using the input template:



After the function is added (after entering in the **UserName** field), it looks as follows:



If you try to edit this, the screen will be rendered using the edit template, as shown in the following screenshot:



Note that the **UserName** field is already populated.

After the workflow is configured, when the user executes the workflow action, the `Test User` custom field is set with the value **jobinkk**.

See also

- The *Creating a skeleton plugin* recipe in *Chapter 1*, *Plugin Development Process*
- The *Deploying a JIRA 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. Log in 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.
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 on 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, that is 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 limitations aforementioned, 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.

Modifying workflows in the JIRA database

The following are the steps to modify the workflows in the JIRA 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. Make 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 the 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. Reindex JIRA.

Making an issue editable/non-editable based on workflow status

We know that the edit permission on an issue is controlled through the **Edit Issue** permission. This is used within the permission's schemes tied to a project and it blocks/allows editing of the issue, irrespective of which status it is in! But often 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 will have a quick look at how to achieve this, using **workflow properties**. Workflow properties are nothing but key/value pairs defined on a workflow step or a transition.

How to do it...

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 step's **View properties** link.
5. Enter `jira.issue.editable` in 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*.

How it works...

When an issue is viewed, the edit operation is available only if you have the edit permission and the workflow manager passes the issue as editable. The workflow manager retrieves the list of properties added onto the issue's current status (that is, the step linked to the status) and checks the value of the `jira.issue.editable` property to see if it is set to `false` or not before passing the issue as editable.

The properties against an issue workflow step are retrieved as follows:

```
JiraWorkflow workflow = workflowManager.getWorkflow(issue);
StepDescriptorcurrentStep = workflow.getLinkedStep(getConstantsManag
er().getStatus(status));
Map properties = currentStep.getMetaAttributes();
```

See also

- The *Permissions based on workflow status* recipe

Including/excluding resolutions for specific transitions

If you haven't noticed already, resolutions in JIRA are global. If you have a resolution with a **Resolved** status, 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 as **Resolved** when you are rejecting an issue.

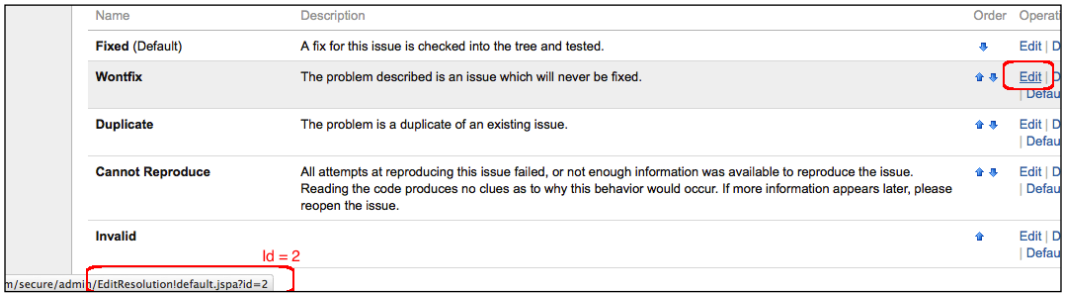
Let us see how we can pick and choose resolutions based on workflow transitions.

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** link of this transition.
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:





You can also find the resolution ID by querying the `resolutions` table in the database.

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*.

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, if `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.

Permissions based on workflow status

We have seen how to restrict editing on an issue based on the workflow status. JIRA gives us an even bigger set of options to restrict many of these operations (such as edit, comment, and so on) on the issue or its subtasks, depending on the issue status.

Let us see this in detail.

How to do it...

This is done in a similar way to making an issue editable/non-editable. Here, also, we add a property on the concerned workflow step. The following are the steps to perform this recipe:

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 step's **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]`. Let's talk a bit about the keys used in the property:
   - `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.
   - `permission`: A short name specified in the `Permissions` class (`http://docs.atlassian.com/software/jira/docs/api/latest/com/atlassian/jira/security/Permissions.html`).

   The following are the permitted values, as of JIRA 5.1: `admin`, `use`, `sysadmin`, `project`, `browse`, `create`, `edit`, `update` (same as `edit`), `scheduleissue`, `assign`, `assignable`, `attach`, `resolv`, `close`, `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`.
   - `type`: Type of permission granted/denied. The values can be `group`, `user`, `assignee`, `reporter`, `lead`, `userCF`, and `projectrole`.
   - `suffix`: An optional suffix to make the property unique when you have the same type added more than once! For example, `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 as follows:

```
jira.permission.comment.group=some-group
jira.permission.comment=denied
jira.permission.edit.group.1=some-group-one
jira.permission.edit.group.2=some-group-two
```

```
jira.permission.modifyreporter.user=username
jira.permission.delete.projectrole=10000
jira.permission.subtasks.delete.projectrole=10000
```

You can even use the value as `denied` when the type is not used. For example, `jira.permission.comment=denied` means the comment feature is disabled at this state. In the other examples, the permission is limited to the given group, user, role, and so on.

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, and cannot 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.

See also

- The *Making an issue editable/non-editable based on workflow status* recipe

Internationalization in workflow transitions

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 such as the workflow action name, button name, and so on are configured in the workflows 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's submit button name or the action name using the properties `jira.i18n.submit` or `jira.i18n.title`, respectively. The following are the steps to perform this recipe:

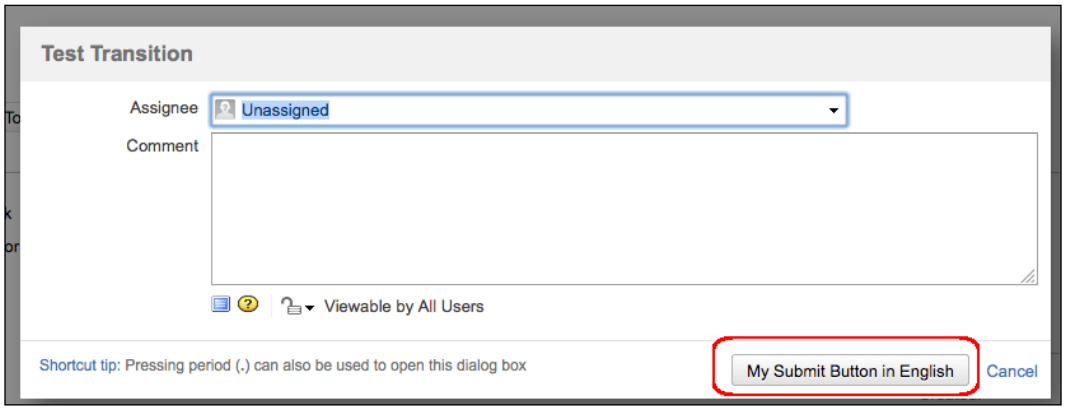
1. Open the JAR file `atlassian-jira/WEB-INF/lib/language_<language code>_<country code>.jar`. From JIRA 4.3 onwards, the JAR filename is of the form `jira-lang-<language code>_<country code>-<jira version>.jar`.
2. Edit the file `\com\atlassian\jira\web\action\JiraWebActionSupport_<language code>_<country code>.properties` inside the JAR. You can use a utility such as 7-Zip 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 transition's **View Properties** link.
9. Enter `jira.i18n.submit` or `jira.i18n.title` into the **Property Key** field, depending on whether you want to modify a submit button name or 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`.
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!

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...

Let's perform the following steps to obtain available workflow actions programmatically:

1. Retrieve the JIRA workflow object associated with the issue:

```
JiraWorkflowworkFlow = ComponentAccessor.getWorkflowManager().
getWorkflow(issue);
```

   Here, `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 action descriptors and get the details for each action, depending on the requirement! The name of the available actions 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. In this recipe we use it to retrieve the workflow object, based on the issue selected.

Once we retrieve the JIRA workflow, we get the current step using the status. As you have seen earlier in this chapter, a workflow status is linked to *one and only one* workflow step. Once we get the step, we can get lots of information from it, including the available actions from that step. Pretty good, isn't it?



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.

There's more...

Even though the recipe shows how to retrieve an action name, it is the ID that is used while programmatically progressing an issue through its workflow.

Getting the action IDs, given the 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 transit the issues through its workflows. Let us have a look at how to do this using the JIRA API.

How to do it...

Since JIRA 4.1, transitioning issues is done using `IssueService` (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/bc/issue/IssueService.html`). The following steps shows how to programmatically transit the issues workflows:

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 (the number within brackets alongside the transition name) if you know it is not going to change, or retrieve it 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);
```

   The arguments of `validateTransition` are described as follows:

   - `user`: The current user or the user who will be performing the transition
   - `12345L`: The issue ID
   - `10000L`: The action ID
   - `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()){
  IssueResulttransitionResult = 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! That will transit the issue to the appropriate state.

Prior to `IssueService`, the transition was done using `WorkflowTransitionUtil` (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/workflow/WorkflowTransitionUtil.html`). It is still supported, but `IssueService` is recommended.

The following is how the transitioning is done using `WorkflowTransitionUtil`:

1. Get the `WorkflowTransitionUtil` object:

```
WorkflowTransitionUtil workflowTransitionUtil =
(WorkflowTransitionUtil) JiraUtils .loadComponent(WorkflowTransiti
onUtilImpl.class);
```

2. Create a map of parameters that need to be updated on the issue:

```
Map paramMap = EasyMap.build();
paramMap.put(IssueFieldConstants.RESOLUTION, "10000");
paramMap.put(IssueFieldConstants.COMMENT, comment);
```

3. Populate the `workflowTransitionUtil` object with the details:

```
workflowTransitionUtil.setParams(paramMap);
workflowTransitionUtil.setIssue(12345L);
workflowTransitionUtil.setUsername(user);
workflowTransitionUtil.setAction(10000L);
```

4. Validate the transition:

```
ErrorCollection c1 = workflowTransitionUtil.validate();
```

5. If there is no error, progress with the workflow:

```
ErrorCollection c2 = workflowTransitionUtil.progress();
```

And we should have the issue in its new status! Check the error collection to handle errors, if any.

How it works...

Once the action ID is correct and the parameters are validated properly, `IssueService` or `WorkflowTransitionUtil` 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 writing the SQL query to extract workflow changes for a given issue.

Getting ready

Make sure you have a SQL client installed and configured that will help you to connect to the JIRA database.

How to do it...

Let's perform the following steps to obtain the workflow history from the JIRA database:

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 in hand, you can get it from the database using the issue key, shown as follows:

   ```
   select id from jiraissue where pkey = "JIRA-123"
   ```

   Here, `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 in to a single change group 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'
   ```

   Here, `10010` is the issue ID, 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 group ID values `10000`, `10010`, and so on are IDs extracted in the previous step. Also, `oldstring` is the original value on the issue and `newstring` is the updated value.

5. 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');
   ```

6. Another example to extract the details along with the created date is to use an 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!

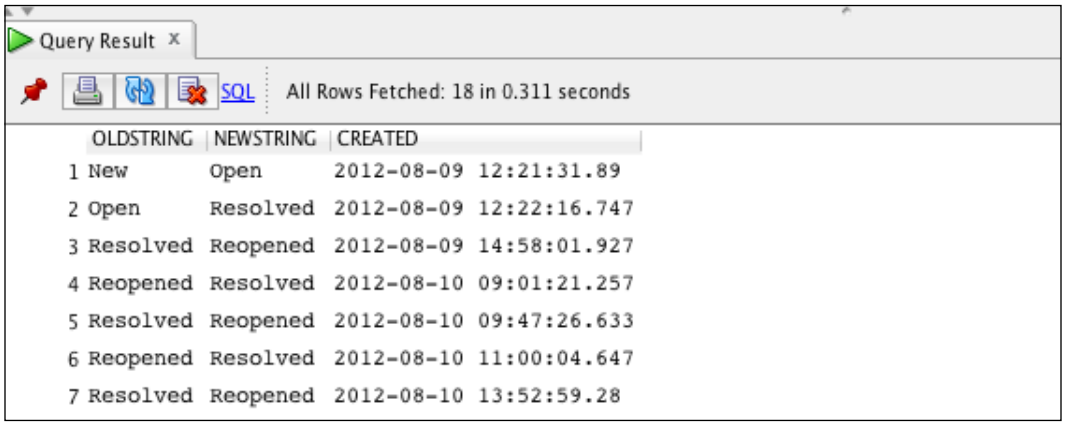
How it works...

As mentioned earlier, 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 hence 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:



	OLDSTRING	NEWSTRING	CREATED
1	New	Open	2012-08-09 12:21:31.89
2	Open	Resolved	2012-08-09 12:22:16.747
3	Resolved	Reopened	2012-08-09 14:58:01.927
4	Reopened	Resolved	2012-08-10 09:01:21.257
5	Resolved	Reopened	2012-08-10 09:47:26.633
6	Reopened	Resolved	2012-08-10 11:00:04.647
7	Resolved	Reopened	2012-08-10 13:52:59.28

See also

- The *Retrieving workflow details from a database* recipe in *Chapter 10*, *Dealing with the JIRA Database*

Reordering workflow actions in JIRA

On a JIRA workflow, the available actions that appear in the **View Issue** page are normally ordered in the sequence those transitions were created. 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 can modify the ordering using this property instead of tampering with the workflow.

How to do it...

Let's perform the following steps to reorder workflow actions in JIRA:

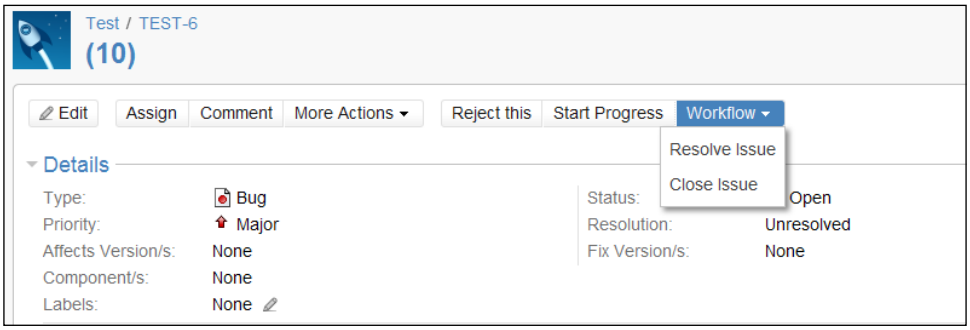
1. Log in as a JIRA administrator.
2. Navigate to **Administration** | **Issues** | **Workflows**.
3. Create a draft of the workflow if it is active. Navigate to the transition, which needs to be modified.
4. Click on the transition's **View Properties** link.
5. Enter **opsbar-sequence** into the **Property 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*.

How it works...

Let us consider the following example where the **Reject this** workflow action appears first:



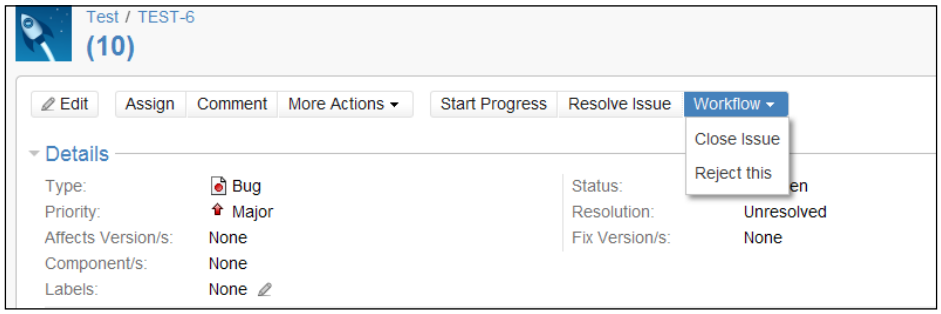
Normally, people would want to see this as the last option because it is most likely the least used operation.

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
Start Progress	10
Resolve Issue	20
Close Issue	30
Reject this	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 them as 10, 20, and so on, so that we can 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:





Remember, the order of the workflow actions is 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 those 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 end up modifying it in 10 places. This is where OSWorkflow's common actions come to our rescue.



A little bit on the theory can be read at `https://confluence.atlassian.com/display/JIRA/Configuring+Workflow#ConfiguringWorkflow-Workingwith%27commontransitions%27`.

JIRA already makes use of common actions in its default workflow. We can't modify the default workflow, but we can, if we make a copy of it and rename the **Resolve Issue** transition to **Resolve Issue – Modified**. It appears as shown in the following screenshot:

Diagram | Text | Download as XML

Step Name (id)	Linked Status	Transitions (id)	Operations
Open (1)	Open	*Start Progress* (4) >> In Progress *Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed	Add Transition | Delete Transitions | Edit | View Properties
In Progress (3)	In Progress	Stop Progress (301) >> Open *Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed	Add Transition | Delete Transitions | Edit | View Properties
Resolved (4)	Resolved	Close Issue (701) >> Closed *Reopen Issue* (3) >> Reopened	Add Transition | Delete Transitions | Edit | View Properties
Reopened (5)	Reopened	*Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed *Start Progress* (4) >> In Progress	Add Transition | Delete Transitions | Edit | View Properties



Note that the transition is renamed at all the three places where it appears!

In this recipe, let us look at adding a new common transition.

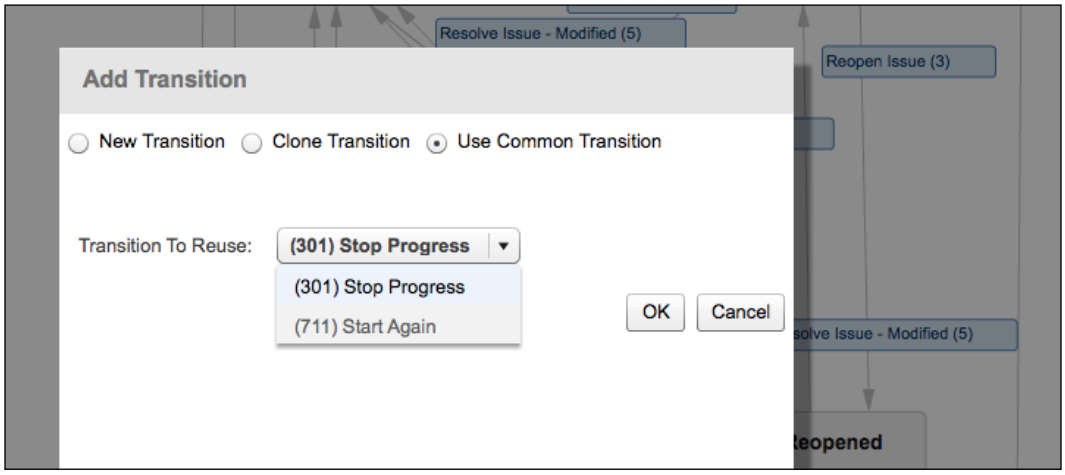
How to do it...

There are two ways of adding common transitions:

- Edit the workflow using the workflow designer in diagram mode
- Create the workflow using XML

Let's see the first way, which is pretty easy! The following are the steps to edit the workflow using the workflow designer:

1. Create a common transition from any step to another step as you normally do.
2. Go to the next step where you want to make use of the already existing transition
3. Select source and target statuses and you will get the **Add Transition** screen.
4. Choose the **Use Common Transition** option and select the previously created transition, as shown in the following screenshot:



5. Click on **OK**. The common transition is created! You can use the same transition again by following steps 3 and 4.

If you need to take the XML route for to some reason, the following are the steps:

1. To make things easier, export the existing workflow that needs to be modified into an XML. You can do this using the **XML** link on the **View Workflows** page. You could create a workflow XML from scratch, but that needs a lot of effort and knowledge of OSWorkflow. In this case, we export the standard JIRA workflow.
2. Identify the `common-actions` section in the workflow XML. It comes at the start, immediately after `initial-actions`.
3. Add our new `common-action` section. There are a few things that we need to note here. The action ID should be a unique ID within the XML. You will find examples of all this in the standard workflow XML, or you can read more about them in the OSWorkflow's documentation.

   The following is how a simple action looks:

```
<action id="711" name="Start Again">
  <meta name="jira.description"></meta>
  <results>
    <unconditional-result old-status="Not Done" status="Done"
step="1">
      <post-functions>
        <function type="class">
          <arg name="class.name">
com.atlassian.jira.workflow.function.issue.
UpdateIssueStatusFunction</arg>
        </function>
        <function type="class">
          <arg name="class.name">
com.atlassian.jira.workflow.function.misc.
CreateCommentFunction</arg>
        </function>
        <function type="class">
          <arg name="class.name">
com.atlassian.jira.workflow.function.issue.
GenerateChangeHistoryFunction</arg>
        </function>
        <function type="class">
          <arg name="class.name">
com.atlassian.jira.workflow.function.issue.
IssueReindexFunction</arg>
        </function>
        <function type="class">
          <arg name="class.name">
com.atlassian.jira.workflow.function.event.
FireIssueEventFunction</arg>
```

```
            <arg name="eventTypeId">13</arg>
          </function>
        </post-functions>
      </unconditional-result>
    </results>
</action>
```

4. Make sure you modify the name, description, status, step, event type ID, and the post functions. Here we used `Not Done` as the value of `old-status` as it is used in the other common actions in the JIRA standard workflow. You can also add new meta attributes, conditions, validators, and so on, but it is probably a good idea to start simple and modify everything else in the JIRA UI once it is imported to JIRA.

5. Include the common action in the other steps, wherever required:

```
<step id="1" name="Open">
  <meta name="jira.status.id">1</meta>
  <actions>
    <common-action id=".." />
    ..................
    <common-action id="711" />
    <action id=" .....
    ....................
    </action>
  </actions>
</step>
```

   Note that the ID here should be the action ID of the `common-action` section we added in the earlier steps. Also, the common actions should appear before the `action` elements in the *step* in order to comply with the OSWorkflow syntax.

6. Import the modified XML as a workflow to JIRA. You can do this using the **Import a Workflow from XML** link. The workflow is now ready to use.

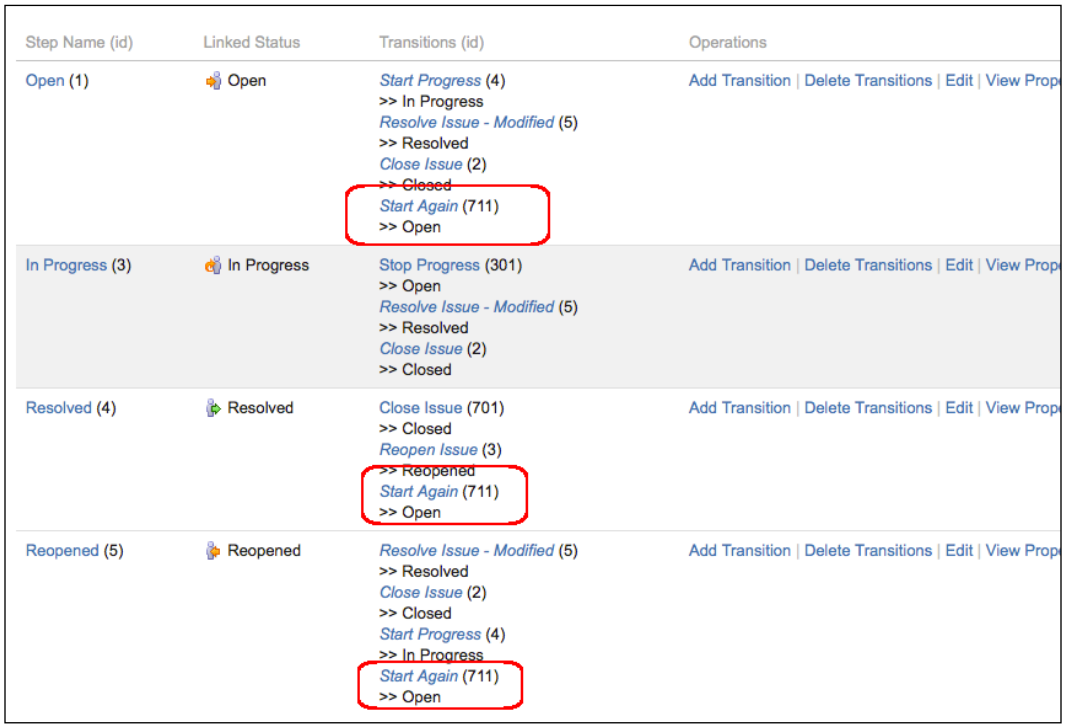


Check out `https://confluence.atlassian.com/display/JIRA/Configuring+Workflow#ConfiguringWorkflow-UsingXMLtocreateaworkflow` for more details on *Configuring Workflow*.

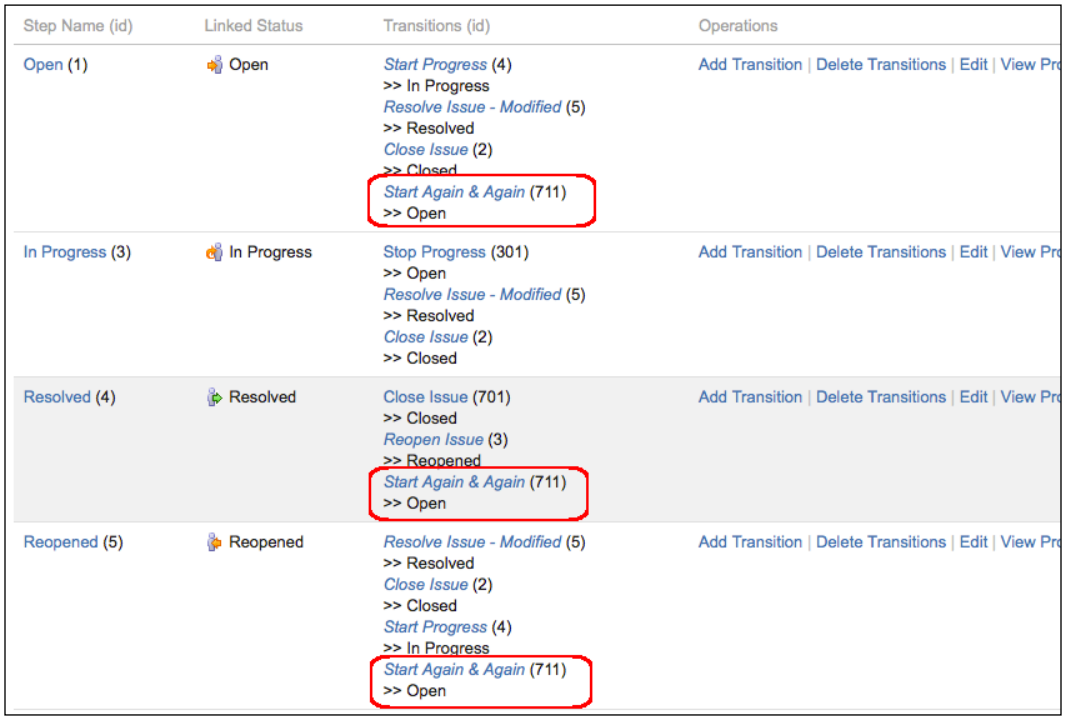
How it works...

JIRA workflows fundamentally use OpenSymphony's OSWorkflow, as we saw in *Chapter 2*, *Understanding the Plugin Framework*. OSWorkflow gives us the flexibility to add common actions by modifying the workflow XML. We have used this feature by modifying the existing workflow XML and importing it back in to JIRA.

The following screenshot is how the updated workflow looks:



Note that the new transition **Start Again** is added to all the steps other than the last one. Suppose we want to modify the name to **Start Again & Again**, it can be done just by editing one of these transitions. The modified workflow looks as follows:



Step Name (id)	Linked Status	Transitions (id)	Operations
Open (1)	Open	*Start Progress* (4) >> In Progress *Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed *Start Again & Again* (711) >> Open	Add Transition | Delete Transitions | Edit | View Pr
In Progress (3)	In Progress	Stop Progress (301) >> Open *Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed	Add Transition | Delete Transitions | Edit | View Pr
Resolved (4)	Resolved	Close Issue (701) >> Closed *Reopen Issue* (3) >> Reopened *Start Again & Again* (711) >> Open	Add Transition | Delete Transitions | Edit | View Pr
Reopened (5)	Reopened	*Resolve Issue - Modified* (5) >> Resolved *Close Issue* (2) >> Closed *Start Progress* (4) >> In Progress *Start Again & Again* (711) >> Open	Add Transition | Delete Transitions | Edit | View Pr

We can similarly modify any attributes on the transition, and it will be reflected in all the places where the transition is used.

Creating global transitions in workflows

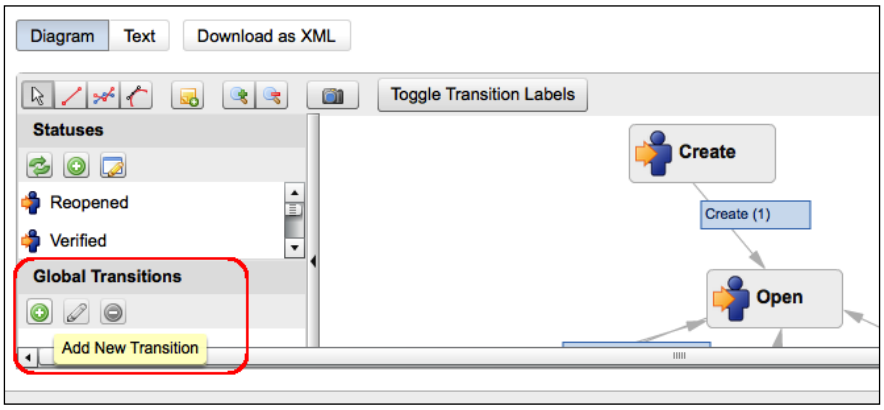
We have seen how to configure common transitions in the previous recipe. Global transitions in workflows are a similar useful feature. A **global transition** is a transition in which the destination step has all other steps in the workflow as incoming steps. That is, this transition will act as a transition 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 diagram mode.

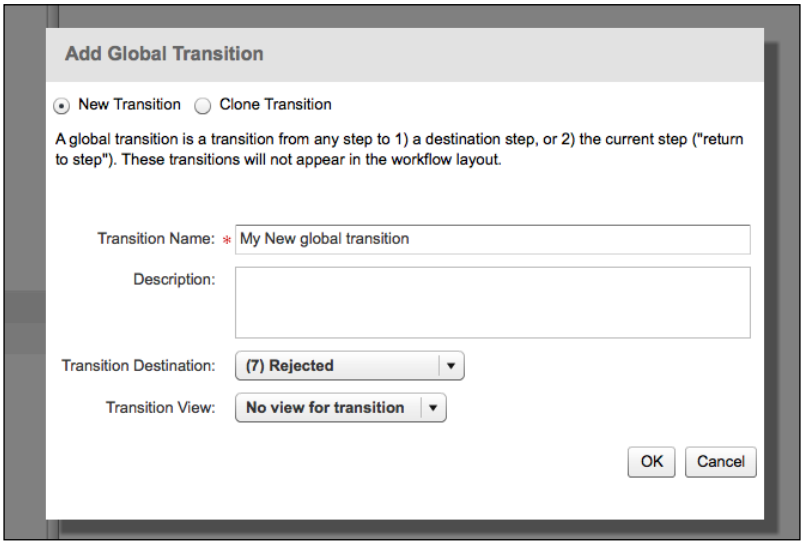
How to do it...

The following are the steps to create a global transition:

1. Go to the workflow editor in diagram mode.
2. Click on the **+** icon on the **Global Transitions** panel, as shown in the following screenshot:



3. Provide a transition name and select a destination step (to which the transition will be added from all other steps in the workflow). Also select a transition screen if needed. The following is an example screenshot:



4. Click on **OK**. The global transition is created!

How it works...

The following screenshot shows how the updated workflow looks:



This is very useful in cases where you need a single transition from all steps to a closing step, for example, if your workflow needs a **Reject** action from every single step.

Jelly escalation

Before winding up this chapter, let us have a quick look at how we can use one of the useful features of JIRA to escalate inactive issues by transitioning them to a predefined workflow status.

Jelly Service is a built-in service in JIRA which we can use to run useful Jelly scripts at regular intervals. Atlassian explains in its documentation (`https://confluence.atlassian.com/display/JIRA/Jelly+Escalation`) about running a Jelly script to move issues that were not updated in the last 7 days to an inactive status. Note that it is 7 calendar days and *not* business days. JIRA has no concept of business days, out of the box.

Let us have a look at this recipe on how to modify the script and transition issues into different workflow statuses.

Getting ready

Make sure Jelly is turned on in your JIRA instance. It is disabled by default due to security concerns. You can turn it on by setting the `jira.jelly.on` property to `true`.

You can set the property by adding `-Djira.jelly.on=true` into the `JAVA_OPTS` variable. Adding this variable depends on the server and operating system.

For example, the property can be set on the Tomcat server in Windows by adding it into `setenv.bat` under the `/bin` folder.

How to do it...

The following are the steps to close issues that have been inactive for the last 15 days:

1. Create a filter that displays issues that have not been updated for the last 15 days. You can do this by executing the following JQL query:

   ```
   updated<= -15d
   ```

   Save the filter with a name and make a note of the filter ID.

   You can find the filter ID from the browser URL while viewing a filter. The URL will be something like `http://localhost:8080/secure/IssueNavigator.jspa?mode=hide&requestId=10000`, where `filterId` is 10000. Alternatively, you can get the same from the `searchrequest` table in the JIRA database.

2. Modify the Jelly script provided by Atlassian in order to include the new filter ID, workflow step name, username, and password. Also modify the comment accordingly.

   The following is the modified script:

   ```
   <JiraJelly xmlns:jira="jelly:com.atlassian.jira.jelly.
   enterprise.JiraTagLib" xmlns:core="jelly:core"
   xmlns:log="jelly:log" >
     <jira:Login username="jobinkk" password="[password here]"><]">
       <log:warn>Running Inactivate issues service</log:warn>
       <!-- Properties for the script -->
       <core:setvar="comment">Closing out this issue since it has
   been inactive for 15 days!</core:set>
       <core:setvar="workflowStep" value="Close Issue" />
       <core:setvar="workflowUser" value="jobinkk" />
       <core:setvar="filter15Days" value="10000" />

       <!-- Run the SearchRequestFilter -->
   ```

```
    <jira:RunSearchRequestfilterid="${filter15Days}" var="issues"
/>


    <core:forEachvar="issue" items="${issues}">
      <log:warn>Inactivating issue ${issue.key}</log:warn>
      <jira:TransitionWorkflow key="${issue.key}"
user="${workflowUser}" workflowAction="${workflowStep}"
comment="${comment}"/>
    </core:forEach>
  </jira:Login>
</JiraJelly>
```

3. Save the script and put it under a location in the server where JIRA is running.
4. Go to **Administration** | **System** | **Advanced** | **Services** in JIRA.
5. Add the escalation service:
   - **Name**: `Escalation Task`
   - **Class**: Click on **Built-in services** and select **Run Jelly Script**. The class will be selected as `com.atlassian.jira.jelly.service.JellyService`.
   - **Delay**: Select a suitable delay in minutes.
   - Click on **Add Service**.
6. On the **Add Service** page, enter the following details:
   - **Input File**: Path to the script file we saved in the server
   - **Output File**: Path to an output log file
   - **Delay**: Modify if required

   The script will now run the configured delay.

How it works...

JIRA has its own API for Jelly scripting. As you can see from the script, the following are the steps executed:

1. The script runs the search request on the filter we saved in step 1 using the `RunSearchRequest` method. It then stores the retrieved results in variable issues.
2. The script then iterates on the issues, and transitions each of them in the workflow using the `TransitionWorkflow` method. It makes use of the key from the issue, the workflow user we configured, and the workflow action. It also adds the comment we entered in the script.



Note that the workflow action should be available on the issue from its current status. If it is not, the transition will not work. For example, a closed workflow action will throw an error if attempted on an issue that is already closed.

We can modify the script to transition the issue to any workflow status based on any filter criteria.



A lot of other useful things about Jelly scripting can be found at `http://confluence.atlassian.com/display/JIRA/Jelly+Tags`.

5
Gadgets and Reporting in JIRA

In this chapter, we will cover:

- Writing a JIRA report
- Reports in an Excel format
- Data validation in JIRA reports
- Restricting access to reports
- Object-configurable parameters for reports
- Creating a pie chart in JIRA
- Writing JIRA gadgets
- Invoking REST services from gadgets
- Configuring user preferences in gadgets
- Accessing gadgets outside of JIRA

Introduction

Reporting support in an application such as JIRA is inevitable! With so much data spanning across different projects, issues, and with a lot 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**: From 4.x onward, the JIRA dashboard was revamped to include gadgets, replacing the legacy portlets. These gadgets are mini applications built using HTML and JavaScript that can run on any OpenSocial gadget container. They communicate with JIRA using the 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 on to other containers such as 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 `http://confluence.atlassian.com/display/JIRA/Generating+Reports`.

In addition to the gadgets and reports that JIRA provides, there are a lot 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 see more details on writing JIRA reports and gadgets, converting legacy portlets into gadgets, and so on.

In addition to that, we will also have a quick look at the **JIRA Query Language** (**JQL**), which provides advanced searching capabilities within the issue navigator. JQL helps us to generate a lot of the reports in the issue navigator and export them into convenient views such as Excel, Word, and so on.

Writing a JIRA report

As previously mentioned, a JIRA report can display statistical information based on all the 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`	This is a 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`	The human-readable name of the report. Appears in the plugins page. Default is the plugin key.

Elements:

Name	Description
`description`	The description of the report.
`label`	The user visible name of the report.
`resource type="velocity"`	The velocity templates for the report views.
`resource type="18n"`	The JAVA properties file for the i18n localization
`properties`	The reports-configurable parameters that are 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 a little business logic in it. The following example has been chosen to display the key and summary of all 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 key="allissues-report" name="All Issues Report"
   class="com.jtricks.AllIssuesReport">
     <description key="report.allissues.description">This
   report shows details of all issues a specific project.
   </description>
      <!-- the label of this report, which the user will use to
   select it -->
      <label key="report.allissues.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. The keys entered, such as `report.allissues.label`, will be mapped to a key within the property file:

   ```
   <!-- this is a .properties file containing the i18n keys for
   this report -->
   <resource type="i18n" name="i18n" location="com.jtricks.
   allissues.AllIssuesReport" />
   ```

   Here, the `AllIssuesReport.properties` file will be present in the `com.jtricks.allissues` 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:

```
<!-- the 'view' template is used to render the HTML result
-->
<resource type="velocity" name="view" location="templates/
allissues/allissues-report.vm" />
```

Here we have defined the velocity templates that will be used to render the HTML and the Excel views 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>Project</name>
    <description>report.allissues.project.description
</description>
     <!-- valid types are string, text, long, select, date
etc-->
     <type>select</type>
     <!-- the values generator is a class which will
generate values for this select list -->
     <values class="com.jtricks.ProjectValuesGenerator"/>
  </property>
</properties>
```

This is a list of properties that will be rendered appropriately on the report input 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 upcoming recipes.

e. 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 key="allissues-report" name="All Issues Report"
class="com.jtricks.AllIssuesReport">
   <description key="report.allissues.description">This
report shows details of all issues a specific project.
</description>
  <label key="report.allissues.label" />
  <resource type="velocity" name="view" location="templates/
allissues/allissues-report.vm" />
  <resource type="i18n" name="i18n" location="com.jtricks.
allissues.AllIssuesReport" />
```

```
    <properties>
      <property>
          <key>projectId</key>
          <name>Project</name>
          <description>report.allissues.project.description
</description>
          <type>select</type>
          <values class="com.jtricks.ProjectValuesGenerator"/>
      </property>
      </properties>
</report>
```

2. Create the `i18n` resource properties file. As mentioned, it will be created in the `com.jtricks.allissues` package under the `resources` folder. The name of the file will be `AllIssuesReport.properties`. We have used three properties so far, which will be populated with the appropriate values:

   ```
   report.allissues.description=Displays all Issues from a project
   report.allissues.label=All Issues report
   report.allissues.project.description=Project to be used as the
   basis of the report
   ```

   You can create `AllIssuesReport.proprties_{language}_{countrycode}` to support other locales.

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{
     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 therefore it 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 as the property in the `atlassian-plugin.xml` file:

```
  final String projectid = (String) reqParams.get("projectId");
  final Long pid = new Long(projectid);
```

   We 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 = ComponentAccessor.
getComponent(SearchService.class).search(ComponentAccessor.
getJiraAuthenticationContext().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:

```
public class AllIssuesReport extends AbstractReport {

  public String generateReportHtml(ProjectActionSupport action,
Map reqParams) throws Exception {
    return descriptor.getHtml("view", getVelocityParams(action,
reqParams));
```

```
  }

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 = ComponentAccessor.
getComponent(SearchService.class).search(ComponentAccessor.
getJiraAuthenticationContext().getLoggedInUser(), query,
PagerFilter.getUnlimitedFilter());
    return results.getIssues();
  }
}
```

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:

```
<table id="allissues-report-table" border="0" cellpadding="3"
cellspacing="1" width="100%">
   <tr class="rowNormal">
   <th width="30%">Key</th>
   <th width="70%">Summary</th>
   </tr>
   #foreach ($issue in $issues)
       <tr class="rowNormal">
         <td width="30%"><a href="/browse/$issue.key">$issue.key</
a></td>
         <td width="70%">$issue.summary</td>
       </tr>
   #end
</table>
```

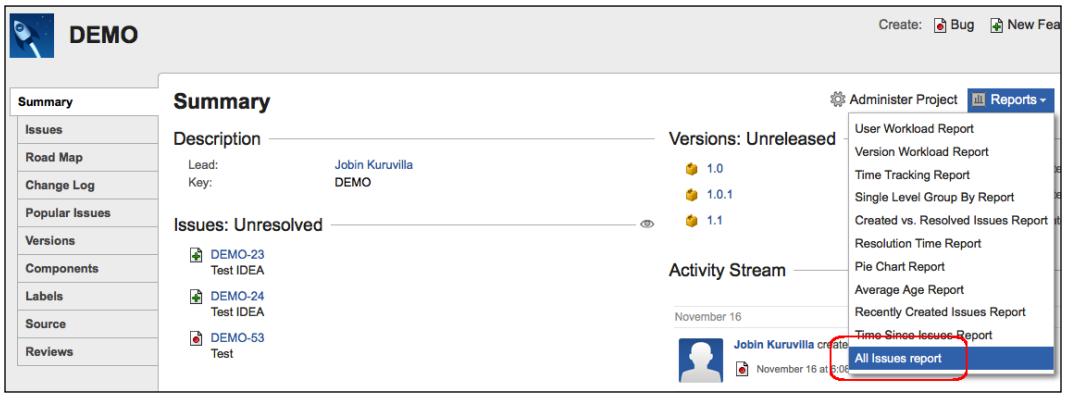
6. With that, our report is ready. Package the plugin and deploy it. We will see more on creating Excel reports, validation within reports, and so on in the coming recipes.

How it works...

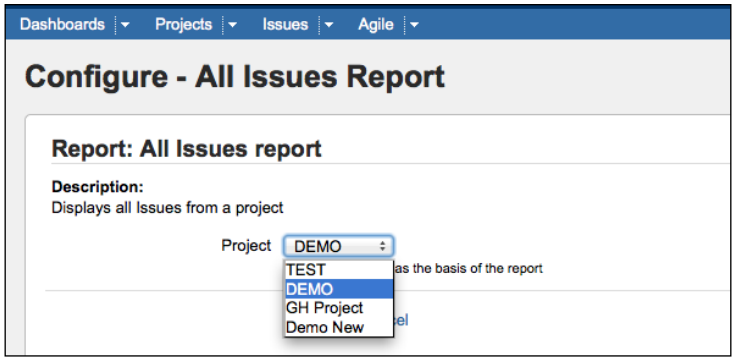
The whole logic of how it works can be outlined as follows:

- The input view of the 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.
- The report class gets the properties, uses them to retrieve the details required in the report, and populates the velocity context with the details.
- Velocity templates use the details in its context to render the report.

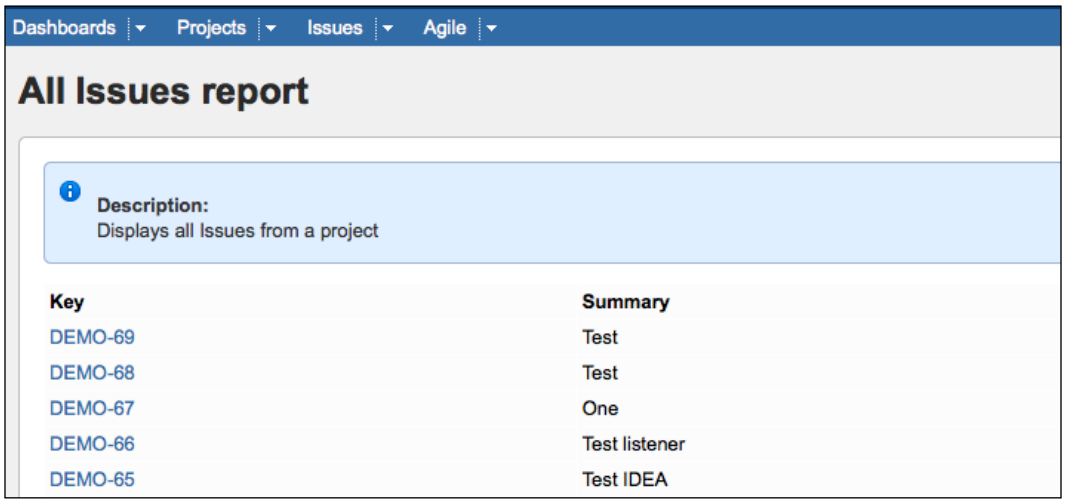
After the plugin is deployed, you can see the report among other JIRA reports in the **Browse Project** section, as shown in the following screenshot:



After clicking on the report, the input screen is displayed, which is constructed using the properties entered in the plugin descriptor; a **Project** drop-down in our case:



Once you click on **Next**, the report will be generated using the Report class and will be rendered using the velocity 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 an 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 `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` method 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.

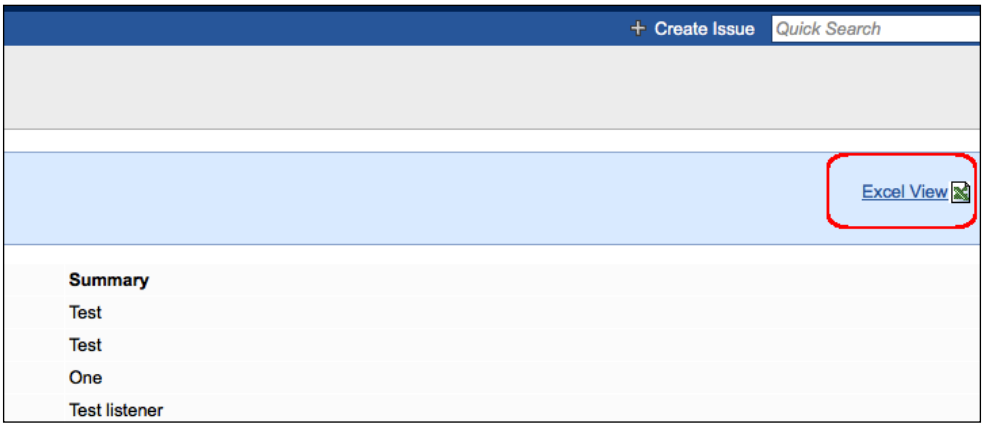
   In our example, all we have is a list of issues with its summary and the key. Therefore, we can even use the same template for Excel. It appears as follows:

```
<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 entitled **Excel View** will appear in the upper-right side of the generated reports, as shown in the next screenshot:



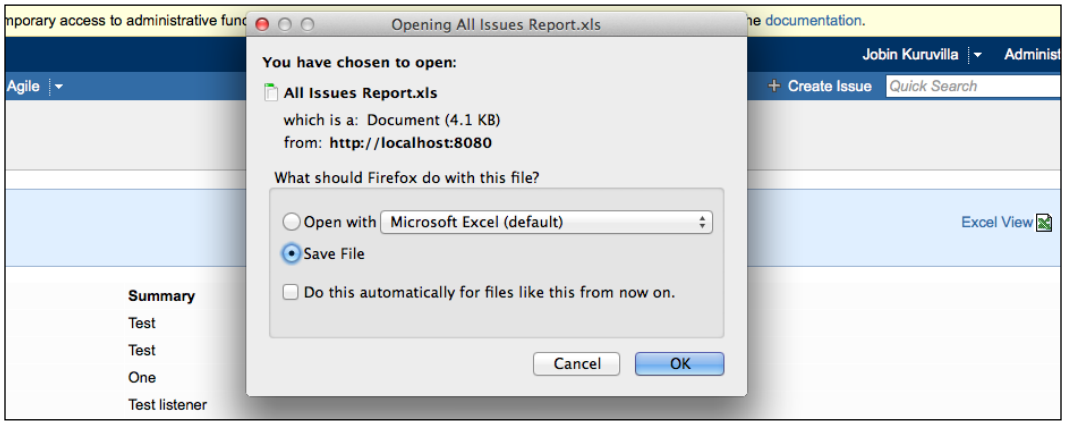
On clicking the link, the `generateReportExcel` method is executed, which in turn will generate the report and render the Excel view using the appropriate template that is defined in the plugin descriptor.

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 in the following:

```
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 on this.

See also

- The *Writing a JIRA report* recipe.

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 previous 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 low, as the project is selected from a valid list of available projects. However, 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 of this chapter.

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");
   ```

   In this case, `reqParams` 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 `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 on top of the field.

   You can use internationalization here as well to include appropriate error messages.

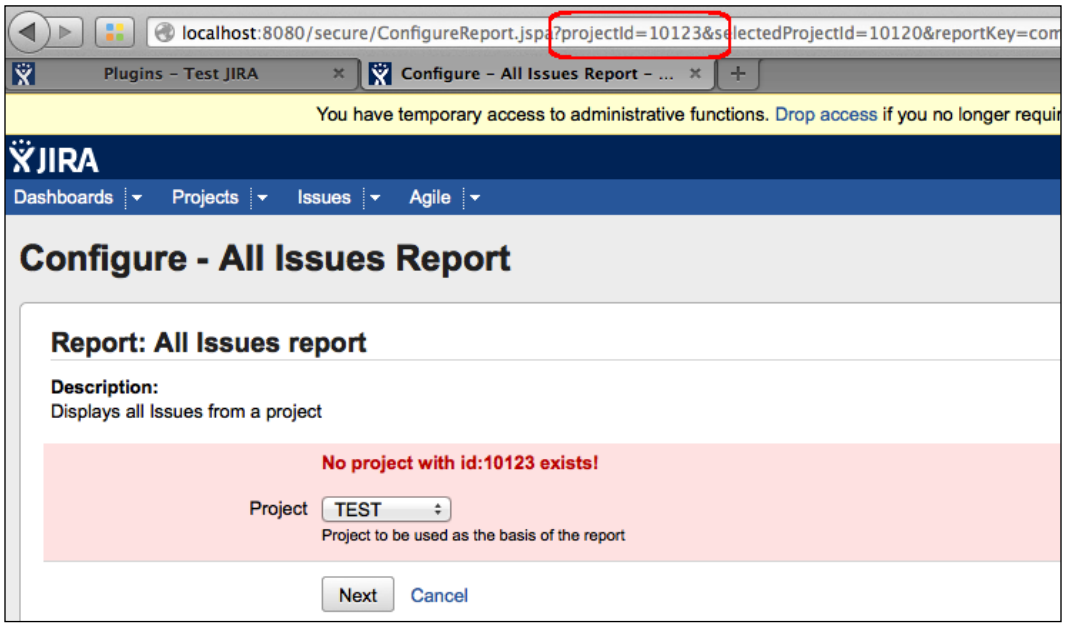
5. Add similar validations for all the interested parameters. The following is how the method looks in our example:

   ```
   @Override
   public void validate(ProjectActionSupport action, Map reqParams) {
     // Do your validation here if you have any!
      final String projectid = (String) reqParams.get("projectId");
      final Long pid = new Long(projectid);

      if (ComponentAccessor.getProjectManager().getProjectObj(pid) ==
   null) {
       action.addError("projectId", "No project with id:"+projectId+"
   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 10123.

See also

- The *Writing a JIRA report* recipe.

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 of this chapter.

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 to do this:

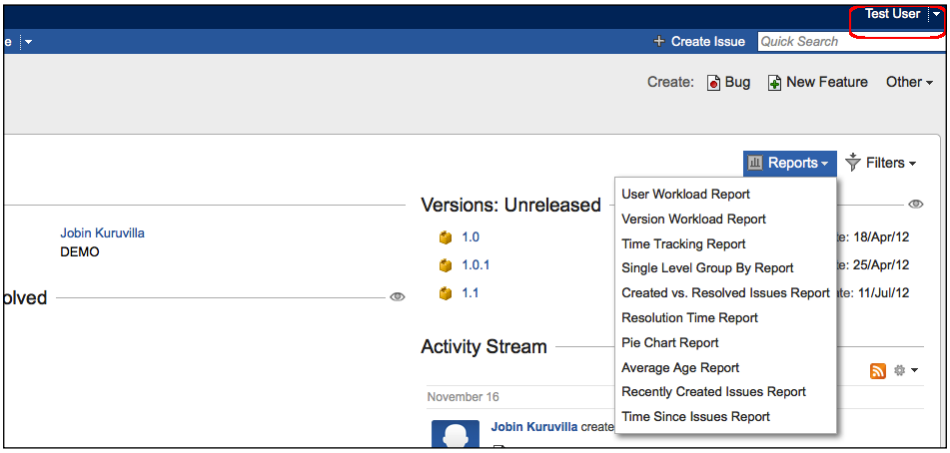
1. Override the `showReport` method in the report class we created in the previous recipes.
2. Implement the logic to return true only if the condition is satisfied. In our example, the report should be visible only to JIRA Administrators, and hence we should return true only if the current user is a JIRA Administrator:

```
@Override
public boolean showReport() {
  User user = ComponentAccessor.getJiraAuthenticationContext().
getLoggedInUser();
  return ComponentAccessor.getUserUtil().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 **Browse** projects area. If not, the report link won't be visible. We can include similar conditions and evaluate them in the `showReport` method before returning true:



The user **Test User**, highlighted in the preceding screenshot, is not a JIRA Administrator, and therefore is not able to see the **All Issues** report.

See also

- The *Writing a JIRA report* recipe

Object-configurable parameters for reports

We have seen how to write JIRA reports and 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 by JIRA 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 5.1.*, the following are the property types supported for reports:

Type	Input HTML type
`string`	Text Box
`long`	Text Box
`hidden`	NA. Hidden to the user.
`date`	Text Box with a Calendar pop-up
`user`	Text Box with a User Picker
`text`	Text Area
`select`	Select List
`multiselect`	Multi-select List
`checkbox`	Check Box
`filterpicker`	Filter Picker
`filterprojectpicker`	Filter or Project Picker
`cascadingselect`	Cascading Select List. Dependant 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 as shown in the previous code snippet.

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:

1. **Using a Value Generator class**: This 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 in the previous recipe here:

```
<property>
  <key>projectId</key>
  <name>Project</name>
  <description>report.allissues.project.description</description>
  <type>select</type>
  <values class="com.jtricks.ProjectValuesGenerator"/>
</property>
```

In this example, `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;
  }
}
```

2. **Using predefined key/value pairs in the property**: The following is an example of how to do this:

```
    <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 seeing the field 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 this 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: This is used to create a Filter or Project Picker. The Java data type is String, and it will be the ID preceded by 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: This is 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 values 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` class 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 `key1` key selected!

After selecting the appropriate values, they can be retrieved in the report class, as shown in the following lines of code:

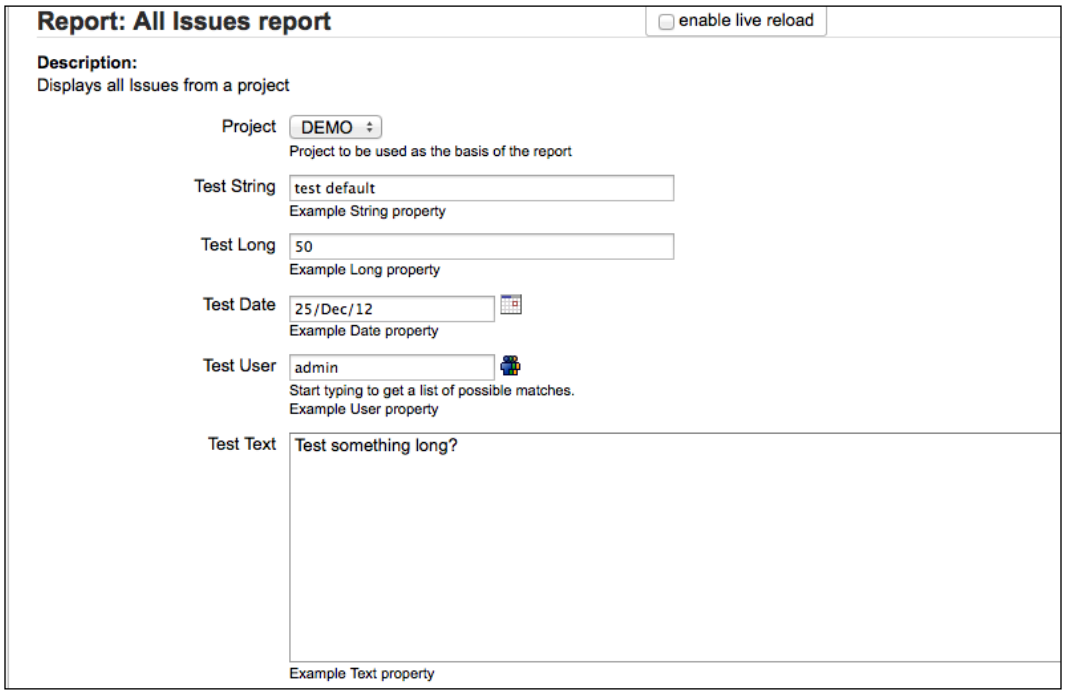
```
final String testString = (String) reqParams.get("testString");
final String testLong = (String) reqParams.get("testLong");
final String testHidden = (String) reqParams.get("testHidden");
final String testDate = (String) reqParams.get("testDate");
final String testUser = (String) reqParams.get("testUser");
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");
```

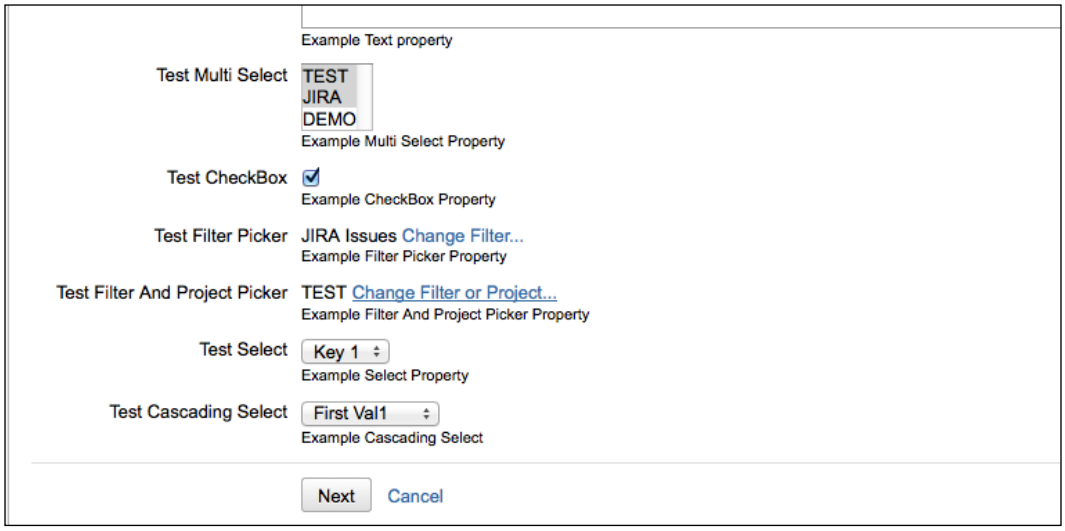
Special mention should be given to `filterprojectpicker`. The value will be `filter-10000` if a filter with ID 10000 is selected. The value will be `project-10000` if a project with ID 10000 is selected.

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 we previously discussed. We can then retrieve their values in the report class and process them to generate the report.

The following two screenshots show how these properties appear on the input screen:





If you print the extracted values into the console in the report class, it will appear as follows:

```
[INFO] [talledLocalContainer] ***********************************
[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:25/Dec/12
[INFO] [talledLocalContainer] Test User:admin
[INFO] [talledLocalContainer] Test Text:Test something long?
[INFO] [talledLocalContainer] Test Multi Select:[10001, 10002]
[INFO] [talledLocalContainer] Test Checkbox:true
[INFO] [talledLocalContainer] Test Filter Picker:10000
[INFO] [talledLocalContainer] Test Filter Project Picker:project-10001
[INFO] [talledLocalContainer] Test Select:key1
[INFO] [talledLocalContainer] Test Cascading Select:One1
[INFO] [talledLocalContainer] ***********************************
```

Hopefully, that gives you a fair idea of how to use object-configurable parameters in JIRA reports.

See also

- The *Writing a JIRA report* recipe

Creating a pie chart in JIRA

As we have already seen in the previous recipes, JIRA ships with a bunch of built-in reports. It also lets us write our own reports using the report plugin module. One of the reports that attracts a lot of users in JIRA is the **Pie Chart**. While the existing JIRA pie reports are really good at what they are meant for, sometimes the need arises to write our own pie charts. For example, you might want to display a pie chart that shows the distribution of issues based on status over a given period of time. There can be many other use cases like this.

Writing a pie chart in JIRA is easy because JIRA already supports `JFreeChart` and has utility classes, which do most of the work in creating these charts. In this recipe, we will see how to write a simple pie chart with the help of Atlassian Utility classes.

Getting ready

Create a skeleton plugin using Atlassian Plugin SDK.

How to do it...

Let us try to create a very simple pie chart without any business logic. To keep things simple and to concentrate on the pie chart, let us go for a report without any input parameters and with just the HTML view. The following are the steps to accomplish this:

1. Add the report plugin module in the plugin descriptor:

```
<report key="pie-chart" name="Pie Chart" class="com.jtricks.
PieChart">
  <description>Sample Pie Chart</description>
  <label>Example Pie Chart</label>
  <resource type="velocity" name="view" location="templates/pie/
pie-chart.vm" />
</report>
```

   All this has is a class and a velocity template for the HTML view.

2. Create the report class. As usual, it should implement the `AbstractReport` class. All we do here is populate the velocity templates with parameters from the pie chart we created using a custom `PieChartGenerator` class.

   Here is how the `generateReportHtml` looks:

```
public String generateReportHtml(ProjectActionSupport action, Map
reqParams) throws Exception {
  final Map<String, Object> params = new HashMap<String,
Object>();
  params.put("report", this);
  params.put("action", action);
  params.put("user", authenticationContext.getLoggedInUser());

  final Chart chart = new JTricksPieChartGenerator().generateChart
(authenticationContext,REPORT_IMAGE_WIDTH, REPORT_IMAGE_HEIGHT);

  params.putAll(chart.getParameters());

  return descriptor.getHtml("view", params);
}
```

   The `Chart` class is an Atlassian class of the type `com.atlassian.jira.charts.Chart`. The business logic of creating the chart is done inside a custom utility class, `JTricksPieChartGenerator`, which we will see next.

3. Create the `JTricksPieChartGenerator` utility class, which generates the pie chart.

   This is where the business logic of creating the chart is done, and therefore, we will look at them in detail:

   - Create the `DefaultPieDataset` class, which will be the data set for the pie chart. This is a `JFreeChart` class, for which the Java Docs can be found at `http://www.jfree.org/jfreechart/api/javadoc/org/jfree/data/general/DefaultPieDataset.html`.

     ```
     DefaultPieDataset dataset = new DefaultPieDataset();
     ```

   - Populate the values in `dataset`:

     ```
     dataset.setValue("One", 10L);
     dataset.setValue("Two", 15L);
     ```

     In the previous example, we just populated two key/value pairs with a name and a number value. This is the data, using which the pie chart is generated. When we generate custom charts, we should replace this data with the appropriate data that we are interested in.

   - Create the chart:

     ```
     final ChartHelper helper = new PieChartGenerator(dataset,
     authenticationContext.getI18nHelper()).generateChart();
     helper.generate(width, height);
     ```

     In this example we use the `com.atlassian.jira.charts.jfreechart.PieChartGenerator` class to generate the chart using the dataset we just created and the `i18nHelper`. Make sure you invoke the generate method, as shown in the preceding snippet.

   - Populate a map with all the required parameters from the generated `ChartHelper` and return a `Chart` object, as shown next:

     ```
     params.put("chart", helper.getLocation());
     params.put("chartDataset", dataset);
     params.put("imagemap", helper.getImageMap());
     params.put("imagemapName", helper.getImageMapName());
     params.put("width", width);
     params.put("height", height);

     return new Chart(helper.getLocation(), helper.getImageMap(),
     helper.getImageMapName(), params);
     ```

     You can add all the parameters available, but we are limiting it to the absolute minimum. `params.putAll(chart.getParameters())` in the report class will then populate the velocity context with all these parameters.

- The `generateChart` method will now look as follows:

```
public Chart generateChart(JiraAuthenticationContext
authenticationContext, int width, int height) {
    try {
      final Map<String, Object> params = new HashMap<String,
Object>();
      // Create Dataset
      DefaultPieDataset dataset = new DefaultPieDataset();

      dataset.setValue("One", 10L);
      dataset.setValue("Two", 15L);

      final ChartHelper helper = new
PieChartGenerator(dataset, authenticationContext.
getI18nHelper()).generateChart();
      helper.generate(width, height);

      params.put("chart", helper.getLocation());
      params.put("chartDataset", dataset);
      params.put("imagemap", helper.getImageMap());
      params.put("imagemapName", helper.getImageMapName());
      params.put("width", width);
      params.put("height", height);

      return new Chart(helper.getLocation(), helper.
getImageMap(), helper.getImageMapName(), params);

    } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Error generating chart",
e);
    }
}
```

4. Create the velocity template for the HTML view using the context we populated in the report class. In our example, the template is `templates/pie/pie-chart.vm`. It looks like the following block of code:

```
Sample Chart: <br><br>
<table width="100%" class="report">
  <tr>
    <td>
      #if ($chart)
         #if ($imagemap)
          $imagemap
        #end
```

```
        <p class="report-chart">
          <img src='$baseurl/charts?filename=$chart' border='0'
#if ($imagemap) usemap="\#$imagemapName" #end/>
        </p>
      #end
    </td>
  </tr>
</table>
```

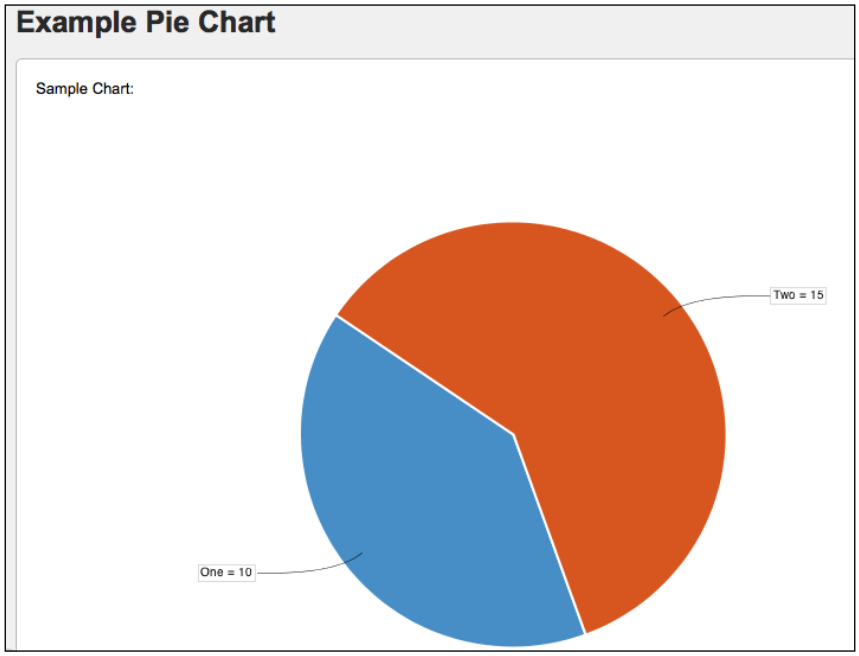
In this example we display the chart we created. The chart is available at the `$baseurl/charts?filename=$chart` URL, where `$chart` is the location generated by the helper class. We populated this earlier in the context.

5. Package the plugin, deploy it, and test it!

How it works...

In short, the bit we need to do here is to create `DefaultPieDataset` and everything else is done by JIRA for you. The `generateChart` method may take more arguments depending on the complexity of the reports we are going to create. For example, `startDate`, `endDate`, and so on. The dataset will then be created using these arguments instead of our hardcoded values!

In our example, the chart appears as follows:



There's more...

Have a look at the other helper classes under the `com.atlassian.jira.charts.jfreechart` package, for example, `StackedBarChartGenerator`, `HistogramChartGenerator`, `CreatedVsResolvedChartGenerator`, and so on, for other types of charts!

See also

- The *Writing a JIRA report* recipe

Writing JIRA 4 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**. This includes the title, description, author's name, and so on.
  - **The Screenshot and a thumbnail image**. 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 Required features** that the gadget container must provide for the gadget.
  - **User preferences**, which will be configured by the gadget users.
  - The **Gadget content** created using HTML and JavaScript.
- A screenshot and thumbnail image will be used during preview and while selecting the gadget from the container
- An `i18n` property file will be used for internationalization in the gadget
- Optional CSS and JavaScript files will be 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 greetings from JTricks!

1. Modify the plugin descriptor with the gadget module and the resources required for our gadget:
   - Add the `Gadget` module in the plugin descriptor:

     ```
     <gadget key="hello-gadget" name="Hello Gadget"
     location="hello-gadget.xml">
        <description>Hello Gadget! </description>
     </gadget>
     ```

     As you can see, this has a unique key and 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 will stick with one.

   - Include the thumbnail, screenshot images, and the downloadable resources in the plugin descriptor. We have seen details of this in the previous chapter, and more can be learned at `http://confluence.atlassian.com/display/JIRADEV/Downloadable+Plugin+Resources`. In our example, the resources are added on to the plugin descriptor as the following:

     ```
     <resource type="download" name="screenshot.png" location="/
     images/screenshot.png"/>
     <resource type="download" name="thumbnail.png" location="/
     images/thumbnail.png"/>
     ```

     The location is relative to the `src/main/resources` folder in the plugin. As mentioned before, the screenshot is optional.

2. Add 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>
   ```

   The `atlassian-plugin.xml` file will now look like this:

   ```
   <atlassian-plugin key="com.jtricks.gadgets" name="Gadgets Plugin"
   plugins-version="2">
       <plugin-info>
         <description>Gadgets Example</description>
   ```

```
      <version>2.0</version>
      <vendor name="JTricks" url="http://www.j-tricks.com/" />
    </plugin-info>
    <gadget key="hello-gadget" name="Hello Gadget"
location="hello-gadget.xml">
      <description>Hello Gadget!</description>
    </gadget>

    <resource type="download" name="screenshot.png" location="/
images/screenshot.png"/>
    <resource type="download" name="thumbnail.png" location="/
images/thumbnail.png"/>

    <resource type="download" name="i18n/messages.xml"
location="i18n/messages.xml">
      <param name="content-type" value="text/xml; charset=UTF-8"/>
    </resource>
  </atlassian-plugin>
```

3. Add the screenshot and thumbnail images under the `src/main/resources/images` folder. The thumbnail image should be of the size 120 x 60 pixels.

4. Add the `i18n` properties file under the `src/main/resources/i18n` folder. The name of the file we define 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 use 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>
   ```

5. 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 of each of them in this example. The entire set of attributes and elements and other details of the gadget specification can be read at `http://confluence.atlassian.com/display/GADGETDEV/Creating+your+Gadget+XML+Specification`.

- Write the `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="__MSG_gadget.title__"
             title_url="__MSG_gadget.title.url__"
             description="__MSG_gadget.description__"
             author="Jobin Kuruvilla"
             author_email=jobinkk@gmail.com
             screenshot='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "screenshot.png")'
             thumbnail='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "thumbnail.png")' height="150"  >
</ModulePrefs>
```

  As you can see, it holds information such as the title, title URL (to which the gadget title will link to), description, author name and e-mail, height of the gadget, and URLs to the 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 200 by default, but this is optional. The images are referenced using `#staticResourceUrl`, where the first argument is the fully qualified gadget module key, which is of the form `${atlassian-plugin-key}:${module-key}`. In our example, the plugin key is `com.jtricks.gadgets` and the module key is `hello-gadget`.

- 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 this example, we add the category as `Other`.

  Other values that are supported for the 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 the categories within the `Param` element, each in a new line.

- Include `Required` features if there are any under the `require` XML tag. A full list of supported features can be found at `http://confluence.atlassian.com/display/GADGETDEV/Including+Features+into+your+Gadget`.
- Add the `Locale` element to point to the `i18n` properties file:

  ```
  <Locale messages="__ATLASSIAN_BASE_URL__/download/resources/com.jtricks.gadgets/i18n/messages.xml"/>
  ```

  In this example, 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.gadgets`, where `com.jtricks.gadgets` is the Atlassian plugin key. The path to the XML file `/i18n/messages.xml` is what is defined in the resource module earlier.

- Add user preferences if required using the `UserPref` element. We will omit this in our example, as the "Hello Gadget" doesn't take any inputs from the user.
- 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 `<![CDATA[ and ]]>` so that it 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's XML is now ready and looks like the following block of code:

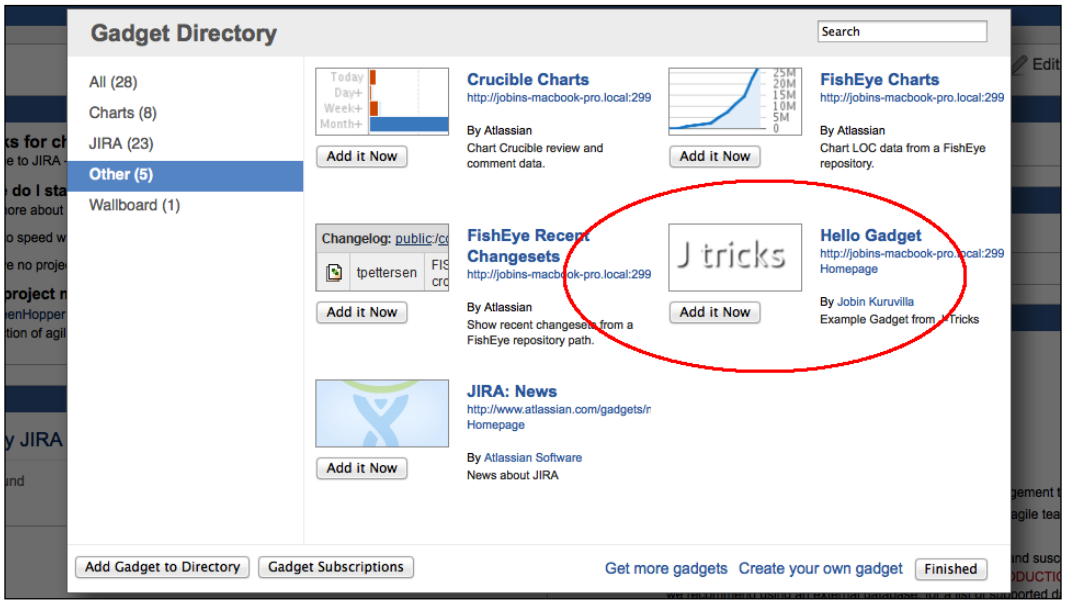
```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="__MSG_gadget.title__"
               title_url="__MSG_gadget.title.url__"
               description="__MSG_gadget.description__"
               author="Jobin Kuruvilla"
               author_email=jobinkk@gmail.com
               screenshot='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "screenshot.png")'
               thumbnail='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "thumbnail.png")' height="150" >
```

```
    <Optional feature="gadget-directory">
      <Param name="categories">
        Other
      </Param>
    </Optional>
    <Locale messages="__ATLASSIAN_BASE_URL__/download/resources/
com.jtricks.gadgets/i18n/messages.xml"/>
  </ModulePrefs>
  <Content type="html" view="profile">
    <![CDATA[ Hello From JTricks ]]>
   </Content>
</Module>
```

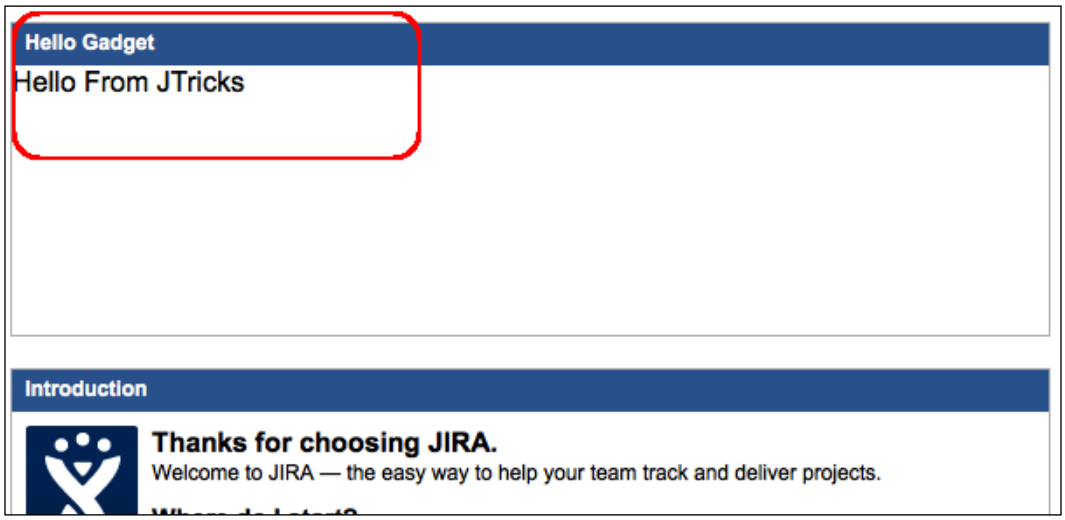
6. Package the plugin, deploy it, and test it.

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"/>
```

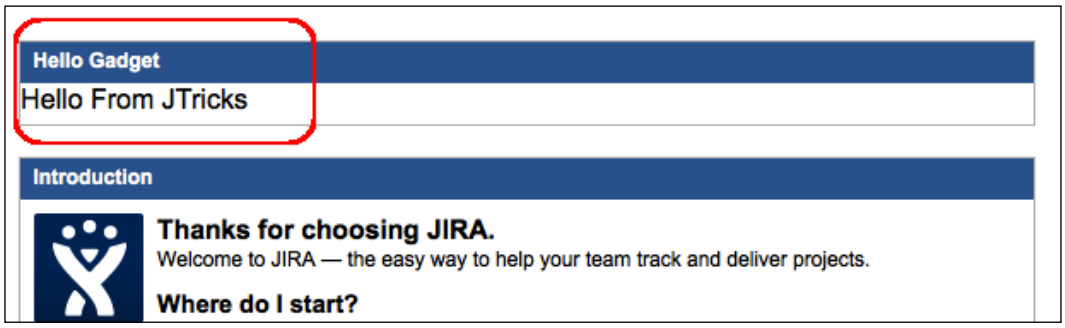
We should then invoke `gadgets.window.adjustHeight()` whenever the content is reloaded. For example, we can do it in a `window onload` event, as shown next:

```
<script type="text/javascript" charset="utf-8">
    function resize()
    {
      gadgets.window.adjustHeight();
    }
    window.onload=resize;
</script>
```

The `gadget.xml` file, in this case, will look like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
    <ModulePrefs title="__MSG_gadget.title__"
                 title_url="__MSG_gadget.title.url__"
                 description="__MSG_gadget.description__"
                 author="Jobin Kuruvilla"
                 author_email="jobinkk@gmail.com"
                 screenshot='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "screenshot.png")'
                 thumbnail='#staticResourceUrl("com.jtricks.
gadgets:hello-gadget", "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.gadgets/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 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 inbuilt REST methods, one of which is to retrieve the details of the current user. The method can be reached in the following 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 required is in the gadget XML, which is `hello-gadget.xml` in our example. Only the `Content` element needs to be modified, which will now invoke the REST service and render the content.

The following are the steps to do this:

1. Include the common Atlassian gadget resources:

   ```
   #requireResource("com.atlassian.jira.gadgets:common")
   #includeResources()
   ```

   `#requireResource` will bring the JIRA gadget JavaScript framework into the gadget's context. `#includeResources` will write out the HTML tags for the resource in place. Check out `http://confluence.atlassian.com/display/GADGETDEV/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:

   - `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.
   - `useOauth`: An optional parameter. This is used to configure the type of authentication, which must be a URL. `/rest/gadget/1.0/currentUser` is commonly used.
   - `config`: Another optional parameter. This is only used if there are any configuration options for the gadget.
   - `view`: This is 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 in the following code snippet 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:

   - `enableReload`: This is optional and used to reload the gadget at regular intervals.
   - `onResizeReload`: This is also optional and is used to reload the gadget when the browser is resized.
   - `onResizeAdjustHeight`: This is once more optional and used along with the `dynamic-height` feature. This will adjust the gadget height when the browser is resized.
   - `template`: This creates the actual view.
   - `args`: This is an array of objects or a function that returns an array of objects. It has two attributes: `Key`— which is 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 invoke the `/rest/gadget/1.0/currentUser` method and use 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":"jobinkk@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);
    }
```

In this case, `args.user["fullName"]` will retrieve the user's `fullName` from the REST output. The 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>
```

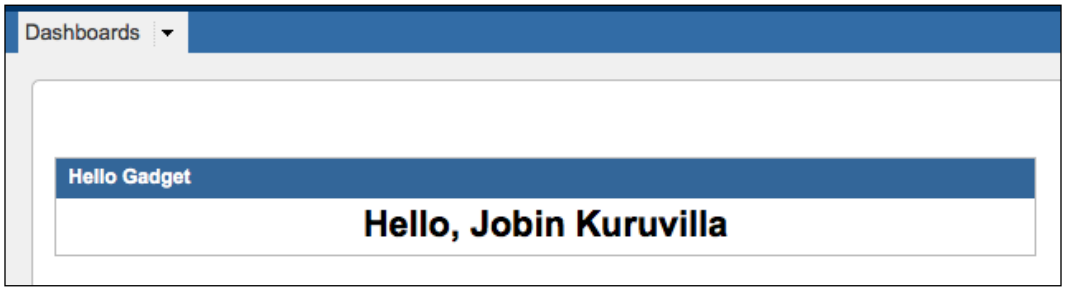
4. Package the gadget and deploy it.



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

Configuring user preferences in gadgets

In the previous two recipes, we saw how to create gadgets from both 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 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` property is set to `false`, the greeting message will be **Hello!**

The following are the steps to configure the 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` element. This element defines the various user preferences. The element supports the following fields:

   - `name`: The name of the user preferences. The value of this can then be accessed using `gadget.getPref("name")`.
   - `display_name`: The display name of the field. By default, it will be the same as the name.
   - `urlparam`: The optional string to pass as the parameter name for content `type="url"`.
   - `datatype`: The data type of the field. Valid options include `string`, `bool`, `enum`, `hidden`, or `list`. Default is `string`.
   - `required`: This marks the field as required. Default is false.
   - `default_value`: This 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}
},
...
```

In this case, `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"
            }
          ]
        }
      ]
     };
  }
}
```

In this case, 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 `http://confluence.atlassian.com/display/GADGETDEV/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` attribute named `isConfigured`:

```
<UserPref name="isConfigured" datatype="hidden" default_
value="false"/>
```

   The user preferences are set every time the gadget loads, and we use this property as it 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__",
```

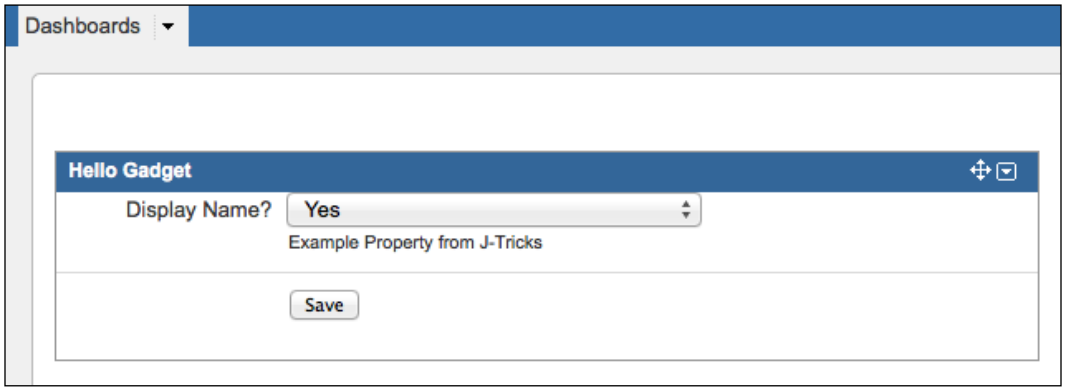
```
    config: {
      descriptor: function (args) {
       var gadget = this;
       return  {
        fields: [
         {
         userpref: "displayName",
         label: gadget.getMsg("property.label"),
         description:gadget.getMsg("property.description"),
         type: "select",
         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")
         {
         varuserDetails = AJS.$("<h1/>").text("Hello,
          "+args.user["fullName"]);
         } else
         {
         varuserDetails = AJS.$("<h1/>").text("Hello!");
         }
        gadget.getView().html(userDetails);
    },
  args: [{
  key: "user",
  ajaxOptions: function() {
  return {
```

```
        url: "/rest/gadget/1.0/currentUser"
                        };
                    }
                }]
        }
        });
            })();
</script>
]]>
</Content>
```

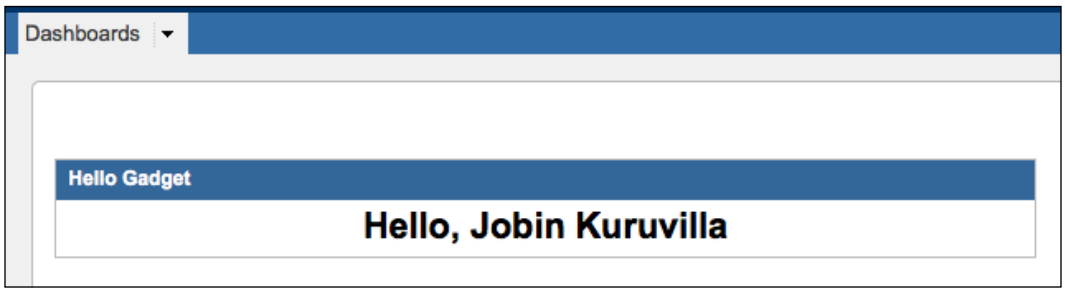
8. Package the gadget and deploy it.

How it works...

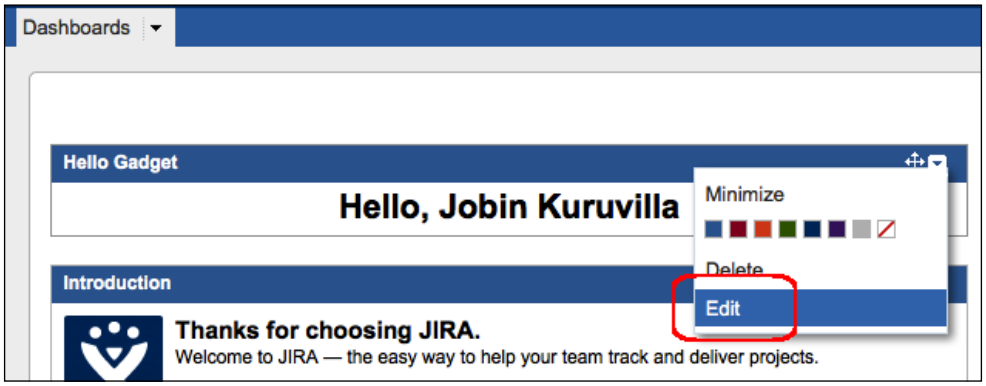
Once the user configurable properties are added, upon its creation, the gadget will ask the user to configure the `displayName` property, as shown next. The default value will be true (label: **Yes**) as we have configured it:



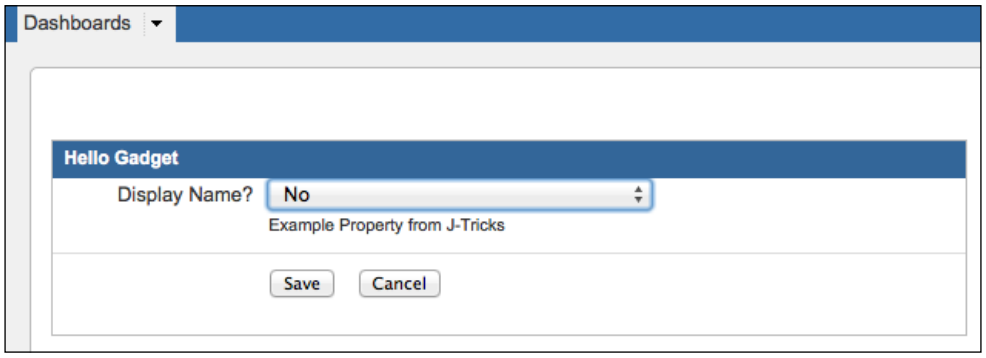
When **Yes** is selected, it appears exactly as it was in the previous recipe:



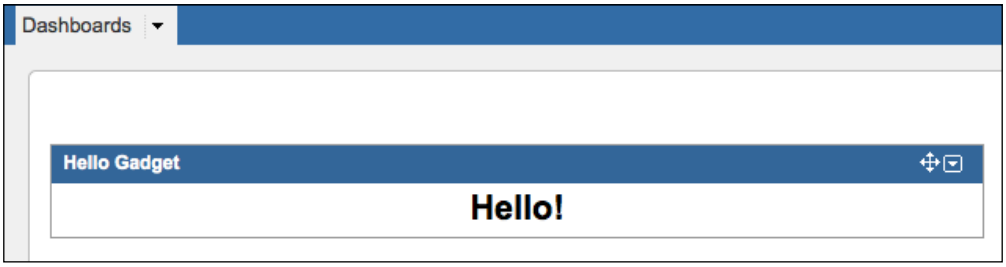
If you click on the gadget options now, you can see the **Edit** option, as shown in the following screenshot:



The following screenshot appears while clicking 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 this.

There are only a couple of things you need do to implement this feature:

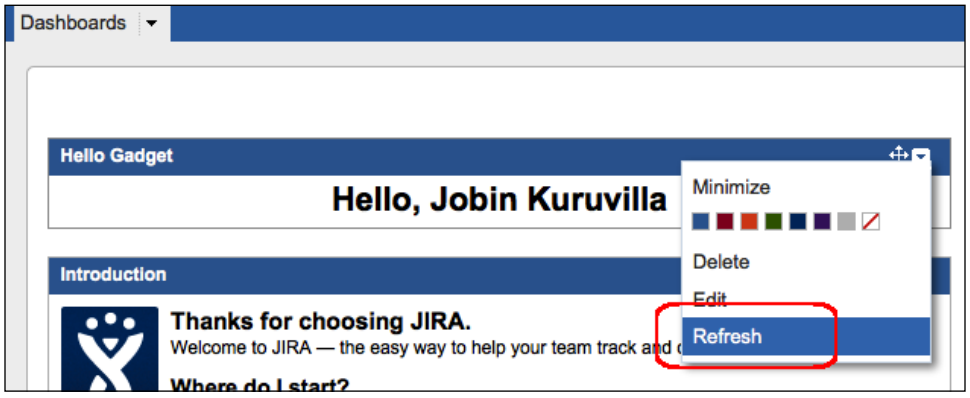
1. Add the `refresh` user preference:

```
<UserPref name="refresh" datatype="hidden" default_value="false"/>
```

2. Include the `enableReload: true` property in the view:

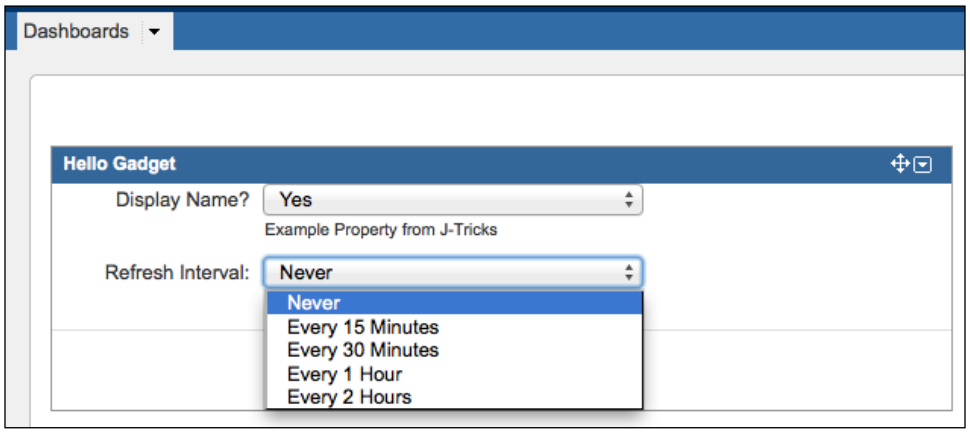
```
view: {
  enableReload: true,
   template: function(args) {
     ...
  },
  args: [{
     ...
   }]
}
```

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
- The *Invoking REST services from gadgets* recipe

Accessing gadgets outside of JIRA

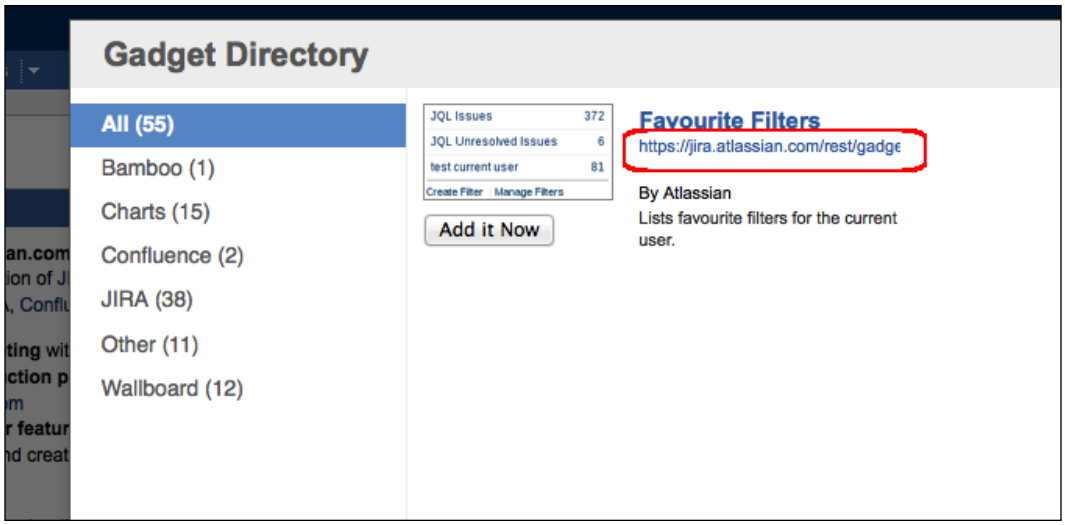
We have seen how to write a gadget and add it into the JIRA Dashboard. But have we made use of all the advantages of an OpenSocial gadget? How about adding them onto other OpenSocial containers such as Gmail or iGoogle?

In this recipe, we will see how to add a gadget in to Gmail. The process is pretty much similar for other containers as well.

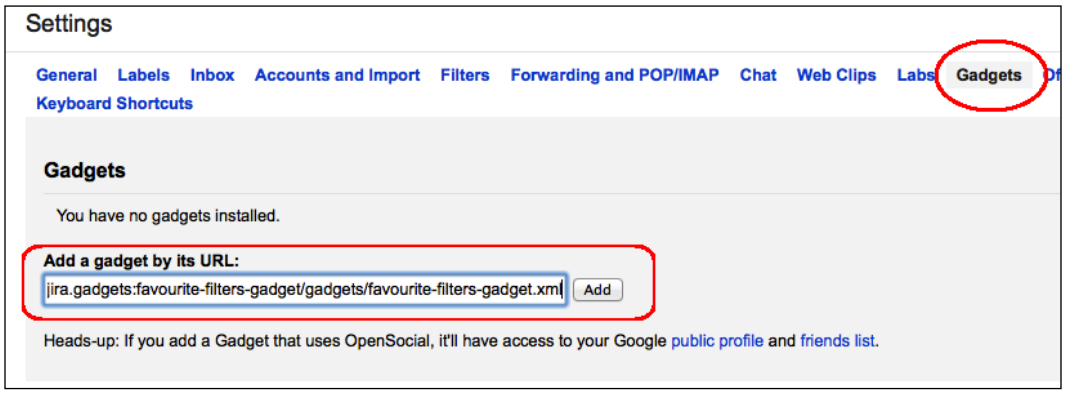
How to do it...

The following is a quick step-by-step procedure to add a gadget to Gmail:

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 this example, we choose to add the **Favourite Filters** gadget:



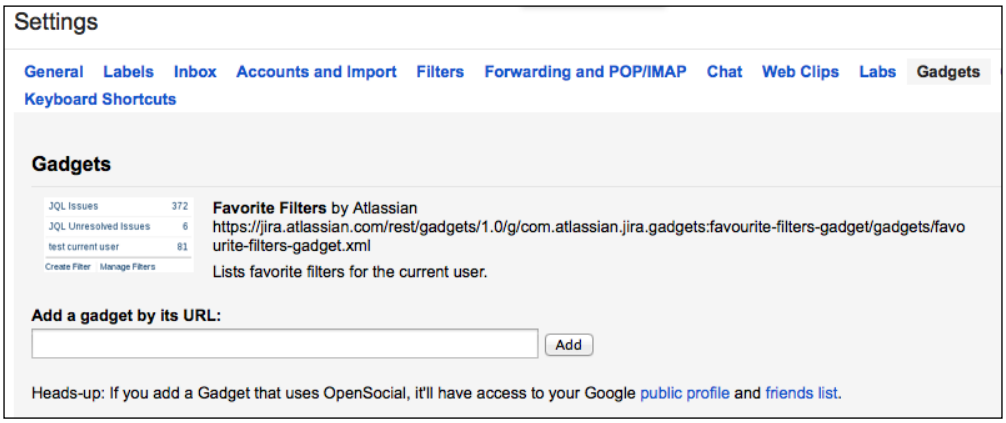
2. Go to **Gmail** | **Settings** | **Gadgets** and enter the URL, as shown in the next screenshot:





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.

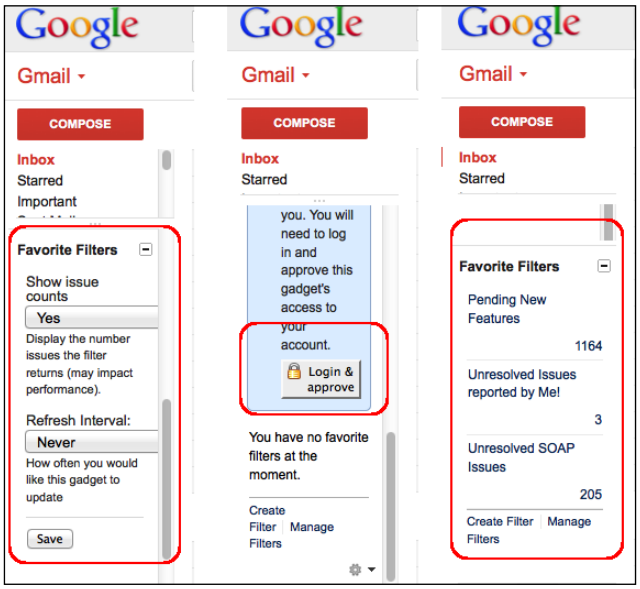
3. Once added, the gadget will appear in the settings as shown in the following screenshot:



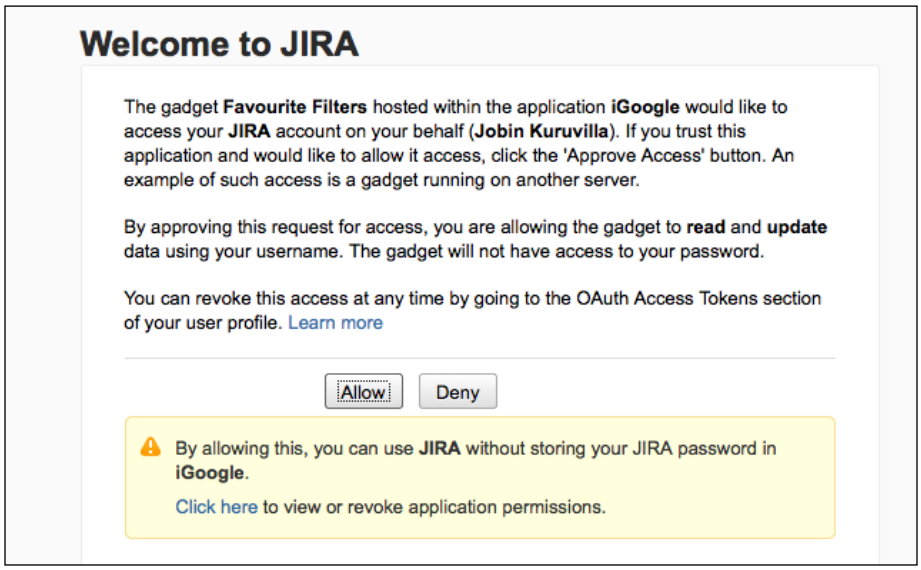
4. The gadget should now be available under the list of gadgets you have in your Gmail sidebars. Save the configurations. In our example, we need to choose whether or not to display the count of issues and the refresh interval.

   The gadget initially will not show any results because we haven't connected to JIRA with a proper username/password. Edit the gadget settings and you will see an option, **Login & Approve**, which allows you to log in to your JIRA instance and approve the retrieval of data to be displayed in Gmail.

   Once approved, results will be shown, just the way you wanted it! The following screenshot shows the three stages:



5. **Login & approve**, which is the second step in preceding screenshot, takes us to our JIRA instance, as shown in the following screenshot:



6. Once you click on **Allow**, the approval is done. If you were not logged in prior to JIRA, you will be asked enter username/password of your JIRA instance.

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.

See also

- The *Writing JIRA 4 gadgets* recipe
- The *Invoking REST services from gadgets* recipe

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 JQL queries in plugins
- Linking directly to search queries
- Index and de-index programmatically
- 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.

Before we start, it would make sense to look at one of the major enhancements in JIRA 4 and above, that is, **JIRA Query Language** (**JQL**). JQL brings to the table advanced searching capabilities, using which the users can search for issues in their JIRA instance, and then exploit all the capabilities of the Issue Navigator.

In addition to the previous searching capabilities, now called **simple searching**, JQL, or the **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 autocomplete features and maintains a query history to navigate easily to the recent searches. 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**, and ending with a **value** or **function**. To find out all the issues in a project, we can use `project = "TEST"`, where `project` is the field, `=` is the operator, and `TEST` is the value.

Similarly, we can find all the issues assigned to the current user using `assignee = currentUser()`, where `assignee` is the field, `=` is the operator, and `currentUser()` is a JQL function.

At this point in time, JQL doesn't support 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 `http://confluence.atlassian.com/display/JIRA/Advanced+Searching`.

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 `http://confluence.atlassian.com/display/JIRA/Advanced+Searching#AdvancedSearching-FunctionsReference`. 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 the arguments at runtime.

In our example, let us consider creating a function, `projects()`, which can take a list of project keys and return all the issues in the supplied projects, for example, `project in projects("TEST", "DEMO")`.

This will be equivalent to `project in ("TEST","DEMO")`, and also to `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 key="jql-projects" name="Projects Function"
class="com.jtricks.ProjectsFunction">
  <!--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`:

   - `fname`: This holds the JQL function name that is visible to the user. This will be used in the JQL query.
   - `list`: This 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 `NOTIN`, whereas a scalar (single item) can be used with operators `=`, `!=`, `<`, `>`, `<=`, `>=`, `IS`, and `ISNOT`.

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 as follows:

   - `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` data type as follows:

     ```
     public JiraDataType getDataType() {
       return JiraDataTypes.PROJECT;
     }
     ```



Check out the `JiraDataTypes` class (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/JiraDataTypes.html`) to see other supported data types.

   - `getMinimumNumberOfExpectedArguments`: This method 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 pre-populated function will then look like `projects("")`.

   - `validate`: This method is used for the validation of the arguments we have passed. In our example, we need to check if the method has at least one argument or not and make sure all the arguments passed are valid project keys. The `validate` method looks like the following code snippet:

     ```
     public MessageSet validate(User searcher, FunctionOperand
     operand, TerminalClause terminalClause) {
       List<String> projectKeys = operand.getArgs();
       MessageSet messages = new MessageSetImpl();
       if (projectKeys.isEmpty()) {
         messages.addErrorMessage("At least one project key
     needed");
       } else {
         for (String projectKey : projectKeys) {
     ```

```
      if (projectManager.getProjectObjByKey(projectKey) ==
null) {
        messages.addErrorMessage("Invalid Project Key:" +
projectKey);
      }
    }
  }
  return messages;
}
```

In this example, 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 that 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` class 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), we can validate it using the following line of code:

```
MessageSet messages = validateNumberOfArgs(operand, 1);
```

This code adds an error if the number of arguments is not one.

- `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:

```
public List<QueryLiteral> getValues(QueryCreationContext
context, FunctionOperand operand, TerminalClause
terminalClause) {
  notNull("queryCreationContext", context);
  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` object 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` value, and construct it with a long and it represents a `Long` value.

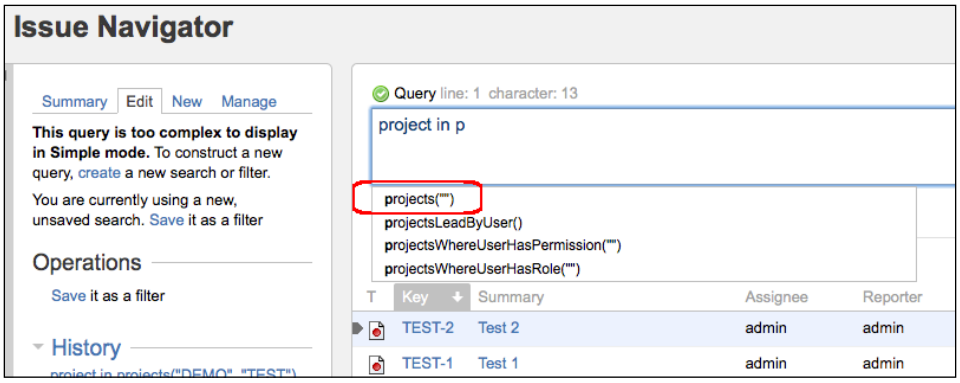
In our example, we use the project ID (`Long`) that is unique across projects. For projects, we can even use the key (`String`) or name (`String`), as they are also unique. However, it may not work with fields such as `fix for version`, as you might find two `fix for version` fields with the same name. 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.

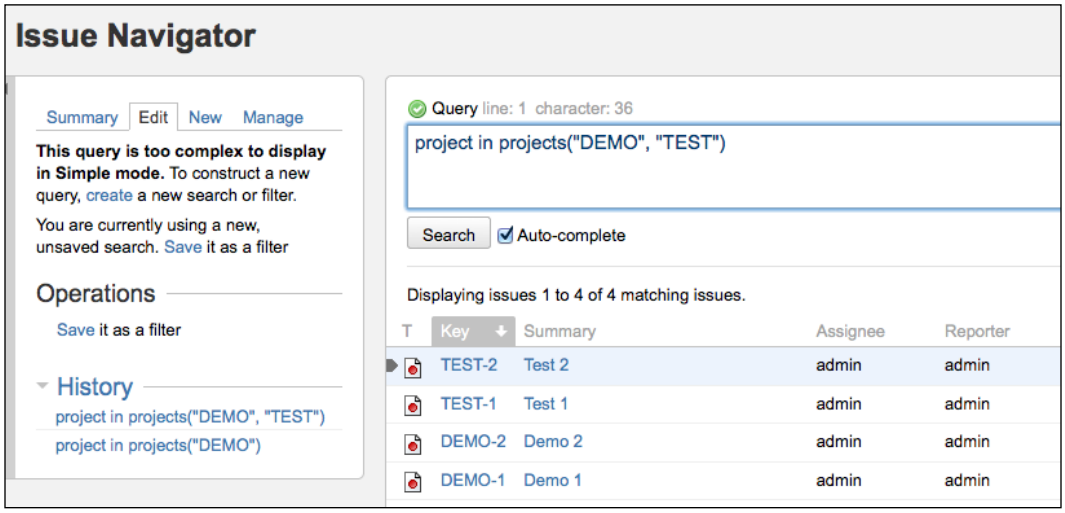
3. Package the plugin and deploy it.

How it works...

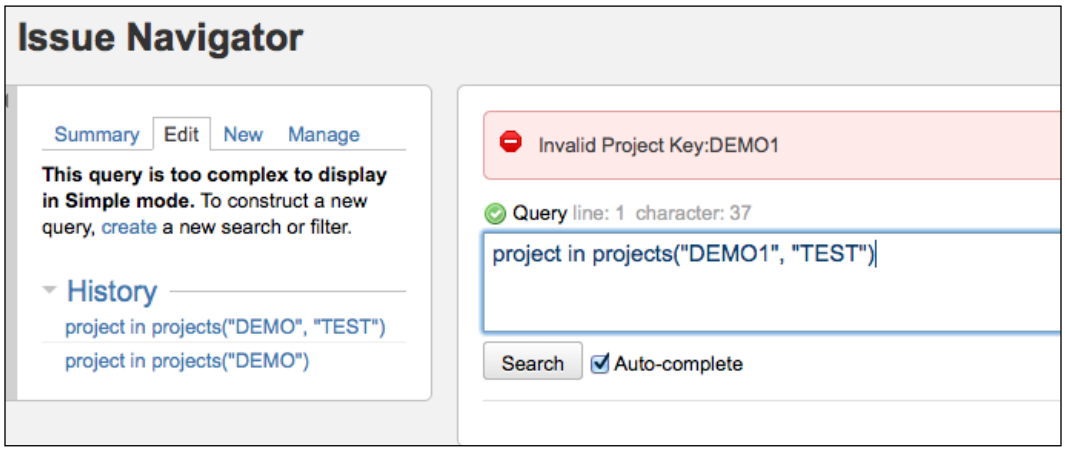
Once the plugin is deployed, we can go to the **Issue Navigator** page and open the advanced search to start using our brand new function! When you start typing `project in p`, JIRA auto-populates the available options including our new function, as shown in the following screenshot:



Once the function with appropriate arguments is added, the search is executed and the 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 a JIRA 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 any issues in a predefined 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 he/she will surely see the name of the project in the JQL query that is used in the filter.

This is where sanitizing of 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 this by sanitizing the JQL function we created in the previous recipe.

Getting ready

Develop the JQL function, as explained in the previous *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 the 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:

```
@NotNull FunctionOperand santiseOperand(User searcher, @NotNull
FunctionOperand operand);
```

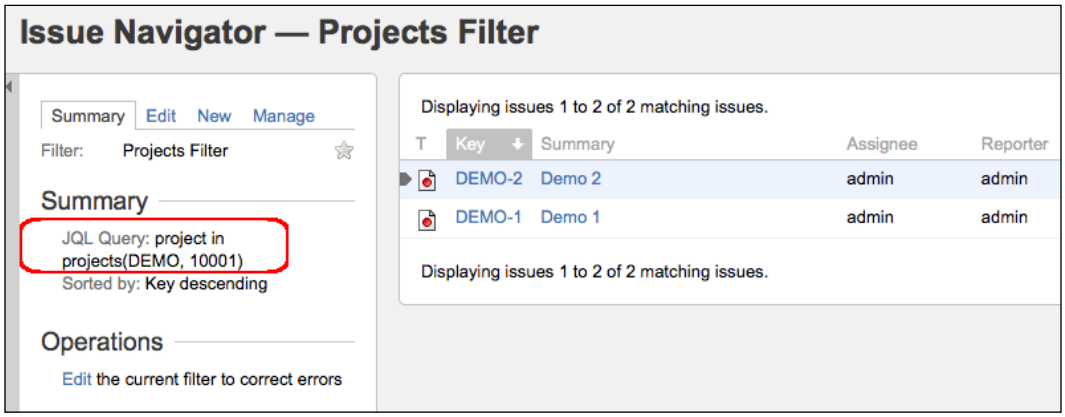
   In this case, we read all the existing arguments of the JQL function from the `FunctionOperand` argument and modify it to include project IDs instead of keys wherever the user doesn't have **Browse** permissions:

```
public FunctionOperand sanitiseOperand(User 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(Permissions.BROWSE, project, user)) {
      newArgs.add(project.getId().toString());
      argChanged = true;
    } else {
      newArgs.add(pKey);
    }
  }

  if (argChanged) {
    return new FunctionOperand(functionOperand.getName(),
newArgs);
  } else {
    return functionOperand;
  }
}
```

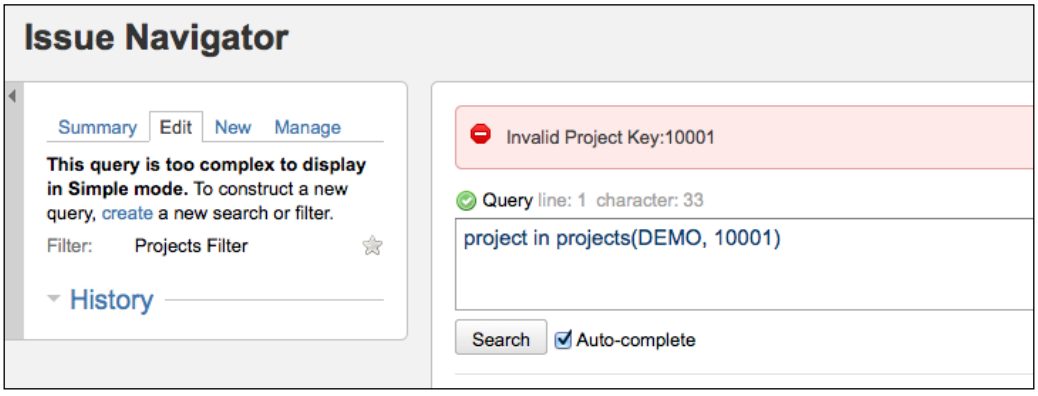
3. Package and deploy the modified plugin.

How it works...

Once the plugin is deployed, if a user doesn't have the permission to browse a project, he/she will see the project ID instead of the key that was originally entered when the filter was created. Following is a sample screenshot of how the query will look in this case. As you can see, 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!



What if you try to edit the filter now? Our validation will now kick in as it is not able to find a project with the ID, as shown in the following screenshot! Nice, eh?



This is only an example, and we can sanitize the query in a similar way in every other case.

See also

- The *Writing a JQL function* recipe

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, and to export them into Excel, Word, XML, and so on with the help of the predefined search request views.

In this recipe, we will see how we can add a simple HTML view into 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 the Atlassian plugin SDK.

How to do it...

As mentioned earlier, 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 to do this:

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.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. Following are the other attributes:

- `name`: The name that will appear in the Issue Navigator for the view.
- `class`: The search request view class. This is where we populate the Velocity contexts with the necessary information.
- `contentType`: The content type of the file that is generated, for example, `text/html`, `text/xml`, `application/rss+xml`, `application/vnd.ms-word`, `application/vnd.ms-excel`, and so on.
- `fileExtension`: The extensions of the file generated, for example, `html`, `xml`, `doc`, `xls`, and so on.
- `state`: The value of this field is either `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`, to be implemented!

   This method takes a `writer` argument, using which we can generate the output using the various template views we define, for example:

   ```
   writer.write(descriptor.getHtml("header", headerParams));
   ```

   This will identify the Velocity template with the view named as `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` file 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 default Velocity context parameters:

      ```
      final Map defaultParams = JiraVelocityUtils.
      getDefaultVelocityParams
      (authenticationContext);
      ```

b. Populate the map with the variables that we need in the context to render the header template and write the header. In our example, let us keep the header fairly simple and just use the filter name and the current user:

```
final Map headerParams = new HashMap(defaultParams);
headerParams.put("filtername", searchRequest.getName());
headerParams.put("user", authenticationContext.
getLoggedInUser());
writer.write(descriptor.getHtml("header", headerParams));
```

c. Now we need to write the search results. We should iterate over each issue in the search results and write it to the writer using the format we defined earlier. To ensure that this doesn't result in a huge memory consumption, only one issue should be loaded into the memory at a time. This can be guaranteed by using a **HitCollector**. This collector is responsible for writing out each issue as it is encountered in the search results. It will be called for each search result by the underlying Lucene search code:

```
final IndexSearcher searcher = searchProviderFactory.getSear
cher(SearchProviderFactory.
ISSUE_INDEX);
final Map issueParams = new HashMap(defaultParams);
final DocumentHitCollector hitCollector = new IssueWriterHit
Collector(searcher, writer, issueFactory){
  protected void writeIssue(Issue issue, Writer writer)
throws IOException{
    //put the current issue into the velocity context and
render the single issue view
    issueParams.put("issue", issue  writer.write(descriptor.
getHtml("body", issueParams));
  }
};
searchProvider.searchAndSort(searchRequest.getQuery(), user,
hitCollector, searchRequestParams.getPagerFilter());
```

All we do here is define the HitCollector and invoke the `searchAndSort` method, which will then use the HitCollector to generate the view for each issue. Here we can add more variables if we need them in the view.

d. We can now write the footer before we finish. Let us again put the user just for educational purposes:

```
writer.write(descriptor.getHtml("footer", EasyMap.
build("user", user)));
```

In this case, we have created a simple map just to show that we need only the variables that we use in the view. The method will now look as follows:

```
@Override
public void writeSearchResults(final SearchRequest
searchRequest, final SearchRequestParams
searchRequestParams, final Writer writer) throws
SearchException {
  final Map defaultParams = JiraVelocityUtils.
getDefaultVelocityParams
(authenticationContext);
  final Map headerParams = new HashMap(defaultParams);
  headerParams.put("filtername", searchRequest.getName());
headerParams.put("user", authenticationContext.
getLoggedInUser());
  try{
    //Header
    writer.write(descriptor.getHtml("header",
headerParams));

    //Body
    final IndexSearcher searcher = searchProviderFactory.
getSearcher
(SearchProviderFactory.ISSUE_INDEX);
    final Map issueParams = new HashMap(defaultParams);
    final DocumentHitCollector hitCollector = new IssueWrite
rHitCollector(searcher, writer, issueFactory) {
      protected void writeIssue(Issue issue, Writer writer)
throws IOException{
        //put the current issue into the velocity context
and render the single issue view
        issueParams.put("issue", issue);
        writer.write(descriptor.getHtml("body",
issueParams));
      }
    };
    searchProvider.searchAndSort(searchRequest.
getQuery(), authenticationContext.getUser(),hitCollector,
searchRequestParams.getPagerFilter());

    //Footer
    writer.write(descriptor.getHtml("footer", EasyMap.
build("user", authenticationContext.getUser())));
  }catch (IOException e){
    throw new RuntimeException(e);
  }catch (SearchException e){
    throw new RuntimeException(e);
  }
}
```

3. Write the Velocity templates. As we saw earlier, we are using three views:

   - **Header**: The Velocity template is `templates/searchrequest-html-header.vm`. The following code snippet is how it looks:

     ```
     Hello $user.displayName , have a look at the search
     results!<br><br>
     #set($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.

   - **Body**: The Velocity template is `templates/searchrequest-html-body.vm`. The following code snippet 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.

   - **Footer**: The Velocity template is `templates/searchrequest-html-footer.vm`. The following code snippet shows how it looks:

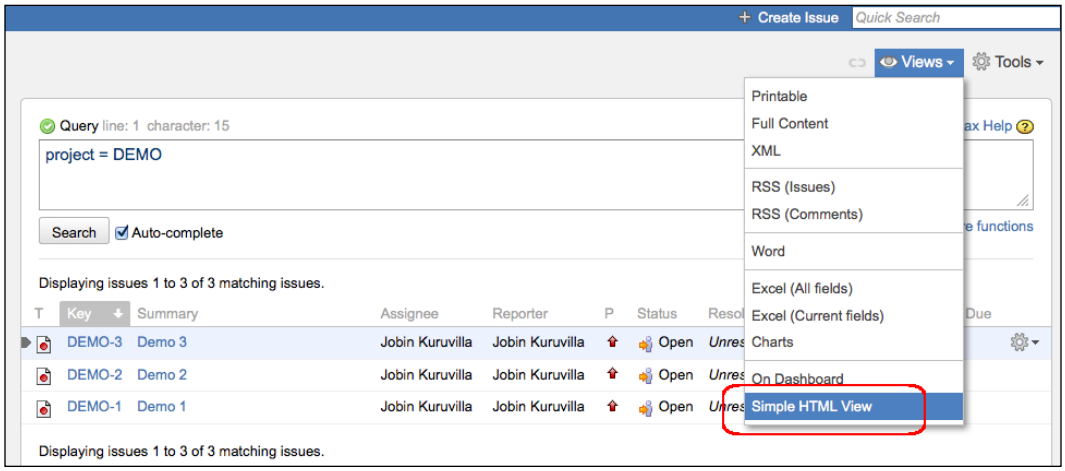
     ```
     </table>
     <br><br>...And that's all we have got now, $user.
     displayName!
     ```

     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 this view, the current search results will appear as shown in the following screenshot. Note the underlined text, which was rendered using the Velocity variables for user name and filter name:

Hello Jobin Kuruvilla , Have a look at the search results!

Filter : Anonymous

DEMO-3 Demo 3
DEMO-2 Demo 2
DEMO-1 Demo 1

...And that's all we have got now , Jobin Kuruvilla !

If the results belong to a filter, it will display the filter name instead of **Anonymous**:

Hello Jobin Kuruvilla , Have a look at the search results!

Filter : All DEMO Issues

DEMO-3 Demo 3
DEMO-2 Demo 2
DEMO-1 Demo 1

...And that's all we have got now , Jobin Kuruvilla !

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/display/JIRADEV/Search+Request+View+Plugin+Module`.

See also

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

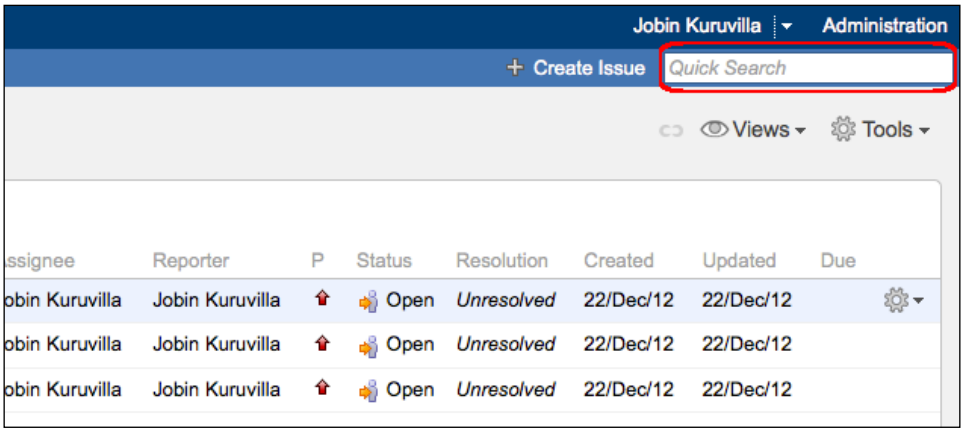
Smart querying using quick search

The name says it all! JIRA allows smart querying using its **quick search** functionality, and it enables the users to find critical information with ease. There is a predefined 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 in the right-hand top corner of JIRA, as shown in the following screenshot:



The following is how we can search on some of the fields as of JIRA 5.1. 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.



And there's more! If you are browsing a project or viewing an issue, and if 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 the 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, and it can find all the issues reported by me or the specified user. `r:none` is also supported and it returns the issues without any reporter.

For example, `r:me` will retrieve all the issues reported by me, whereas `r:admin` will retrieve all the issues reported by the user (admin).

- **Date fields**: A quick search can be done based on the three major date fields on the issue: `created`, `updated`, and `due`. 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 the issues created on the date
  - `updated:-5d` will retrieve all the issues updated in the last 5 days
  - `due:-2w,1w` will retrieve all the issues due in the last 2 weeks and in the next week



You can also use the `overdue` keyword to retrieve all the issues that are overdue (has a past due date).

- **Priority**: A quick search can be done using the priority values such as `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`.

- **Issue type**: Issue type names can be used in the quick search as long as they don't have any spaces in them. 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 the quick search to find issues that have those values as resolution. For example, `fixed`, `duplicate`, and so on.
- **Versions**: The 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 a wildcard search. The search will also find all the 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**: The quick search can find issues with component names using the prefix `c:` followed by the component name. It will retrieve all the 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 such as `jql`, `jql performance`, `advanced jql`, and so on.

There's more...

The 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. This 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 the bugs that are opened and assigned to me. It is equivalent to the following JQL line:

```
issuetype = Bug AND assignee = currentUser() AND status = Open
```

The `my open bugs jql` will retrieve all the bugs that are opened and assigned to me and has the word `jql` in its 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 `http://confluence.atlassian.com/display/JIRA/Advanced+Searching`.

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 `orderBy` clause, which are built using the `JqlClauseBuilder` class. 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 assign them to the current user within our plugin. The JQL equivalent for this is as follows:

```
project = "DEMO" AND assignee = currentUser()
```

The following are the steps to do this programmatically:

1. Create a `JqlQueryBuilder` object.

   A `JqlQueryBuilder` object is used to build the query that is used to perform issue searching. The following is how a `JqlQueryBuilder` object is created:

   ```
   JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
   ```

2. Create a `where` clause that returns a `JqlClauseBuilder` object. 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 as argument. The `project` clause will return a `ConditionBuilder` object:

   ```
   builder.where().project(10000L)
   ```

4. Add the `assignee` clause using the `and` condition on the `ConditionBuilder` object:

   ```
   builder.where().project(10000L).and().assigneeIsCurrentUser();
   ```



We can have numerous clauses added like the preceding instruction using the different conditions. More examples are discussed in the *There's More...* section of this recipe.

5. Add ordering, if you have any, using the `orderBy` clause. We can sort based on the assignee as follows:

   ```
   builder.orderBy().assignee(SortOrder.ASC);
   ```

   `SortOrder.DESC` can be used for descending orders.

6. Build the `Query` object `com.atlassian.query.Query`:

   ```
   Query query = builder.buildQuery();
   ```

   The `Query` object is immutable; once it is created it cannot be changed. The `JqlQueryBuilder` object represents the mutable version of a `Query` object. We can create a `Query` object from an already existing `Query` by calling `JqlQueryBuilder.newBuilder(existingQuery)`.

7. Get an instance of the `SearchService` class. 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 search results:

   ```
   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.



For more information on `SearchResults` go to `http://docs.atlassian.com/jira/latest/com/atlassian/jira/issue/search/SearchResults.html`.

9. Retrieve the issues from the search results:

   ```
   List<Issue> issues = results.getIssues();
   ```

   The entire method will look as follows:

   ```
   private List<Issue> getIssues(User user) {
     JqlQueryBuilder builder = JqlQueryBuilder.newBuilder();
     builder.where().project(10000L).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, this 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");
  ```

  `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. The JQL representation of the query will be as follows:

  ```
  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 function* recipe

Parsing JQL queries 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 instead 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 as we saw in the previous recipe:

```
project = "DEMO" AND assignee = currentUser()
```

The following is how we can use the JQL query for searching:

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 parsed result:

```
Query query = parseResult.getQuery()
```

4. Search for the issues and retrieve the search results, as we saw 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...

In this example, 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* recipe

Linking directly to search queries

Haven't you wondered how we can link to a query from a template, or link a 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. Get an instance of the `SearchService` class. It could be injected in the constructor of your plugin using a dependency injection, or it can be retrieved from the `ComponentAccessor` class as follows:

   ```
   SearchService searchService = ComponentAccessor.
   getComponent(SearchService.class);
   ```

3. Retrieve the query string from the `Query` object using `SearchService`, shown as follows:

   ```
   String queryString = searchService.getQueryString(user, query);
   ```

4. Construct the link using the context path. In JSPs, you can do it as shown in the following lines of code:

   ```
   <a href="<%= request.getContextPath() %>/secure/IssueNavigator.
   jspa?reset=true<ww:property value="/queryString" />&amp;mode=hide"
   title="">Show in Navigator</a>
   ```

   In this case, `getQueryString()` in the action class returns the preceding query string.

   In Velocity templates, you can construct the link as follows:

   ```
   <a href="$requestContext.baseUrl/secure/IssueNavigator.jspa?reset=
   true$queryString&amp;mode=hide" title="">Show in Navigator</a>
   ```

   In this case, `$queryString` is the preceding query string in context!

   The `mode` parameter can have the values `hide` or `show` depending on whether you want to open the Issue Navigator in view or edit mode!

How it works...

The `getQueryString` method in `SearchService` returns the query string in a manner in which it can be used in a URL. It starts with `&jqlQuery=`, followed by the actual query as a web URL.

So, `reset=true<ww:property value="/queryString" />&mode=hide` will then be `reset=true&jqlQuery=someQuery&mode=hide`.

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://<Context_Path>/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 as follows:

```
http://localhost:8080/secure/QuickSearch.jspa?searchString=my+open
```



Please note that the spaces in **Quick Search** are replaced by `+`, while substituting `%s`.

Following are a few other examples:

- The URL `http://localhost:8080/secure/QuickSearch.jspa?searchString=my+open+critical` retrieves all open critical issues assigned to you
- The URL `http://localhost:8080/secure/QuickSearch.jspa?searchString=created:-1w+my` retrieves all the issues assigned to you that were created in the past week

Index and de-index programmatically

As we have seen in the JIRA architecture explained in *Chapter 2*, *Understanding the Plugin Framework*, searching in JIRA is based on Apache Lucene. The Lucene indexes are stored in the filesystem 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 indexing off or on 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 examples.

How to do it...

Most of the indexing operations can be done with the help of `IssueIndexManager`. An instance of `IssueIndexManager` can be created either by injecting in to the constructor or by creating it as follows:

```
IssueIndexManager indexManager = ComponentAccessor.
getIssueIndexManager();
```

The following are the important operations supported by `IssueIndexManager`:

- `reIndexAll()`: This indexes all the issues in JIRA. This is a good method if you want a custom admin operation to do indexing too!
- `reIndex(Issue issue)`: This is used to selectively index an issue by passing the `Issue` object.
- `deIndex(Issue issue)`: This is a 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` interface should be overridden so that it won't index the issue again.

- `reIndexIssues(final Collection<GenericValue> issues)` or `reIndexIssueObjects(final Collection<? extends Issue>issueObjects)`: Both of these can be used to index a collection of issues.



Check out the Javadocs at `http://docs.atlassian.com/software/jira/docs/api/latest/com/atlassian/jira/issue/index/IssueIndexManager.html` for more available methods on the `IssueIndexManager` interface.

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

Managing filters programmatically

Be it 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 following aspects of managing the filters one by one:

- Creating a filter
- Updating a filter
- Deleting a filter
- Retrieving a filter
- Sharing a filter

Most of the operations on managing filters are done using `SearchRequestService`. For **creating** a filter, perform the following steps:

1. Create the `Query` object to be saved as a filter. The `Query` object can be created using `JqlQueryBuilder`, as we have seen in the earlier recipes.

2. Create a `SearchRequest` object from the query:

   ```
   SearchRequest searchRequest = new SearchRequest(query);
   ```

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);
   ```

   In this case, `user` is the user for which the filter should be created.

4. Get an instance of `SearchRequestService`. It can be either injected in the constructor or implemented as follows:

   ```
   SearchRequestService searchRequestService = ComponentAccessor.getC
   omponent(SearchRequestService.class);
   ```

5. Create the filter:

   ```
   final SearchRequest newSearchRequest = searchRequestService.
   createFilter(ctx, searchRequest, favourite);
   ```

   In this case, `favourite` is a Boolean which can be set to true if you want the filter to be made a favorite.

Updating a filter is very much similar to creating a filter. Once the search request is updated and the context is created, we need to invoke the following method to update and persist the filter with the new search parameters in the database, that is, the new query:

```
SearchRequest updatedSearchRequest = searchRequestService.
updateSearchParameters
(JiraServiceContext serviceCtx, SearchRequest request);
```

To update the attributes, such as name, description, and so on, one of the following methods is invoked depending on whether we want to make the filter a favorite or not:

```
SearchRequest updatedFilter = searchRequestService.
updateFilter(JiraServiceContext serviceCtx, SearchRequest request);
```

Alternatively, we can use the following:

```
SearchRequest updatedFilter = searchRequestService.
updateFilter(JiraServiceContext serviceCtx, SearchRequest request,
boolean isFavourite);
```

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 using the following instruction:

```
deleteAllFiltersForUser(JiraServiceContext serviceCtx, User user);
```

The `SearchRequestService` interface also has a few methods to **retrieve** favorite filters, filters owned by a user, non-private filters, and so on. A few key methods are listed as follows:

```
Collection<SearchRequest> getFavouriteFilters(User user);
Collection<SearchRequest> getOwnedFilters(User user);
Collection<SearchRequest> getNonPrivateFilters(User user);
Collection<SearchRequest> getFiltersFavouritedByOthers(User user);
```

The method names are self-explanatory.

In order to **share** a filter, we need to retrieve the relevant filter and set the permissions on it using the following instruction:

```
searchRequest.setPermissions(permissions);
```

In this case, `permissions` is a set of `SharePermission` objects. The `SharePermission` objects can be created from a JSONArray using the `SharePermissionUtils` utility class. A JSONObject can have three keys: `Type`, `Param1`, and `Param2`.

The `Type` key can have the following values: `global`, `group`, or `project`. The relationship among these values are listed as follows:

- When `Type` is `global`, `Param1` and `Param2` are not required
- When it is `group`, `Param1` is populated with the group name
- When it is `project`, `Param1` is the ID of the project and `Param2` is the ID of the project role

Examples of JSON arrays are as follows:

```
[{"type":"global"}]
[{"type":"group","param1":"jira-administrators"},{"type":"project","pa
ram1":"10000","param2":"10010"}]
```

See also

- The *Searching in plugins* recipe

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:

- **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` to `59`	`, - * /`
`Minute`	`0` to `59`	`, - * /`
`Hour`	`0` to `23`	`, - * /`
`Day-of-month`	`1` to `31`	`, - * / ? L W C`
`Month`	`1` to `12` or `JAN` to `DEC`	`, - * /`
`Day-of-week`	`1` to `7` or `SUN` to `SAT`	`, - * / ? L C #`
`Year` (optional)	`1970` to `2099`	`, - * /`

  The special characters denote the following:

Special character	Usage
`,`	This specifies a list of values. For example, `MON,WED,FRI` means "every Monday, Wednesday, and Friday".
`-`	This specifies a range of values. For example, `MON-WED` means "every Monday, Tuesday, Wednesday".
`*`	This specifies all the possible values. For example, `*` in the `Hour` field means "every hour of the day".

Special character	Usage
/	This specifies the increments to the given value. For example, `1/3` in the `Hour` field means "every 3 hours during the day, starting from 1.00 A.M.".
?	This specifies 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	This specifies the 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	This specifies the weekday (`MON` to `FRI`) nearest to the given day of the month. For example, `1W` means "nearest working day to the first 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.
#	This specifies the nth occurrence of a given day of the week. For example, `MON#3` means "third 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 A.M. every day
- `0 0/15 15 * * ?`: Every 15 minutes starting at 3.00 P.M. ending at 3:59 P.M.



You can find more examples in the Atlassian documentation for filter subscriptions at `http://confluence.atlassian.com/display/JIRA/Receiving+Search+Results+via+Email`.

- **Group name**: This is the group that we want to subscribe the filter to. If the value is null, it will be considered as a personal subscription and the user in the context will be used.
- **E-mail on empty**: This is a Boolean value, which is set to `true` if you want the subscription to send an e-mail even when it has no results.

Now let us look at the steps to subscribe to a known filter:

1. Get an instance of the `FilterSubscriptionService` class. You can either inject the class in to 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);
   ```

   In this case, `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, even when the number of results is zero or not:

   ```
   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);
   }
   ```

   In this case, `filterId` is the ID of the filter we want to subscribe to, and it 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 with the following method:

```
filterSubscriptionService.updateSubscription(ctx, subId, groupName,
cronExpression, emailOnEmpty);
```

In this case, `subId` is the existing subscription ID.

How it works...

Each subscription that we create is stored as **Quartz** scheduled jobs in the system, which runs 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` class using the parameters from the web form. The various attributes supported in the form can be found from the `CronEditorBean` class. Once the `CronEditorBean` class is created, it can be parsed into a cron expression as follows:

```
String cronExpression = new CronExpressionGenerator().
getCronExpressionFromInput
(cronEditorBean);
```



The Javadocs for `CronEditorBean` can be found at `http://docs.atlassian.com/software/jira/docs/api/latest/com/atlassian/jira/web/component/cron/CronEditorBean.html`.

See also

- The *Searching in plugins* recipe

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
- Discarding fields while cloning
- JavaScript tricks on issue fields
- Creating issues and comments from an e-mail

Introduction

We have so far seen how to develop custom fields, workflows, reports and gadgets, JQL functions, and other pluggable things associated with them. In this chapter, we will learn about programming issues, that is, creating, editing, or deleting issues, creating new issue operations, managing the various other operations available on issues via JIRA APIs, and so on.

Creating an issue from a plugin

In this recipe, we will see how to programmatically create an issue from a plugin. Prior to Version 4.1, JIRA used `IssueManager` to create an issue. From JIRA 4.1, there is this `IssueService` class that drives the issue operations. Since `IssueService` is recommended over `IssueManager`, we will use it in our recipes to create an issue.

How to do it...

The main advantage of using `IssueService` over the `IssueManager` class is that it takes care of the validation and error handling. The following are the steps to create an issue using the `IssueService` class:

1. Create an instance of the `IssueService` class. You can either inject it in the constructor or get it from the `ComponentAccessor` class, 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 = new
      IssueInputParametersImpl();
      ```

   b. Populate the `IssueInputParameters` class with the values required to create the issue as shown in the next few lines of code:

      ```
      issueInputParameters.setProjectId(10000L).
      setIssueTypeId("5").setSummary("Test Summary").
      setReporterId("admin").setAssigneeId("admin").
      setDescription("Test Description").setStatusId("1").
      setPriorityId("2").setFixVersionIds(10000L, 12121L);
      ```

   c. Make sure all the required values such as project, issue type, summary, and other mandatory values required when the issue is created using the user interface, are set on the `IssueInputParameters` class.

   d. Here, we have used test values, but make sure to replace them with appropriate values. For example, the project, issue type ID, priority ID, Fix version IDs, reporter, and assignee 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` variable 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` object will have all non-field-specific errors such as permission issue-related errors, and so on, but any field-specific errors, such as 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 whether 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, if you 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);
  }
}
```

Here again, we just print the error to the console.

6. If `createResult` is valid, then the issue is created successfully and you can retrieve it as:

```
MutableIssue issue = createResult.getIssue();
```

How it works...

By using `IssueService`, JIRA now validates the inputs we give it using the rules we have set up in JIRA via the user interfaces, such as the mandatory fields, permission checks, individual field validations, and so on. Behind the scenes, it still uses the `IssueManager` class.

There's more...

As mentioned before, prior to JIRA 4.1, we needed to use the `IssueManager` class to create the issues. It can still be used in JIRA 4.1 and higher, but this is not recommended as it overrides all the validations.

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`.

Using IssueManager to create the issue

Follow these steps to create an issue with the `IssueManager` class:

1. Initialize an issue object using the `IssueFactory` class:

```
MutableIssue issue = ComponentAccessor.getIssueFactory().
getIssue();
```

2. Set 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 catch 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 the 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 the progress on all its subtasks, although people use it for a lot of other purposes too.

How to do it...

There are two steps to 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 these steps in more detail:

1. Create the subtask issue object similar to how we created the issue in the previous recipe. Here, the `IssueInputParameters` interface is constructed (after changing the methods such as `setIssueTypeId()` appropriately).

   For this issue, we will use the `validateSubTaskCreate` method—which takes an extra parameter named `parentId`— instead of `validateCreate`:

   ```
   CreateValidationResult createValidationResult = issueService.
   validateSubTaskCreate(user, parent.getId(), issueInputParameters);
   ```

   Here, `parent` is the issue object on which we are creating the subtask.

2. Create an issue after checking for errors, as we have seen previously:

   ```
   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` as follows:

      ```
      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.

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, 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 option 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 this:

1. Create the `IssueInputParameters` object with the input fields that need to be modified:

```
IssueInputParameters issueInputParameters = new
IssueInputParametersImpl();
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.setRetainExistingValuesWhenParameterNotProvid
ed(false);
```

2. Validate the input parameters using `IssueService`:

```
UpdateValidationResult updateValidationResult = issueService.
validateUpdate(user, issue.getId(), issueInputParameters);
```

   Here, `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 when creating the issue.

4. Validate `updateResult` and handle the error, if any. If it is not valid, the updated issue object can be retrieved as:

```
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 this:

1. Validate the delete operation on the issue using `IssueService`:

```
DeleteValidationResult deleteValidationResult = issueService.
validateDelete(user, issue.getId());
```

   Here, `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 `deleteValidationResult` is invalid, handle the errors appropriately.
4. Confirm whether the deletion was successful by checking the `deleteErrors` collection:

```
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 **Edit Issue**, **Clone Issue**, 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 **Web Item 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.

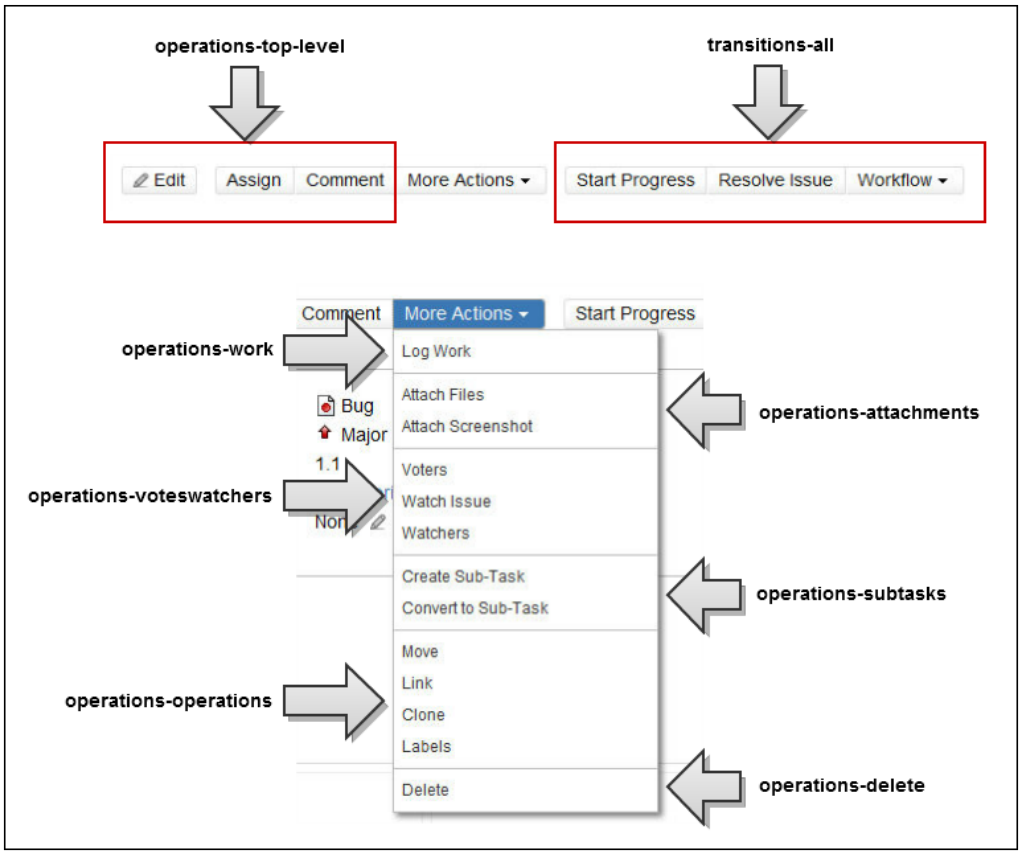
How to do it...

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 **Web Section** 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 these sections. The following is a diagram from the Atlassian documentation detailing with each of the available web sections for the issue operations:



2. 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 key="manage-project" name="Manage Project"
section="operations-operations" weight="100">
  <label>Manage Project</label>
  <tooltip>Manages the Project  in which the issue belongs </
tooltip>
  <link linkId="manage-project-link">/plugins/servlet/project-
config/${issue.project.key}</link>
</web-item>
```

As you can see, this 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.

`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, the `link` attribute, is the most important one, as it 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.

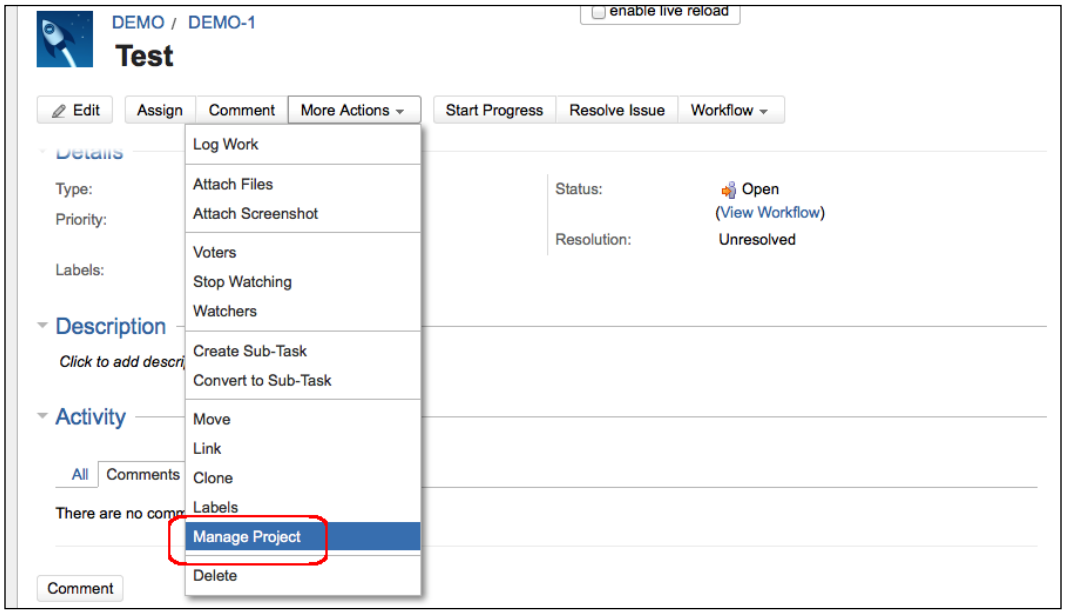
In our example, we need to take the user to the administer project area. Luckily, in this case, we know the servlet to be invoked, as it is an existing servlet in JIRA. All we need to do is to invoke the `project-config` servlet by passing the project key as well. The issue object is available on the 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 **View Issue** 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 have noticed, 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 **Issue Operations** module was used in the creation of new issue operations. This is outside the scope of this book, though you can find details about it in the Atlassian documentation at `http://confluence.atlassian.com/display/JIRADEV/Issue+Operations+Plugin+Module`.

See also

- The *Extending a Webwork action in JIRA* recipe in *Chapter 2*, *Understanding the Plugin Framework*

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 else. JIRA allows conditions to be added while defining the web items, and when the conditions are not satisfied, the web item won't show up!

In this recipe, we will lock down the new issue operation we created in the previous recipe to Project Administrators exclusively.

Getting ready...

Create the **Manage Project** 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. The class should implement the `com.atlassian.plugin.web.Condition` interface, but it is recommended to extend `com.atlassian.jira.plugin.webfragment.conditions.AbstractIssueCondition` when creating an issue condition.

While extending `AbstractIssueCondition`, we will have to implement the `shouldDisplay` method, as shown here:

```
public class AdminCondition extends AbstractIssueCondition {
  private final PermissionManager permissionManager;

  public AdminCondition(PermissionManager permissionManager) {
    this.permissionManager = permissionManager;
  }

  @Override
  public boolean shouldDisplay(User user, Issue issue, JiraHelper
jiraHelper) {
        return this.permissionManager.hasPermission(Permissions.
PROJECT_ADMIN, issue.getProjectObject(), user);
  }
}
```

Here, a `true` value is returned if the user has the `PROJECT_ADMIN` permission on the project. That is all we need for the `condition` class.

2. Include the `condition` class in the web item:

```
<web-item key="manage-project" name="Manage Project"
section="operations-operations" weight="100">
  <label>Manage Project</label>
  <tooltip>Manages the Project in which the issue belongs </
tooltip>
  <link linkId="manage-project-link">
    /secure/project/ViewProject.jspa?pid=${issue.project.id}
  </link>
  <condition class="com.jtricks.conditions.AdminCondition"/>
</web-item>
```

It is possible to invert a condition by using the `invert` flag, as shown:

```
<condition class="com.jtricks.conditions.AdminCondition"
invert="true"/>
```

Once inverted, this condition will check if the user is an admin and will return true only for non-admin users.

Condition elements can also take optional parameters, as shown:

```
<condition class="com.atlassian.jira.plugin.webfragment.
conditions.JiraGlobalPermissionCondition">
   <param name="permission">sysadmin</param>
</condition>
```

The parameters can be retrieved in the `condition` class by overriding the `init(Map params)` method. In this case, `params` is a map of string key/value pairs that hold these parameters, in which case, the Map will have permission as the key, and the value passed (`sysadmin` in our example) can be accessed using the key and can then be used in passing or failing the condition.

For example, the following code in the `conditions` class will get you the appropriate permission type:

```
int permission = Permissions.getType((String) params.
get("permission"));
// Permissions.SYSTEM_ADMIN in this case
```

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 to both Project Administrators as well JIRA System Administrators, we can do so using an `OR` condition, as shown here:

```
<conditions type="OR">
  <condition class="com.atlassian.jira.plugin.webfragment.
conditions.JiraGlobalPermissionCondition">
  <param name="permission">sysadmin</param>
  </condition>
  <condition class="com.jtricks.conditions.AdminCondition"/>
</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 a Project Administrator (or a JIRA system admin, depending on which condition you used), you will see the operation. If the user doesn't have the permissions, 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, namely **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` interface can be retrieved either by injecting it in to the constructor or from the `ComponentAccessor` class, as shown here:

```
AttachmentManager attachmentManager = ComponentAccessor.
getAttachmentManager();
```

Creating an attachment

An attachment can be created on an issue using the `createAttachment` method on the `AttachmentManager` interface, as shown:

```
ChangeItemBean changeBean = attachmentManager.createAttachment(new
File(fileName), newFileName, "text/plain", user, issue);
```

The following are the arguments:

- The `fileName` value 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.
- `newFileName` is the name with which the file will be attached to the issue, and it can be different from the original filename.
- 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`.
- `user` is the user who is attaching the file.
- `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 and create the attachment at a specific time, it can be done using the overloaded method `createAttachment`, which takes two extra parameters—`attachmentProperties`, which is a Map containing the key/value properties, and `createdTime`, which is of the type `java.util.Date`.

Such attachment properties will be stored in the database using `PropertySet`.

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.getAuthor());
}
```

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 such as `attachmentsEnabled`, `isScreenshotAppletEnabled()`, `isScreenshotAppletSupportedByOS()`, and so on, to check whether the respective functionality is 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 you 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 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 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 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, such as creating worklogs, updating them, or deleting them, and it does this in all the four different modes we have seen previously.

We will look at 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).groupLevel(null).
   roleLevelId(null);
   ```

   Here, the 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` here can be the date from when the work has started. We can also optionally add comments and set the worklog visibility to certain groups or project roles! Set these parameters as null when the worklog is visible to all.

3. Create the `WorklogInputParameters` object from the builder and validate it using the `WorklogService` class:

   ```
   WorklogResult result = this.worklogService.validateCreate(jiraServ
   iceContext, builder.build());
   ```

4. Create the worklog using `WorklogService`:

   ```
   Worklog worklog = this.worklogService.createAndAutoAdjustRemaining
   Estimate(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 `WorklogResult` object 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 previously. The only difference is that the method invoked on `WorklogService` is `createAndRetainRemainingEstimate` instead of `createAndAutoAdjustRemainingEstimate`.

The full code is as shown here:

```
JiraServiceContext jiraServiceContext = new
JiraServiceContextImpl(user);
final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue).timeSpent(timeSpent).
startDate(new Date()).comment(null).groupLevel(null).
roleLevelId(null);
WorklogResult result = this.worklogService.validateCreate(jiraServiceC
ontext, 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 previously 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).groupLevel(null).
   roleLevelId(null);
   ```

3. Create the New Estimate Input Parameters from the Builder object:

   ```
   final WorklogNewEstimateInputParameters params = builder.
   newEstimate(newEstimate).buildNewEstimate();
   ```

   Here, we specify `newEstimate`, which is a String representation similar to `timeSpent`. `newEstimate` and will be set as the remaining estimate on the issue.

4. Create the `WorklogResult` interface from `WorklogNewEstimateInputParameters` using `WorklogService`:

```
WorklogResult result = this.worklogService.validateUpdateWithNewEs
timate(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.createWithNewRemainingEstima
te(jiraServiceContext, (WorklogNewEstimateResult) result, false);
```

   Here, the method used is `createWithNewRemainingEstimate`, which sets `newEstimate` as the remaining estimate on the issue after logging the work using `timeSpent`! As you can see, the result object is converted to `WorklogNewEstimateResult`.

Logging work and adjusting the remaining estimate by a value

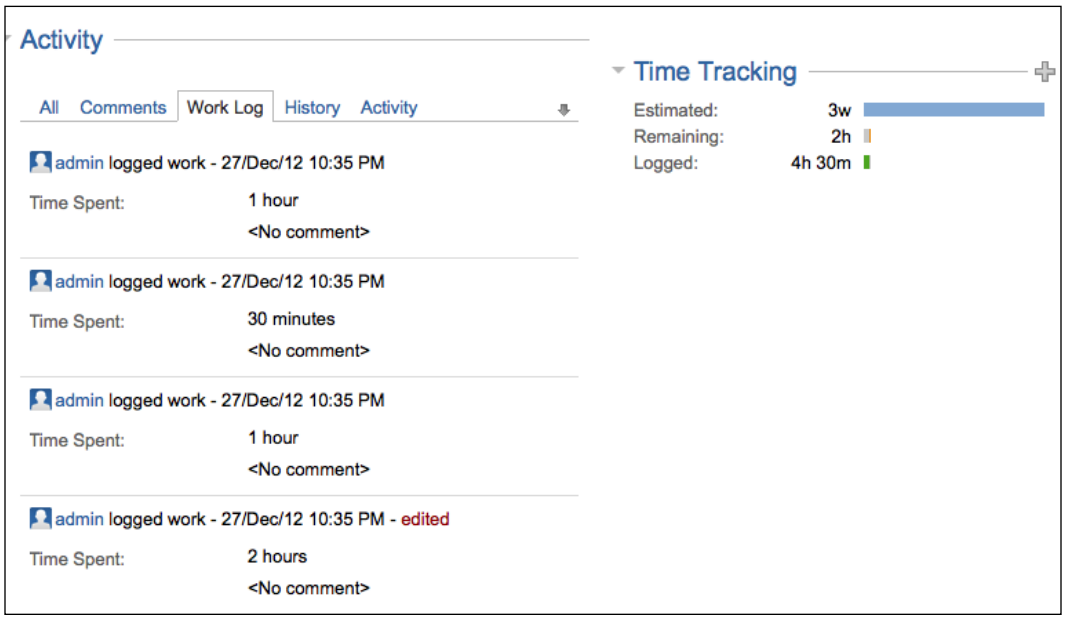
Here the process is much similar to the preceding method. The only difference is that the `adjustmentAmount` method is used on Builder instead of `newEstimate` and `validateCreateWithManuallyAdjustedEstimate` is used on `WorklogService` to create the worklog. Also, `WorklogResult` is an instance of `WorklogAdjustmentAmountResult`.

The code is as follows:

```
JiraServiceContext jiraServiceContext = new
JiraServiceContextImpl(user);
final WorklogInputParametersImpl.Builder builder =
WorklogInputParametersImpl.issue(issue).timeSpent(timeSpent).
startDate(new Date()).comment(null).groupLevel(null).
roleLevelId(null);
final WorklogAdjustmentAmountInputParameters params = builder.adjustme
ntAmount(estimateToReduce).buildAdjustmentAmount();
WorklogResult result = worklogService.validateCreateWithManuallyAdjust
edEstimate(jiraServiceContext, params);
Worklog worklog = this.worklogService.createWithManuallyAdjustedEstima
te(jiraServiceContext, (WorklogAdjustmentAmountResult) result, false);
```

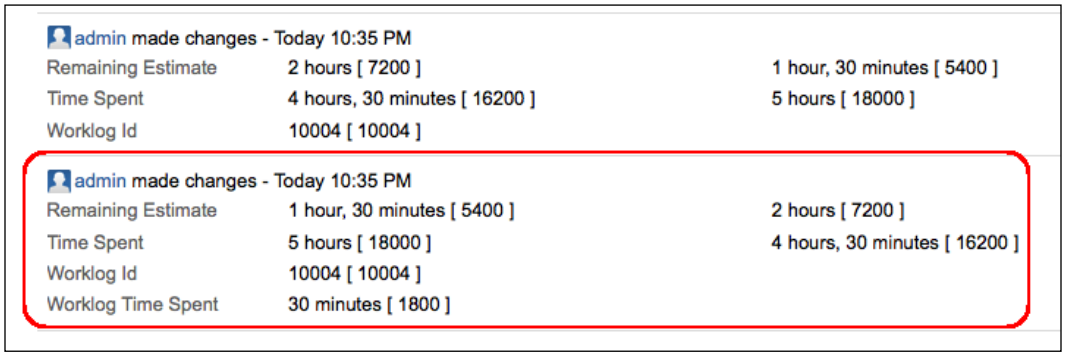
How it works...

Once we create or update the worklogs using the `WorklogService` API, the changes will be reflected on the issue under the **Work Log** 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 under the **Change History** as shown here:



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 when creating a worklog. Also, 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).
groupLevel(null).roleLevelId(null);
WorklogResult result = this.worklogService.validateUpdate(jiraServiceC
ontext, builder.build());
Worklog updatedLog = this.worklogService.updateAndAutoAdjustRemainingE
stimate(jiraServiceContext, result, false);
```

As you can see, a builder is created by passing the worklog ID, which is unique across issues. `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).
groupLevel(null).roleLevelId(null);
final WorklogNewEstimateInputParameters params = builder.
newEstimate(newEstimate).buildNewEstimate();
WorklogResult result = this.worklogService.validateUpdateWithNewEstima
te(jiraServiceContext, params);
Worklog updatedLog = this.worklogService.updateWithNewRemainingEstimat
e(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 the 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(jiraServic
eContext, worklog.getId());
worklogService.deleteAndAutoAdjustRemainingEstimate(jiraServiceConte
xt, 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, except that the `deleteAndRetainRemainingEstimate` method is used instead of `deleteAndAutoAdjustRemainingEstimate`:

```
JiraServiceContext jiraServiceContext = new
JiraServiceContextImpl(user);
WorklogResult worklogResult = worklogService.validateDelete(jiraServic
eContext, 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, while validating, `newEstimate` is used to create `WorklogResult`, which is an instance of `WorklogNewEstimateResult`. The code is as follows:

```
JiraServiceContext jiraServiceContext = new
JiraServiceContextImpl(user);
WorklogResult worklogResult = worklogService.validateDeleteWithNewEsti
mate(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.validateDeleteWit
hManuallyAdjustedEstimate(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 all of 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 as follows:

```
Comment comment = this.commentService.create(user, issue,
commentString, false, new SimpleErrorCollection());
```

Here, `commentString` is the comment we are adding, user is the user adding the comment, and `issue` is the issue on which the comment is added. The fourth argument is a Boolean that determines whether an event should be dispatched or not. If it is true, an `Issue Commented` event is thrown.

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 overridden `create` method on the `CommentService` class that takes the role ID and group name along with the other attributes. Only one of them should be passed at one time.

In order to restrict the comment visibility to groups, the `Comment Visibility` property under **General Configuration** should be set to **Groups & Project Roles**. The default is to allow restricting comments only for project roles.

For example, the comment can be restricted to a **group** as follows:

```
Comment comment = this.commentService.create(user, issue,
commentString, group, null, false, new SimpleErrorCollection());
```

In this group, `group` is the name of the group, and the fifth parameter (`null`) is `roleId`.

Restricting to a **role** is done as follows:

```
Comment comment = this.commentService.create(user, issue,
commentString, null, roleId, false, new SimpleErrorCollection());
```

In this case, `group` is null and `roleId` is the unique ID of the `ProjectRole` that we need to restrict the comment to.

The Boolean to dispatch events can be used in both cases.

Updating comments

Following are the steps to update a comment:

1. Create the `MutableComment` object from the comment to be updated:

   ```
   MutableComment comm = this.commentService.getMutableComment(user,
   comment.getId(), new SimpleErrorCollection());
   ```

2. Modify the comment with the following statement:

   ```
   comm.setBody("New Comment");
   ```

   Here, we update the body of the comment, though we can also update other attributes such as the author, group level, role level, and so on.

3. Update the comment using `CommentService`:

   ```
   this.commentService.update(user, comm, false, new
   SimpleErrorCollection());
   ```

Deleting comments

A comment can be deleted as shown here:

```
this.commentService.delete(new JiraServiceContextImpl(user), comment,
false);
```

Here, `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`. A `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 `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 the `IssueFieldConstants` class. For example, `IssueFieldConstants.SUMMARY` represents the issue summary.

   The third and fourth arguments are the old value and the new value 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` value 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 change log using the items in the `changeHolder` object using the `ChangeLogUtils` class:

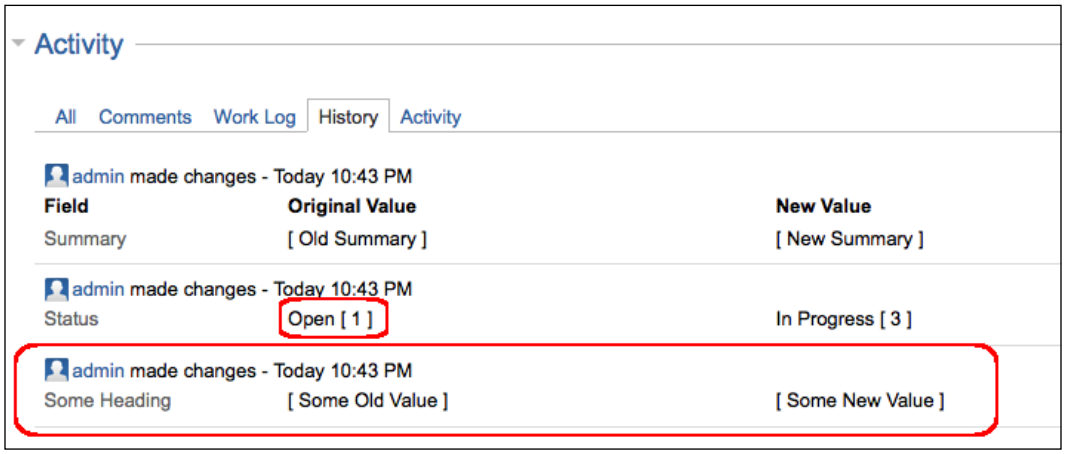
```
GenericValue changeLog = ChangeLogUtils.createChangeGroup(user,
issue, issue, changeHolder.getChangeItems(),  false);
```

Here, `user` is the user making the change. The second and third arguments are the original issue and the issue after the changes. You can give both as 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 change log 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 the inward links.

Getting Ready

Make sure the **Issue Linking** feature is turned **ON** in JIRA and the 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 to the constructor:

```
IssueLinkTypeManager issueLinkTypeManager = ComponentAccessor.getC
omponent(IssueLinkTypeManager.class);
IssueLinkType linkType = issueLinkTypeManager.getIssueLinkTypesByN
ame("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 to 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`, and is 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 links or just display them.

Deleting issue links

Following are the steps to do this:

1. Retrieve the `IssueLinkType`, as we did earlier:

```
IssueLinkTypeManager issueLinkTypeManager = ComponentAccessor.getC
omponent(IssueLinkTypeManager.class);
IssueLinkType linkType = issueLinkTypeManager.getIssueLinkTypesByN
ame("Duplicate").iterator().next();
```

2. Get the issue link to be deleted using the `IssueLinkManager` class:

```
IssueLink issueLink = issueLinkManager.getIssueLink(sourceIssue.
getId(), destIssue.getId(), linkType.getId());
```

   Here, the `sourceIssue` and `destIssue` parameters represent 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();
```

Discarding fields while cloning

Cloning of issues in JIRA is an easy way to replicate an existing issue. While cloning, a new issue is created exactly similar to the original issue with identical values for all its fields, except for a few special ones. The special ones include `created date`, `updated date`, `issue key`, `status`, and so on.

But, in addition to the special fields JIRA has chosen, we might want to ignore a few other fields while cloning an issue. How about a unique custom field? We surely don't want to replicate that while cloning?

Here is an easy way to discard any such fields while cloning an issue.

Getting ready...

Create a Skeleton plugin using the Atlassian Plugin SDK. It is recommended to read the *Extending JIRA actions* recipe before proceeding.

How to do it...

As we have seen while extending JIRA actions in the second chapter, all we need to do here is to create a new webwork action that extends the existing JIRA clone action and overrides the required method. In this specific case, we will be overriding the `setFields()` method to remove the cloning of the specific fields we are interested in!

Let us, for example, say that we want to avoid cloning a unique number field named `Test Number`. Following are the steps to do this:

1. Add a new webwork module in the `atlassian-plugin.xml` file with a new action class and the same alias as JIRA's clone action, `CloneIssueDetails`. Once we do that, the new action class will be executed while cloning issues:

```
<webwork1 key=" jtricks-clone-issue-details" name="JTricks Clone
Issue Details" >
  <actions>
    <action name="com.jtricks.JTricksCloneIssueDetails"
alias="CloneIssueDetails">
      <view name="input">/secure/views/cloneissue-start.jsp</view>
      <view name="error">/secure/views/cloneissue-start.jsp</view>
    </action>
  </actions>
</webwork1>
```

2. Create the new class by extending the existing action class:

```
public class JTricksCloneIssueDetails extends CloneIssueDetails{
  ...
}
```

3. Override the `setFields()` method to set a null value for the fields we do not want to clone:

```
@Override
protected void setFields() throws FieldLayoutStorageException {
  super.setFields();
  // Set null values for interested fields here
}
```

4. Add the code to set null values. In our example, we set a null value for the `Test Number` custom field:

```
CustomField customField = customFieldManager.getCustomFieldObjectB
yName("Test Number");
getIssueObject().setCustomFieldValue(customField, null);
```

   Here we get the cloned issue using the `getIssueObject` method and set the null value for the custom field. Don't forget to use the `getCustomFieldObject` method by passing the custom field ID if the field name is not unique!

   If we want to set null values for a system field, such as fix for versions, the method is the same:

```
getIssueObject().setFixVersions(null);
```

5. Package the plugin and deploy it.



An action can be overridden only once. Care must be taken not to override it again in another plugin (this might be a third-party plugin) as only one will be picked up.

How it works...

Once the clone operation is invoked, the new action we have created will be executed. The clone operation creates a new issue object and copies the values to its fields from the original issue. This is done in the `setFields` method.

It is only logical to override this method and set null values for fields we do not want to clone. As shown previously, the `setFields` method from the super class, which is the JIRA's built-in class, is executed first. Once this method is executed, the new issue object, which can be retrieved using the `getIssueObject` method, has all the values populated. We just reset some of the values by setting them to null.

See also

- The *Extending a webwork action in JIRA* recipe in *Chapter 2*, *Understanding the Plugin Framework*

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 some 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 built-in option to do so.

In that case, 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 need developing a plugin—is to drive this using JavaScript. To make things better, JIRA offers a 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 it 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 **Why Critical?**. The field should be shown only if the priority of the issue is critical.

Following are the simple steps to achieve this using JavaScript:

1. Write the JavaScript to achieve the functionality.

   In our example, we need to show the **Why Critical?** field only when the priority is critical. 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 also 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">
(function($){
  $(document).ready(function(){
    var priority = document.getElementById('priority');
    hideOrShow(priority.value);
    priority.onchange=function() {
      hideOrShow(priority.value);
    };
  });

  function hideOrShow(priorityVal){
    if (priorityVal == '2'){
      AJS.$("#customfield_10170").closest('div.field-
group').show();
    } else {
      AJS.$("#customfield_10170").closest('div.field-
group').hide();
    }
  }
})(AJS.$);
</script>
```

d. Here, `10170` is the ID of the custom field, and hence `customfield_10170` represents the unique custom field ID! Also, `2` is the ID of the priority system field.

In this example, we created a page load event where the script looks at the priority value and sets the visibility of the `div` element surrounding the custom field as hidden or shown.

The following part captures the on load event of the page where the custom field is in edit mode:

```
(function($){
$(document).ready(function(){
    ...
  });
})(AJS.$);
```

The following code shows the field, if the priority is `2`:

```
AJS.$("#customfield_10170").closest('div.field-group').
show();
```

For every other priority value, the `closest` div surrounding field is hidden.

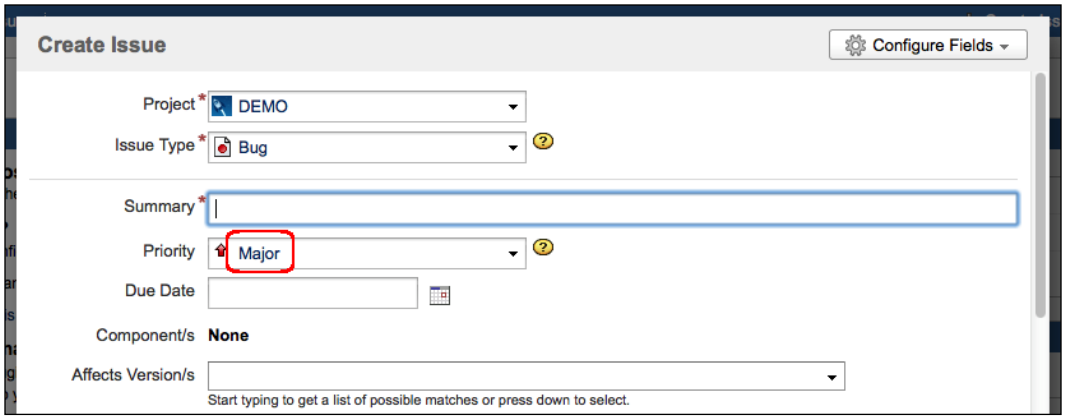
2. Add the preceding JavaScript to the description of the custom field.

The field behavior will be effective on the next reload after the JavaScript is added on to the field description.

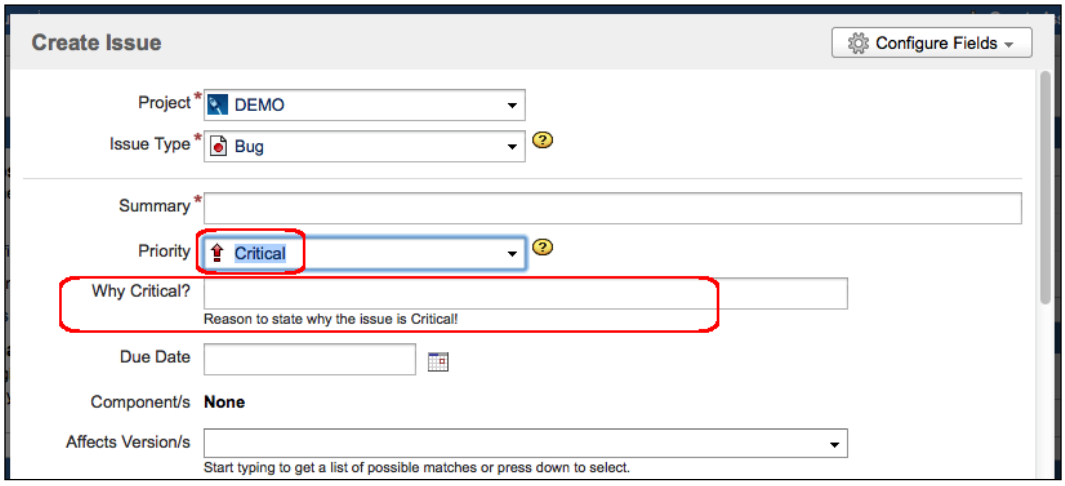
How it works...

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 critical, as shown in the next screenshot:



Here, the priority is **Major** (value 3), and hence the **Why Critical?** field is not available. But the moment the priority is changed to **Critical**, 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 an 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 works on issues raised, such as this one!

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 that 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-mails:

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 filesystem. JIRA will periodically scan this mailbox and create issues or comments based on the e-mail.
2. Navigate to JIRA's **Administration** | **System** | **Mail** | **Incoming Mail**.
3. Click on the **Add POP / IMAP mail server** button.
4. Enter the details for the POP or IMAP mail server created in Step 1 and click on **Add**.
5. Click on the **Add incoming mail handler** button:
    - **Name**: The name of the mail handler.
    - **Server**: Pick one from the servers configured above or select the **Local Files** option for an external mail service that writes messages to the filesystem.
    - **Delay**: Choose a delay for the handler to run and scan the mails.
    - **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 **Create a new issue** or add a comment to an existing issue handler.

- **Folder Name**: For IMAP Server, specify the folder name if it is a folder other than the 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, as this is 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:
   - **Project**: Project where the issue should be created.
   - **Issue Type**: Type of the issue to be created. For example, if we want the issue to be created as a bug, select **Bug**.
   - **Strip Quotes**: If enabled, this strips previous messages from the replies.
   - **Catch Email Address**: If this is 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.
   - **Bulk**: Determines how to handle "bulk" e-mails. The possible options are:
     - `ignore`: Ignore the e-mail and do nothing.
     - `forward`: Forward the e-mail to the address set in the **Forward Email** text field.
     - `delete`: Delete the e-mail permanently.
     - `accept`: Accept the e-mail for processing.
   - **Forward Email**: Error notifications and unhandled e-mails (used in conjunction with bulk forward handle parameter) will be forwarded to this address.
   - **Create Users**: If this is set to true, accounts will be created for new senders. This option is not compatible with the **Default Reporter** option.
   - **Default Reporter**: Can be used to create an 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.
   - **Notify Users**: Only used if **Create Users** is checked. It indicates whether users should get a e-mail notification for the new accounts created.
   - **CC Assignee**: If this is set, the new issue will be assigned to a matching user in the **To**, **Cc**, or the **Bcc** field in the given order depending on where the user is matched.
   - **CC Watchers**: If this is 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 e-mails to the newly added mailbox.

How it works...

The handler we have 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 had 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 configured in the handle parameters. The e-mail subject will become the issue summary and e-mail body will become 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 while creation, such as custom field values. Some of them have far better filtering mechanisms as well.

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
- 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
- Creating the project tab panel using fragments
- Adding new tabs in the Browse Version screen
- Adding new tabs in the Browse Component screen
- Adding issue link renderers
- Extending a webwork action to add UI elements
- Displaying dynamic notifications/warnings on issues
- Re-ordering issue operations in the View Issue page
- Re-ordering fields in the View Issue page

Introduction

One of the good things about JIRA is that it has a simple but powerful user interface. A lot has changed between 3.13.*x* and 5.*x* in terms of the user interface, but it continues to keep 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 more . While it is the same 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 UI without actually modifying the core JIRA files or with little modification to them.



If we modify the JIRA files, it should be noted that maintaining the changes over various JIRA versions, that is, during upgrades, would be more difficult than usual.

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, coloring scheme, and so on, with some simple configurations. In this recipe, we will see some examples of how easy it is to make those changes.

There are four main things that can be configured to change JIRA's appearance:

- **Logo**: Understandably, this is one thing everyone wants to change.
- **Colors**: JIRA has a nice theme of colors revolving around a theme of blue. But we can easily change these colors to suit our taste, or rather the company's taste!

- **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 a simple configuration.
- **Date and time formats**: The date and time formats in JIRA could be modified easily to suit our needs, provided they are in a valid format supported by Java's `SimpleDateFormat` (`http://download.oracle.com/javase/1.4.2/docs/api/java/text/SimpleDateFormat.html`).

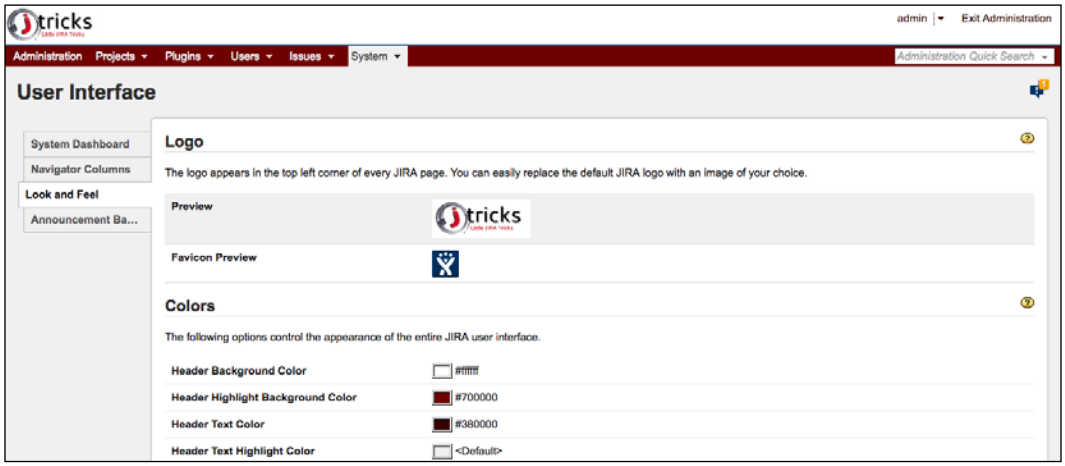
How to do it...

The following are the steps to make changes to the basic JIRA look and feel:

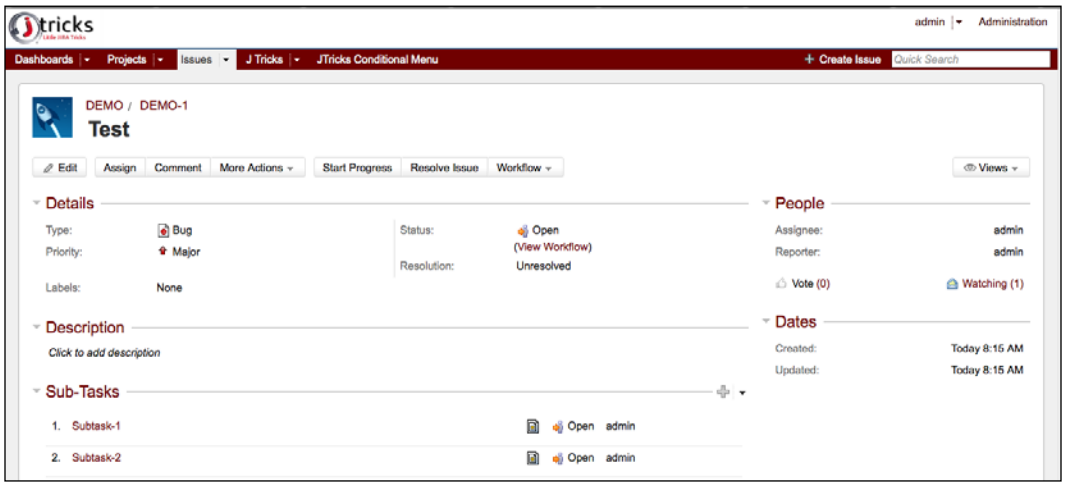
1. Log in to JIRA as an administrator.
2. Navigate to **Administration** | **System** | **User Interface** | **Look and Feel**.
3. Click on **Edit Configuration**.
4. Make the appropriate changes:
    - **Logo**: You can either upload your new image or provide a valid URL to the image that is uploaded elsewhere. This can be done for the favicon as well.
    - **Colors**: Specify the hexadecimal notations (HEX values) of the interested colors if the color scheme needs to be changed.
    - **Gadget Colors**: Again, specify the hexadecimal notations (HEX values) of the interested colors so that gadget users can pick from the new set of colors.
    - **Date and Time formats**: Enter the new date and time formats, provided it is a valid format supported by Java's `SimpleDateFormat` (`http://download.oracle.com/javase/1.4.2/docs/api/java/text/SimpleDateFormat.html`).
5. Click on **Update**.

Repeat the cycle until the desired result is achieved. We can always get back to the defaults by clicking on **Reset Defaults** while editing the configurations.

With some simple changes, the JIRA UI can look a lot different from how it normally looks. The following screenshot is a small example of it:



In this case, the **View Issue** page will look like the following screenshot:



This is a small, yet powerful change!

Adding new web sections in the UI

A web *fragment* is a link or a section of links at a particular location of the JIRA web interface. It can be a menu in the JIRA's top navigation bar, a new set of issue operations, or a new section in the **Administration** UI section.

There are two types of plugin modules to add new web fragments in JIRA, namely, the **Web Section** plugin module and the **Web Item** plugin module. A web section is a collection of links that are displayed together at 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...

The following are the steps to add a new web section into 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` file:

```
<web-section key="jtricks-admin-section" name="JTricks Section"
location="admin_plugins_menu" weight="10">
      <label>J Tricks</label>
</web-section>
```

3. As with all other plugin modules, it has a unique module key. Here, the two other important attributes of the `web-section` element are `location` and `weight`. `location` defines the location in UI where the section should appear and `weight` defines the order in which it should appear.

   In the previous example, `location` is `admin_plugins_menu`, which will create a new web section under the **Administration** | **Plugins** menu, just like the existing sections: **Plugins**, **Application Links**, and so on.

4. 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 UI. The `resource` element can be used to include resource files such as JavaScript or CSS 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:

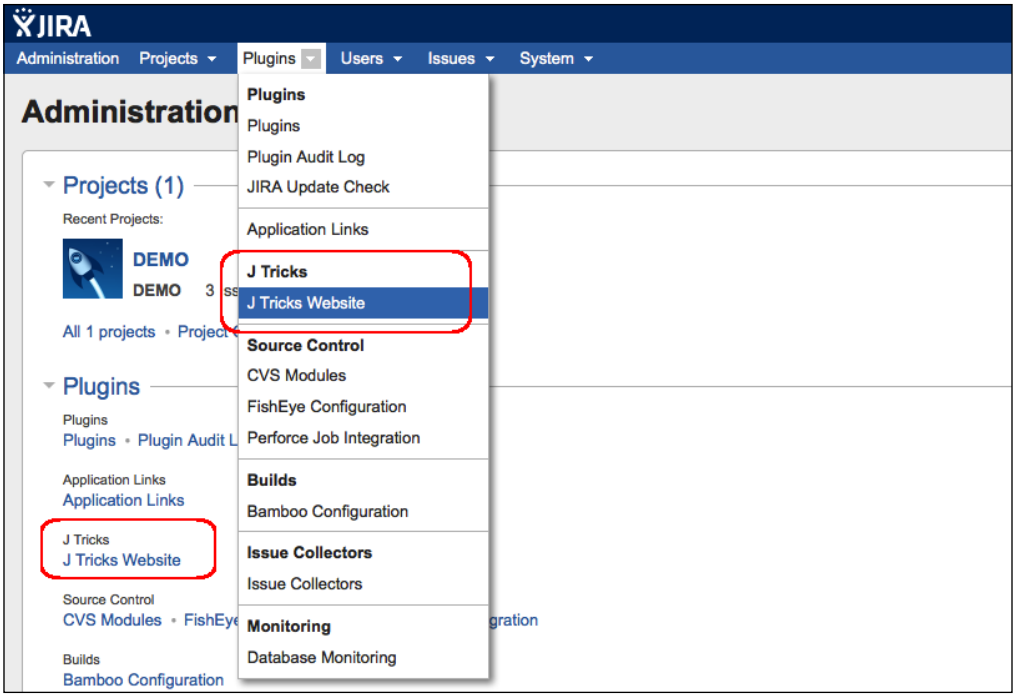
```
<label key="some.valid.key">
  <param name="param0">$somevariable</param>
</label>
```

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 you to insert values into the label using the 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.

5. Deploy the plugin.

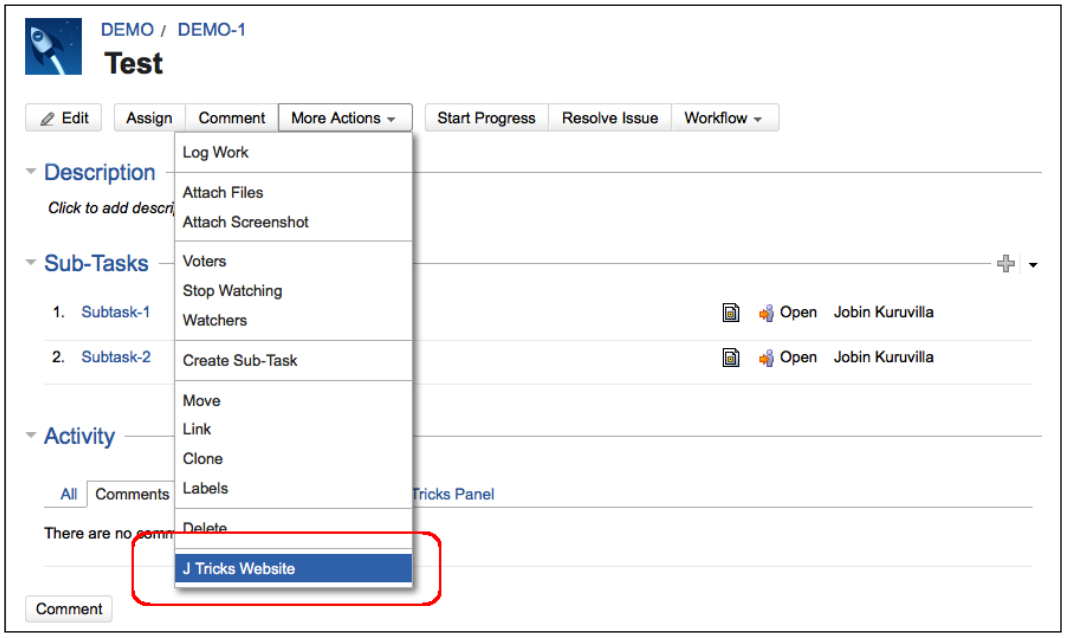
How it works...

Once the plugin is deployed, we can see that a new section is created in the **Administration** screen of JIRA, as shown in the following screenshot. The web item is explained in detail in the next recipe.



We can add the section at 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 following screenshot.

The web item's `section` attribute must be changed to match the new location as well, that is, `jtricks-admin-section`.



Note that the web section's label may not always be visible because in some cases the section is just used to group the links. For example, in the case of issue operations, the section is just used to group the links together as previously shown.

See also

- The *Adding new web items in the UI* recipe

Adding new web items in the UI

A **web item** is a new link that can be added at various places in the JIRA UI. The link will typically go under a web section. It 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...

The following are the steps to add a new web item into JIRA:

1. Identify the web section where the new link should be added. We have already learned how to create a new web section. A link is then added into a section created as mentioned previously 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` value of the web section in which it should appear.

   For example, if we want to place a link in the web section previously created, the `section` element will have the value `admin_plugins_menu/jtricks-admin-section`.

2. Add the new `web-item` module into the `atlassian-plugin.xml` file:

```
<web-item key="jtricks-admin-link" name="JTricks Link"
section="admin_plugins_menu/jtricks-admin-section" weight="10">
    <label>J Tricks Website</label>
    <link linkId="jtricks.admin.link">http://www.j-
                tricks.com</link>
</web-item>
```

3. The `web-item` module also has a unique key. The other two important attributes of the module are `section` and `weight`. `section` defines the web section where the link is placed, as mentioned previously, and `weight` defines the order in which the links will appear.

   A web item also has all the elements of a web section: `condition/conditions`, `context-provider`, `description`, `param`, `resource`, and `tooltip`. 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 following examples:

```
<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.

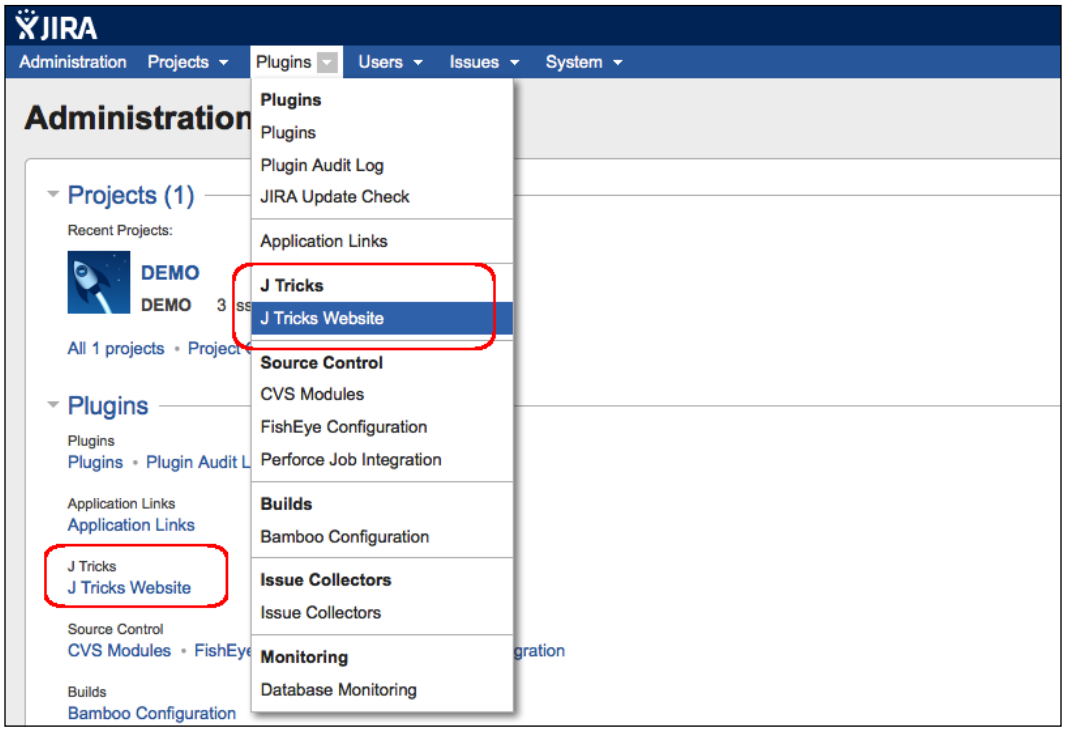
An `icon` element is used when we need to add an icon alongside the link:

```
<icon height="16" width="16">
  <link>/images/avatar.gif</link>
</icon>
```

4. 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 earlier under the **Administration** screen of JIRA.



We can add the item in various places, just by changing the `section` attribute. We have seen an example when creating a new issue operation in the previous recipe.

See also

- The *Adding new web sections in the UI* recipe

Use of decorators

We have seen web sections and web items. In most cases, these web items points to custom actions created via plugins and will most certainly have views rendered using custom Velocity templates or JSPs. How do we provide a user experience similar to other standard JIRA pages?

Everyone knows how using the proper decorators can get your desired look and feel. Have you been paying attention to the JIRA decorators as well?

Getting ready

Let us assume we are developing a simple `webwork` plugin and have added a `web-item` module that points to the new action. This is how the respective modules look in the `atlassian-plugin.xml` file:

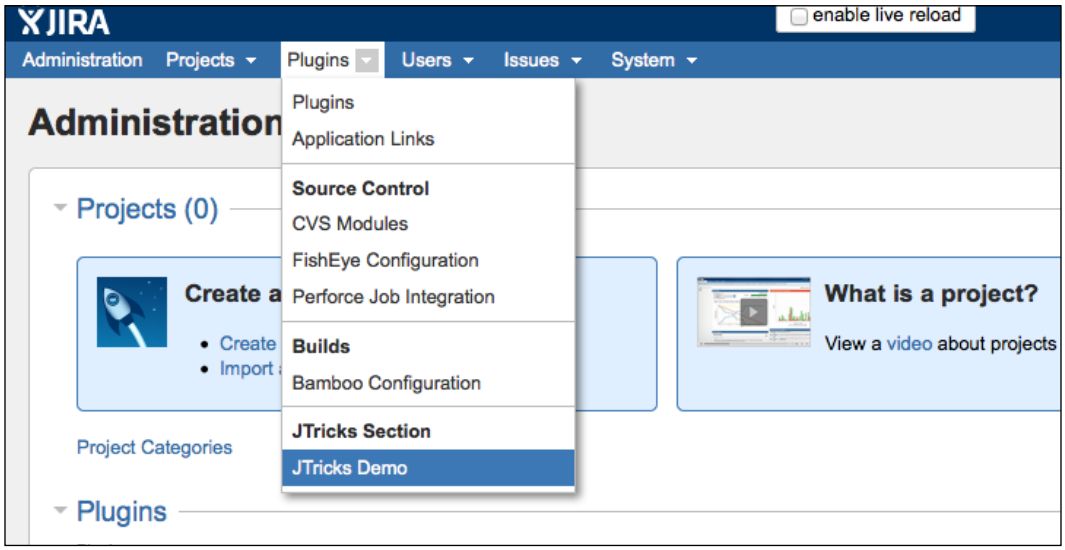
```
<webwork1 key="j-tricks-demo-action" name="JTricks Demo Action" i18n-
name-key="j-tricks-demo-action.name">

  <description key="j-tricks-demo-action.description">
          The JTricks Demo Action Plugin
  </description>
  <actions>
     <action name="com.jtricks.jira.webwork.JTricksDemoAction"
                  alias="JTricksDemoAction">
        <view name="input">/templates/j-tricks-demo-
                  action/jtricksdemoaction/input.vm</view>
     </action>
  </actions>
 </webwork1>

<web-section key="jtricks-demo-section" name="JTricks Section"
location="admin_plugins_menu" i18n-name-key="j-tricks-section.name"
weight="900">

  <label key="j-tricks-section.label"></label>
  <description>J Tricks Section</description>
  <tooltip>J Tricks - Little JIRA Tricks</tooltip>
</web-section>

<web-item name="JTricks Demo" i18n-name-key="j-tricks-demo.name"
key="j-tricks-demo" section="admin_plugins_menu/jtricks-demo-section"
weight="1000">
    <description key="j-tricks-demo.description">The JTricks Demo
                     Plugin</description>
    <label key="j-tricks-demo.label"></label>
```

```
        <link linkId="j-tricks-demo-link">
          /secure/JTricksDemoAction!default.jspa</link>
    </web-item>
```

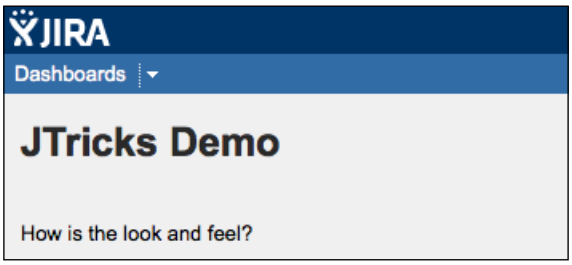
As you see, we have a webwork action with an `input` view. There is a web section under **Administration** | **Plugins** and a web item under the new section, which points to our action. On clicking on the new item, the `input` view is rendered. All the usual stuff!

How to do it...

Using the JIRA decorators, the look and feel of the page changes drastically. After deploying this little plugin, we will see the menu item we created, as shown in the following screenshot:



Click on the menu item and you will see our view page rendered, as shown in the following screenshot:



But, as you can see, the look and feel is a bit odd and our link is shown under the context of dashboard even though we created it under **Administration** | **Plugins**. How do we give it the admin look and feel?

In order to provide the correct look and feel, we need to apply the proper decorators. This can be achieved in three simple steps:

1. Add a `meta` tag under the `HEAD` tag of the HTML document with `name` as `decorator` and the `content` value as `atl.admin`, as shown in the following code snippet:

   ```
   <meta name="decorator" content="atl.admin" />
   ```

   The `decorator` name will vary depending on the location.

2. Add another `meta` tag with the meta `name` as `admin.active.section` and the `content` value as the section name. In our example, section is `admin_plugins_menu/jtricks-demo-section`.

   ```
   <meta content="admin_plugins_menu/jtricks-demo-section"
   name="admin.active.section">
   ```

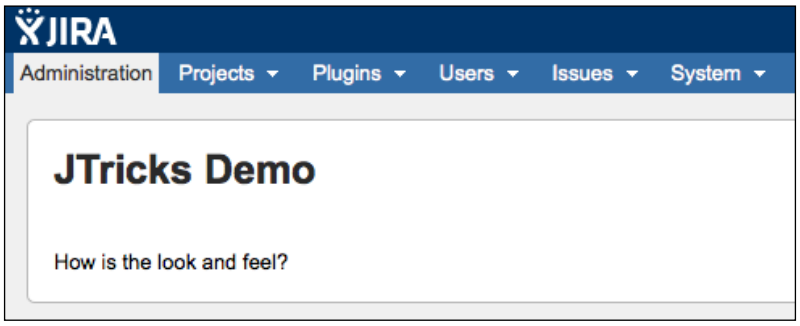
3. Add another `meta` tag with the meta `name` as `admin.active.tab` and the `content` value as the web item link ID. This is required only if a section has more than one tab and we want to show one of those tabs as active. In our example, we don't have to use this. But an example would be **FishEye Configuration** provided by JIRA `Fisheye` plugin.

How it works...

After the `atl.admin` decorator is added as a `meta` tag under the `HEAD` tag of the HTML document, the updated `input.vm` template will look as follows:

```
<html>
    <head>
        <title>J-Tricks Demo</title>
        <meta name="decorator" content="atl.admin" />
    </head>
    <body>
        <h1>JTricks Demo</h1>
        <br><br>
        How is the look and feel?
    </body>
</html>
```

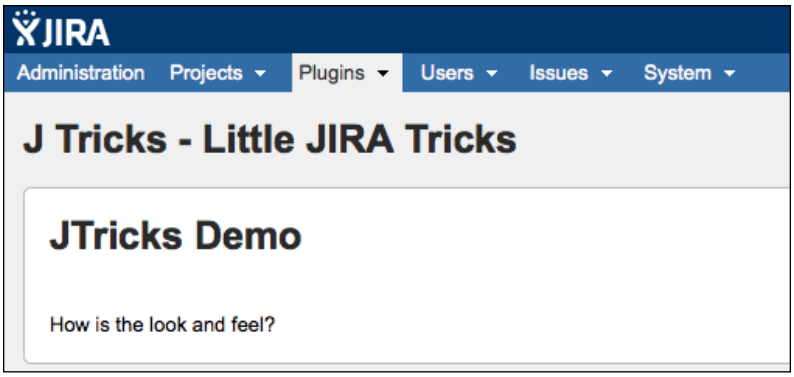
The web item is now rendered as shown in the following screenshot:



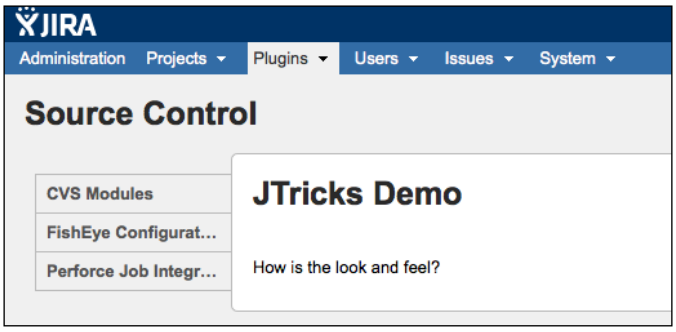
Once the `admin.active.section` meta tag is added, the `input.vm` template will look like this:

```
<html>
    <head>
        <title>J-Tricks Demo</title>
        <meta name="decorator" content="atl.admin" />
        <meta name="admin.active.section"
         content="admin_plugins_menu/jtricks-demo-section" />
    </head>
    <body>
        <h1>JTricks Demo</h1>
        <br><br>
        How is the look and feel?
    </body>
</html>
```

And now the page will look like the following screenshot:



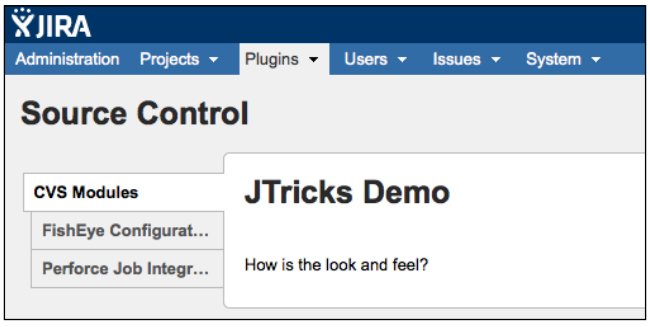
In order to demonstrate `admin.active.tab` meta tag, let us assume the section is changed to `admin_plugins_menu/source_control`. The view will now be rendered as follows:



And as you can see, none of the tabs are selected. Let us assume that we want to select the **CVS Modules** tab. The link ID for the **CVS Modules** web item is `cvs_modules`. Note that we need the link ID and not the `web-item` key. The updated code will now be as follows:

```
<html>
    <head>
        <title>J-Tricks Demo</title>
        <meta name="decorator" content="atl.admin" />
        <meta content="admin_plugins_menu/source_control"
                      name="admin.active.section">
        <meta content="cvs_modules" name="admin.active.tab">
    </head>
    <body>
        <h1>JTricks Demo</h1>
        <br><br>
        How is the look and feel?
    </body>
</html>
```

And the page will now look like the following screenshot:



As you can see, the appropriate section and tab are highlighted and the page now has an admin look and feel.

Similarly, the look and feel of various JIRA areas can be achieved by using the appropriate decorators. The full list of available decorators can be found at `https://developer.atlassian.com/display/JIRADEV/Changes+to+markup,+CSS+and+Decorators+in+JIRA+5.0`.

See also

- The *Adding new web sections in the UI* recipe
- The *Adding new web items in the UI* recipe

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 stop 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 that 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 a 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` 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 for a permission, as of 5.1, are `admin`, `use`, `sysadmin`, `project`, `browse`, `create`, `edit`, `update` (same as `edit`), `scheduleissue`, `assign`, `assignable`, `attach`, `resolv`, `close`, `comment`, `delete`, `work`, `worklogdeleteall`, `worklogdeleteown`, `worklogeditall`, `worklogeditown`, `link`, `sharefilters`, `groupsubscriptions`, `move`, `setsecurity`, `pickusers`, `viewversioncontrol`, `modifyreporter`, `viewvotersandwatchers`, `managewatcherlist`, `bulkchange`, `commenteditall`, `commenteditown`, `commentdeleteall`, `commentdeleteown`, `attachdeleteall`, `attachdeleteown`, `viewworkflowreadonly`. The 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. Carry out the following steps:

1. Write a condition class that should extend the `AbstractJiraCondition` class and override the following abstract method.

   ```
   public abstract boolean shouldDisplay(User user, JiraHelper
   jiraHelper);
   ```

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 AbstractJiraCondition {
     @Override
     public boolean shouldDisplay(User user, JiraHelper
     jiraHelper) {
       return user != null  &&
     ComponentAccessor.getGroupManager().getGroupNamesForUser
     (user).contains("jira-developers");
     }
   }
   ```

3. Add the new condition class in the `web-item` module:

   ```
   <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.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. But the link will be visible only if the condition `DeveloperCondition`, returns `true`.

We can easily invert a condition using the `invert` flag as follows:

```
<condition class="com.jtricks.conditions.DeveloperCondition"
invert="true"/>
```

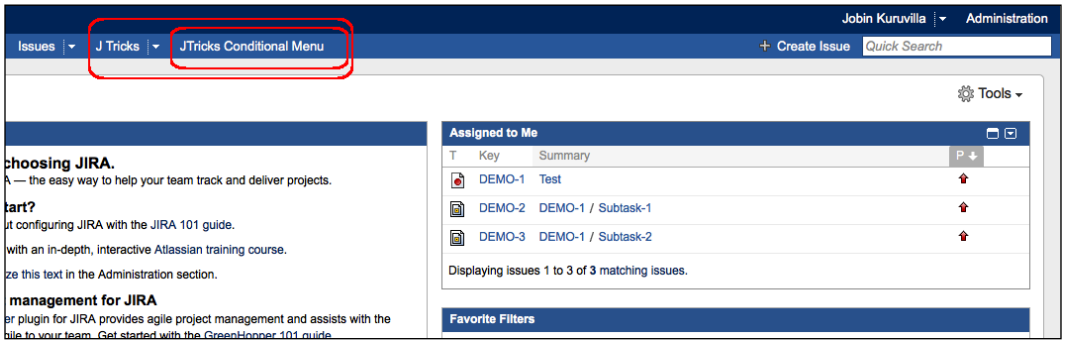
This will display the link if the user is not logged in or not in the group of JIRA developers!

4. Deploy the plugin.

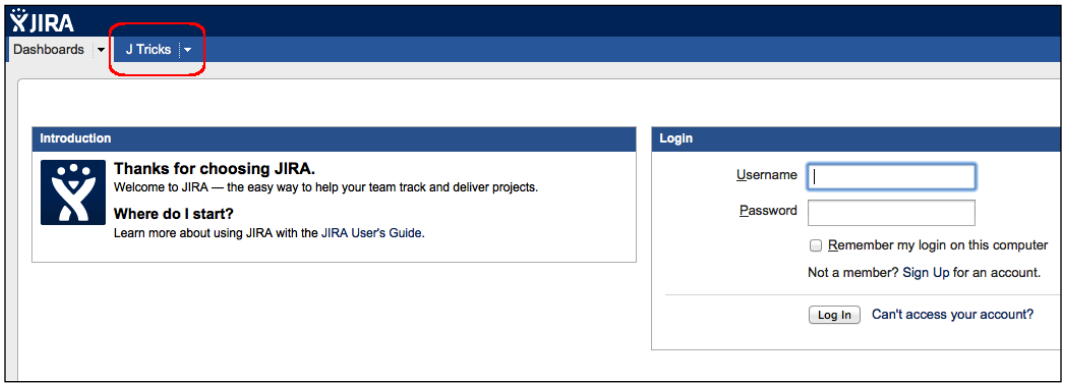
How it works...

Once the plugin is deployed, we can see that the new **JTricks Conditional Menu** tab 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, the menu is not shown as seen in the following screenshot. In this case, we only have web items that don't have conditions defined for them!



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 includes `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` element 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(User user, JiraHelper
   jiraHelper);
   ```

   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(User user, JiraHelper helper) {
      return EasyMap.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 this example, we'll 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.

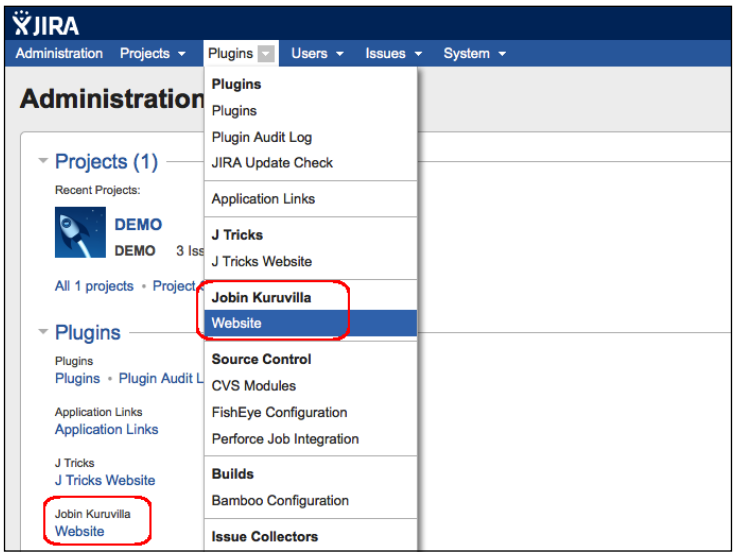
```
<web-section key="jtricks-admin-context-section" name="JTricks
Context Section" location="admin_plugins_menu" weight="910">

  <label>$userName</label>
  <context-provider
   class="com.jtricks.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 **Administration** UI screen, 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 need a web item first to be placed in the system's top navigation bar and then have a web section declared under it. The web section can then have a list of web items created under it that will form the links on the drop-down menu.

To do this, carry out the following steps:

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 JTricks' website. It's important to note here that the `linkId` value should be the same as the `key` value. In this case, both are `jtricks-menu`.

2. Define a web section located under the previous 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 previous web section:

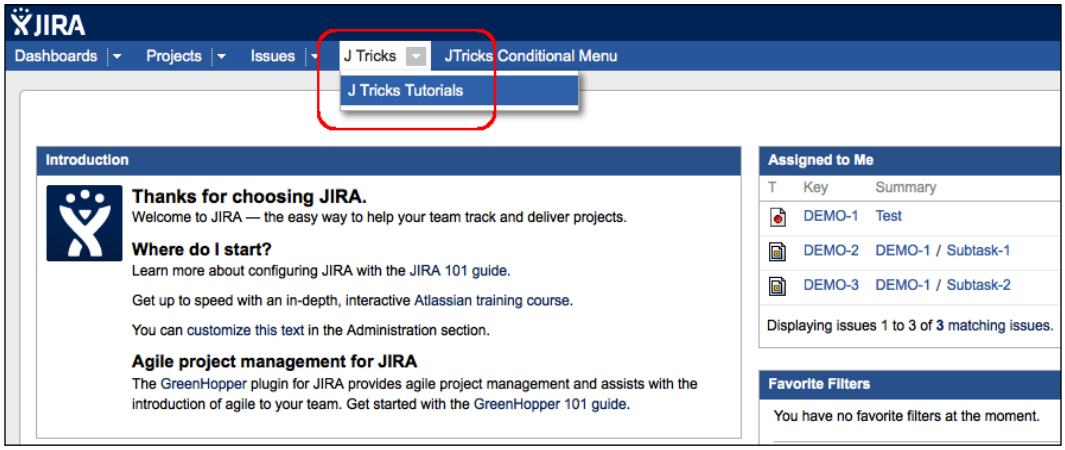
```
<web-item key="jtricks-item" name="Jtricks Item" section="jtricks-
menu/jtricks-section" weight="210">
  <description>J Tricks Tutorials</description>
  <label>J Tricks Tutorials</label>
  <tooltip>Tutorials from J Tricks</tooltip>
  <link linkId="jtricks.link">http://www.j-tricks.com/tutorials</
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` module and `jtricks-section` is the key for the previous `web-section` module.

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, **J Tricks**, under which a list of links are grouped into 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 all the cases, we knew what links we 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's top navigation bar, as discussed in the previous recipe.

How to do it...

Let's assume we want to create some links in the system's 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! Here the links change based on the user's status and hence need to be created dynamically.

To do this, carry out the following steps:

1. Create a **Favourites** web section in the system's 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 `simple-link-factory` tag in the `atlassian-plugin.xml` file. A **simple link factory** defines a new link factory that creates the set of links dynamically. It always hangs off an already declared web section, `favourites-section` in our case.

```
<simple-link-factory key="favourites-factory" name="Favourites
Link Factory" section="favourites-menu/favourites-section"  i18n-
name-key="jtricks.favourites.factory" weight="10" lazy="true"
class="com.jtricks.web.links.FavouritesLinkFactory"/>
```

3. As you can see, a simple link factory has a unique key and it points to an already available section. In our case, the section is `favourites-menu/favourites-section`, which is declared in the first step.

   The most important attribute is the `class` attribute, `FavouritesLinkFactory`. Also, notice that the attribute `lazy` is declared as `true` to represent lazy loading.

Create the simple link factory class. The class should implement the `SimpleLinkFactory` interface, as follows:

```
public class FavouritesLinkFactory implements SimpleLinkFactory {

  public List<SimpleLink> getLinks(User user, Map<String, Object>
arg1) {
    ...
  }

  public void init(SimpleLinkFactoryModuleDescriptor arg0) {
  }
}
```

All we need to do is implement two methods, `getLinks` and `init`. The `init` method needs to be implemented only when you need to initialize anything in your plugin. This will be invoked only once, that is, at JIRA startup.

The `getLinks` method is the actual method that we need to implement.

4. Implement the `getLinks` method. In this method, we need to return a collection of links that will then be displayed as a web item under the section we defined earlier.

   Each link we return must be an instance of the `SimpleLink` class. A `SimpleLink` object is the Java representation of a web item we normally declare in the `atlassian-plugin.xml` file. It has all the same attributes such as the `label`, `title`, `iconUrl`, `style`, `url`, and `accesskey`.

   The following is the method for our example:

```
public List<SimpleLink> getLinks(User user, Map<String, Object>
arg1) {
  List<SimpleLink> links = new ArrayList<SimpleLink>();

  if (user != null) {
    links.add(new SimpleLinkImpl("id1", "Favourites 1", "My
Favourite One", null, null, "http://www.google.com", null));
     links.add(new SimpleLinkImpl("id2", "Favourites 2", "My
Favourite Two", null, null, "http://www.j-tricks.com", null));
   } else {
    links.add(new SimpleLinkImpl("id1", "Favourite Link", "My
Default Favourite", null, null, "http://www.google.com", null));
  }
  return links;
}
```

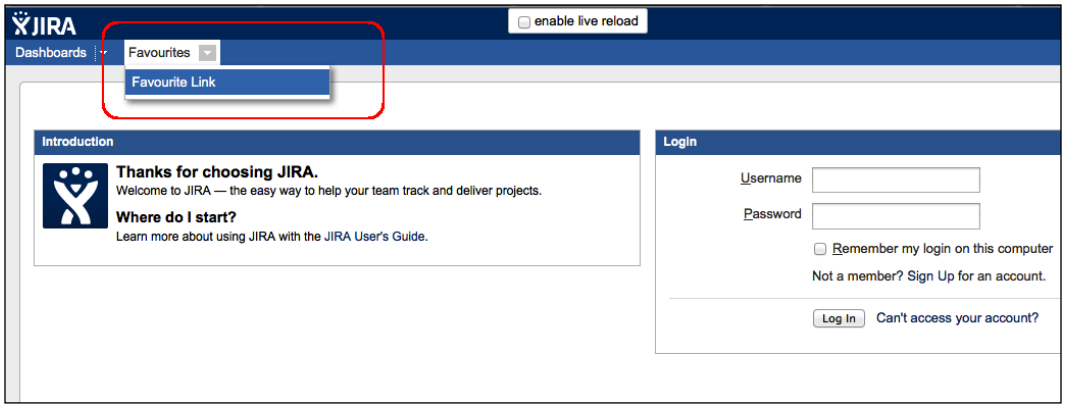
Here, we just create different links based on whether the user is `null` or not. A user is `null` if he/she is not logged in. As you can see, each link has different attributes as mentioned earlier.

5. Package the plugin and deploy it.

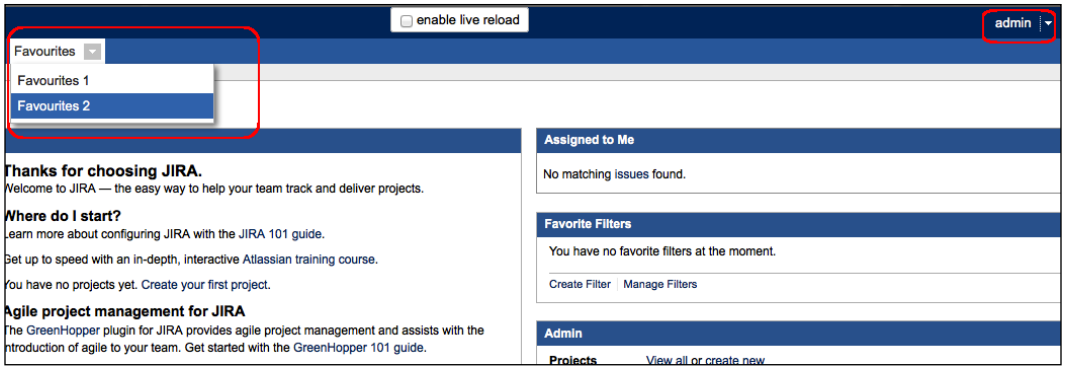
The links should now be created dynamically.

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 `getLinks` method.



The same idea can be used to create dynamic links based on a different criterion, and of course, at various different places in UI.

Adding new tabs in the View Issue screen

We have seen how to enhance the UI by adding new sections and links at 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 on the **View Issue** page can be done with the Issue Tab Panel plugin module. Perform the following steps 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` file:

```
<issue-tabpanel key="jtricks-issue-tabpanel" i18n-name-
key="issuetabpanel.jtricks.name" name="Issue Tab Panel"
class="com.jtricks.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` attribute and should define a `class` attribute that implements the tab panel. It also has a list of elements, explained as follows:

   - `description`: A description of the tab panel.
   - `label`: A human-readable label for the panel.
   - `resource`: Defines the Velocity template that renders the tab panel view.
   - `order`: Defines the order in which the panels will appear on the **View Issue** page.
   - `sortable`: Defines whether the content of the panel is 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.

   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 this example we will just return `true`.

```
public boolean showPanel(Issue issue, User remoteUser) {
  return true;
}
```

3. 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 each instance of the class renders an item in the tab. This class also returns the *time* at which the action is performed and this *time* will be used for sorting the items, if `sortable = true` in the module definition in step 1. Sorting is done based on time returned by the `getTimePerformed()` method.

4. In the example, let us create a single action class as follows:

```
public class JTricksAction extends AbstractIssueAction{
  private final JiraAuthenticationContext authenticationContext;

  public JTricksAction(IssueTabPanelModuleDescriptor
  descriptor, JiraAuthenticationContext
  authenticationContext) {
    super(descriptor);
    this.authenticationContext = authenticationContext;
  }

  @Override
  public Date getTimePerformed() {
    return new Date();
  }
    @Override
  protected void populateVelocityParams(Map params)
{   params.put("user", this.authenticationContext.getUser().
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 the key name as `user`.

5. Implement the `getActions` method in the tab panel class to return a list of `IssueActions`. As mentioned in the previous step, each instance of the class renders an item in the tab. In our example, we just return a list that contains the new `JTricksAction`. That will be a single item in the tab.

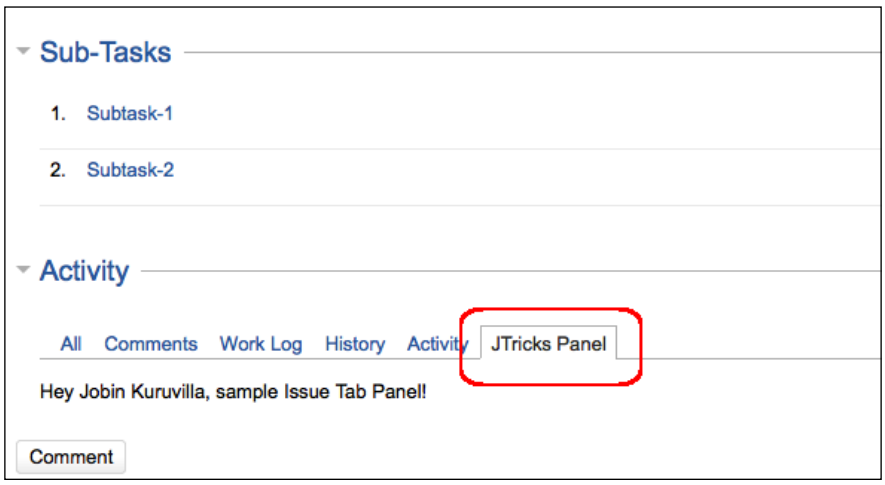
```
public List getActions(Issue issue, User remoteUser) {
  List<JTricksAction> panelActions = new
ArrayList<JTricksAction>();
  panelActions.add(new JTricksAction(descriptor,
authenticationContext));
  return panelActions;
}
```

   Here, `descriptor` is an instance variable of the super class. All we do here is create an instance of the action class and return a list of such actions.

6. Create a `view` template in the location specified earlier. `Hey $user, sample Issue Tab Panel!` is all we need and the user here is populated into the context in the action class.
7. Package the plugin and deploy it.

How it works...

Once the plugin is deployed, a new tab panel will appear in the **View Issue** page, as shown in the following screenshot:



As you can see, the greeting message is populated using the Velocity context and the attributes in it.

There's more...

From JIRA 5.0 onwards, it is possible to load the issue tab panels asynchronously. This will improve the user experience if a custom tab takes more time to load.

Loading issue tab panel asynchronously

All we need to do is 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.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. But 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 is 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. The following is an example of this:

```
<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 **Browse Project** screen.

Getting ready

Create a new skeleton plugin using the Atlassian Plugin SDK.

How to do it...

The following are the steps to create a new project tab panel:

1. Define the project tab panel in `atlassian-plugin.xml`.

```
<project-tabpanel key="jtricks-project-panel" i18n-name-
key="projectpanels.jtricks.name" name="JTricks Panel" class="com.
jtricks.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` attribute and should define a `class` that implements the tab panel. It also has a list of elements, explained as follows:

   - `description`: A description of the tab panel
   - `label`: A human-readable label for the panel
   - `resource`: Defines the Velocity template that renders the tab panel view
   - `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 this example we will just return `true`.

```
public boolean showPanel(Issue issue, User remoteUser) {
  return true;
}
```

3. Create a `view` template in the location specified earlier. The template we defined is as follows:

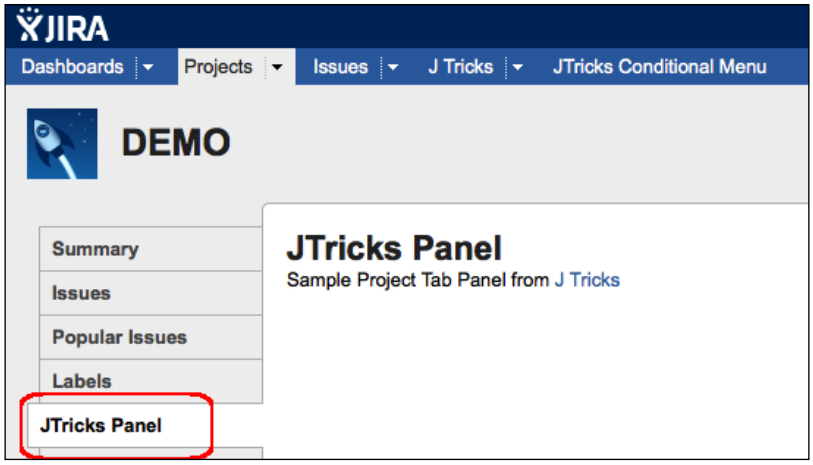
```
Sample Project Tab Panel from <a href="http://www.j-tricks.com">J Tricks</a>
```

   If we need extra Velocity parameters in this context, they can be populated in the project tab panel class by overriding the `createVelocityParams` method.

4. Package the plugin and deploy it.

How it works...

Once the plugin is deployed, a new tab panel will appear in the **Browse Project** page, as shown in the following screenshot:



Creating the project tab panel using fragments

We have seen how to create a new project tab panel in the previous recipe. While this works in most cases, sometimes we would like to create the nice fragmented view in JIRA 4.1+. Here, each project tab panel has a list of fragments organized in two columns. We can create the fragments and order them so that they appear in a formatted way on clicking the new tab panel.

In this recipe, we will see how to create the project tab panel using fragments. Before we start, there are a couple of things worth mentioning:

- We need to use the same package structure, `com.atlassian.jira.plugin.projectpanel.impl`, to create the fragment class, as we need to override the protected methods in it
- The components used in creating the fragments are not available in the OSGI v2.0 plugins, and hence we have to create a v1.0 plugin

How to do it...

Following are the steps to create a fragmented project tab panel:

1. Add the project tab panel module in `atlassian-plugin.xml`.

```
<project-tabpanel key="jtricks-project-fragment-panel" i18n-
name-key="projectpanels.fragments.jtricks.name" name="JTricks
Frag Panel" class="com.atlassian.jira.plugin.projectpanel.impl.
JTricksFragProjectTabPanel">
  <description>A sample Project Tab Panel with fragments
  </description>
  <label>JTricks Fragments Panel</label>
  <order>910</order>
</project-tabpanel>
```

   The attributes and elements are similar to a normal project tab panel, except that they don't have a `view` Velocity resource defined. The HTML here is constructed with the help of fragments.

2. Create the fragments needed in the project panel. Let's assume we need two fragments, `FragmentOne` and `FragmentTwo`, for our example.

   Each fragment must extend the `AbstractFragment` class. We need to override the following three methods for a fragment:

   - `getId`: This defines the ID of the fragment, which will also be the name of the Velocity template used to render this fragment

- `getTemplateDirectoryPath`: This returns the path where the Velocity template is placed
- `showFragment`: This defines whether the fragment is visible to the user or not

A fourth method, `createVelocityParams`, can be overridden if we need to pass extra parameters to the Velocity context. `FragmentOne` looks as follows:

```
public class FragmentOne extends AbstractFragment{
  protected static final String TEMPLATE_DIRECTORY_PATH =
"templates/project/fragments/";
  public FragmentOne(VelocityTemplatingEngine templatingEngine,
JiraAuthenticationContext jiraAuthenticationContext) {
    super(templatingEngine, jiraAuthenticationContext);
  }
  public String getId() {
    return "fragmentone";
  }
  public boolean showFragment(BrowseContext ctx) {
    return true;
  }
  @Override
  protected String getTemplateDirectoryPath() {
    return TEMPLATE_DIRECTORY_PATH;
  }
  @Override
  protected Map<String, Object> createVelocityParams(BrowseContext
ctx) {
    Map<String, Object> createVelocityParams = super.
createVelocityParams(ctx);
    createVelocityParams.put("user", ctx.getUser().
getDisplayName());
    return createVelocityParams;
  }
}
```

Here, the Velocity template will be `fragmentone.vm`, placed under `templates/project/fragments/`. The fragment is always shown, but this can be modified to include complex logic. We also add a new variable, `user`, to the context, which then holds the full name of the current user. Note that the `user` variable is already in the context, but this is just for an example.

`FragmentTwo` will be similar, as shown in the next few lines of code:

```
public class FragmentTwo extends AbstractFragment {
  protected static final String TEMPLATE_DIRECTORY_PATH =
"templates/project/fragments/";
```

```
  public FragmentTwo(VelocityTemplatingEngine templatingEngine,
JiraAuthenticationContext jiraAuthenticationContext) {
    super(templatingEngine, jiraAuthenticationContext);
  }
  public String getId() {
    return "fragmenttwo";
  }
  public boolean showFragment(BrowseContext ctx) {
    return true;
  }
  @Override
  protected String getTemplateDirectoryPath() {
    return TEMPLATE_DIRECTORY_PATH;
  }
}
```

Here, the Velocity template will be `templates/project/fragments/fragmenttwo.vm`. Note that we don't override the `createVelocityParams` method here, as we don't need any extra parameters in the context.

3. Create the fragment-based project tab panel class. In our example, the class is `JTricksFragProjectTabPanel`. The class must extend the `AbstractFragmentBasedProjectTabPanel` class. We need to implement three methods in this class:
    - `getLeftColumnFragments`: This returns a list of the `ProjectTabPanelFragment` classes that form the left column of the panel
    - `getRightColumnFragments`: This returns a list of the `ProjectTabPanelFragment` classes that forms the right column of the panel
    - `showPanel`: This determines whether the panel can be shown or not

    The class will look as follows:

```
public class JTricksFragProjectTabPanel extends
AbstractFragmentBasedProjectTabPanel {
  private final FragmentOne fragmentOne;
  private final FragmentTwo fragmentTwo;
  public JTricksFragProjectTabPanel(VelocityTemplati
ngEngine templatingEngine, JiraAuthenticationContext
jiraAuthenticationContext) {
    this.fragmentOne = new FragmentOne(templatingEngine,
jiraAuthenticationContext);
    this.fragmentTwo = new FragmentTwo(templatingEngine,
jiraAuthenticationContext);
```

```
  }

  @Override
  protected List<ProjectTabPanelFragment> getLeftColumnFragments(B
rowseContext ctx) {
    final List<ProjectTabPanelFragment> frags = new ArrayList<Proj
ectTabPanelFragment>();
    frags.add(fragmentOne);
    return frags;
  }
  @Override
  protected List<ProjectTabPanelFragment> getRightColumnFragments(
BrowseContext ctx) {
    final List<ProjectTabPanelFragment> frags = new ArrayList<Proj
ectTabPanelFragment>();
    frags.add(fragmentTwo);
    return frags;
  }

  public boolean showPanel(BrowseContext ctx) {
    return true;
  }
}
```

Here, we just construct the fragment objects and return them to the appropriate column list.

4. Create the Velocity templates for the fragments. In our example, `fragmentone.vm` looks like the following:

```
<div class="mod-header">
  <h3>Fragment 1</h3>
</div>
<div class="mod-content">
  <ul class="item-details">
    <li>Welcome, $user!</li>
    <li>This is fragment 1.</li>
  </ul>
</div>
```

Notice the use of `$user`, which was populated in the Velocity context of the `FragmentOne` class. Also, the various `div` elements are used for the UI conformance.

Similarly, `fragmenttwo.vm` is as follows:

```
<div class="mod-header">
  <h3>Fragment 2</h3>
</div>
```

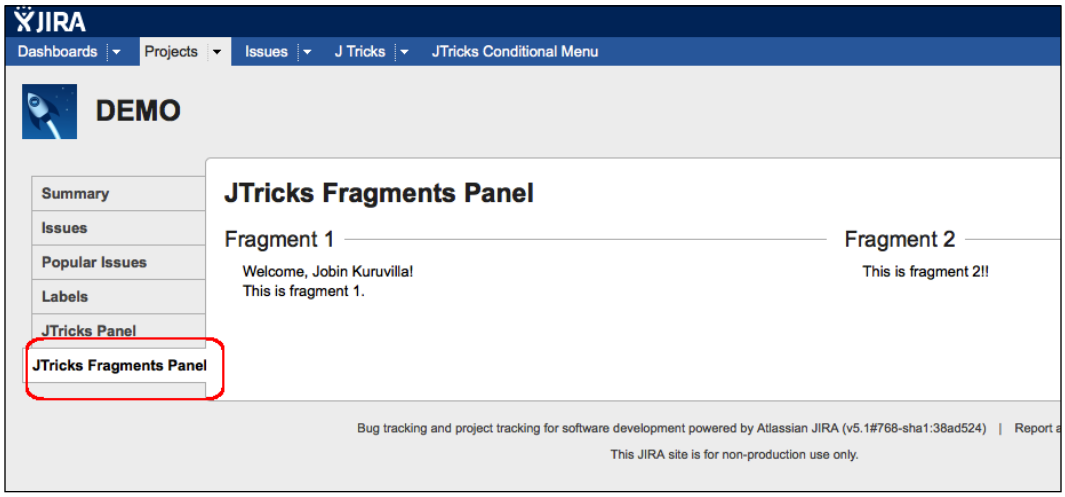
```
<div class="mod-content">
  <ul class="item-details">
    <li>This is fragment 2!!</li>
  </ul>
</div>
```

The only difference is that we don't use the Velocity variable here.

5. Package the plugin and deploy it.

How it works...

Once the plugin is deployed, a new fragment-based tab panel will appear in the **Browse Project** page, as shown in the following screenshot:



Adding new tabs in the Browse Version screen

In this recipe, we will see how to add a new tab in the **Browse Version** screen. This screen holds details of a particular version in JIRA.

Getting ready

Create a new skeleton plugin using the Atlassian Plugin SDK.

How to do it...

The following are the steps to create a new version tab panel. It is very similar to creating a new project tab panel, except for the obvious changes in the files and keywords involved.

1. Define the version tab panel in `atlassian-plugin.xml`:

```
<version-tabpanel key="jtricks-version-panel" i18n-name-
key="versionpanels.jtricks.name" name="jtricks Version Panel"
class="com.jtricks.JTricksVersionTabPanel">
  <description>A sample Version Tab Panel</description>
  <label>JTricks Panel</label>
  <order>900</order>
  <resource type="velocity" name="view" location="templates/
version/version-panel.vm" />
</version-tabpanel>
```

   Here, the plugin module has a unique `key` attribute and should define a class that implements the tab panel. It also has the following list of elements:

   - `description`: A description of the tab panel
   - `label`: A human-readable label for the panel
   - `resource`: Defines the Velocity template that renders the tab panel view
   - `order`: Defines the order in which the panels will appear on the **Browse Version** screen

2. Implement the version tab panel class.

   The class should extend the `GenericTabPanel` class, which in turn implements the `TabPanel` interface. We need to implement only the `showPanel` and `createVelocityParams` methods, the latter only if we need to add extra variables into the Velocity context.

   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 this example we will just return `true`.

```
public boolean showPanel(Issue issue, User remoteUser) {
  return true;
}
```

Let's just override the `createVelocityParams` method to add a new variable user to the Velocity context:

```
@Override
protected Map<String, Object> createVelocityParams(BrowseVersionCo
ntext context) {
  Map<String, Object> createVelocityParams = super.
createVelocityParams(context);
  createVelocityParams.put("user", context.getUser().
getDisplayName());
  return createVelocityParams;
}
```

This variable will now be available on the view template that we are using.

3. Create the view template in the location specified in the plugin descriptor. Let's just create a simple template with a greeting to the current user as follows:

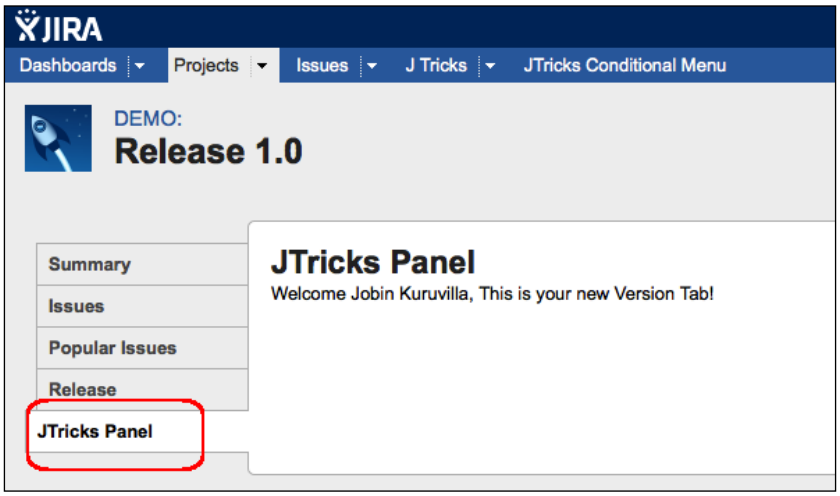
   ```
   Welcome $user, This is your new Version Tab!
   ```

   Note that we have used the `$user` variable that we populated in the previous step.

4. Package the plugin and deploy it.

How it works...

Once the plugin is deployed, a new tab panel will appear in the **Browse Version** page, as shown in the next screenshot:



Adding new tabs in the Browse Component screen

In this recipe, we will see how to add a new tab in the **Browse Component** screen. This screen holds details of a particular component in JIRA, and the process of adding a new tab is very similar to adding a new version tab or project tab panel.

Getting ready

Create a new skeleton plugin using the Atlassian Plugin SDK.

How to do it...

The following are the steps to create a new component tab panel:

1. Define the component tab panel in `atlassian-plugin.xml`:

```
<component-tabpanel key="jtricks-component-panel" i18n-name-
key="componentpanels.jtricks.name" name="jtricks Component Panel"
class="com.jtricks.JTricksComponentTabPanel">
  <description>A sample Component Tab Panel</description>
  <label>JTricks Panel</label>
  <order>900</order>
  <resource type="velocity" name="view" location="templates/
component/component-panel.vm" />
</component-tabpanel>
```

   As in the case of a version tab panel, the component tab panel also has a unique `key` attribute and should define a class that implements the tab panel. It also has the following list of elements:

   - `description`: A description of the tab panel
   - `label`: A human-readable label for the panel
   - `resource`: Defines the Velocity template that renders the tab panel view
   - `order`: Defines the order in which the panels will appear on the **Browse Component** screen

2. Implement the `Component Tab Panel` class.

   The class should extend the `GenericTabPanel` class, which in turn implements the `ComponentTabPanel` interface. We need to implement only the `showPanel` and `createVelocityParams` methods, the latter only if we need to add extra variables into the Velocity context.

   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 this example we will just return `true`.

   ```
   public boolean showPanel(Issue issue, User remoteUser) {
     return true;
   }
   ```

   As in the previous recipe, let's override the `createVelocityParams` method to add a new variable, `user`, to the Velocity context:

   ```
   @Override
   protected Map<String, Object> createVelocityParams(BrowseVersionCo
   ntext context) {
     Map<String, Object> createVelocityParams = super.
   createVelocityParams(context);
     createVelocityParams.put("user", context.getUser().
   getDisplayName());
     return createVelocityParams;
   }
   ```

   This variable will now be available on the view template that we are using.

3. Create the view template in the location specified in the plugin descriptor. Let's create a simple template, as in the previous recipe, as follows:

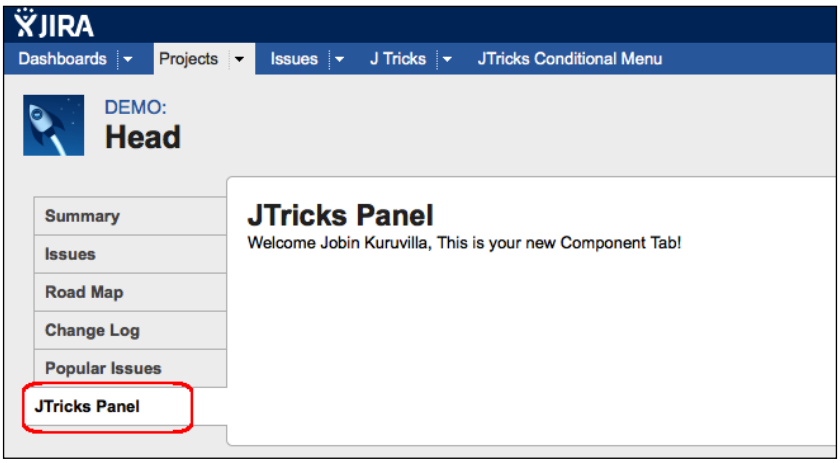
   ```
   Welcome $user, This is your new Component Tab!
   ```

   Note that we have used the `$user` variable, which we populated in the previous step.

4. Package the plugin and deploy it.

How it works...

Once the plugin is deployed, a new tab panel will appear in the **Browse Component** page, as shown in the following screenshot:



Adding issue link renderers

One of the great features introduced in JIRA5 is the ability to create links between JIRA issues and other remote entities. The remote entity can be anything that is accessible via URL, for example a confluence page, a remote JIRA issue, a ticket in another system, and so on.

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 we 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 the Atlassian Plugin SDK.

Let's consider an interesting example where a given user's Twitter status and the number of followers are displayed in an issue as a remote link.

In order to communicate with Twitter using java, you have different mechanisms, but in this recipe let's use Twitter4j. More details about Twitter4j can be found at `http://twitter4j.org/en/index.html`. The required libraries can be pulled into 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://dev.twitter.com/` to communicate with Twitter. Now for the serious part.

How to do it...

The part where you integrate with Twitter is not important for this recipe, but the code is pretty simple and looks something like this:

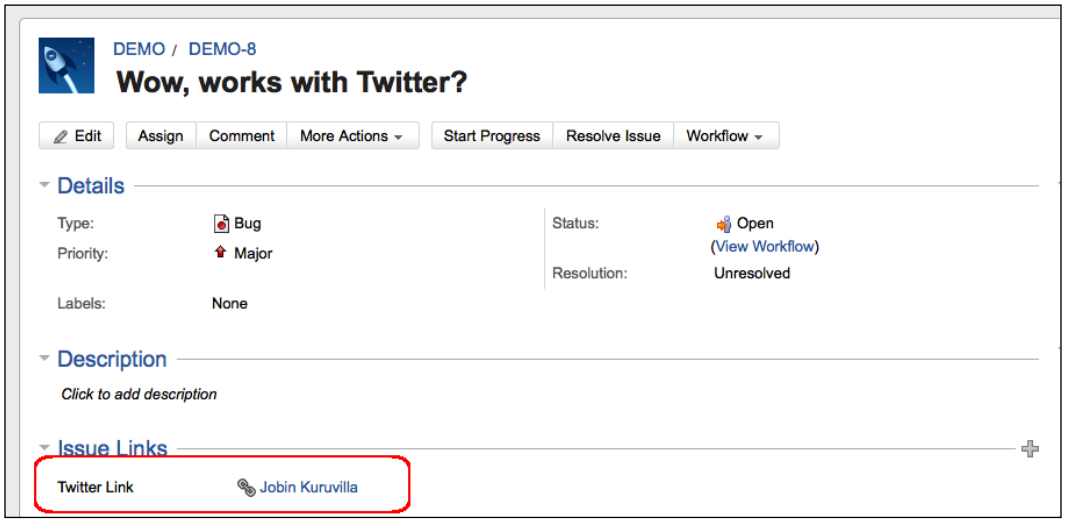
```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.setDebugEnabled(true)
  .setOAuthConsumerKey("************")
  .setOAuthConsumerSecret("***************")
  .setOAuthAccessToken("***************")
  .setOAuthAccessTokenSecret("***********");
TwitterFactory tf = new TwitterFactory(cb.build());
Twitter twitter = tf.getInstance();
```

You can now use the `Twitter` object to get various details from the account. Here, the `OAuth` tokens will be available from the application that we registered in the beginning.

As you read in the beginning, issue renderers are associated with a specific application type so let's create a new application type, Twitter!

```
String tweeter = "jobinkk";
final RemoteIssueLink remoteIssueLink = new RemoteIssueLinkBuilder().
url("https://twitter.com/"+ tweeter)
  .title(twitter.showUser(tweeter).getName()).globalId (tweeter).
issueId(issue.getId())
  .relationship("Twitter Link").applicationName("Twitter").
applicationType ("Twitter").build();
```

You can create a simple issue operation or a servlet to create this new remote link in your JIRA issue. If you use the **Link** operation, it won't let you use a custom application type (`Twitter` in this case). When the remote link is created using the previous code, it appears as follows in the **View Issue** page:



The step-by-step process to implement an issue link renderer that will transform this link to something better and with more details, is as follows:

1. Define the issue link renderer module in `atlassian-plugin.xml`:

```
<issue-link-renderer key="twitterLinkRenderer" application-
type="Twitter" class="com.jtricks.renderer.TwitterLinkRenderer">
    <resource name="initial-view" type="velocity"
location="templates/twitterlink.vm"/>
</issue-link-renderer>
```

   As with any other plugin module, `issue-link-renderer` has a unique `key` attribute. It also has a human-readable name and can have a `i18n-name-key` attribute 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` name 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 is as follows:

```
public class TwitterLinkRenderer extends
AbstractIssueLinkRenderer{

  public static final String DEFAULT_ICON_URL = "/download/
resources/com.jtricks.link-renderer:twitter-resources/images/
twitter.jpg";

  @Override
  public Map<String, Object> getInitialContext(RemoteIssueLink
twitterLink, Map<String, Object> context) {
    final I18nHelper i18n = 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(RemoteIssueLink
remoteIssueLink, I18nHelper i18n, 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 next!

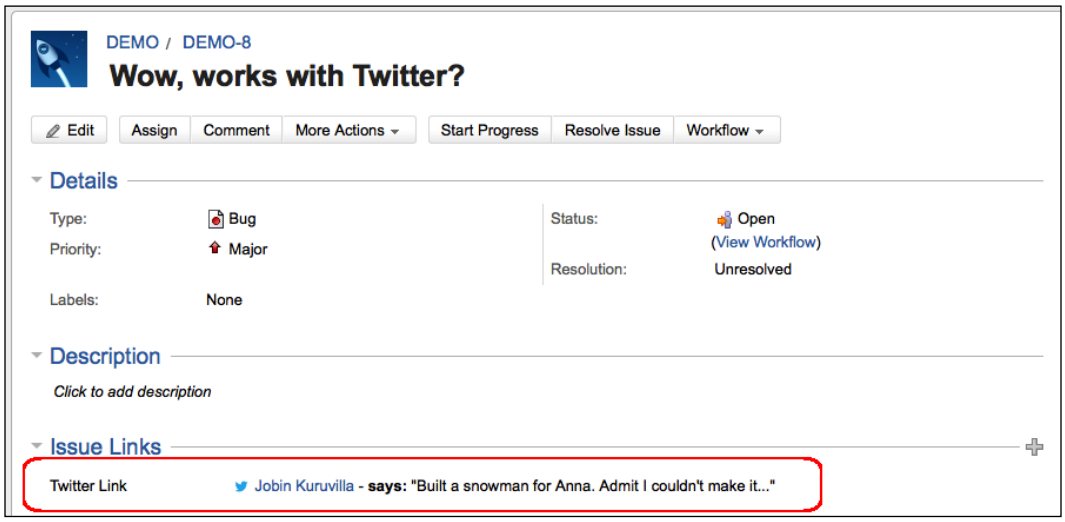
You might also have noticed that the `DEFAULT_ICON_URL` string points to a downloadable resource. You can either use a custom image declared as a resource in the plugin or go with a JIRA image.

3. Create a renderer Velocity template. In this example, we only have `initial-view` defined and the Velocity template for it is `twitterlink.vm`. We can render it using the variables available in the context. An example would be as follows:

```
<p>
    #renderIssueLinkIcon(${iconUrl} ${iconTooltip} ${iconTooltip}
${textutils})
    <span title="${textutils.htmlEncode($tooltip)}">
        <a href="${url}" class="link-title">${textutils.
htmlEncode($title)}</a>
    </span>
    #if ($status) -
    <span title="Status: $status">
      <span class="status"><b>says:</b> "$status"</span>
    </span>
    #end
</p>
```

Here we render the view with an icon, the title (which was nothing but the tweeter's name), the URL to the tweeter's 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!

4. That's it! The **View Issue** page will now have the same link but rendered slightly differently, along with the current status of the tweeter!



What if you want to show more details but load them asynchronously? That is where the `final-view` resource name 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.renderer.TwitterLinkRenderer">
    <resource name="initial-view" type="velocity"
location="templates/twitterlink.vm"/>
    <resource name="final-view" type="velocity"
location="templates/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.

   The `getFinalContext` method can populate the variable required in the final context. Let's 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(RemoteIssueLink
twitterLink, Map<String, Object> context) {
  final I18nHelper i18n = 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();
}
```

The `requiresAsyncLoading` method will just return `true`:

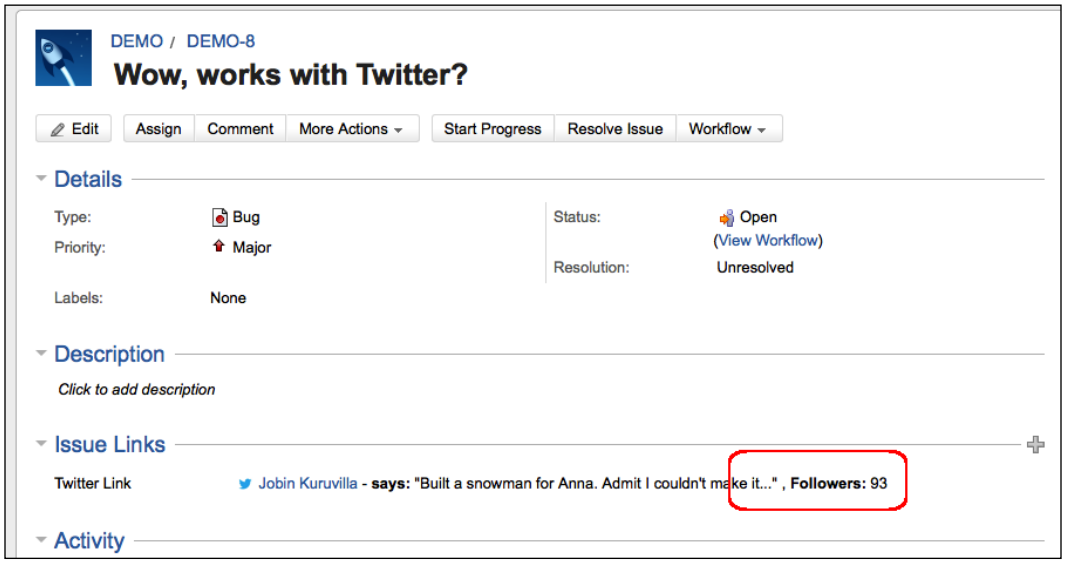
```
@Override
public boolean requiresAsyncLoading(RemoteIssueLink
remoteIssueLink) {
  return true;
}
```

3. Create the Velocity template required for `final-view`, `twitterlink-final` in this case:

```
<p>
    #renderIssueLinkIcon(${iconUrl} ${iconTooltip} ${iconTooltip}
${textutils})
    <span title="${textutils.htmlEncode($tooltip)}">
        <a href="${url}" class="link-title">${textutils.
htmlEncode($title)}</a>
    </span>
    #if ($status) -
    <span title="Status: $status">
      <span class="status"><b>says:</b> "$status"</span>
    </span>
    #end
    #if ($followers) ,
    <span title="Followers: $followers">
     <span class="followers"><b>Followers:</b> $followers</span>
    </span>
    #end
</p>
```

As you can see, this is pretty much the same as the first template, except for the part where the follower count is added.

And now, the link is rendered as follows:



How's that?

How it works...

Be it Twitter or any other application, 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 **View Issue** page. It is wise to provide only the basic information here, in order to avoid delays in loading the issue details page.
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, like getting more details from the remote link.

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 did so in the example just to show how both are loaded separately!

See also

- The *Programming issue links* recipe in *Chapter 7*, *Programming Issues*

Extending a webwork action to add UI elements

In *Chapter 2*, *Understanding the Plugin Framework*, we have seen how to extend a webwork action. In this recipe, let us work on it and see how we can add more UI elements to an existing JIRA form.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it...

Let's consider a simple example this. The clone issue operation in JIRA creates a copy of the original issue with the value of almost all of its fields copied across, barring few fields such as issue key, created date, updated date, estimates, number of votes, and so on.

What if we want to copy across the number of votes on the issue as well? Let's say we want to add a checkbox on the clone issue form, leaving the decision to the user whether to copy the votes across or not. If the user selects the checkbox, votes will be copied across and if not, the cloned issue will be created with zero votes as it happens in JIRA by default. This example will give a rough idea about adding new UI elements on JIRA forms and using them in the action classes.

The following is the step-by-step process that can be used to implement our example:

1. Override the JIRA web action for clone issue by creating an entry for it in `atlassian-plugin.xml`:

```
<webwork1 key="clone-issue" name="Clone Issue with new UI
elements" class="java.lang.Object">
  <description>Sample Webwork action with extended UI elements</
description>
  <actions>
    <action name="com.jtricks.web.action.
ExtendedCloneIssueDetails" alias="CloneIssueDetails">
```

```
       <view name="input">/secure/views/extended-cloneissue-start.
jsp</view>
      <view name="error">/secure/views/extended-cloneissue-start.
jsp</view>
    </action>
  </actions>
</webwork1>
```

As we have seen in *Chapter 2*, *Understanding the Plugin Framework*, the alias of the action remains the same as that of the clone issue action. We have gone for a custom class `ExtendedCloneIssueDetails` that extends the JIRA action class `CloneIssueDetails`.

Here, we also use a copy of the original JSP, named `extended-cloneissue-start.jsp`, just to keep track of the modified files. It is entirely possible to modify directly on the JIRA-supplied `cloneissue-start.jsp`.

2. Create the new action class extending the original action class. It is also possible to create an entirely new action class, but extending the original action will be easier as it leaves us with the task of only adding the extra bits that are needed.

```
public class ExtendedCloneIssueDetails extends CloneIssueDetails {

  public ExtendedCloneIssueDetails(ApplicationProperties
applicationProperties, PermissionManager permissionManager,
IssueLinkManager issueLinkManager, RemoteIssueLinkManager
remoteIssueLinkManager, IssueLinkTypeManager
issueLinkTypeManager, SubTaskManager subTaskManager,
AttachmentManager attachmentManager, FieldManager fieldManager,
IssueCreationHelperBean issueCreationHelperBean, IssueFactory
issueFactory, IssueService issueService) {
  super(applicationProperties, permissionManager,
  issueLinkManager, remoteIssueLinkManager,
  issueLinkTypeManager,  subTaskManager, attachmentManager,
  fieldManager, issueCreationHelperBean, issueFactory,
  issueService);
  }
  ...
}
```

3. Declare a variable for the new field we are going to add in the clone issue form. The variable name will be the same as the name of the UI element.

   The variable that we declare should match the type of the UI element. For example, a checkbox will have a Java Boolean as the variable type.

```
private boolean cloneVotes = false;
```

   Different UI elements are mapped to different Java types, such as text fields to Java string, number fields to long, and so on.

4. Create getters and setters for the new field:

```
public boolean isCloneVotes() {
  return cloneVotes;
}

public void setCloneVotes(boolean cloneVotes) {
  this.cloneVotes = cloneVotes;
}
```

   These getter and setter methods will be used to get values from and to the JSPs.

5. Add the new UI element into the JSP. In our case, the JSP is `extended-cloneissue-start.jsp`, which is a copy of `cloneissue-start.jsp`. The UI element that we add should follow the rules of the templates used in the version of JIRA we are using. This is important as the UI keeps changing between versions.

   As in JIRA 5.1, the new checkbox can be added as follows:

```
<page:applyDecorator name="auifieldgroup">
  <aui:checkbox label="'Clone Votes?'" name="'cloneVotes'"
fieldValue="'true'" theme="'aui'">
    <aui:param name="'description'">
      <ww:text name="'Clone the votes on the original issue?'"/>
    </aui:param>
  </aui:checkbox>
</page:applyDecorator>
```

   Note that the checkbox field has the same name as that of the class variable in the action class. The rest of the code revolves around the decorator to use, attributes and elements to be passed, and so on.

   Put it under the `<page:applyDecorator name="auifieldset">` tag, in case you are wondering where to put the previous code.

6. Capture the checkbox value in the action class when the form is submitted. The value can be retrieved in the class using the getter method. We can now do all the things that we want using the checkbox value.

   In this case, if the `cloneVotes` checkbox is checked, the votes field should be copied across from the original issue to the cloned one. As we have seen in the previous chapter while discarding field values during cloning, we can override the `setFields` method to do that.

```
@Override
protected void setFields() throws FieldLayoutStorageException {
  super.setFields();
  if (isCloneVotes()) {
    getIssueObject().setVotes(getOriginalIssue().getVotes());
  }
}
```

The important bit here is that the checkbox value is passed across to the action class behind the scenes and we are using the same to decide whether to copy the votes' values across.

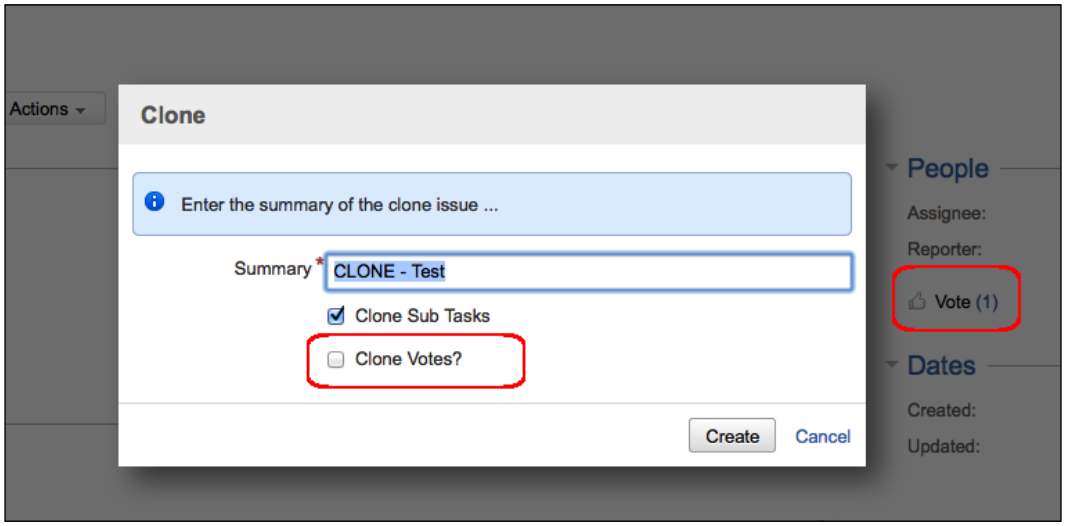
7. Package the plugin, deploy it, and see it in action. Don't forget to copy the modified JSP file to the `/secure/views` folder.



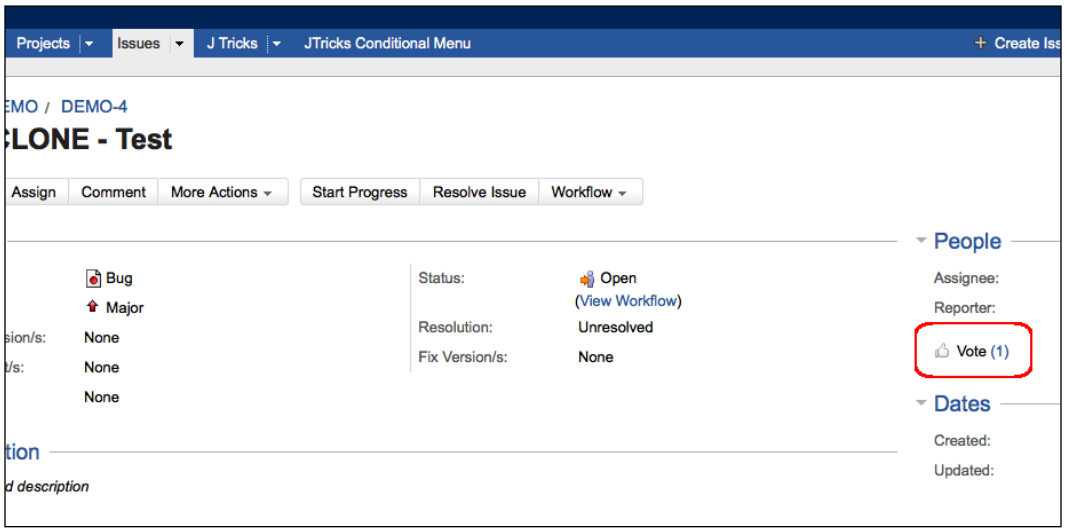
An action can be overridden only once. Care must be taken not to override it again in another plugin (it might be a third party one), as only one will be picked up.

How it works...

Once the plugin is deployed and the JSP file is copied across to the right-hand side location, we can find the modified UI while cloning an issue. The form will have the new field, **Clone Votes?**, as shown in the following screenshot:



Note that the issue we are cloning has one vote on it. If the `cloneVotes` field is checked, the cloned issue will have the votes on it, as shown in the following screenshot:



A similar approach can be used to add any new fields into the JIRA forms and use the fields in the action classes, as required. You can find more information about the various elements and its attributes by looking at the `webwork.tld` file, residing in the `WEB-INF/tld/` folder.

See also

- The *Extending a webwork action in JIRA* recipe in *Chapter 2*, *Understanding the Plugin Framework*

Displaying dynamic notifications/warnings on issues

JIRA has an interesting feature, the **Announcement Banner**, which can be used to make announcements to its user community via JIRA itself. But sometimes, it isn't enough to satisfy all its users. Power users of JIRA sometimes want to see warnings or notifications while they are viewing an issue based on some attributes of the issue.

In this recipe, we will see how to add a warning or error message on an issue based on whether the issue has subtasks or not.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

Here, as in the previous recipe, the core logic is in extending the JIRA action and modifying the existing JSP files.

In this recipe, we will be overriding the `ViewIssue` action and the constructor for this action needs the `PagerManager` class. Unfortunately, `PagerManager` is not available for v2 plugins yet and hence we will have to write this as a v1 plugin.

How to do it...

The following are the steps to display warnings/errors based on the number of subtasks on a standard issue type:

1. As in the previous recipe, extend the JIRA action (in this case, the `view` action) by adding a webwork module in `atlassian-plugin.xml`:

```
<webwork1 key="view-issue" name="View Issue with warning"
class="java.lang.Object">
  <description>View Issue Screen with Warnings</description>
    <actions>
      <action name="com.jtricks.web.action.ExtendedViewIssue"
alias="ViewIssue">
        <view name="success">/secure/views/issue/extended-
viewissue.jsp</view>
        <view name="issuenotfound">/secure/views/issuenotfound.
jsp</view>
        <view name="permissionviolation">/secure/views/
permissionviolation.jsp</view>
        <command name="moveIssueLink" alias="MoveIssueLink">
          <view name="error">/secure/views/issue/viewissue.jsp</
view>
        </command>
      </action>
    </actions>
</webwork1>
```

2. Override the action class (`ViewIssue`) with `ExtendedViewIssue`, previously mentioned, and add a public method to check whether the issue has any subtasks or not. Make sure the method is public so that it can be invoked from the JSP.

```
public boolean hasNoSubtasks(){
  return !getIssueObject().isSubTask() && getIssueObject().
getSubTaskObjects().isEmpty();
}
```

3. It just checks whether the issue is a subtask or not and returns true if it is a standard issue-type and has no subtasks of its own.
4. Modify the `extended-viewissue.jsp` file to add a warning at the top, if the issue has no subtasks. Here we add a condition to check if the public method returns true, and if so, the warning is added as follows:

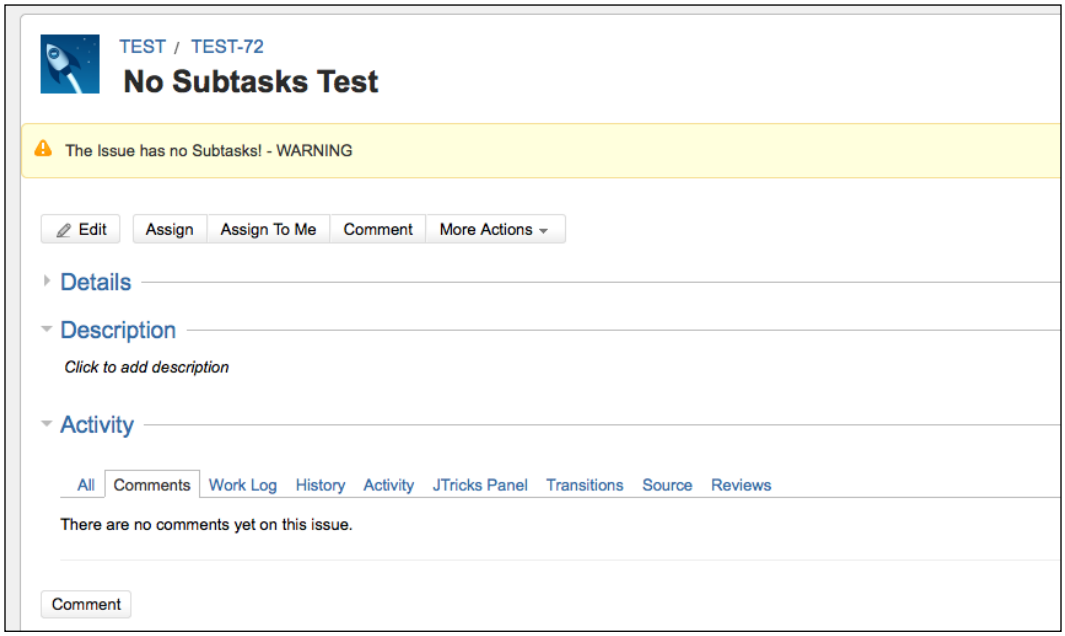
```
<ww:if test="/hasNoSubtasks() == true">
  <div class="aui-message warning">
    <span class="aui-icon icon-warning"></span>
    The Issue has no Subtasks!  - WARNING
  </div>
</ww:if>
```

   As you can see, we have used the JIRA styles to add a warning icon and a warning `div` container.

5. Package the plugin and deploy it to see it in action.

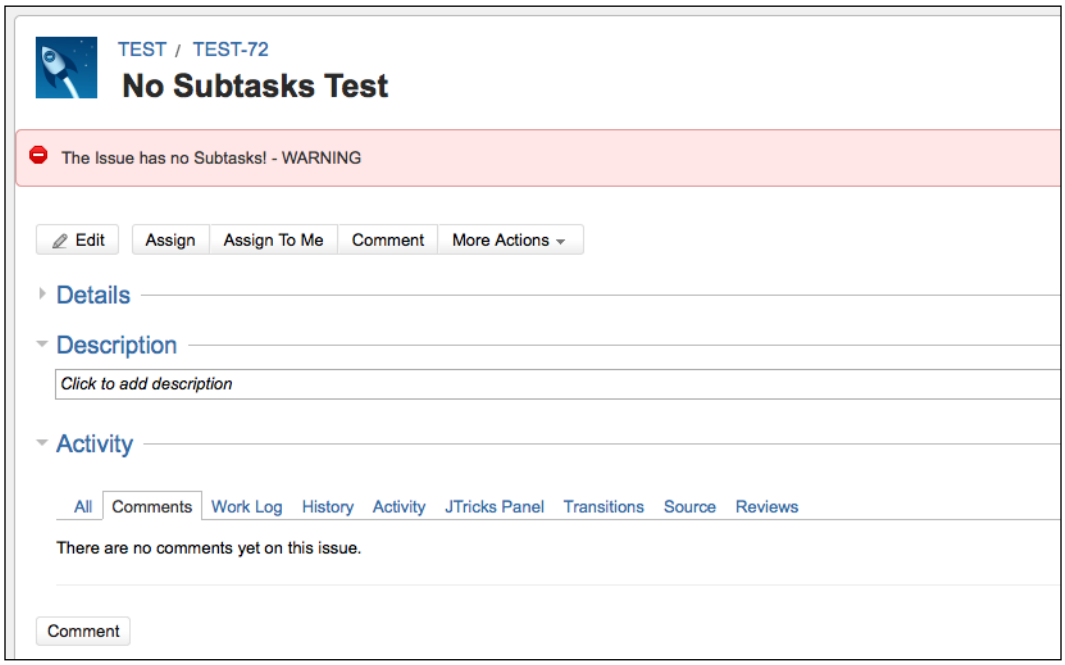
How it works...

Once the plugin is deployed, the user will see a warning, as shown in the following screenshot, if a standard issue has no subtasks—assuming the warning code was added immediately after the summary field:



Just by modifying the CSS to add error styles, the message will appear as shown:

```
<ww:if test="/hasNoSubtasks() == true">
  <div class="aui-message error">
    <span class="aui-icon icon-error"></span>
    The Issue has no Subtasks! - ERROR
  </div>
</ww:if>
```

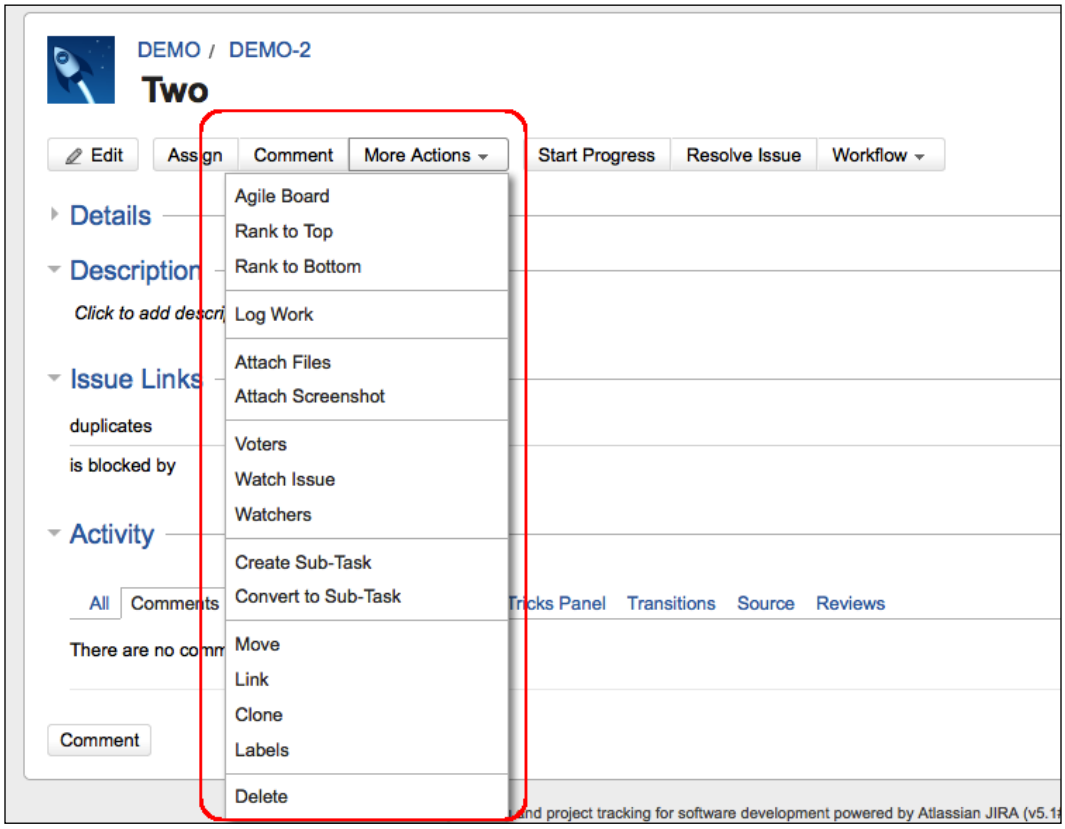


See also

- The *Extending a webwork action in JIRA* recipe in *Chapter 2*, *Understanding the Plugin Framework*

Re-ordering issue operations in the View Issue page

In the previous chapter, we have seen 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 re-order those actions without actually doing any coding! For example, let's assume we want to move **Delete** as the first option in the list and then move the subtask operations up the chart.

How to do it...

The following is the step-by-step process to reorder the issue operations:

1. Go to the `system-issueoperations-plugin.xml` file residing under the `WEB-INF/classes` folder. This is the file where all the issue operations are defined.
2. Modify the `weight` attribute on the relevant plugin modules to order them.

   `weight` is the attribute that defines the order of JIRA web fragments. Issue operations post JIRA 4.1.*x* are stored as web fragments and hence are reordered using `weight`.

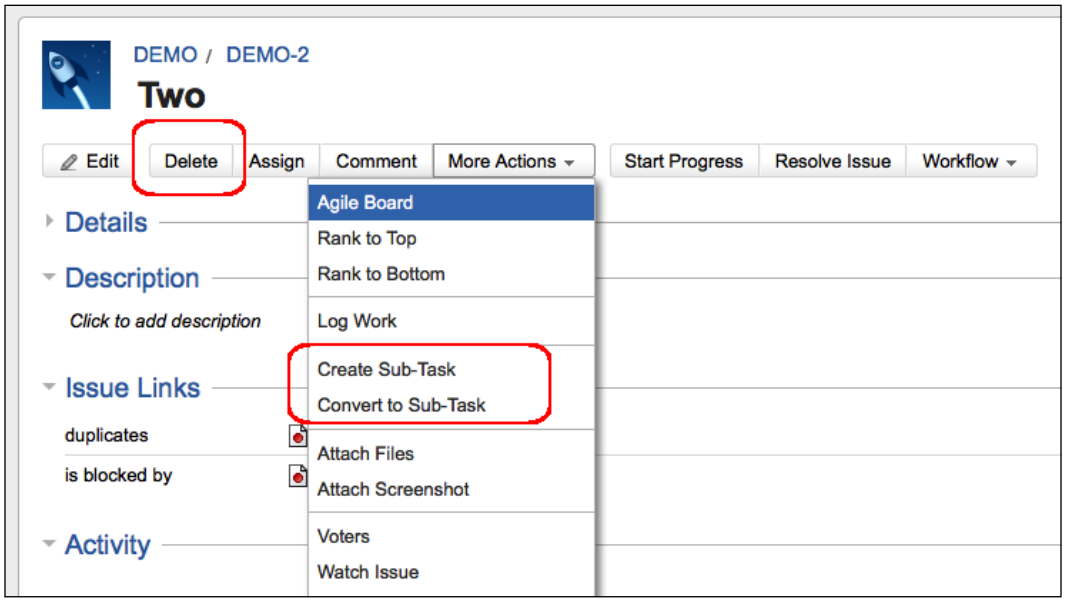
   Prior to JIRA 4.1, issue operations were defined using the Issue Operations plugin module instead of Web Items. In those modules, the `order` attribute was used to define the ordering, which is an equivalent to the current `weight` attribute.

   Lower the weight, and the item will appear first.

3. Save the file and restart JIRA.

How it works...

In our example, we wanted the **Delete** operation to appear first. As `10` is the lowest weight value by default, if we give a weight of `5` to the `delete` web section, that is, `operations-delete`, it will appear first in the list, as shown in the following screenshot. Similarly, we can reorder every other action, as we have done for the subtask operations as well.



See also

- The *Adding new issue operations* recipe in *Chapter 7*, *Programming Issues*

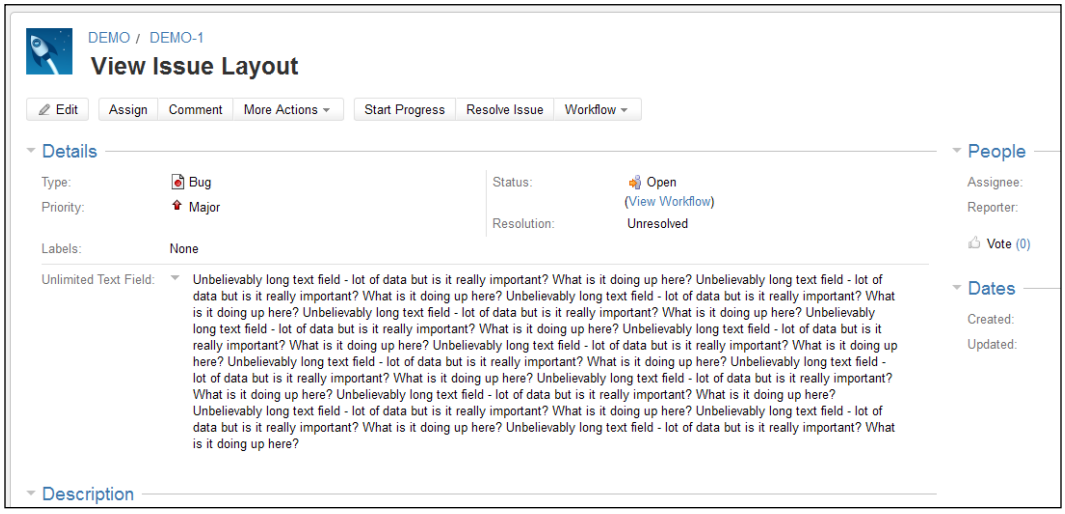
Re-ordering fields in the View Issue page

It is always difficult to satisfy everyone in a big user community and so it proves with the JIRA's **View Issue** page. While some people love it, some think there are simple improvements possible that would result in huge customer satisfaction.

One such area is the layout of the **View Issue** page. While it is neatly organized in terms of the code, the order in which they appear seems to be a strong contender for change in many cases.

For example, in the **View Issue** page, the summary of the issue is followed by standard issue fields like **Status**, **Priority**, **versions**, **Component/s**, and so on. It is then followed by the 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.

The **View Issue** page looks as follows when you have a large custom field:



As you can see, the **Unlimited Text Field** field has a huge value and the **Description** field is way down on the screen. In this recipe, we will see how we can re-order some of the things.

How to do it...

In earlier versions of JIRA, the **View Issue** page was rendered purely as `/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 for this particular example, perform the following steps:

1. Go to the `atlassian-plugin.xml` file of the bundled plugin.

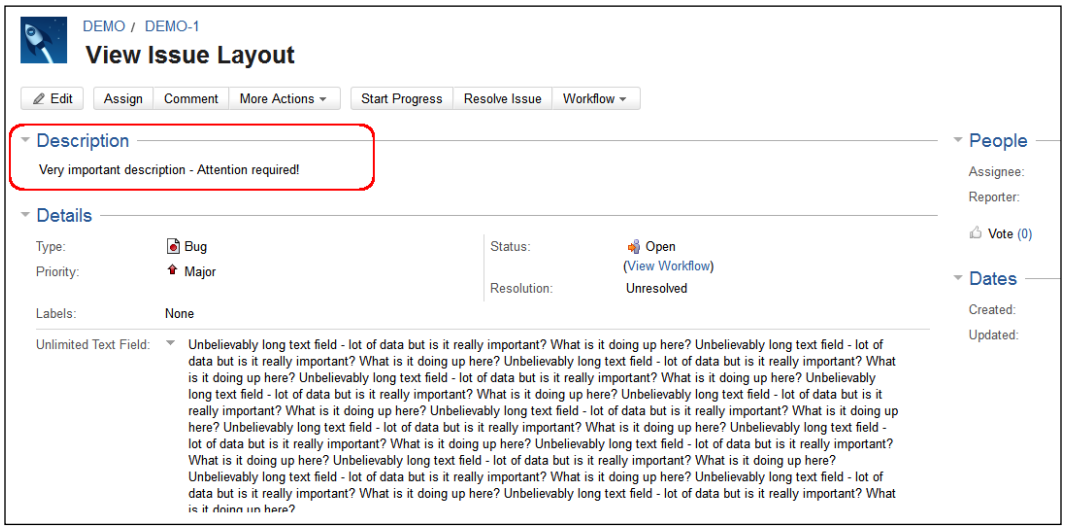
   Identify the definition of the description module. It looks as follows:

   ```
   <!-- 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 on the screen is defined by the `weight` attribute.

2. Modify the `weight` attribute to have a value that is less than that of the details module. The weight of the **Details** block is `100` as you can see in the `atlassian-plugin.xml` and hence we will have to modify the weight of description module to something less than `100`, say `50`.
3. Save everything and put it back as described in the *Modifying Atlassian bundled plugins* recipe in *Chapter 2*, *Understanding the Plugin Framework*.
4. Restart JIRA.

Once this is done, the modified **View Issue** page will look as follows:



There's more...

We can even wrap the custom fields in a separate module. A little more work but sometimes worth it!

Wrapping custom fields in a separate module

As you saw earlier, sections on the **View Issue** page such as **Details**, **Description**, and so on, are different web panels. The following is how we can add a new section:

1. Go to `atlassian-plugin.xml` of `jira-view-issue-plugin.xxx.jar`, which is a bundled plugin.

   Add a new web panel definition for custom fields:

   ```
   <!-- Customfield panel -->
   <web-panel key="cf-module" location="atl.jira.view.issue.left.
   context" weight="150">
           <context-provider class="com.atlassian.jira.plugin.
   viewissue.DetailsBlockContextProvider"/>
           <resource name="view" type="velocity" location="viewissue/
   customfieldblock.vm"/>
           <label>Custom Fields</label>
   </web-panel>
   ```

   As with any web panels, we need a context provider and a Velocity template to render the view using that context. Since the custom fields are already rendered in the details module, we can reuse the same context provider to make things easier!

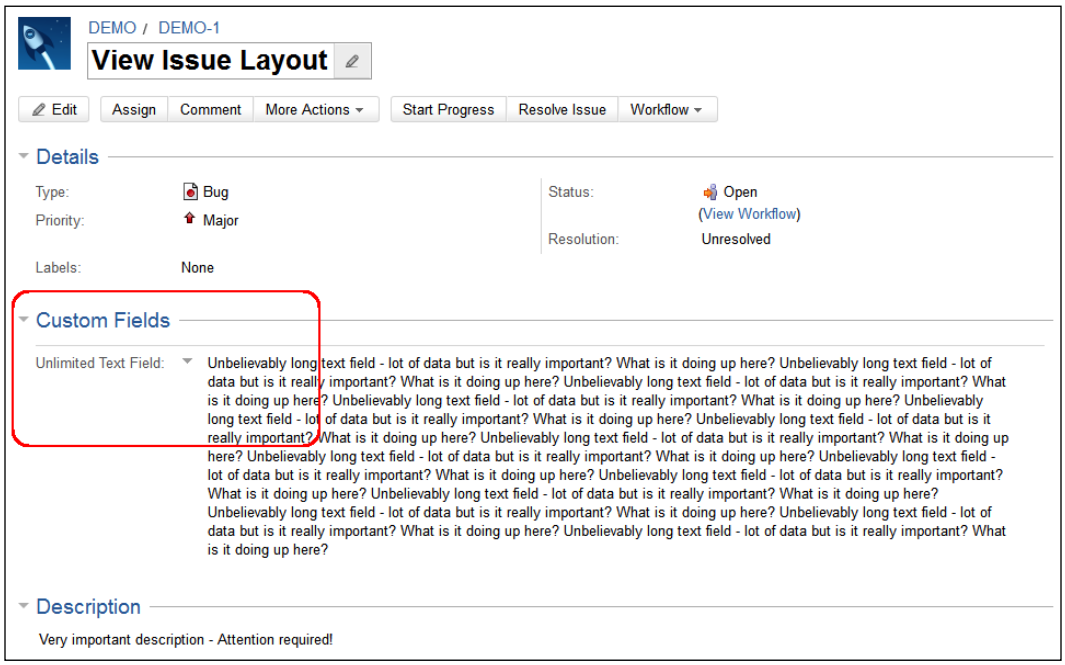
2. Create a new Velocity template to render the new module. In our case it is `customfieldblock.vm` under the `viewissue` folder. The template will actually hold the code that is currently present in `detailsblock.vm` to render the custom fields. The following is the code in JIRA 5.1:

```
#enable_html_escaping()
#if (!${tabs.empty})
<div id="customfieldmodule">
    #if (${tabs.size()} > 1)
## Show tab headings
        <div class="tabwrap tabs2">
            <ul id="customfield-tabs" class="tabs horizontal">
                #foreach (${tab} in ${tabs})
                <li id="tabCell${velocityCount}"
#if(${velocityCount} == 1) class="active" #end>
                    <a rel="${velocityCount}"
href="#"><strong>${tab.name}</strong></a>
                </li>
                #end
            </ul>
        </div>
    #end
## Show the actual tabs with their fields
    #foreach (${tab} in ${tabs})
    <ul id="tabCellPane${velocityCount}" class="property-list
#if(${velocityCount} != 1) hidden #end #if(${tabs.size()} > 1) pl-
tab #end">
## Show tab's fields
        #foreach(${field} in ${tab.fields})
            #if (${field.showField})
                <li id="rowFor${field.id}" class="item">
                    <div class="wrap">
                        <strong title="${field.name}"
class="name">${field.name}:</strong>
                        <div id="${field.id}-val" class="value
${field.styleClass}" data-fieldtype="${field.fieldType}">
                            ${field.fieldHtml}
                        </div>
                    </div>
                </li>
            #end
        #end
    </ul>
    #end
</div>
#end
```

You must copy this code from `detailsblock.vm` of your JIRA version instead of copying it from here. Otherwise, you might introduce bugs or end up losing changes introduced in new JIRA versions.

3. Remove the previously copied code from `detailsblock.vm` to avoid duplication.
4. Save the changes to the bundled plugin 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 as follows:



How's that for a change?

See also

- The *Modifying Atlassian bundled plugins* recipe in *Chapter 2*, *Understanding the Plugin Framework*

9
Remote Access to JIRA

In this chapter we will cover:

- Writing a Java client for the REST API
- Creating a SOAP client
- Writing a Java XML-RPC client
- Working with issues
- Working with attachments
- Remote time tracking
- Working with comments
- Remote user and group management
- Progressing an issue in a workflow
- Managing versions
- Managing components
- Remote administration methods
- Exposing services and data entities as REST APIs
- Deploying a SOAP service in JIRA
- Deploying an XML-RPC service within JIRA

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 exposes its functionalities via **REST**, **SOAP**, or **XML/RPC** interfaces. Only a selected JIRA functionality is exposed via these interfaces but JIRA also lets us extend these interfaces. In this chapter, we will learn how to communicate with JIRA using these interfaces and add more methods into these interfaces by developing plugins. As REST is the preferred method for remote access in JIRA5+, the focus of this chapter is more on REST with examples of other interfaces as well. The core principle for all the interfaces is the same.

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 `http://confluence.atlassian.com/display/JIRADEV/JIRA+RPC+Services`.

Writing a Java client for the 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

Make sure that the **Accept remote API Calls** option is turned **ON** in JIRA under **Administration** | **System** | **General Configuration**.

How to do it...

In order to connect to JIRA using REST APIs, Atlassian has developed a JIRA REST Java client library called JRJC in short. 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 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 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. We can do this by carrying out the following 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</artifactId>
  <version>2.0.0-m2</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/public/com/atlassian/jira/jira-rest-java-client/`. We are using 2.0.0-m2 as the 2.x Version supports `CREATE` methods. 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`.

2. Create a Java project by running `maven eclipse:eclipse` if you are using Eclipse or create the project using your favorite IDE and add all dependencies listed earlier in the class path.

   Once done, create a standalone Java class.

3. Create a connection to the JIRA server.

```
final AsynchronousJiraRestClientFactory factory = new
AsynchronousJiraRestClientFactory();
final URI uri = new URI("http://localhost:8080/");
final JiraRestClient jiraRestClient = factory.createWithBasicHttpA
uthentication(uri, "userName", "password");
```

   Here, we instantiate the `AsynchronousJiraRestClientFactory` method and use the `createWithBasicHttpAuthentication` method to instantiate the REST client by passing the username and password.

   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 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.

```
IssueRestClient issueClient = jiraRestClient.getIssueClient();
UserRestClient userClient = 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 the JIRA REST API is evolving so quickly, JRJC has a lot of potential and is worth investing time in.

Creating a SOAP client

SOAP was the preferred mode of remote access in JIRA prior to JIRA5. SOAP still has the most methods compared to REST or XML/RPC and is probably used the most in the plugins that we find around us. In this recipe, we will start with the basics and see how we can write a simple SOAP client.

Getting ready

Install Maven 2 and configure a Java development environment. Make sure the RPC plugin is enabled in JIRA and the **Accept remote API calls** option is turned **ON** at **Administration** | **System** | **General Configuration**.

How to do it...

The following are the steps to create a JIRA SOAP client:

1. Download the latest demo SOAP client distribution from the Atlassian public repository available at `https://studio.plugins.atlassian.com/svn/JRPC/tags/`. This contains a Maven 2 project configured to use Apache Axis, and a sample Java SOAP client, which creates test issues at `http://jira.atlassian.com`.
2. Modify the `jira.soapclient.jiraurl` property in the `pom.xml` file to point to your JIRA instance, the instance you want to connect to. By default, it points to `http://jira.atlassian.com`.
3. Download the WSDL file of the instance you want to connect to. You can find the WSDL file under the `/src/main/wsdl` location. If it is not there, or if you want to download the WSDL afresh, run the following command:

   ```
   mvn -Pfetch-wsdl -Djira.soapclient.jiraurl=http://{your_jira_instance}/
   ```

4. This will download the WSDL from the configured JIRA instance (as in *step 2*) to `/src/main/wsdl/`. Skip the `jira.soapclient.jiraurl` property to download the Atlassian JIRA WSDL.
5. Create the client JAR. We can do this by running the following command to generate the sources from the WSDL and create the SOAP client:

   ```
   mvn -Pbuildclient
   ```

6. This will generate a JAR file with all the necessary classes required. There is a second JAR file created with the dependencies (such as the axes) embedded in it. The latter will do a world of good if you are executing it from an environment without axes and other dependencies already configured in it.
7. Write the client program. Let us go with the simplest approach in this recipe, that is, to create a simple standalone Java class in Eclipse. Start by creating an Eclipse project by running the following command:

   ```
   mvn eclipse:eclipse
   ```



You can alternatively try other IDEs or even run from the command prompt, whichever is convenient. Make sure you add the client JAR created in *step 4* in the classpath. Now we're all set to write a simple program that just logs into our JIRA instance. From now on, it is just another web service invocation as detailed in the following steps.

8. Create the standalone Java class.
9. Get the SOAP service locator:

   ```
   JiraSoapServiceService jiraSoapServiceLocator = new
   JiraSoapServiceServiceLocator();
   ```

10. Get the SOAP service instance from the locator by passing the URL of your JIRA instance:

    ```
    JiraSoapService jiraSoapService = jiraSoapServiceLocator.
    getJirasoapserviceV2(new URL(your_url));
    ```

11. Start accessing the methods using the SOAP service instance. For example, the log in can be done as follows:

    ```
    String token = jiraSoapService.login(your_username, your_
    password);
    ```

    The token retrieved here is used for all the other operations instead of logging in every time. You can see the token as the first argument in all the other operations.

12. With that, our SOAP client is ready. Let's just try getting an issue using the key and print its key and ID to prove that this stuff works!

    ```
    RemoteIssue issue = jiraSoapService.getIssue(authToken, ISSUE_
    KEY);
    System.out.println("Retrieved Issue:"+issue.getKey()+" with
    Id:"+issue.getId());
    ```

    You will find the output printed with the issue key and ID.

Hopefully, this gives you a fair idea to get started with your first SOAP client! There is a lot more you can do with the SOAP client, some of which we will see in the following recipes.

Writing a Java XML-RPC client

We have already seen how to create a REST client or SOAP client and use it to connect to JIRA from an external third-party application. In this recipe, we will see how to invoke an XML-RPC method from a client application written in Java.

The Javadocs for the XML-RPC client can be found at `http://docs.atlassian.com/software/jira/docs/api/rpc-jira-plugin/latest/com/atlassian/jira/rpc/xmlrpc/XmlRpcService.html`.

Getting ready

Make sure the **Accept remote API Calls** option is turned **ON** in JIRA under **Administration** | **System** | **General Configuration**.

How to do it...

Let's try to retrieve the list of projects using the XML-RPC service deployed within JIRA. The following are the steps:

1. Create a Maven 2 project and add the dependency for `Apache2 xml-rpc` libraries.

```
<dependency>
  <groupId>xmlrpc</groupId>
  <artifactId>xmlrpc</artifactId>
  <version>2.0</version>
</dependency>
```

   Note that the version of the `xml-rpc` libraries we have used in this recipe is Version 2.0. You can find the version used by your JIRA instance by looking at the `xmlrpc-xx.jar` file under the `WEB-INF/lib` folder. For this version, you also need the `commons-codec` dependency.

```
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.4</version>
</dependency>
```

2. Create a Java client. In this example, we will create a standalone Java class with all the libraries in the classpath.

3. Instantiate the `XmlRpcClient` object using the following code:

```
XmlRpcClient rpcClient = new XmlRpcClient(JIRA_URI + RPC_PATH);
```

Here, the `JIRA_URI` is the URI of your JIRA instance, `http://localhost:8080` for example, and `RPC_PATH` will be `/rpc/xmlrpc`, which will be the same even for new methods exposed via plugins. In this case, the complete path will be: `http://localhost:8080/rpc/xmlrpc`.

Note that we are using XML-RPC v2 here. Check out the syntax for the version you are using!

4. Log in to Java XML-RPC client JIRA by invoking the `login` method as shown:

```
// Login and retrieve logon token
Vector loginParams = new Vector(2);
loginParams.add(USER_NAME);
loginParams.add(PASSWORD);
String loginToken = (String) rpcClient.execute("jira1.login",
loginParams);
System.out.println("Logged in: " + loginToken);
```

As you can learn from the Javadocs, the method expects a username and password, which are passed into the `execute` method on the client as a `Vector` object. The first argument is the method name which is preceded with the namespace under which the methods are exposed. In this case, the namespace is `jira1` and is equivalent to the service path we have seen in the previous recipe. The complete method name will hence become `jira1.login`.

In the case of the `login` method, the return object is an authentication token which is a String object.

5. Retrieve the list projects using the `getProjectsNoSchemes` method:

```
// Retrieve projects
Vector loginTokenVector = new Vector(1);
loginTokenVector.add(loginToken);
List projects = (List) rpcClient.execute("jira1.
getProjectsNoSchemes",   loginTokenVector);
```

Here again, we need to send a `Vector` object as input along with the method name, in this case, with the authentication token in the `Vector` object. If we need to invoke a method that needs a complex object in scenarios such as creating an issue, we should first create a `HashTable` object Java XML-RPC client with the input parameters as key/value pairs and add it into the `Vector` object.

The return type in this case is type cast into a `List`. This will be a `List` of map objects, each map representing a `RemoteProject` object with the details of the project in it as key/value pairs. For example, the name of the project can be accessed from the map using the key name, as shown in the following step.

6. Retrieve the details of projects from the list. The details will be the attributes of the project published with the getter/setter methods in the `RemoteProject` object, such as name, lead, and so on.

```
for (Iterator iterator = projects.iterator(); iterator.hasNext();)
{
  Map project = (Map) iterator.next();
  System.out.println(project.get("name") + " with lead " +
project.get("lead"));
}
```

   As mentioned in the previous step, the details can be retrieved as key/value pairs from the `Map` objects representing a project. This same logic applies to all `XML-RPC` methods where complex objects are retrieved as maps with key/value pairs in them.

7. Log out from JIRA:

```
Boolean bool = (Boolean) rpcClient.execute("jira1.logout",
loginTokenVector);
```

   Here the output is converted to Boolean as the method returns a Boolean.

Working with issues

So far we have seen how to write the clients for REST, SOAP, and XML/RPC methods. Now it is time to move on to some real examples. In this recipe, we will have a look at creating an issue using REST APIs.

Getting ready

Create a JIRA REST client as mentioned in the *Writing a Java client for the REST API* recipe.

How to do it...

The following are the steps to create an issue with the standard fields populated on it:

1. As mentioned in the *Writing Java clients for the REST API* recipe, initialize the Rest client:

```
final AsynchronousJiraRestClientFactory factory = new
AsynchronousJiraRestClientFactory();
final URI uri = new URI("http://localhost:8080/");
final JiraRestClient jiraRestClient = factory.createWithBasicHttpA
uthentication(uri, "userName", "password");
```

2. Retrieve the `IssueRestClient` from `JiraRestClient` as we are dealing with issues in this recipe:

```
IssueRestClient issueClient = jiraRestClient.getIssueClient();
```

3. Initialize the `IssueInputBuilder` object using the project key, `issuetype Id`, and `Summary`.

```
IssueInputBuilder issueInputBuilder = 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 an ID.

   Whether to use the name or ID, for various fields, can be found at Javadocs (`http://docs.atlassian.com/jira-rest-java-client/0.6-m9/apidocs/com/atlassian/jira/rest/client/domain/input/IssueInputBuilder.html`).

5. Set the custom field values on the issue. The way to set values into `IssueInputBuilder` differs based on the type of the field. The following are some examples for the most-used field types.

   - Setting values for single values fields is fairly easy.

```
issueInputBuilder.setFieldValue("customfield_10000", "Test
text Val")
```

   - For a multi-user picker, you need to send a list of `User` objects.

```
User jobin = jiraRestClient.getUserClient().
getUser("jobinkk").get();
User testUser = jiraRestClient.getUserClient().
getUser("test").get();
issueInputBuilder.setFieldValue("customfield_10000", Arrays.
asList(jobin, testUser));
```

- For a multi-valued field, such as 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", "Two"))
```

- For a multi select list, you need to set a list of `ComplexIssueInputFieldValue` objects.

```
List<ComplexIssueInputFieldValue> fieldList = new ArrayList<
ComplexIssueInputFieldValue>();
String[] valuesList = new String[] { "10", "20" };
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);
```

- 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", "Retail");
cascadingValues.put("child", ComplexIssueInputFieldValue.
with("value", "Two"));
issueInputBuilder.setFieldValue("customfield_10000", new Com
plexIssueInputFieldValue(cascadingValues));
```

6. In all the examples, `customfield_10000` denotes the custom field ID. Other field types such as number fields, check boxes, and so on, will fit into one of the preceding categories.
7. Build the `IssueInput` object from the builder.

```
IssueInput issueInput = issueInputBuilder.build();
```

8. Create the issue using the `IssueRestClient` object.

```
Promise<BasicIssue> createdIssue = jiraRestClient.
getIssueClient().createIssue(issueInput);
Issue issue = createdIssue.get();
```



Updating issues is not supported as I write this but you can watch `https://ecosystem.atlassian.net/browse/JRJC-64` to get an update on it.

Browsing an existing issue couldn't be any simpler.

```
final Promise<Issue> issue = jiraRestClient.getIssueClient().
getIssue(issueKey);
Issue browsedIssue = issue.get();
```

While reading field values, standard fields are straightforward using getter methods whereas custom fields differ based on the type.

```
Iterable<Field> fields = browsedIssue.getFields();
```

If the field is a multiuser field, you can retrieve the values as shown here:

```
JSONArray array = (JSONArray) field.getValue();
for (inti = 0; i<array.length(); i++) {
  JSONObject obj = 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 multi-select fields, use `value` instead of `displayName`.

```
JSONArray array = (JSONArray) field.getValue();
for (inti = 0; i<array.length(); i++) {
  JSONObject obj = array.getJSONObject(i);
  System.out.println("Value:" + obj.getString("value"));
}
```

For single select, the field value will be a `JSON` object instead of array and everything else will be the same.

For cascading select:

```
JSONObject obj = (JSONObject) field.getValue();
String parentVal = obj.getString("value");
String childVal = obj.getJSONObject("child").getString("value");
```

For normal fields:

```
Object value = field.getValue();
```

Hopefully, that covers creating and browsing issues using REST.

Working with attachments

In this recipe, we will see how to add attachments on an issue via REST and browse existing attachments.

Getting ready

Create a JIRA REST client as mentioned in the *Writing a Java client for the 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 on to an issue. The following are the three options and how they are used.

Using the input stream and a new filename

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 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.

   ```
   jiraRestClient.getIssueClient().addAttachment(attachmentURI , in,
   "file1.txt");
   ```

   Here the last argument, `file1.txt`, is the name with which the file appears on the issue.

Using AttachmentInput

This is very similar to the preceding case. The only difference is that an `AttachmentInput` object is created before calling the `addAttachments` method. Note that the method name is `addAttachments`.

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 = jiraRestClient.getIssueClient().
   getIssue(key);
   Issue browsedIssue = issue.get();
   ```

3. Get the attachment URI from the issue.

   ```
   URI attachmentURI = browsedIssue.getAttachmentsUri();
   ```

4. Create the `AttachmentInput` object.

   ```
   AttachmentInput input = new AttachmentInput("file2.txt", in1);
   ```

5. Add the attachment.

   ```
   jiraRestClient.getIssueClient().addAttachments(attachmentURI ,
   input);
   ```

Using file and a new filename

Here we use a `file` object instead of creating `InputStream`.

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 = jiraRestClient.getIssueClient().
   getIssue(key);
   Issue browsedIssue = issue.get();
   ```

3. Get the attachment URI from the issue.

   ```
   URI attachmentURI = browsedIssue.getAttachmentsUri();
   ```

4. Add the attachment.

   ```
   jiraRestClient.getIssueClient().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.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 the users log the 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 the 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 the third-party products—when it is necessary to log the work using remote APIs. In this recipe, we will see how to log work using the REST API.

Getting ready...

Create a JIRA REST client as mentioned in the *Writing a Java client for the 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:

```
WorklogInput  worklogInput  = 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:

```
WorklogInput  worklogInput  = 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 the adjustment type and its meaning 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:

```
WorklogInput  worklogInput  = 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:

```
WorklogInput  worklogInput  = 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:

```
WorklogInput  worklogInput  = 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 the `addWorklog` method:

```
jiraRestClient.getIssueClient().addWorklog(browsedIssue.
getWorklogUri(),worklogInput);
```

In all the examples, `minutesSpent` is an integer whereas `adjustEstimateValue` is a String, although both represent minutes.

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 a Java client for the REST API* recipe.

How to do it...

Adding a comment on an issue using the 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 the 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);
   ```

For updating comments, use the same methods but pass the comment URI, update author name, and update time instead of null values.

Remote user and group management

Let's now have a look at the user and group management using remote APIs. This is really useful when the users and groups need to be managed from a third-party application.

As opposed to the previous recipes, this one uses SOAP methods because user and group management is not yet available via REST.

Getting ready

Create a SOAP client as mentioned in the *Creating a SOAP client* recipe.

How to do it...

Creating a group and user are pretty straightforward. The following is how we do it once the client is created:

```
//Create group jtricks-test-group
RemoteGroup group = jiraSoapService.createGroup(authToken, "jtricks-
test-group", null);
//Create user jtricks-test-user
RemoteUser user = jiraSoapService.createUser(authToken, "jtricks-test-
user", "password", "Test User", "support@j-tricks.com");
```

Here, the first snippet creates a group with the name `jtricks-test-group`. The third argument is `RemoteUser` which can be added to the group as the first user when the group is created. We can leave it as `null` if an empty group has to be created.

The second snippet creates a user with the relevant details, such as name, password, full name, and e-mail.

A user can be added to a group as follows:

```
jiraSoapService.addUserToGroup(authToken, group, user);
```

Here, the group and user are `RemoteGroup` and `RemoteUser` objects respectively.

An existing user or group can be retrieved as follows:

```
RemoteUser user = jiraSoapService.getUser(authToken, "jtricks-test-
user");
RemoteGroup group = jiraSoapService.getGroup(authToken, "jtricks-test-
group");
```

The users in a group can be retrieved from the `RemoteGroup` object as shown here:

```
RemoteUser[] users = group.getUsers();
for (RemoteUser remoteUser : users) {
  System.out.println("Full Name:"+remoteUser.getFullname());
}
Deleting a user or group is also straightforward as shown next:
//Delete User.
jiraSoapService.deleteUser(authToken, user1.getName());
//Delete Group.
jiraSoapService.deleteGroup(authToken, group1.getName(), swapGroup.
getName());
```

Here, `swapGroup` identifies the group to change the comment and to make the worklog visible.

Progressing an issue in a 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 a Java client for the 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 = jiraRestClient.
   getIssueClient().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(resolveTransi
tion.getId(), fieldInputs, comment);
```

6. Invoke the transition method on `IssueRestClient`.

```
jiraRestClient.getIssueClient().transition(browsedIssue.
getTransitionsUri(), transitionInput);
```

Thus the issue can be transitioned to a given status.

Managing versions

We have seen how to add versions as a 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 a Java client for the 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.

```
VersionInput versionInput = new VersionInput("DEMO", "JRJC",
"Test", new DateTime(), false, false);
```

2. Use the `createVersion` method on `VersionRestClient`.

```
Promise<Version> version = jiraRestClient.getVersionRestClient().
createVersion(versionInput);
```

3. Once a version is created, you can retrieve it any time using the version URI.

```
Promise<Version> version = jiraRestClient.getVersionRestClient().
getVersion(versionURI);
```

4. 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));
```

5. 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);
```

6. You can move it after another version for which you have the URI.

```
jiraRestClient.getVersionRestClient().moveVersion(version.get().
getSelf(), anotherVersionURI);
```

7. A version can be deleted using the `removeVersion` method. Here, we need to specify another version, which will be used to replace "Fix for version" in all the issues that has current version as "Fix for version" and another version which will be used to replace "Affected version" in all the issues that has current version as "Affected version".

```
jiraRestClient.getVersionRestClient().removeVersion(version.
get().getSelf(), moveFixIssuesToVersionUri,
moveAffectedIssuesToVersionUri);
```

8. 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 in 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 a Java client for the 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.

```
ComponentInput componentInput = new ComponentInput("JRJC", "Test",
"jobinkk", AssigneeType.COMPONENT_LEAD);
```

2. Use the `createComponent` method on `ComponentRestClient`.

```
Promise<Component> component = jiraRestClient.
getComponentRestClient().createComponent("DEMO", componentInput);
```

3. Once a component is created, you can retrieve it any time using the component URI.

```
Promise<Component> component= jiraRestClient.
getComponentRestClient().getComponent(componentURI);
```

4. 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));
```

5. 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);
```

6. You can get the list of issues related to this component as follows:

```
Promise<Integer> number = jiraRestClient.getComponentClient().getC
omponentRelatedIssuesCount(component.get().getSelf());
```

This covers the most important methods for managing components in JIRA.

Remote administration methods

Before we wind up the various useful methods for using remote APIs, we can have a look at the administration methods. In this recipe, we will be concentrating 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 REST is not evolved enough to do administration tasks, we will use the SOAP API in this chapter.

Getting ready

Create the SOAP client as we discussed in the *Creating a SOAP client* recipe.

How to do it...

We can have a look at the journey of creating a permission scheme, creating a project using it, and adding some users into the project roles. Other schemes used during the creation of the project, such as the notification scheme and issue security scheme, are not supported via SOAP.

The following are the steps for our journey:

1. Create the new Permission scheme:

   ```
   RemotePermissionScheme permScheme = jiraSoapService.createPermissi
   onScheme(authToken, "Test P Scheme", "Test P Description");
   ```

   Here, we use the `createPermissionScheme` method to create a new permission scheme by passing the authentication token, a name, and a description. Note that we can instead get an existing permission scheme from the list retrieved using the `getPermissionSchemes` method.

2. Add relevant permissions to the newly created permission scheme using the `addPermissionTo` method. This step is relevant only if we are creating a new permission scheme:

   ```
   RemotePermissionScheme modifiedPermScheme = jiraSoapService.
   addPermissionTo(authToken, permScheme, adminPermission, user);
   ```

   Here, `adminPermission` should be a remote permission, which is in the list of `RemotePermission` objects retrieved using the `getAllPermissions` method. For example, the `AdministerProject` permission can be obtained as follows:

   ```
   RemotePermission[] permissions = jiraSoapService.
   getAllPermissions(authToken);
   RemotePermissionadminPermission = null;
   ```

```
for (RemotePermission remotePermission : permissions) {
  if (remotePermission.getPermission().equals(23L)) {
adminPermission = remotePermission;
    break;
  }
}
```

Here, `23L` is the ID of the `AdministerProject` permission. The IDs of other permissions can be found at the `com.atlassian.jira.security.Permissions` class.

The final argument to the `addPermissionTo` method is a remote entity, which can be a `RemoteUser` or a `RemoteGroup` object. We have seen how to access users and groups via SOAP in the previous recipes. In our example, we get a user by name as follows:

```
RemoteUser user = jiraSoapService.getUser(authToken, "jobinkk");
```

3. Create the project using the `createProject` method:

```
RemoteProject project = jiraSoapService.createProject(authToken,
"TEST", "Test Name", "Test Description", "http://www.j-tricks.
com", "jobinkk", permScheme, null, null);
```

The following are the arguments:

token	Authentication token
key	Project key
name	Project name
description	Project description
url	URL of the project
lead	Project lead
`permissionScheme`	Permission scheme for the project of `RemotePermissionScheme` type
`notificationScheme`	Notification scheme for the project of `RemoteScheme` type
`issueSecurityScheme`	Issue security scheme for the project of `RemoteScheme` type

For example, let's use the newly created permissions scheme and leave the other two as `null`. We can specify a specific notification scheme or issue security scheme with a `RemoteScheme` object created by populating the correct ID of the relevant schemes.

4. Add an actor to the newly created project:

```
jiraSoapService.addActorsToProjectRole(authToken, new String[] {
"jobinkk" }, adminRole, project,  "atlassian-user-role-actor");
```

   Here, the `addActorsToProjectRole` method takes an array of actors (only `jobinkk` in this case), the role to which the actor should be added, the project we have created, and the type of the actor.

   The project role can be retrieved using the role ID as shown :

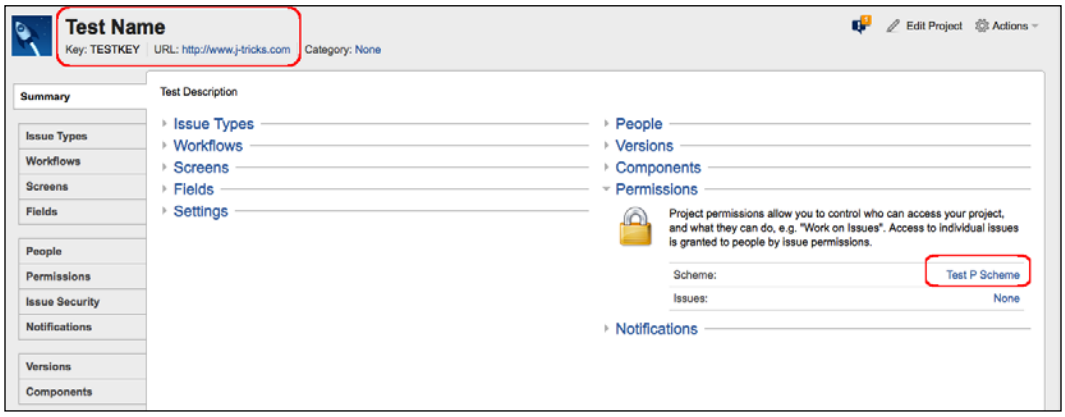
```
RemoteProjectRole adminRole = jiraSoapService.
getProjectRole(authToken, 10020L);
```

   The actor type can either be `atlassian-user-role-actor` or `atlassian-group-role-actor` depending on whether the actor we have added in the array is a user or a group.

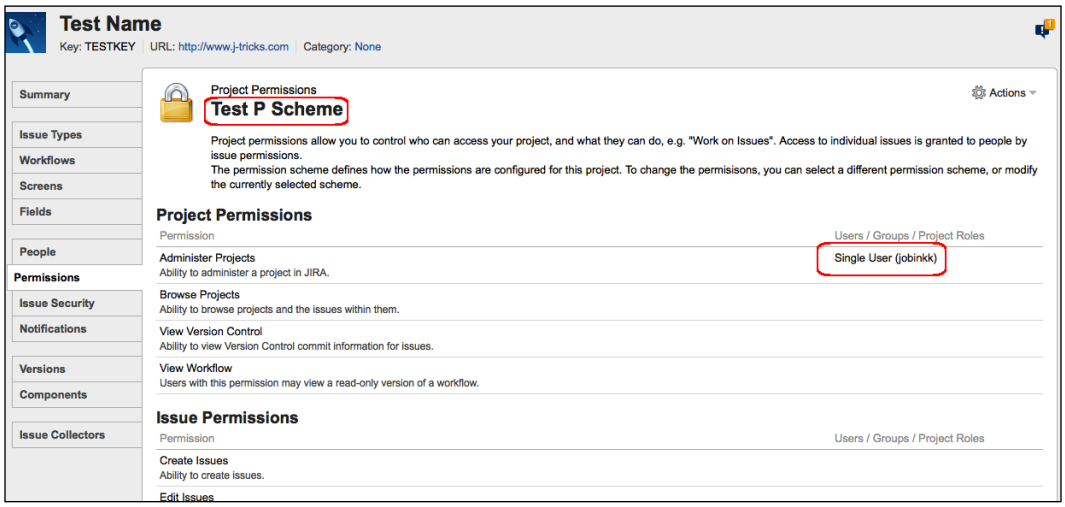
We should now have the project created with the new permission scheme and with the member(s) we added to the relevant roles.

How it works...

Once the method is executed, we can find the project created as follows:

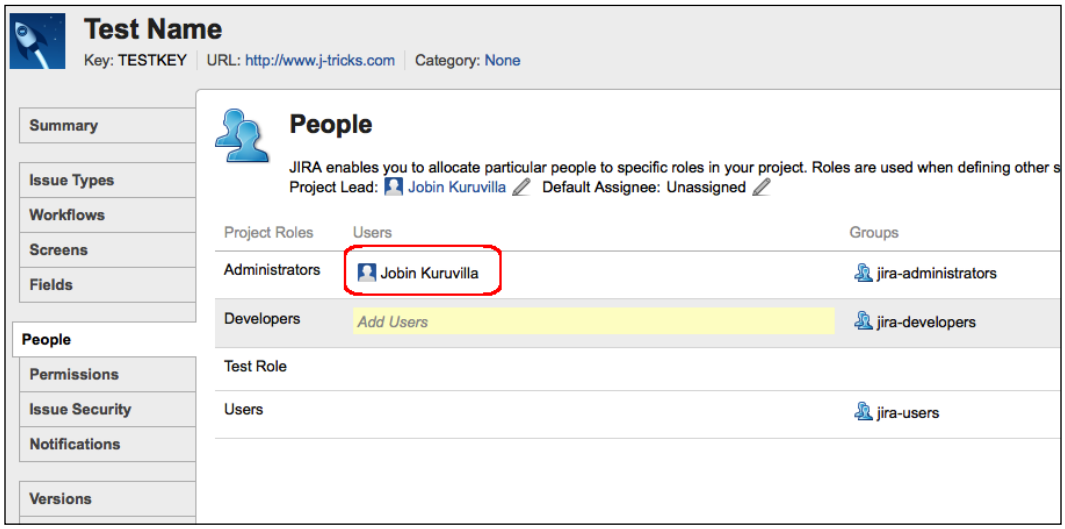


The new permission scheme is created as shown in the following screenshot:



Here a single permission is added, that is, to administer the project. We can add the rest in a similar fashion.

Similarly, the project members are added as shown in the following screenshot:



As you can see, the default actors will be part of the membership in addition to the one we have added!

There are lots of other useful methods in the SOAP API that can be found at `http://docs.atlassian.com/software/jira/docs/api/rpc-jira-plugin/latest/com/atlassian/jira/rpc/soap/JiraSoapService.html`.

Make sure you look at the right Javadocs for your version of JIRA!

Exposing services and data entities as REST APIs

So far we have seen various methods to perform various operations in JIRA via REST. What about operations that are not supported by REST? That little something that prevents you from integrating your 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. The plugin should be v2 for it to work.

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 required for REST to the `pom.xml` file:

```
<dependency>
  <groupId>javax.ws.rs</groupId>
  <artifactId>jsr311-api</artifactId>
  <version>1.0</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>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.3</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 key="rest-service-resources" path="/jtricks" version="1.0">
  <description>
    Provides the REST resource for the tutorial plugin.
  </description>
</rest>
```

Here, the path and version defines the full path where the resources will be available only 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-based 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 String id;
  @XmlElement
  private String name;
  public Category(){
  }
  public Category(String 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 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 the package level, `@Path("/Y")` at the class level, and `@Path("/Z")` at the 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 each other. In our example, let us define a `/category` path at the class level:

```
package com.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 (GenericValue category : categories) {
    categoryList.add(new Category(category.getId(), category.
getName()));
  }
Response.ResponseBuilder responseBuilder = 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:

`@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.

```
@PathParam
@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.ResponseBuilder responseBuilder = 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 as well in the path and get the details of that category alone; we can use the `PathParam` object here as shown earlier. In that case, the URL to this method will be as follows:

```
BASE_URL/rest/jtricks/1.0/category/10010
```

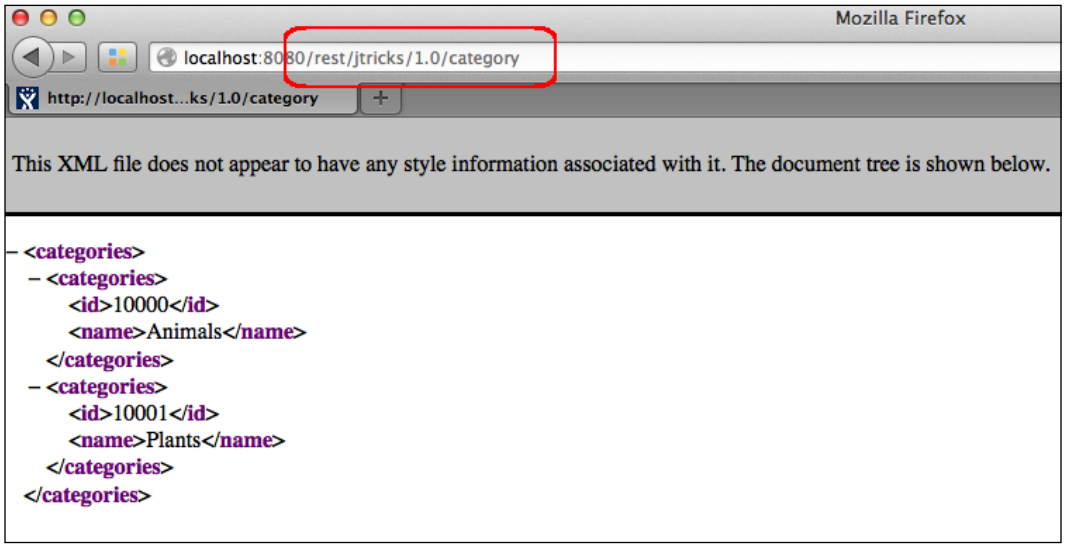
Here, 10010 is the category ID passed into the previously described method as ID.

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.

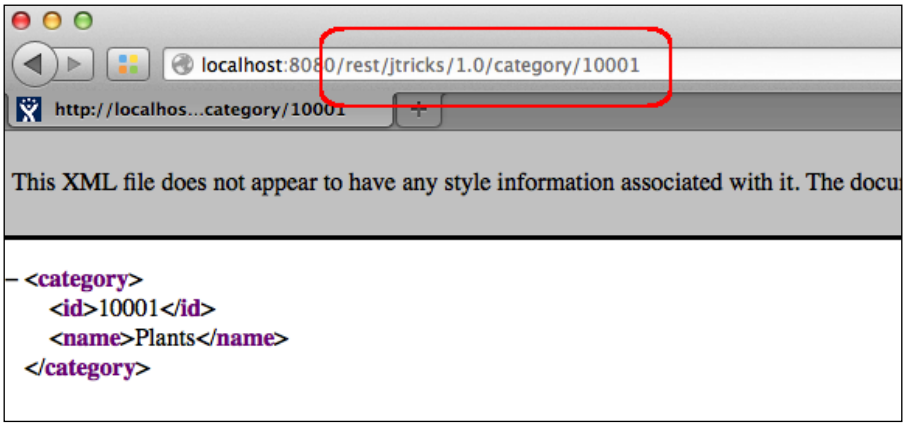
8. Package the plugin and deploy it.

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 your production version of the REST service plugin. Also check out `http://confluence.atlassian.com/display/REST/REST+API+Developer+Documentation` for more details.

Deploying a SOAP service in JIRA

How about exposing more functionality via SOAP? Like we did for REST in the earlier recipe? That is where the RPC endpoint plugin module is useful.

The RPC endpoint plugin module lets us deploy new SOAP and XML-RPC endpoints within JIRA. The new end points added will not be a part of the existing WSDL. Instead, they are available on a new URL and hence you will have to access both the web services if you want to access the new methods and other existing methods.

In this recipe, we will see how to deploy a new SOAP endpoint to perform a new operation.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK. As opposed to earlier JIRA versions, JIRA5 supports the `rpc-soap` module in v2 plugins.

Also, make sure the **Accept remote API Calls** option is turned on under **Administration** | **System** | **General Configuration**.

How to do it...

Let us create a SOAP RPC plugin to expose a new method, `getProjectCategories`, that retrieves all the project categories in the JIRA instance. This is a simple method but will hopefully help us in covering all the basics of creating a new SOAP RPC endpoint.

1. Add the RPC plugin dependency in the `pom.xml` file to get hold of the existing RPC classes. Change the version accordingly.

```
<dependency>
    <groupId>atlassian-jira-rpc-plugin</groupId>
    <artifactId>atlassian-jira-rpc-plugin</artifactId>
    <version>5.0.5</version>
</dependency>
```

2. Declare the new RPC service in `atlassian-plugin.xml`.

```
<rpc-soap key="jtricks-soap-service" name="JTricks SOAP Service"
class="com.jtricks.JTricksSoapServiceImpl">
  <description>JTricks SOAP service.</description>
```

```
  <service-path>jtricksservice</service-path>
  <published-interface>com.jtricks.JTricksSoapService</published-
interface>
</rpc-soap>
```

Here, the SOAP RPC plugin module has a unique key and it declares a new interface for your SOAP module and an implementation class for it. In this case, we have `JTricksSoapService` and `JTricksSoapServiceImpl`. The service path `jtricksservice` defines where in the URL namespace the services will be published and will appear in the URL of the WSDL.

3. Create a `ComponentPlugin` module for this new class to avoid the client getting a null pointer exception:

```
<component key="jtricks-soap-component" name="JTricks SOAP
Component" class="com.jtricks.JTricksSoapServiceImpl">
  <interface>com.jtricks.JTricksSoapService</interface>
</component>
```

4. Declare the new method in the interface as shown here:

```
public interface JTricksSoapService {
  String login(String username, String password);
  // Method to return Project Categories
  RemoteCategory[] getProjectCategories(String token) throws
RemoteException;
}
```

As you can see we have added a method, `getProjectCategories`, that returns an array of `RemoteCategory` objects. We have added a `login` method as well so that we can test this by accessing just the new WSDL.

5. Create the `RemoteCategory` bean. Make sure the new bean extends the `AbstractNamedRemoteEntity` class. The bean should have all the required attributes with getters and setters defined for it. `AbstractNamedRemoteEntity` already exposes the field name and hence will be available for `RemoteCategory`. We will add a new field description.

```
public class RemoteCategory extends AbstractNamedRemoteEntity {
  private String description;
  public RemoteCategory(GenericValue value) {
    super(value);
    this.description = value.getString("description");
  }

  public void setDescription(String description) {
    this.description = description;
  }
  public String getDescription() {
```

```
    return description;
  }
}
```

As you can see, the constructor takes a generic value and sets the description from it. In the super class, `AbstractNamedRemoteEntity`, the name is set in the same way.

6. Implement the `getProjectCategories` method in the implementation class:

```
public RemoteCategory[] getProjectCategories(String token) throws
RemoteException {
  validateToken(token);
  Collection<GenericValue> categories = projectManager.
getProjectCategories();
  RemoteCategory[] remoteCategories = new
RemoteCategory[categories.size()];
  inti = 0;
  for (GenericValue category : categories) {
    remoteCategories[i++] = new RemoteCategory(category);
  }
  return remoteCategories;
}
```

Here, all we do is get the collection of project categories and return an array of `RemoteCategory` objects initialized using the category `GenericValue` objects. Note that the `getProjectCategories()` method is deprecated from JIRA 4.4 but we still use it because `AbstractNamedRemoteEntity` expects a generic value in the constructor.

If you have noticed, we validate the token first before returning the categories. The validation is done as follows:

```
private void validateToken(String token) {
  try {
    User user = tokenManager.retrieveUserNoPermissionCheck(token);
  } catch (RemoteAuthenticationException e) {
      throw new RuntimeException("Error Authenticating!,"+e.
toString());
  } catch (RemotePermissionException e) {
      throw new RuntimeException("User does not have permission
for this operation,"+e.toString());
  }
}
```

We retrieve the user using the token and throw the appropriate error if the token is not valid. The `ProjectManager` and `TokenManager` classes can be injected in the constructor as shown:

```
public JTricksSoapServiceImpl(ProjectManager projectManager,
TokenManager tokenManager) {
  this.projectManager = projectManager;
  this.tokenManager = tokenManager;
}
```

Note that individual methods will have to do the permission checks as the token is retrieved without any permission checks.

7. Implement the `login` method to return the token.

```
public String login(String username, String password) {
  try {
    return tokenManager.login(username, password);
  } catch (RemoteAuthenticationException e) {
    throw new RuntimeException("Error Authenticating!,"+e.
toString());
  } catch (com.atlassian.jira.rpc.exception.RemoteException e) {
    throw new RuntimeException("Couldn't login,"+e.toString());
  }
}
```

It simply uses `TokenManager` to return a token created from the username and password.

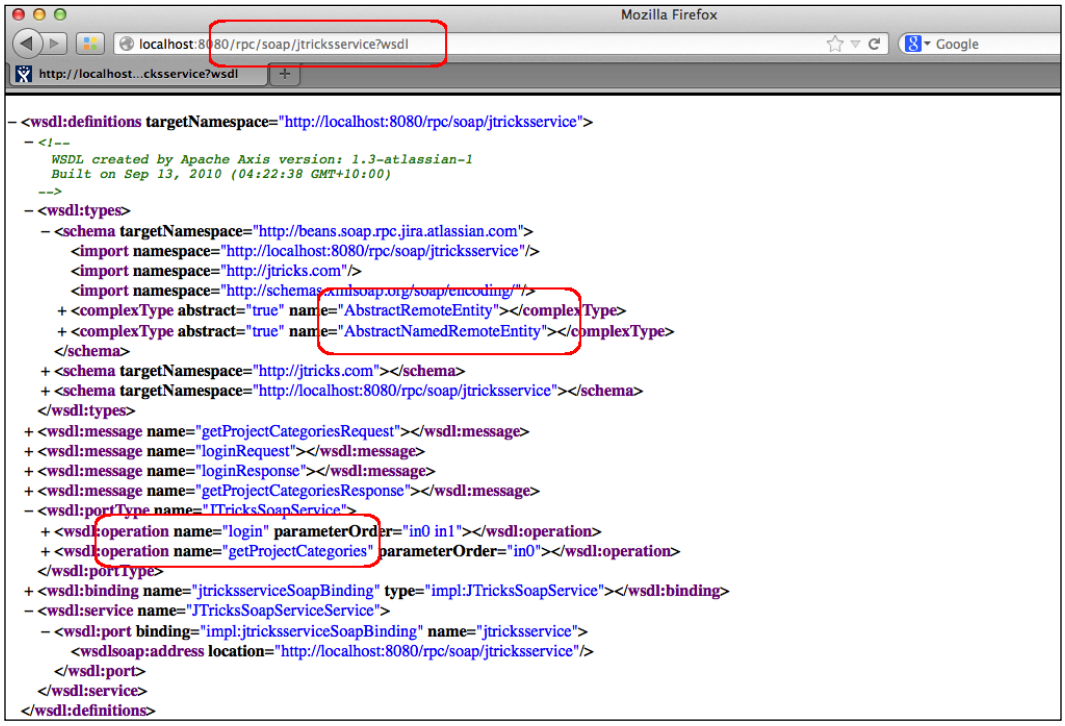
8. Use the component import module in `atlassian-plugin.xml` to import `TokenManager` as it is a v2 plugin.

```
<component-import key="tokenManager">
  <interface>
    com.atlassian.jira.rpc.auth.TokenManager
  </interface>
</component-import>
```

9. Compile the plugin and deploy it.

How it works...

Once the plugin is deployed, the new WSDL should be available at `{your_jira_url}/rpc/soap/jtricksservice?WSDL`.



As you can see, the new methods that we exposed via the interface are now visible in the WSDL file.

Deploying an XML-RPC service within JIRA

In the previous recipe, we have seen how to deploy a SOAP service within JIRA. In this one, we will see how to deploy an XML-RPC service.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK. Here again, make sure the **Accept remote API Calls** option is turned **ON**, as in the previous recipe.

How to do it...

As in the case of the SOAP plugin, let us expose a new method, `getProjectCategories`, that retrieves all the project categories in the project. The following are the steps:

1. Add the RPC plugin dependency in the `pom.xml` file to get hold of the existing RPC classes. Change the version accordingly:

```
<dependency>
    <groupId>atlassian-jira-rpc-plugin</groupId>
    <artifactId>atlassian-jira-rpc-plugin</artifactId>
    <version>5.0.5</version>
</dependency>
```

2. Declare the new RPC service in `atlassian-plugin.xml`:

```
<rpc-xmlrpc key="xmlrpc" name="JTricks XML-RPC Services"
class="com.jtricks.XmlRpcServiceImpl">
  <description>The JTricks sample XML-RPC services.</description>
  <service-path>jtricks</service-path>
</rpc-xmlrpc>
```

   Here, we define an `XmlRpcServiceImpl` class and a `service-path`. The `jtricks` service path is used to access the new methods in place of the default `jira1` path used in accessing the existing methods.

3. Create an interface for the class, with `XmlRpcServiceImpl` named as `XmlRpcService` and define the new methods in it.

```
public interface XmlRpcService {
  String login(String username, String password) throws Exception;
  Vector getprojectCategories(String token);
}
```

   As before, we have a `login` method as well. If you have noticed, the return type of the `getprojectCategories` method is a `Vector` object instead of an array of `RemoteCategory` objects.

   All the methods in the RPC interface should return a `Vector` object if it is returning a list of objects and a `HashTable` if it is returning a single object (`GenericValue`). The `Vector` object will be made of one or more Hash tables, each representing a generic value in the list.

4. Define the `RemoteCategory` object as defined in the previous recipe.

```
public class RemoteCategory extends AbstractNamedRemoteEntity {
  private String description;
  public RemoteCategory(GenericValue value) {
    super(value);
    this.description = value.getString("description");
```

```
  }
  public void setDescription(String description) {
    this.description = description;
  }
  public String getDescription() {
    return description;
  }
}
```

We will then create a `Vector` object from an array of `RemoteCategory` objects to return the project category details.

5. Implement the `XmlRpcServiceImpl` class. The `getprojectCategories` method is implemented as follows:

```
public Vector getprojectCategories(String token) {
  validateToken(token);
  Collection<GenericValue> categories = projectManager.
getProjectCategories();
  RemoteCategory[] remoteCategories = new
RemoteCategory[categories.size()];
  inti = 0;
  for (GenericValue category : categories) {
    remoteCategories[i++] = new RemoteCategory(category);
  }
  return RpcUtils.makeVector(remoteCategories);
}
```

Here we create an array of `RemoteCategory` objects and then create a `Vector` object from it using the `RpcUtils` utility class. The class, behind the scenes, converts the array of `RemoteCategory` objects into a `Vector` object of Hashtables, each Hashtable representing a `RemoteCategory`.

If we want to return a single `RemoteCategory` object instead of an array, we should return it as a `Hashtable` object constructed as follows:

```
RpcUtils.makeStruct(remoteCategory);
```

Implement the `login` and `validateToken` methods, as discussed in the previous recipe.

6. Compile the plugin and deploy it.

How it works...

Once the plugin is deployed, the new methods can be accessed using the new service path as `jtricks.getprojectCategories`. More details about how to access an XML-RPC method can be found in the following recipe.

If we try to get the list of categories using the new method exposed in the previous recipe, the code will look similar to the following:

```
// Retrieve Categories
Vector loginTokenVector = new Vector(1);
loginTokenVector.add(loginToken);
List categories = (List) rpcClient.execute("jtricks.
getprojectCategories", loginTokenVector);
for (Iterator iterator = categories.iterator(); iterator.hasNext();) {
  Map category = (Map) iterator.next();
  System.out.println(category.get("name"));
}
```

Note that the method name here is prefixed with `jtricks`, as it is the service path used in the RPC endpoint plugin module. Everything else works the same.

See also

- The *Writing a Java XML-RPC client* recipe

10
Dealing with the JIRA Database

In this chapter, we will cover:

- Extending the JIRA database with a custom schema
- Accessing database entities from plugins
- Persisting plugin information in the JIRA database
- Using Active Objects to store data
- Accessing the 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 the 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`; 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 used to do in earlier JIRA versions, because the database configuration is now done in `dbconfig.xml` residing in the JIRA `Home` folder.

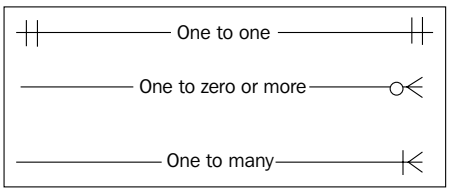
More about connecting to various other databases can be read at `http://confluence.atlassian.com/display/JIRA/Connecting+JIRA+to+a+Database`.

Read more about entity modeling concepts at `http://ofbiz.apache.org/docs/entity.html`.

In this chapter, we will also see glimpses of the JIRA database architecture, which is also explained in detail at `http://confluence.atlassian.com/display/JIRADEV/Database+Schema`.



Entity diagrams throughout this chapter use the following legend:



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 a 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** (`http://docs.atlassian.com/jira/latest/com/atlassian/jira/ofbiz/OfBizDelegator.html`), details of which we will see in the upcoming recipes.

The following are quick 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 the 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 type 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 is declared in `entityengine.xml`, as shown in the following code snippet:

```
<field-type name="oracle10g" loader="maincp" location="entitydefs/
fieldtype-oracle10g.xml"/>
```

If you look inside `fieldtype-oracle10g.xml`, you will notice that `numeric` is mapped to NUMBER(18,0) and `long-varchar` is mapped to VARCHAR2(255).

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 primary key 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" title="Parent" 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 `http://ofbiz.apache.org/docs/entity.html#Entity_Modeling`.

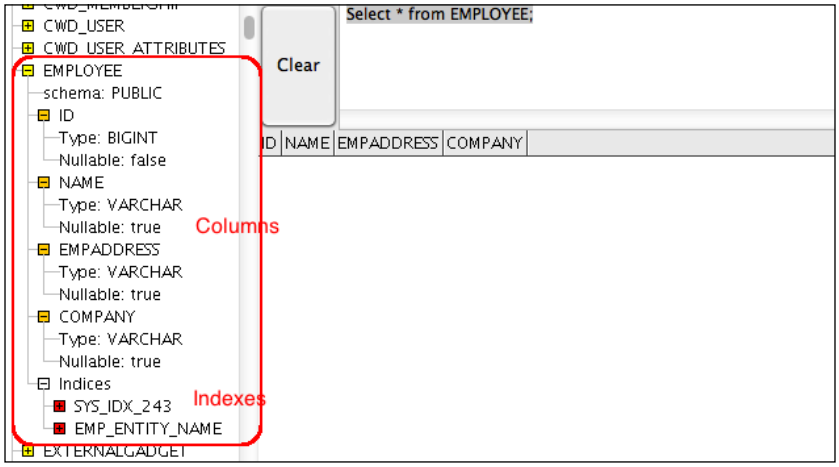
4. Restart JIRA after the changes have been made.

How it 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:

```
2013-02-15 00:05:52,093 main INFO      [atlassian.jira.startup.JiraStartupLogger]
JIRA pre-database startup checks completed successfully.
2013-02-15 00:05:53,715 main INFO      [jira.config.database.SystemTenantDatabaseC
onfigurationLoader] Reading database configuration from /Users/jobinkk/Softwares/J
IRA/5.1/dbconfig.xml
2013-02-15 00:05:53,957 main INFO      [jira.config.database.DatabaseConfiguration
ManagerImpl] Now running Database Checklist Launcher
2013-02-15 00:05:55,653 main WARN      [core.entity.jdbc.DatabaseUtil] Entity "Emp
loyee" has no table in the database
2013-02-15 00:05:56,015 main INFO      [atlassian.jira.startup.DatabaseChecklistLa
uncher] JIRA database startup checks completed successfully.
2013-02-15 00:05:56,064 main INFO      [jira.config.database.DatabaseConfiguration
ManagerImpl] Now running Post database-configuration launchers
2013-02-15 00:06:03,715 main INFO      [jira.config.database.SystemTenantDatabaseC
onfigurationLoader] Reading database configuration from /Users/jobinkk/Softwares/J
IRA/5.1/dbconfig.xml
2013-02-15 00:06:05,463 main WARN      [atlassian.jira.startup.DatabaseLauncher]
```

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 previous 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 (`http://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 `ComponentManager`, as follows:

```
OfBizDelegator delegator = ComponentAccessor.
getComponent(OfBizDelegator.class);
```

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",
EasyMap.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<GenericValue> employees = this.delegator.
findByCondition("Employee", new EntityExpr("id",EntityOperator.
GREATER_THAN,"15000"), EasyList.build("id","name"));
```

Here, we find all employee records with an ID greater 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:

```
OfBizListIterator iterator = this.delegator.findListIteratorByCo
ndition("Employee", new EntityExpr("id",EntityOperator.GREATER_
THAN,"15000"), null, UtilMisc.toList("name"), UtilMisc.toList("name"),
new EntityFindOptions(true, EntityFindOptions.TYPE_SCROLL_INSENSITIVE,
EntityFindOptions.CONCUR_READ_ONLY, true));
List<GenericValue> employees = iterator.getCompleteList();
iterator.close();
```

Here, we search for all records with the ID greater 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 four arguments including `TYPE_SCROLL_INSENSTITVE` and `CONCUR_READ_ONLY`. The first `true` is for `specifyTypeAndConcur` and the last `true` is for distinct select.

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 the previous 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 previously discussed, the last true in the `EntityFindOptions` constructor is for selecting distinct values. 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:

```
GenericValue newEmployee = this.delegator.
createValue("Employee",EasyMap.build("name","Some Guy",
"address","Some Address", "company","J-Tricks"));
```

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. For example, we can retrieve a record with ID `12000` and modify it, as follows:

```
GenericValue employee = delegator.findByAnd("Employee",   EasyMap.
build("id","12000")).get(0);
employee.setString("name","New Name");
employee.store();
```

More useful methods can be found in the Java docs at `http://docs.atlassian.com/jira/latest/com/atlassian/jira/ofbiz/OfBizDelegator.html`.

Persisting plugin information in the JIRA database

While developing plugins, we 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's `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("mt.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("mt.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 the ID is `10000`.

2. Get an instance of `PropertySet`, using `PropertySetManager`, by passing the details of the entity we got:

   ```
   HashMap entityDetails = new HashMap();
   entityDetails.put("delegator.name", "default");
   entityDetails.put("entityName", "User");
   entityDetails.put("entityId", 10000L);
   PropertySet userProperties = PropertySetManager.
   getInstance("ofbiz", entityDetails);
   ```

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.

3. 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");
```

   This will then be stored in the appropriate tables.

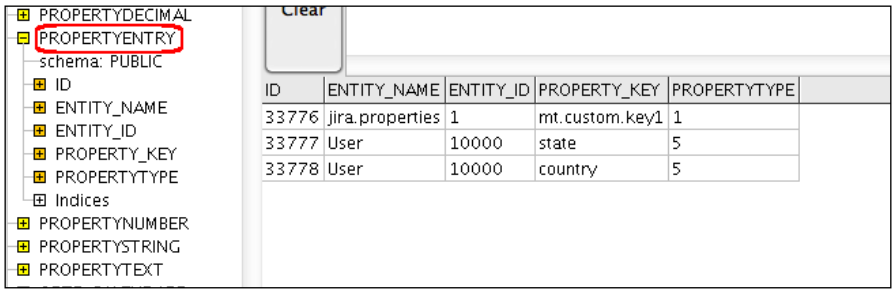
4. We can retrieve these values later by creating the `PropertySet` instance in a similar manner and using the getter methods:

```
System.out.println("Address:" + userProperties.
getString("state")+", "+userProperties.getString("country"));
```

How it works...

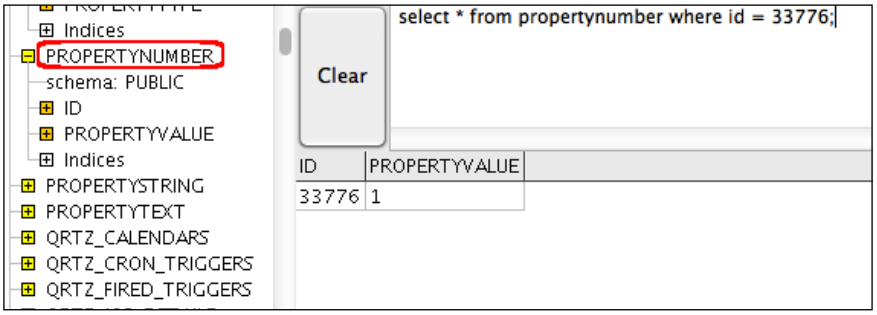
When a property is set using `PropertySet`, instantiated from `PropertiesManager` as we did in the case of the Boolean value, it gets stored in the `propertyentry` table with the `ENTITY_NAME` value as `jira.properties` and `ENTITY_ID` as `1`. It will also have a unique ID, which will then be used to 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:



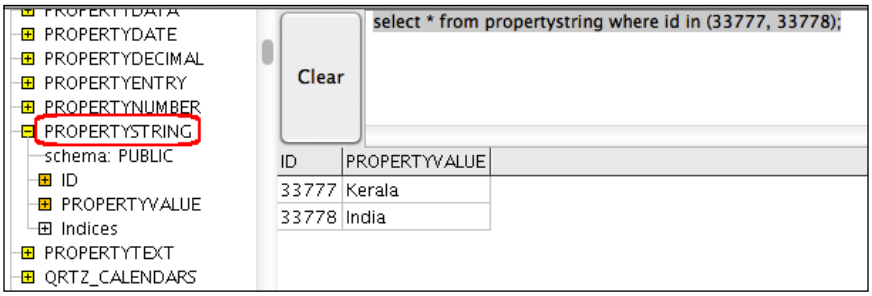
The first one is the Boolean property we added whereas the second and third are the user properties.

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 `33776`, 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 IDs `33777` and `33778`, 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 recently 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, follow the ensuing 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 define 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>
```

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 define 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;
public ManageProperties(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);
```

Hopefully, that gives a fair introduction to Active Objects.

How it works...

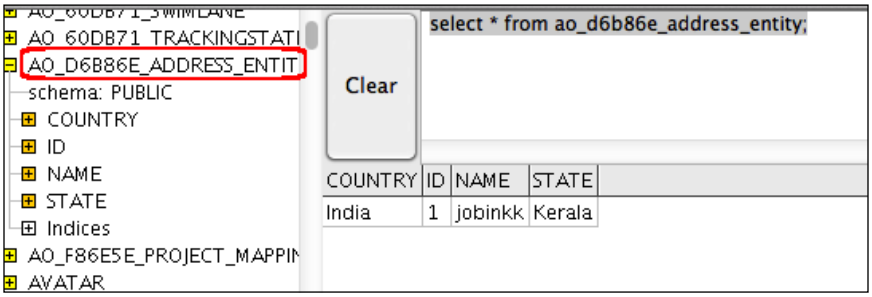
Behind the scenes, separate tables are created in the JIRA database for every entity that we add. The Active Objects service interacts with these tables to do the work.

If you see 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_d6b86e_address_entity`, as follows:



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 lot of global configurations 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 `0` 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 JIRA5.*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();
```

It's 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. But 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 number fields. 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 condition we discussed earlier:

1. Stop the JIRA instance. Updating 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';
   ```

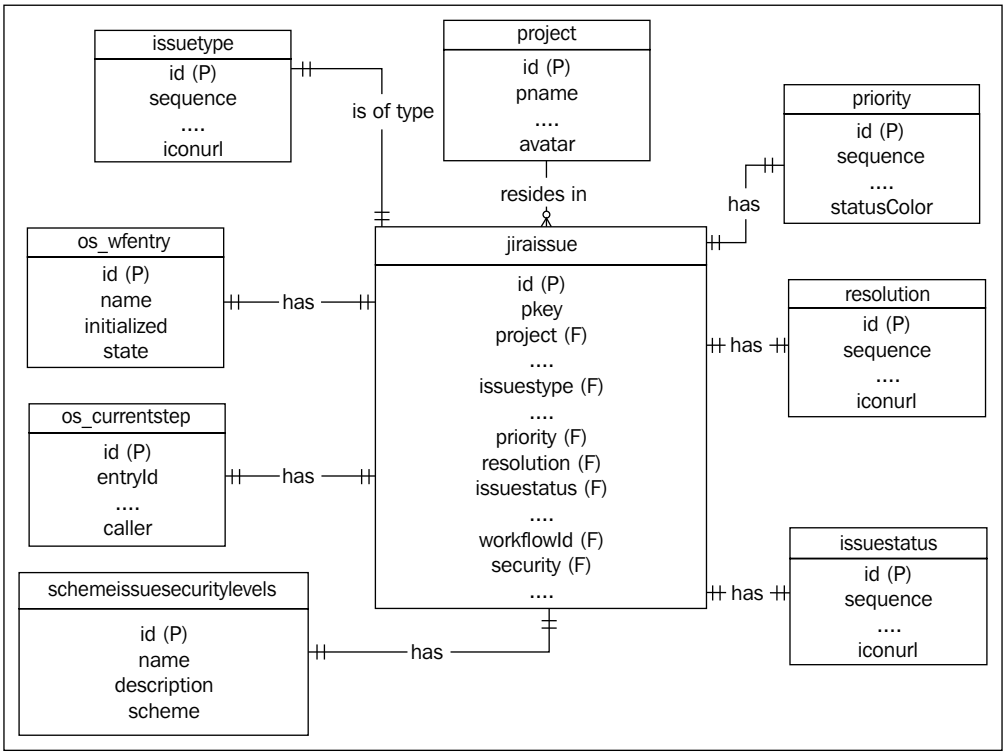
5. Commit the changes and disconnect.
6. Start JIRA.
7. Do a complete re-indexing of the JIRA instance by going to **Administration** | **System** | **Indexing**.

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.

Retrieving issue information from a database

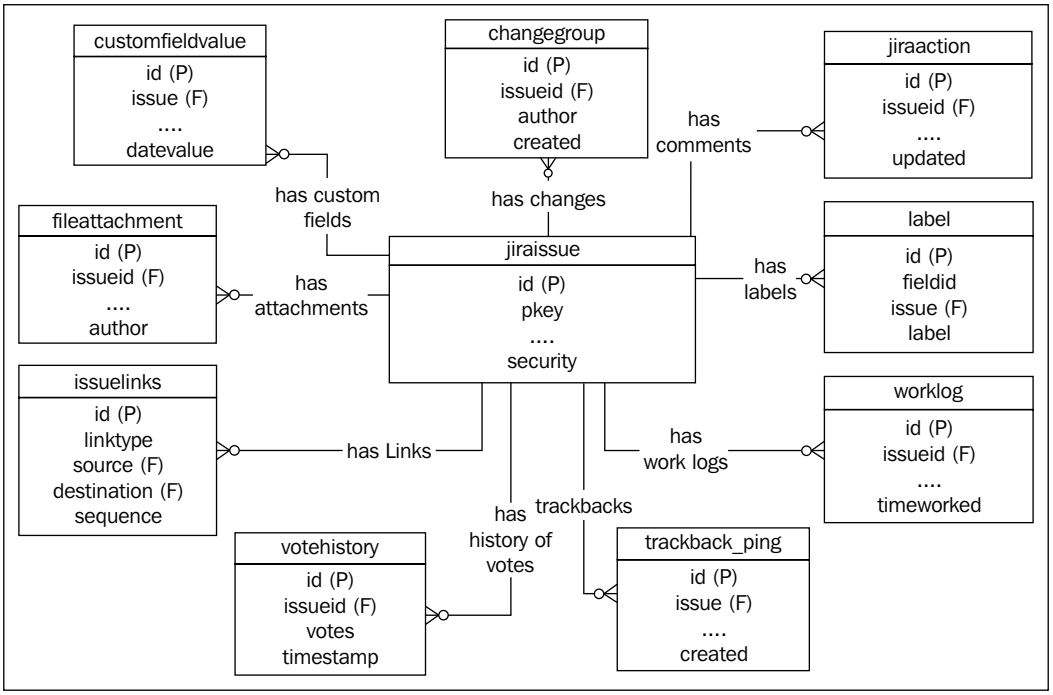
Information about an issue is scattered around in multiple tables in the JIRA database. However, a good starting point is the `jiraissue` table, which is where the issue record is stored. It has foreign keys referencing other tables and, at the same time, the issue ID is referenced in a few other tables.

The following diagram captures the important tables that the `jiraissue` table has a parent relationship with:



As you can see, critical information about an issue such as the project, issue type, status, priority, resolution, security level, and workflow, are all stored in the respective tables but are 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.

Similarly, 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.pkey, 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.

Similarly, all the comments on an issue can be retrieved by the following query:

```
select ji.pkey, ja.actionbody, ja.created, ja.author from jiraissue ji
left join jiraaction ja on ji.id = ja.issueid;
```

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 fixfor versions, affected versions, or components respectively.

We can access the components of an issue as follows:

```
select ji.pkey, comp.cname from nodeassociation na, component comp,
jiraissue ji where comp.id = na.sink_node_id and ji.id = na.source_node_
id and na.association_type = 'IssueComponent' and ji.pkey = 'DEMO-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 have seen 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.

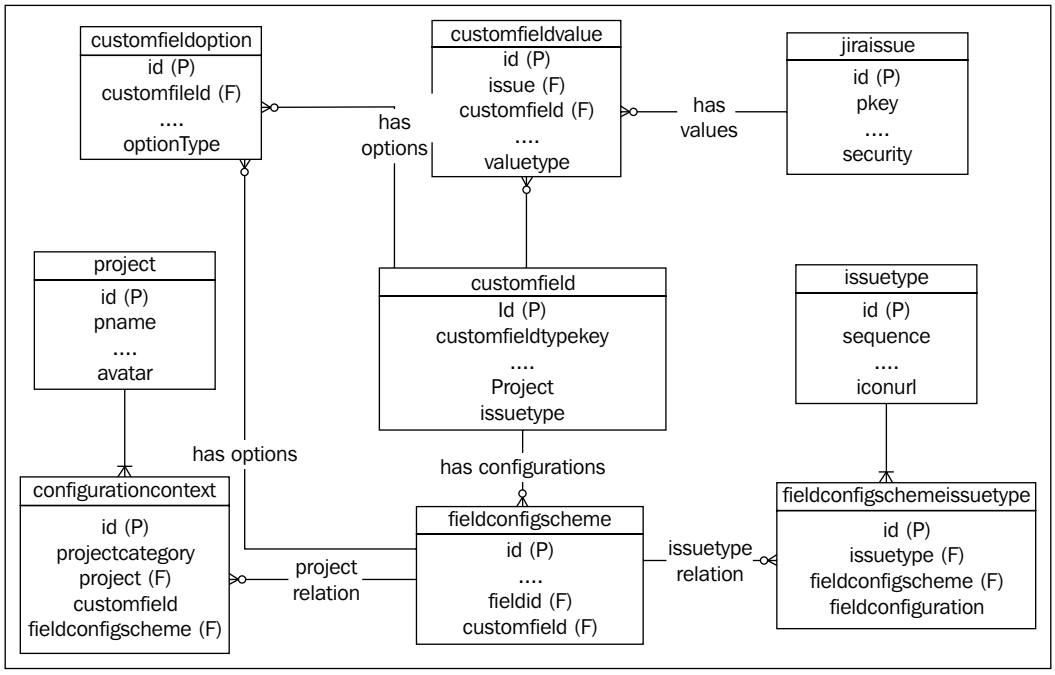
For each custom field, there can be a set of contexts configured. These contexts specify the projects or a list of issue types the field is associated with. For each such context, an entry is made in the `fieldconfigscheme` table with a unique ID. For each `fieldconfigscheme`, there will be entries in 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 like the following:



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:

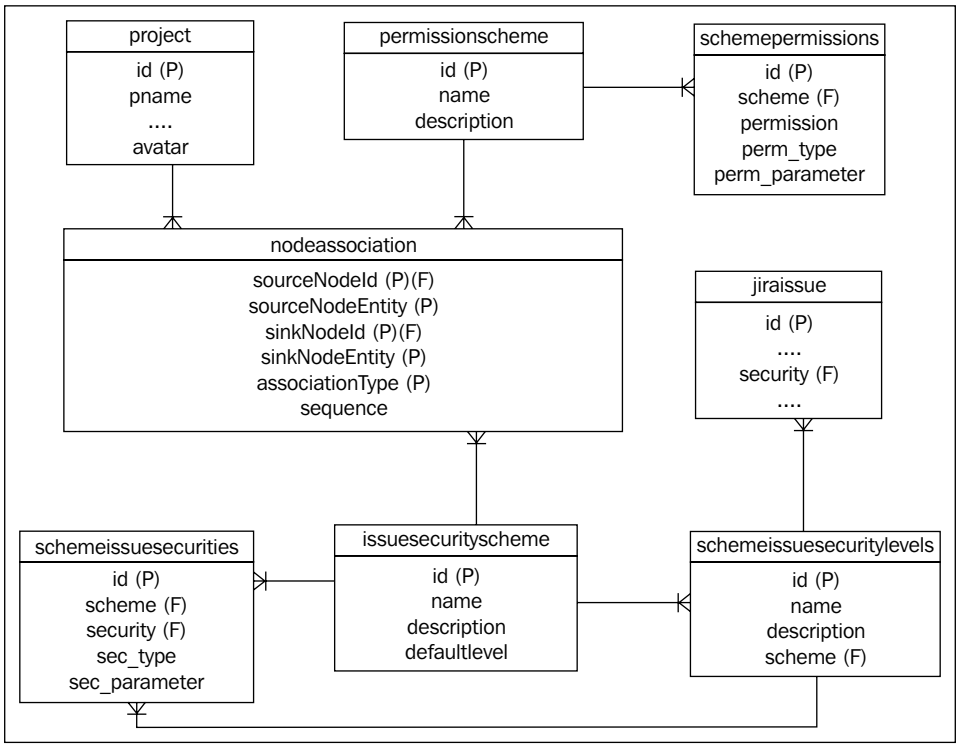
```
select issuetype.pname from fieldconfigschemeissuetype inner join
issuetype on fieldconfigschemeissuetype.issuetype = issuetype.id where
fieldconfigscheme in (select id from fieldconfigscheme where fieldid =
'customfield_12345');
```

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 in the command line:

```
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 `9`, 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 = 9 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 give 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.

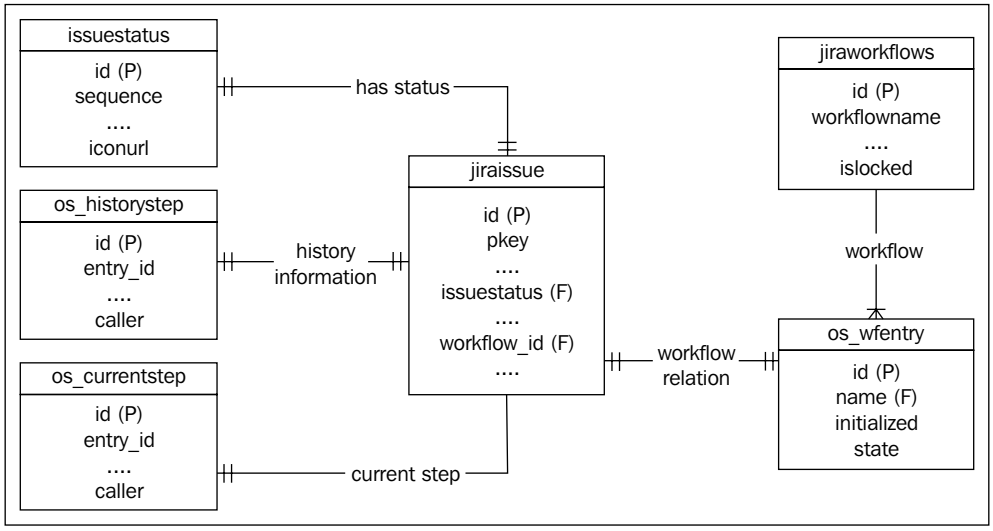
`jiraissue` also has another column, `workflow_id`, which points to the workflow the issue is associated with and the current step in the workflow the issue is in. The first bit of information, that is, the workflow an issue is associated with, is stored in the `os_wfentry` table. Here, the `workflow_id` 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, `DEMO-123`, 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.pkey='DEMO-123';
```

The details of the workflow associated with an issue can be retrieved, as follows:

```
select * from os_wfentry where id=(select workflow_id from jiraissue
where pkey='DEMO-123');
```

You can retrieve the workflow XML for an 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.pkey='DEMO-123';
```

The current step associated with an issue can be retrieved as follows:

```
select * from os_currentstep where entry_id = (select workflow_id from
jiraissue where pkey = 'DEMO-123');
```

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 pkey = 'DEMO-123');
```

Updating the issue status in a database

In this recipe, we will quickly 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.

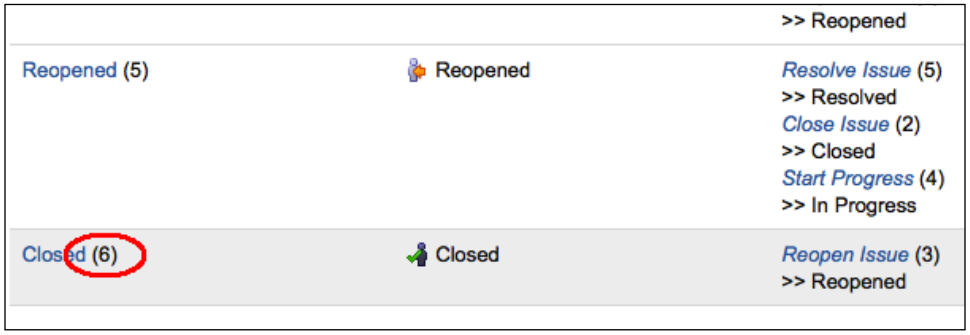
How to do it...

Refer to the following steps to update the status of an issue in JIRA:

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 = 'Closed') where pkey = 'DEMO-123';
```

4. 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:



As you can see, the status **Closed** in the JIRA default workflow is linked to the closed step with an ID value `6`. Now, the `step_id` column can be updated as follows:

```
UPDATE os_currentstep SET step_id = 6 where entry_id = (select
workflow_id from jiraissue where pkey = 'DEMO-123');
```

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**.

Retrieving users and groups from a database

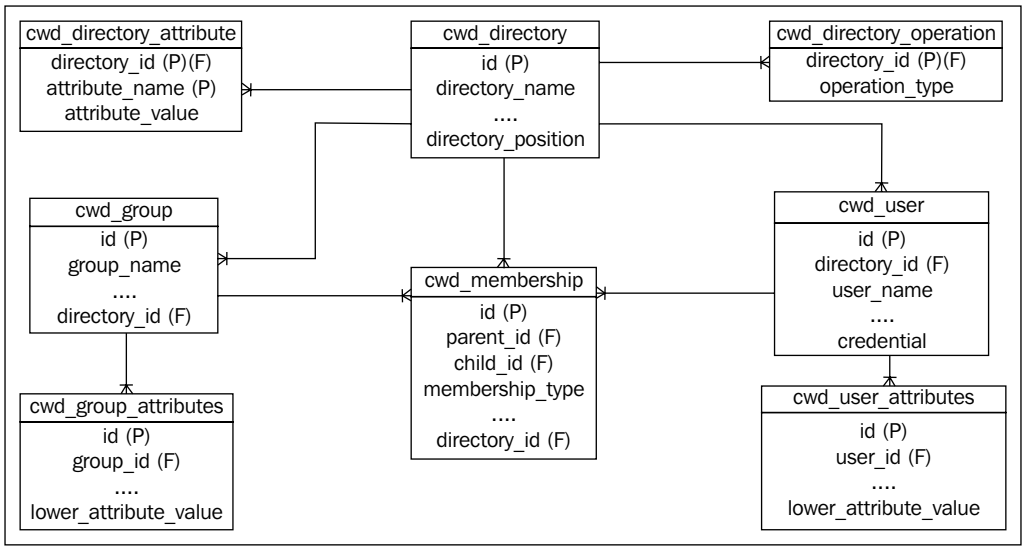
We can find all the information about JIRA users and their groups from the database by running a few simple SQL queries. In this recipe, we will see the various tables involved.

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 highest-priority directory. 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 pkey='DEMO-123');
```

Similarly, all the issues watched by a user can be retrieved as follows:

```
select ji.pkey from jiraissue ji inner join userassociation ua on
ua.sink_node_id =  ji.id where ua.association_type = 'WatchIssue' and
ua.sink_node_entity = 'Issue' and ua.source_name = 'someuser';
```

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 the 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` and `oldstring`, and `newvalue` and `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 pkey =
'DEMO-123') order by cg.created;
```

It is quite easy to modify this 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, as follows:

1. Stop the JIRA server.
2. Connect to the JIRA database.
3. 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,'someguy','2011-06-15');
   ```

   Make sure the ID value (`12345`) is larger than the `max(ID)` value in the table.

4. Insert a change item for this change group. Let us consider the status change we made in the earlier recipe:

   ```
   insert into changeitem values (11111, 12345, 'jira','status','1','
   Open','6','Closed');
   ```

   Note that the `groupid` value here is same as the ID attribute in the previous step. 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.

   Here, also make sure the ID (`11111`) is larger than `max(id)` in the table.

5. Update the `sequence_value_item` table to hold a higher value in the `seq_id` column for the `ChangeGroup` and `ChangeItem` entities. In the previous case, we can give a value of `12346` for `ChangeGroup` and `11112` for `ChangeItem`. Ofbiz normally allocates IDs 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 = 11112 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.

6. Commit the changes and start JIRA.
7. Re-index the JIRA instance by going to **Administration** | **System** | **Advanced** | **Indexing**.

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 different pages in webwork actions
- Adding custom behavior for user details
- Deploying a servlet in JIRA
- Adding shared parameters to Servlet Context
- Writing a Servlet Context Listener
- 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
- Overriding JIRA's default components in plugins
- Internationalization in webwork plugins
- Sharing common libraries across v2 plugins
- Operations using direct HTML links

Introduction

So far, we have grouped the recipes under common themes as different chapters in this book. We have seen all the important themes but we are still left with some useful recipes and a handful of plugin modules that were not covered in the previous chapters.

In this chapter, we will look at some of those powerful plugin points and useful tricks in JIRA that are not covered in the earlier chapters. Not all of these recipes are related, but they are all useful in their own ways.

Writing a service in JIRA

A service that runs at regular intervals is a much-wanted feature in any web application. It is more so if it can be managed with user-configured parameters and without having to reboot, and so on. JIRA offers a mechanism to add new services on to it that run at regular intervals after every startup. 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 on to JIRA.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it...

As opposed to the other JIRA plugin modules, services are not defined inside the plugin descriptor. Instead, it uses a configuration XML. It is still a plugin but without any specific definitions in the `atlassian-plugin.xml` file and has classes, files, and a configuration XML related to the service, residing inside the plugin.

The following are the 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 choosing. 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 under `src/main/resources/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:

   ```
   public void run() {
     System.out.println("Running the JTricks service!!");
   }

   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 regular intervals.

The other key mandatory method is `getObjectConfiguration`. We get the configurations from the XML we have written earlier (in *Step 1*) 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 the same as the ID in the XML file). This ID is used as a key while saving the 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 on 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 of JIRA, such as portlets, though not in the case of services.

5. Install the plugin using the **Universal Plugin Manager** (**UPM**).

Now the service is ready. We can go to **Administration** | **System** | **Advanced** | **Services** and add the new service with the appropriate delay. 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` (this 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. After dropping the JAR file, please restart JIRA.

See also

- The *Adding configurable parameters to a service* recipe.

Adding configurable parameters to a service

For a simple service as the one we just wrote, there is only one parameter that can be configured. It is the delay at which the service runs! 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 to do so:

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 `Tutorial` property from `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—and the property values we entered on the JIRA Admin GUI 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 GUI, it prints **Running the JTricks service Tutorial? I don't like tutorials!** as the tutorial property is not configured yet.

```
2013-02-20 00:00:00,821 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize took: 758ms. Indexer: DefaultIndexManage
r: paths: [/Users/jobinkk/Softwares/JIRA/5.1/caches/indexes/comments, /Users/job
inkk/Softwares/JIRA/5.1/caches/indexes/issues, /Users/jobinkk/Softwares/JIRA/5.1
/caches/indexes/changes]
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize took: 89ms. Indexer: SharedEntityIndexMa
nager: paths: []
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize Indexes complete. Total time: 847ms.
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [issue.index.job.O
ptimizeIndexJob] Indexes Optimized. Took: 847 milliseconds.
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [issue.index.job.O
ptimizeIndexJob] Optimize Index Job complete.
2013-02-20 00:03:39,981 http-8080-1 WARN jobinkk 3x8x1 1cg1hsd 0:0:0:0:0:0:0:1%0
 /secure/admin/EditService!default.jspa [webwork.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!
```

Go to the **Administration** | **System** | **Advanced** | **Services** area, and edit the service to enter a value under the **The tutorial you like** field. Assuming you entered **JTricks Tutorials**, you will see some output as shown here:

```
2013-02-20 00:00:00,821 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize took: 758ms. Indexer: DefaultIndexManage
r: paths: [/Users/jobinkk/Softwares/JIRA/5.1/caches/indexes/comments, /Users/job
inkk/Softwares/JIRA/5.1/caches/indexes/issues, /Users/jobinkk/Softwares/JIRA/5.1
/caches/indexes/changes]
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize took: 89ms. Indexer: SharedEntityIndexMa
nager: paths: []
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [jira.util.index.C
ompositeIndexLifecycleManager] Optimize Indexes complete. Total time: 847ms.
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [issue.index.job.O
ptimizeIndexJob] Indexes Optimized. Took: 847 milliseconds.
2013-02-20 00:00:00,911 QuartzWorker-0 INFO ServiceRunner     [issue.index.job.O
ptimizeIndexJob] Optimize Index Job complete.
2013-02-20 00:03:39,981 http-8080-1 WARN jobinkk 3x8x1 1cg1hsd 0:0:0:0:0:0:0:1%0
 /secure/admin/EditService!default.jspa [webwork.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!
Running the JTricks service!! Tutorial? JTricks Tutorials
Running the JTricks service!! Tutorial? JTricks Tutorials
Running the JTricks service!! Tutorial? JTricks Tutorials
```

See also

- The *Writing a service in JIRA* recipe.

Writing scheduled tasks in JIRA

Have you ever thought of running scheduled tasks within JIRA? Why do we need scheduled tasks when we have JIRA services? We have seen how to write a service in the previous recipe. But in spite of all the advantages we discussed so far, these services have a disadvantage. It always starts when JIRA is restarted and runs at regular intervals after that. So, if you have a service that does some heavy memory-intensive operation and if you restart JIRA in the middle of the day, you will suddenly find your instance's performance compromised! If it is scheduled to run every 24 hours, you will find the same service running in the middle of the day from then on until the next restart.

Scheduled tasks in JIRA are a good way to make sure all such operations happen at quiet times; midnight, for example. In this chapter, we will write a simple scheduled task and see how easy that can be!

As you will see, schedulers cannot be deployed through UPM and hence we need to restart JIRA for changes to take effect.

How to do it...

Let us write a simple scheduled task that prints a line in the console. The following are the steps:

1. Write a Java class that implements the `Quartz job` interface. JIRA internally uses Quartz for scheduling its tasks and so Quartz comes bundled within JIRA.

```
public class JTricksScheduledJob implements Job{
  ...
}
```

2. Implement the `execute` method. This is the method that gets executed every time the job runs. Whatever we do in this method can be as simple as a one liner or as complex as initiating a nuclear explosion! Our scheduled job just prints a line to the console and hence the Java class that we write is simple, as follows:

```
public void execute(JobExecutionContext context) throws
JobExecutionException {
  System.out.println("Running the job at "+(new Date()).
toString());
}
```

3. Package the class in a JAR file and deploy it under the `WEB-INF/lib` folder.

4. Modify the `scheduler-config.xml` file under the `WEB-INF/classes` folder to let JIRA know of our new scheduled task. JIRA stores all the information about scheduled tasks in this file:
    a. Define a job under the `<job>` tag as shown here here:

```
<job name="JTricksJob" class="com.jtricks.
JTricksScheduledJob" />
```

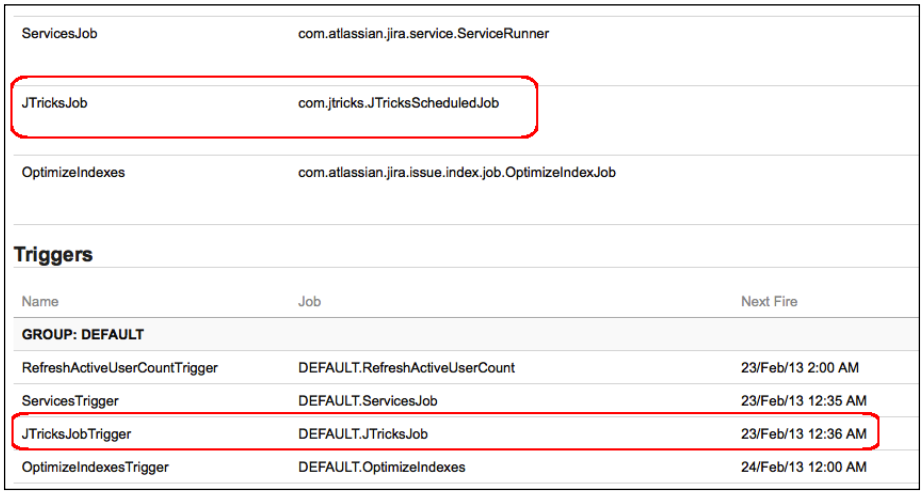
    b. Add a trigger that runs the `JTricksJob` job. This is where we define the cron expression to run the job at defined timings:

```
<trigger name="JTricksJobTrigger" job="JTricksJob"
type="cron">
  <expression>0 0/2 * * * ?</expression><!-- run every 2
minutes -->
</trigger>
```

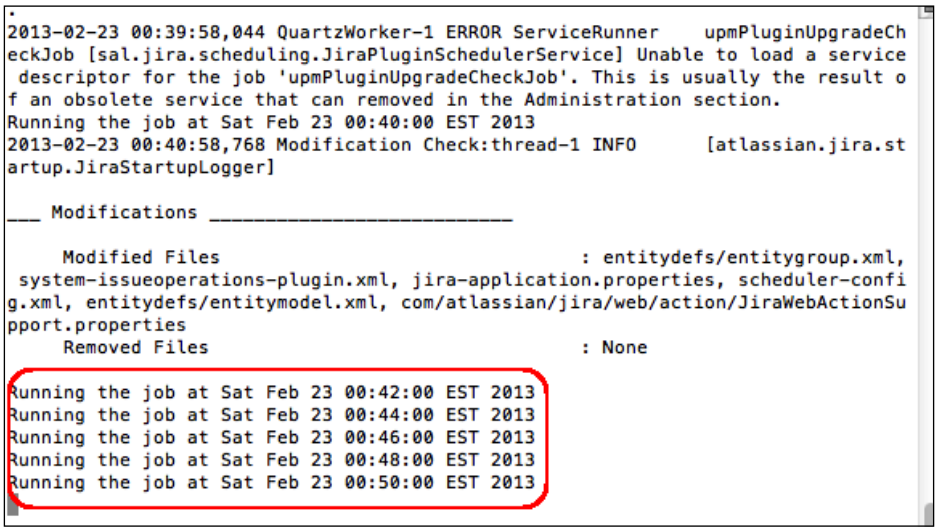
    c. The previous trigger schedules the job to run every two minutes. More details about writing a cron expression can be found at `http://www.quartz-scheduler.org/docs/tutorial/TutorialLesson06.html`.
5. Restart JIRA.

How it works...

Once JIRA is restarted, the new job can be seen at the scheduler details page under **Administration** | **System** | **Troubleshooting and Support** |**Scheduler Details**. We can also verify the next fire time for the task on the same page, as shown in the following screenshot:



And when the job runs, you will see the following printed in the console!



There's more...

It is also possible to write scheduled tasks in JIRA via **Shared Access Layer** (**SAL**). More about this can be found at `https://developer.atlassian.com/display/DOCS/Scheduling+Events+via+SAL+Tutorial`.

Writing listeners in JIRA

Listeners are very powerful features in JIRA. JIRA has a mechanism of throwing events whenever something happens on an issue, such as when creating an issue, updating an issue, progressing on the workflows, or in similar events. Using listeners, we can capture these events and do special things based on our requirements.

There are two different ways that listeners can be implemented in JIRA. The old way of doing it is to extend the `AbstractIssueEventListener` class, which in turn implements the `IssueEventListener` interface. The `AbstractIssueEventListener` class captures the event, identifies its type, and delegates the event to the appropriate method where it is handled. To write a new listener, all we need to do is to extend the `AbstractIssueEventListener` class and override the methods of interest!

The new way of doing this is to use the `atlassian-event` library. Here, we register the listener in the plugin descriptor and implement the listener with the help of the `@EventListener` annotation.

Both ways are supported in JIRA now, though they have their own advantages and disadvantages. For example, it is possible to add properties for a listener in the old way. Adding properties is not supported in the new way, but then the new way doesn't need any configuration as it is registered automatically. The new way can be written as a full-fledged v2.0 plugin.

In this recipe, we will see how to write listeners using both methods.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it...

Writing a listener in the old way, by extending `AbstractIssueEventListener`, is done as follows:

1. Create a listener class that extends the `AbstractIssueEventListener` class:

```
public class OldEventListener extends AbstractIssueEventListener {
  ...
}
```

2. Define the properties for the listener. This is an optional step, needed only if you need to define properties for the listener, which can then be used during execution. An example could be to enter the mail server details if we have a listener that sends custom e-mails using a specific mail server when an event is fired.
    a. Override the `getAcceptedParams` method to return a string array of properties to be defined:

```
@Override
public String[] getAcceptedParams() {
  return new String[] { "prop 1" };
}
```

    Here, we define a property named `prop1`.

    b. Override the `init` method and retrieve the property value entered by the user:

```
@Override
public void init(Map params) {
  prop1 = (String) params.get("prop 1");
}
```

The `init` method is invoked whenever a listener is configured or re-configured. Here, we just retrieve the property value and assign it to a class variable for future use.

3. Override the appropriate listener methods. For example, an issue-created event can be captured by overriding the `issueCreated` method as shown next:

```
@Override
public void issueCreated(IssueEvent event) {
  Issue issue = event.getIssue();
  System.out.println("Issue " + issue.getKey() + " has been
created and property is:"+prop1 +", Old listener");
}
```

Here, we just retrieve the issue that triggered the event—the issue that is newly created in this case—and just print the details along with the listener property. We can write even more complex methods in this method. It is also possible to retrieve other things from the event, such as the change log details, if there are any changes involved, for example, in the case of the `issueUpdated` event or when entering the comment using a transition.



Note that only a handful of events can be listened to like this and there are events such as project creation, which do not throw an event at all! In such cases, you might want to extend the respective action and throw a custom event if needed. All the available events can be found at `http://docs.atlassian.com/jira/latest/com/atlassian/jira/event/issue/IssueEventListener.html`.

4. An important method that is worth mentioning here is the `customEvent` method, which is invoked whenever a custom event is triggered. This happens for all the custom events configured by the user as mentioned in the next recipe. We can capture them as shown here:

```
@Override
public void customEvent(IssueEvent event) {
  Long eventTypeId = event.getEventTypeId();
  Issue issue = event.getIssue();
  if (eventTypeId.equals(10000L)) {
    System.out.println("Custom Event thrown here for issue:" +
issue.getKey()+" and property is:"+prop1+", Old listener");
  }
}
```

Here, `10000` is the ID of the new event.

5. Package the plugin and deploy it via UPM.

6. Configure the listener by going to **Administration** | **System** | **Advanced** | **Listeners**.
    a. Enter the name and the fully qualified class name and click on **Add**.
    b. Edit the listener to add properties, if there are any!

A listener created in the new way, that is, using the `@EventListener` annotation is written as follows:

1. Register the listener in `atlassian-plugin.xml`:

```
<component key="eventListener" class="com.jtricks.
NewEventListener">
  <description>Class that processes the new
JIRA Event</description>
</component>
```

   Here the `class` attribute holds the fully qualified name of the listener class that we are going to write.

2. Import the `EventPublisher` component using the `component-import` plugin module:

```
<component-import key="eventPublisher" interface="com.atlassian.
event.api.EventPublisher"/>
```

3. Write the listener class:
    a. Inject the `EventPublisher` component into the class and use the `register` method to self-register, as shown:

```
public class NewEventListener {
  public NewEventListener(EventPublisher eventPublisher) {
    eventPublisher.register(this);
  }
}
```

    b. Create methods to handle the events using `@EventListener` as shown in the following code:

```
@EventListener
public void onIssueEvent(IssueEvent issueEvent) {
  System.out.println("Capturing event with
ID:"+issueEvent.getEventTypeId()+" in the new listener");
  ...
}
```



Note that the annotation can be used with any number of public methods in the class and all of them will be invoked when an event is fired in JIRA.

c. Handle the events appropriately.

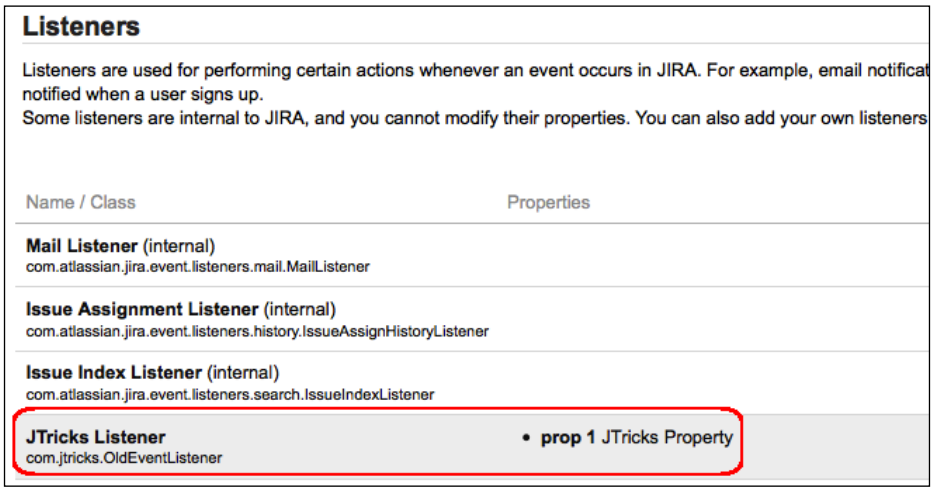
```
  @EventListener
  public void onIssueEvent(IssueEvent issueEvent) {
    System.out.println("Capturing event with ID:" +
issueEvent.getEventTypeId() + " in the new listener");
    Long eventTypeId = issueEvent.getEventTypeId();
    Issue issue = issueEvent.getIssue();
    if (eventTypeId.equals(EventType.ISSUE_CREATED_ID)) {
      System.out.println("Issue " + issue.getKey() + " has
been created, New listener");
    } else if (eventTypeId.equals(10000L)) {
      System.out.println("Custom Event thrown here for
issue:" + issue.getKey()+", New listener");
    }
  }
```

Here, as we can see, the event ID is checked and then handled as appropriate. First, we handled the issue created event and then the custom event.

4. Package the plugin and deploy it using UPM.

How it works...

In both the cases, the listener works exactly the same way once configured. Note that the old listener is registered manually and the property configuration is applicable only for the old way. Once registered, the listener can be seen under **Administration** | **System** | **Advanced** | **Listeners** as follows:



Note that there is the **prop 1** property configured in the listener.

When the event is fired in JIRA, listeners capture them and the appropriate methods are invoked. The old one will print the issue key along with the property name. The new one works the same way, except that there is no property value:

```
2013-02-23 01:21:59,986 Modification Check:thread-1 INFO      [atlassian.jira.st
artup.JiraStartupLogger]

___ Modifications ____________________________

     Modified Files                                : entitydefs/entitygroup.xml,
 system-issueoperations-plugin.xml, jira-application.properties, entitydefs/enti
tymodel.xml, com/atlassian/jira/web/action/JiraWebActionSupport.properties
     Removed Files                                 : None

Capturing event with ID:1 in the new listener
Issue TEST-75 has been created, New listener
Capturing event with ID:1 in the old listener
Issue TEST-75 has been created and property is:JTricks Property, Old listener
Capturing event with ID:10000 in the new listener
Custom Event thrown here for issue:TEST-75, New listener
Capturing event with ID:10000 in the old listener
Custom Event thrown here for issue:TEST-75 and property is:JTricks Property, Old
 listener
```



It is possible to add properties to the listener even in the new way but that needs a separate configuration screen to capture and maintain the properties.

There's more...

It is possible that the plugin may be disabled and re-enabled by the Administrators while the service is still running. The constructor is invoked when the listener is initially loaded at JIRA startup, but we might want to handle the enabling or disabling of plugins separately as they are not captured in the constructor.

Handling, enabling, and disabling of plugins

Atlassian plugins are implemented as Spring dynamic modules, and the `atlassian-plugin.xml` file is transformed into a Spring XML bean configuration before it is actually loaded by the product. In the case of listeners, the event listener will become a Spring bean and hence we can apply the Spring interfaces—`InitializingBean` and `DisposableBean`—to capture the creation and destruction of the bean. In our case, the code is modified as follows:

```
public class NewEventListener implements InitializingBean,
DisposableBean {
```

```
  private final EventPublisher eventPublisher;

  public NewEventListener(EventPublisher eventPublisher) {
    this.eventPublisher = eventPublisher;
  }
  @EventListener
  public void onIssueEvent(IssueEvent issueEvent) {
    System.out.println("Capturing event with ID:" + issueEvent.
getEventTypeId() + " in the new listener");
    Long eventTypeId = issueEvent.getEventTypeId();
    Issue issue = issueEvent.getIssue();
    if (eventTypeId.equals(EventType.ISSUE_CREATED_ID)) {
      System.out.println("Issue " + issue.getKey() + " has been
created, New listener");
    } else if (eventTypeId.equals(10000L)) {
      System.out.println("Custom Event thrown here for issue:" +
issue.getKey()+", New listener");
    }
  }

  public void afterPropertiesSet() throws Exception {
    eventPublisher.register(this);
  }

  public void destroy() throws Exception {
    eventPublisher.unregister(this);
  }

}
```

As you can see, the registering and un-registering happens in the `afterPropertiesSet` method and the `destroy` events respectively. These methods are invoked during the creation/destruction of the bean and that effectively handles the enabling/disabling of the plugin.

Don't forget to add the `spring-beans` JAR in the project build path in this case to avoid compilation issues! Alternatively, the following dependency can be added in `pom.xml`:

```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>2.5.6</version>
            <scope>provided</scope>
</dependency>
```

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 such handling of these events 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 just want to amend the e-mail content?

In this recipe, we will see how we can customize the 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 that is 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 both 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 the cases, the steps are pretty much 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 the 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 to it. We cannot change the templates associated with a system event. For example, an `Issue Updated` event is associated with `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 to 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>
```

4. Here, we can add new mappings, if we are adding new templates:

```
<templatemappings>
  ...
  <templatemapping id="20">
    <name>Demo Event</name>
    <template>demoevent.vm</template>
    <templatetype>issueevent</templatetype>
  </templatemapping>
  ...
</templatemappings>
```

   Make sure the ID we use here is unique in the file.

5. 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 mails and another for Text 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 the 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, `demoevent.vm` in our case, we need to create three templates, one for each subject, HTML body, and the text body, all with the same name put under the respective folders.

6. Restart JIRA after editing the templates appropriately.

How it works...

After a template is newly added and JIRA is restarted, 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.

There's more...

It is possible to add more information about an issue, such as custom fields in the notification e-mails, by editing the relevant velocity templates.

Advanced customization—adding custom field information

All the VM templates have got the `$issue` object in the Velocity context along with other variables that are elaborated on at `http://confluence.atlassian.com/display/JIRADEV/Velocity+Context+for+Email+Templates`. It is fairly easy to use this to retrieve the contents on an issue while generating the e-mail content.

For example, `$issue.summary` will retrieve the issue summary and you can see it in the e-mail subject rendered using `WEB-INF/classes/templates/email/subject/issueupdated.vm`. Similarly, other information on the issue can be easily accessed. For example, custom field details can be accessed as follows:

```
$issue.getCustomFieldValue($customFieldManager.getCustomFieldObject("c
ustomfield_10010"))
```

Here, `10010` is the unique ID for the custom field.

You can see various other examples of formatting at `http://confluence.atlassian.com/display/JIRADEV/Adding+Custom+Fields+to+Email`.

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 or to browse a new project or view another issue after the action is 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 before we redirect?

All you need here is to return `getRedirect(URL)` in the action class' `doExecute` method (or the appropriate method)! This method will redirect to the specified location when the action method is 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.

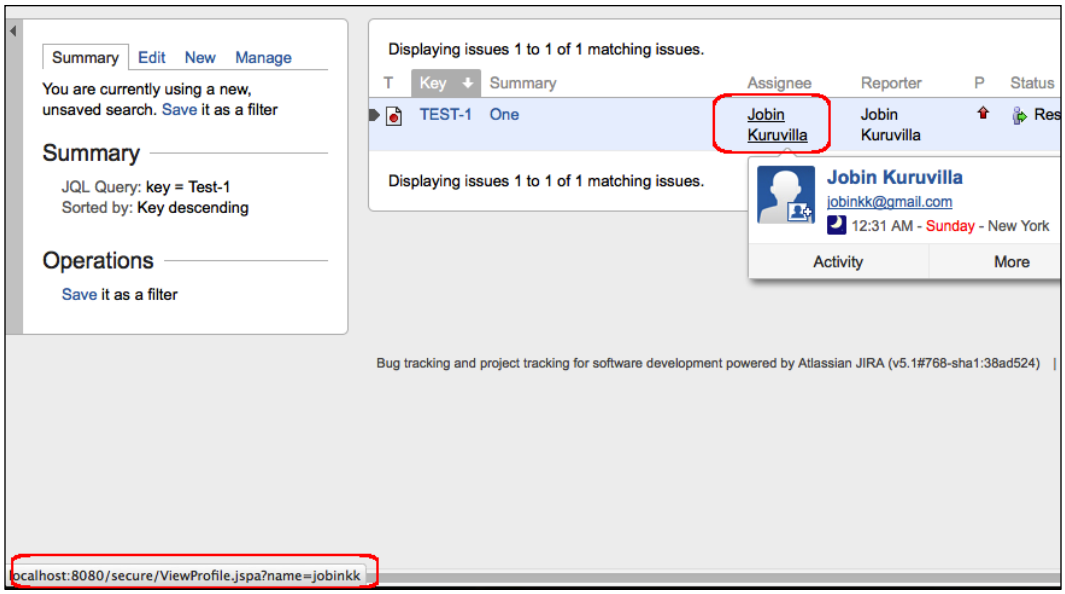
For example, if we need to return to the dashboard on success, we can do it as follows:

```
@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 the 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:



And you can see that the link points to the profile page.

But what if we want to change how the user details are displayed? Say, if we want to display the user avatar alongside the user details? Or, if 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 use them within the existing JIRA displays or within out-custom plugins.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK. In this case, the plugin should be v1 as we are going to use `UserFormatModuleDescriptor` inside the plugin to render the templates.



`UserFormatModuleDescriptor` is not available for v2 plugins.

How to do it...

In this recipe, let us try to create a new user profile that will display the username (instead of 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 into `atlassian-plugin.xml`:

```
<user-format key="twitter-format" name="Twitter User Format"
class="com.jtricks.TwitterUserFormat" system="true">
  <description>User name linking to twitter</description>
<type>twitterLink</type>
  <resource type="velocity" name="view" location="templates/
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 are the types existing in JIRA by default as of Version 5.1: `profileLink`, `fullName`, `fullNameHover`, `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 described in the previous step. The class should implement the `UserFormat` interface.

```
public class TwitterUserFormat implements UserFormat {
  private final UserFormatModuleDescriptor moduleDescriptor;

  public TwitterUserFormat(UserFormatModuleDescriptor
moduleDescriptor){
    this.moduleDescriptor = moduleDescriptor;
  }
}
```

   Here, we inject `UserFormatModuleDescriptor` into the class as it will be used in rendering the Velocity template, as shown in the next step.

3. Implement the required methods. We will have to implement the two overridden format methods.

   The first method takes the `username` and `id` parameters, where `username` is the name of the user, which can also be null, and `id` is an extra argument that can be used to pass an 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 in displaying the assignee in the column view (`/WEB-INF/classes/templates/jira/issue/field/assignee-columnview.vm`), where the ID is the assignee.

   We are not going to use an 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 moduleDescriptor.getHtml(VIEW_TEMPLATE, params);
}
```

   Here, `getInitialParams` just populates the `params` map with the username as shown here:

```
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!

The second method takes `username`, `id`, and `Map`—which is prepopulated with extra values to add more to the context!—as parameters. 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 moduleDescriptor.getHtml(VIEW_TEMPLATE, velocityParams);
}
```

The only difference is that the extra context is also populated into the `params` map.

In both the cases, `moduleDescriptor` renders the Velocity template, defined by the name `VIEW_TEMPLATE` or "view".

4. Write the Velocity template that uses the context populated in the `params` map in the previous step to display the user information:

```
#if ($username)
  #set ($quote = '"')
  #set($author = "<a id=${quote}${textutils.
htmlEncode($username)}${quote} href=${quote}http://twitter.com/#!/
${username}${quote}>$textutils.htmlEncode($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, `http://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. A single quotation mark here is the Velocity syntax and a double quotation mark is the value. It is used to construct the URL where the encoded name, the `href` value, and so on are placed between quotes!

Don't forget to handle the scenario when the user is null. In our case, we just display the name as anonymous when the user is null.

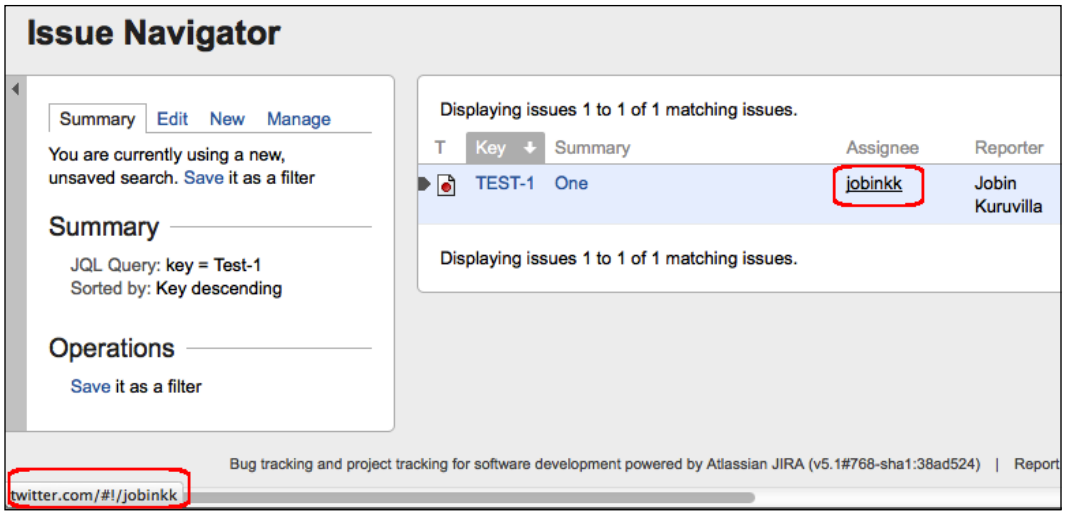
5. Package the plugin and deploy it.

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 the `twitterLink` profile instead of the default `profileLink` as follows:

```
##disable_html_escaping() TODO REENABLE
#if($assigneeUsername)
    #if ($displayParams && $displayParams.nolink)
        $userformat.formatUser($assigneeUsername, 'fullName',
'assignee')
    #else
        <span class="tinylink">$userformat.
formatUser($assigneeUsername, 'twitterLink', 'assignee')</span>
    #end
#else
    <em>$i18n.getText('common.concepts.unassigned')</em>
#end
```

When you do that, the assignee column in the **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 the `formatUser` method 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...

The following are the steps 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:
    - `class`: This is the servlet Java class and it must be a subclass of `javax.servlet.http.HttpServlet`. This attribute is mandatory.
    - `disabled`: This indicates whether the plugin module should be disabled or enabled by default. By default, the module is enabled.
    - `i18n-name-key`: This is the localization key for the human-readable name of the plugin module.
    - `key`: This represents the unique key for the plugin module. This attribute is mandatory.
    - `name`: This is the human-readable name of the servlet.
    - `system`: This indicates whether this plugin module is a system plugin module or not. Only available for non-OSGi plugins.

The following are the child elements supported:

- `description`: The description of the plugin module.
- `init-param`: Initialization parameters for the servlet, specified using the `param-name` and `param-value` sub-elements, just as in `web.xml`. This element and its child elements may be repeated.
- `resource`: The resources for this plugin module. This element may be repeated.
- `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-param` tags, as shown here:

```
<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 initialization parameters—`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 initialization parameters in the `init` method as shown here:

```
@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 enabled back.

As you can see, the initialization parameters we defined in the servlet plugin module can be accessed here from the `ServletConfig` interface. Here, we also initialize the `JiraAuthenticationContext` interface, 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 do what needs to be implemented. For the example, we will just use the `init-param` values 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()));
}
```

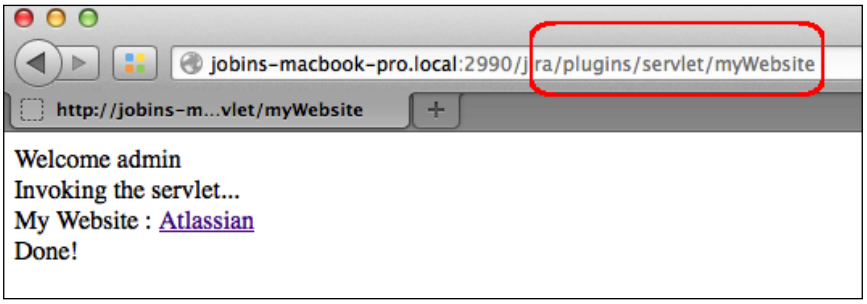
`authenticationContext` retrieves the current username as mentioned earlier.

4. Package the plugin and deploy it.

How it works...

Once deployed, the servlet will be accessible from the following 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 the 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 or servlet filters or context listeners that make 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 `This is a shared Text`. Once the plugin is packaged and deployed, the `sharedText` parameter is available across servlets, filters, and listeners.

In a servlet, we can access the parameter from the `init` method as follows:

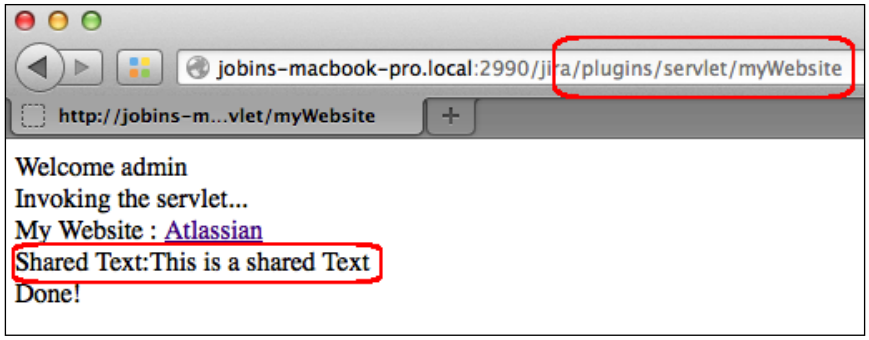
```
@Override
public void init(ServletConfig config) throws ServletException {
  super.init(config);
  String sharedText = config.getServletContext().
getInitParameter("sharedText");
}
```

How it works...

Once the shared text is retrieved, we can use it anywhere, such as 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 this? 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 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 implement `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:

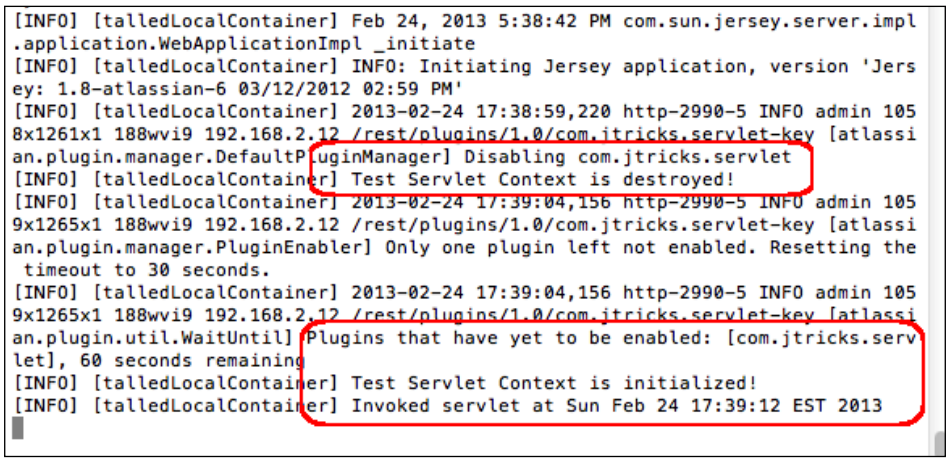
   ```
   public void contextDestroyed(ServletContextEvent event) {
      System.out.println("Test Servlet Context is destroyed!");
   }

   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:



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 do wise things such as profiling, monitoring, content generation, and so on. It works exactly like any normal Java servlet filter and JIRA provides the **Servlet Filter Plugin** Module to add them using plugins. In this recipe, we will learn about how to use a filter to intercept certain queries to JIRA and how we can utilize them!

As with other servlet plugin modules, the `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 named `location`, which denotes the position of the filter in the application's filter chain. The following are the four possible values for `location`:

- `after-encoding`: This is the very top of the filter chain in the application, but after any filters which ensure the integrity of the request
- `before-login`: Before the filter that logs in the user

- `before-decoration`: Before the filter which does a Sitemesh decoration of the response
- `before-dispatch`: At the end of the filter chain, before any servlet or filter which handles the request by default

The `weight` attribute is used in conjunction with `location`. If two filters have the same location, then they are ordered based on the `weight` attribute.

`init-param` as usual takes the initialization 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` element here matches `${baseUrl}/${url-pattern}`. The pattern can use the wildcard characters `*` or `?`, 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.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK.

How to do it...

Let us try to intercept all the issue views, which has the URLs of the format `${baseUrl}/browse/*-*`, and log them. The following are the step-by-step procedures to write a filter and implement the given logic:

1. Add the Servlet Filter plugin module into `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>/browse/*-*</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` will be `/browse/*-*` as the URL to browse has an issue is of the form `${baseUrl}/browse/*-*`. 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.getInitParamete
r("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 defined or parameters defined using the Servlet Context Parameter plugin module.

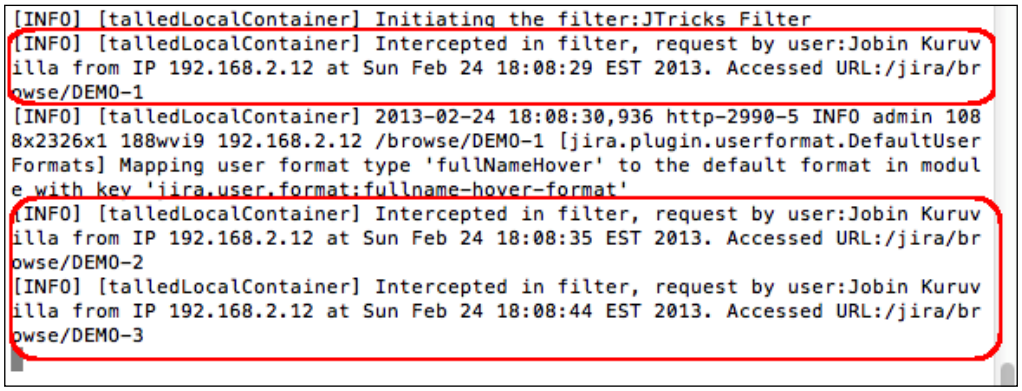
The `destroy` method is invoked whenever a filter is destroyed.

The `doFilter` method is invoked every time the URL matches `url-pattern`. Here, we are just printing the IP address and user details requesting the **View Issue** page and logs the time but we can do many 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 of 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.



With our code in the example, the details are printed, as shown in the previous screenshot, whenever an issue is viewed.

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 are available for use by the core classes and plugins alike. It makes sense sometimes to add custom components to that component system that can then be used with the 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 them from within the plugin and from a separate plugin.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK. For the example, we will use the `RedirectAction` webwork module used in the *Redirecting to a different page in webwork actions* recipe.

How to do it...

First, let us see how we can define a component and use it in the different modules within the same plugin. In our example, we will define a sample component and use the methods exposed by it in the `RedirectAction` webwork. Following are the steps:

1. Create an interface with the required method definitions. The component will expose these methods when used elsewhere:

```
package com.jtricks.provider;

public interface MyComponent {
  public void doSomething();
}
```

2. Create the implementation class and implement the methods:

```
public class MyComponentImpl implements MyComponent{
  private final JiraAuthenticationContext authenticationContext;

  public MyComponentImpl(JiraAuthenticationContext
authenticationContext) {
    this.authenticationContext = authenticationContext;
  }

  public void doSomething() {
    System.out.println("Hey "+authenticationContext.
getLoggedInUser().getDisplayName()+",  Sample method to check
Components");
  }
}
```

   In the implementation class, we can inject the JIRA components as usual and use them for various things. Here, we inject `JiraAuthenticationContext` to retrieve the current user details just to print a personalized message!

3. Declare the component in the `atlassian-plugin.xml` file using the Component Plugin module:

```
<component key="myComponent" name="My Component" class="com.
jtricks.provider.MyComponentImpl">
  <interface>com.jtricks.provider.MyComponent</interface>
</component>
```

   Here, the component module has a unique `key` value and a `class` attribute that points to the implementation class. The `interface` element points to the component interface we created in *Step 1*.

Our component is now ready and available to use within the other plugin modules. For example, we can use this component in the `RedirectAction` class we saw earlier, as follows:

```
public class RedirectAction extends JiraWebActionSupport {
  private final MyComponent myComponent;

  public RedirectAction() {
    this.myComponent = ComponentAccessor.getOSGiComponentInstanceOfTyp
e(MyComponent.class);
  }

  @Override
  protected String doExecute() throws Exception {
    System.out.println("Execute the method in component!");
    this.myComponent.doSomething();
    System.out.println("Succesfully executed. Go to dashboard");
    return getRedirect("/secure/Dashboard.jspa");
  }
}
```

Here, the component is retrieved using the `getOSGiComponentInstanceOfType` method in `ComponentAccessor` and the exposed method is invoked where appropriate.

Exposing components to other plugins

When we create the components as discussed earlier, they remain private and are available only within the plugin even though we can expose these components to other plugins.

To expose a component to other v2 plugins, make sure they are declared public. Following are the steps to do this:

1. Create the interface and implementation class as before.
2. Declare the component in the `atlassian-plugin.xml` file using the Component Plugin module as a public component. For this, we use the `public` attribute on the component module, as shown in the following code:

```
<component key="myComponent" name="My Component" class="com.
jtricks.provider.MyComponentImpl" public="true">
  <interface>com.jtricks.provider.MyComponent</interface>
</component>
```

With that, the component is now ready and available to other plugins.

Importing public components

In order to use the public components in other plugins, we will have to first import them using the `component-import` plugin module. The module is entered in `atlassian-plugin.xml` as follows:

```
<component-import key="myComponent">
   <interface>com.jtricks.provider.MyComponent</interface>
</component-import>
```

Now, the component is available as if it is created within the plugin itself. The `RedirectAction` class will look exactly the same in the new plugin as well if we want to use the component there.

Using service properties in components

It is also possible to define a Map of properties for a public component that can then be used while importing the components with other plugins. It uses the `service-properties` element to define the properties, which have child elements named `entry` and have `key` and `value` attributes. For example, a dictionary service can have the service properties defined with the language as the key, as shown in the following code snippet:

```
<component key="dictionaryService" class="com.myapp.
DefaultDictionaryService" interface="com.myapp.DictionaryService">
    <description>Provides a dictionary service.</description>
    <service-properties>
        <entry key="language" value="English" />
    </service-properties>
</component>
```

It is now possible to use the `filter` attribute on the `component-import` module to import a component only if the service matches the filter. For example, the dictionary service that has English as the language can be imported as follows:

```
<component-import key="dictionaryService" interface="com.myapp.
DictionaryService"  filter="(language=English)" />
```

How it works...

When a component is installed, it generates the `atlassian-plugins-spring.xml` Spring Framework configuration file, transforming the 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 cover, using the 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 the 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 out your own Spring configuration file, stored under the `META-INF/spring` folder in the plugin JAR.



More details about the Component Plugin module and Component Import plugin module can be found in the Atlassian documentation at `http://confluence.atlassian.com/display/PLUGINFRAMEWORK/Component+Plugin+Module` and `http://confluence.atlassian.com/display/JIRADEV/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 type, webwork module type, servlet module type, 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 the Atlassian Plugin SDK.

How to do it...

Let us consider the same example Atlassian have used in their 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.

Following are the steps to define a new plugin module type:

1. Add the module type definition in the `atlassin-plugin.xml` file:

   ```
   <module-type key="dictionary" class="com.jtricks.
   DictionaryModuleDescriptor" />
   ```

   Here, the key must be unique and will be used as the root element when defining the modules of this type. The `class` attribute points to the `ModuleDescriptor` class that is instantiated when a new plugin module of this type is found.

Other useful attributes of this module type includes `name` that holds a human readable name, `i18n-name-key` to hold the localization key for the human-readable name, `disabled` to indicate if the plugin module is disabled by default or not, and `system` to indicate whether this plugin module is a system plugin module or not (available only for non-OSGi). You can also have an optional `description` attribute as a child element.

2. 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.

3. Create the module descriptor class. The class must extend the `AbstractModuleDescriptor` class and should use the interface we created as the generic type:

```
public class DictionaryModuleDescriptor extends AbstractModuleDesc
riptor<Dictionary> {
  ...
}
```

4. Implement the `getModule` method to create the module:

```
public class DictionaryModuleDescriptor extends AbstractModuleDesc
riptor<Dictionary> {
  public DictionaryModuleDescriptor(ModuleFactory moduleFactory) {
    super(moduleFactory);
  }

  public Dictionary getModule() {
    return moduleFactory.createModule(moduleClassName, this);
  }
}
```

   Here, we have used `ModuleFactory` to create a module of this type.

5. Define the attributes and elements that will be used in the new module type and retrieve them in the `init` method. For a dictionary, we need at least one attribute, that is, `language`, to differentiate the various dictionary modules. Let us name that attribute `lang` and retrieve it in the `init` method. The class will now look similar to the following block of code:

```
public class DictionaryModuleDescriptor extends AbstractModuleDesc
riptor<Dictionary> {
```

```
  private String language;

  public DictionaryModuleDescriptor(ModuleFactory moduleFactory) {
    super(moduleFactory);
  }

  @Override
  public void init(Plugin plugin, Element element) throws
PluginParseException {
    super.init(plugin, element);
    language = element.attributeValue("lang");
  }

  public Dictionary getModule() {
    return moduleFactory.createModule(moduleClassName, this);
  }

  public String getLanguage() {
    return language;
  }
}
```

The `init` method takes as argument, `com.atlassian.plugin.Plugin` and `org.dom4j.Element`, the latter holding the module element. We have retrieved the `lang` attribute here and assigned it to a local variable which has a getter method that can be used to get the language value in other plugins/modules.

We can have more attributes or child elements as required by the new module type.

6. With that, the new plugin module is now ready. We can now write the 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" />
```

Note that the root element is the same as the module types' key, `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 US don't know anything other
than JIRA!!";
      }
    }
}

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 UK don't know anything other
than JIRA!!";
    }
  }
}
```

Using the new modules created

Once the new modules are defined, `myUSEnglishDictionary` and `myUKEnglishDictionary` in our example, we can use these in other plugin modules. For example, if we want to use them in a servlet module to find the definition of JIRA in both the dictionaries, it can be done using the following steps:

1. Get all the enabled modules that use the dictionary module descriptor.

   ```
   List<DictionaryModuleDescriptor> dictionaryModuleDescriptors =
   pluginAccessor.getEnabledModuleDescriptorsByClass(DictionaryModule
   Descriptor.class);
   ```

   Here, `pluginAccessor` can be retrieved as follows:

   ```
   PluginAccessor pluginAccessor = ComponentManager.getInstance().
   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:

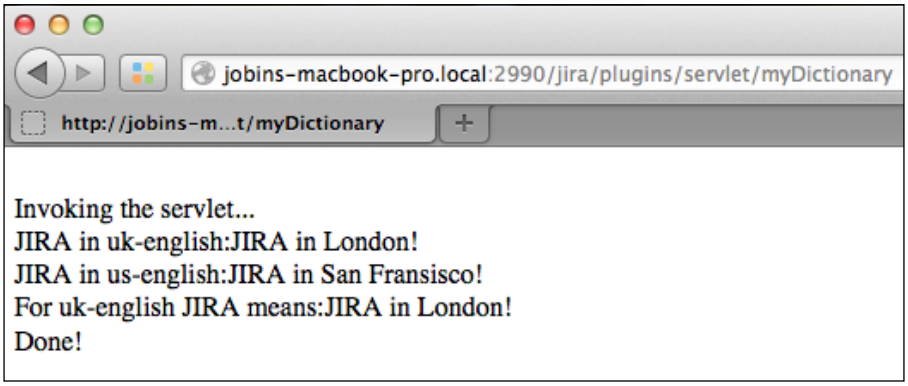
   ```
   private String getJIRADescription(String key) {
     // To get all the enabled modules of this module descriptor
     List<DictionaryModuleDescriptor> dictionaryModuleDescriptors =
   pluginAccessor.getEnabledModuleDescriptorsByClass(DictionaryModule
   Descriptor.class);
   ```

```
  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, it will appear as follows:



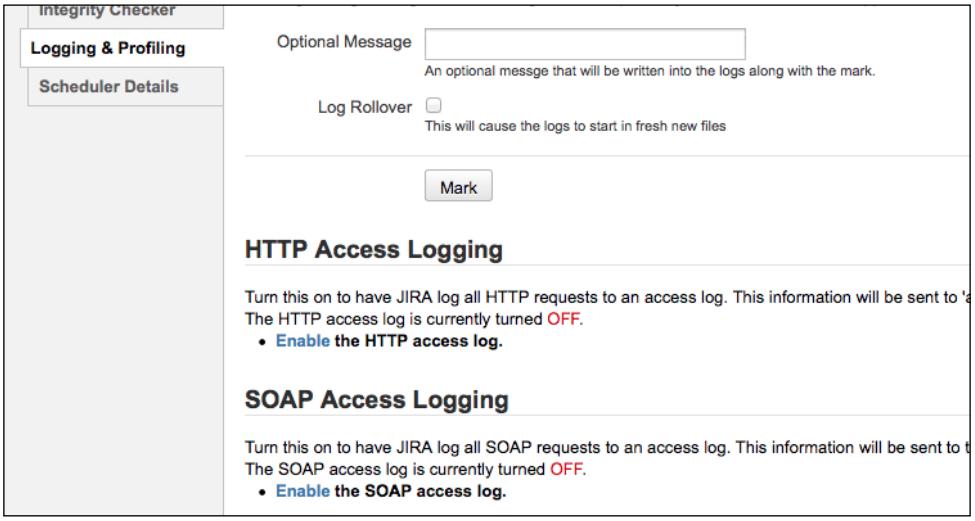
Enabling access logs in JIRA

Access logs are a good way to find 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 from 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.

In JIRA 5.x, 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 and SOAP access logs separately as shown. There is an additional option to turn on the HTTP dump log and SOAP dump log as well. For HTTP, we can also include images in the HTTP access logs.

All these logs are disabled by default and if enabled via GUI, they will be disabled again on 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 `Access logs`, as shown next:

```
#########################################################
# Access logs
#########################################################

log4j.logger.com.atlassian.jira.soap.axis.JiraAxisSoapLog  = OFF,
soapaccesslog
log4j.additivity.com.atlassian.jira.soap.axis.JiraAxisSoapLog = false

log4j.logger.com.atlassian.jira.soap.axis.JiraAxisSoapLogDump  = OFF,
soapdumplog
```

```
log4j.additivity.com.atlassian.jira.soap.axis.JiraAxisSoapLogDump =
false

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 SOAP access logs will be written to `atlassian-jira-soap-access.log`, the SOAP dump logs to `atlassian-jira-soap-dump.log`, the HTTP access logs to `atlassian-jira-http-access.log`, and the HTTP dump logs to `atlassian-jira-http-dump.log` files, everything residing under the `logs` folder.

You can find detailed information in the access logs, similar to the following:

```
0:0:0:0:0:0:0:1%0 o1384x49x1 jobinkk [24/Feb/2013:23:04:34 -0500] "GET
http://localhost:8080/secure/AdminSummary.jspa HTTP/1.1" 200 22795
0.0990 "http://localhost:8080/secure/Dashboard.jspa" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:19.0) Gecko/20100101 Firefox/19.0"
"1kj296"
0:0:0:0:0:0:0:1%0 i1384x51x1 jobinkk [24/Feb/2013:23:04:40 -0500]
"GET http://localhost:8080/secure/admin/ViewLogging.jspa HTTP/1.1"
- - - "http://localhost:8080/secure/AdminSummary.jspa" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:19.0) Gecko/20100101 Firefox/19.0"
"1kj296"
0:0:0:0:0:0:0:1%0 i1384x51x1 jobinkk [24/Feb/2013:23:04:40 -0500]
"GET http://localhost:8080/secure/admin/ViewLogging.jspa HTTP/1.1"
- - - "http://localhost:8080/secure/AdminSummary.jspa" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:19.0) Gecko/20100101 Firefox/19.0"
"1kj296"
```

It is also possible to change the individual logfile's name or path in the `log4j.properties` file by modifying the appropriate properties. For example, the SOAP access logfile can be written to `/var/log/soap-access.log` by modifying the `log4j.appender.soapaccesslog.File` property, as follows:

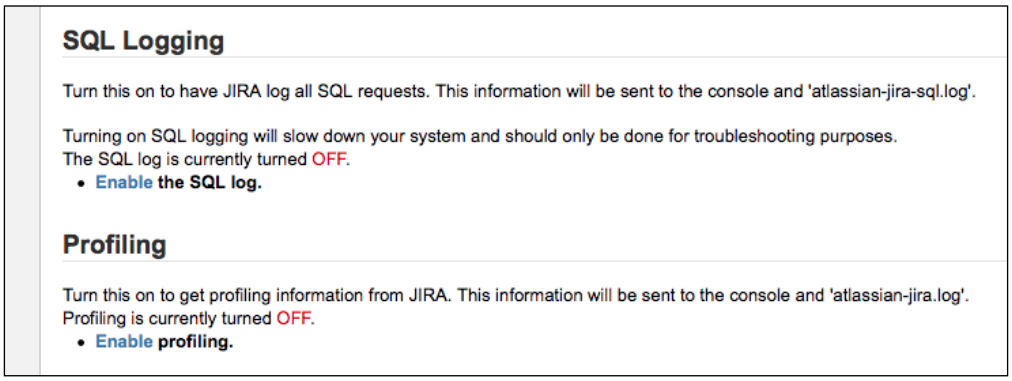
```
log4j.appender.soapaccesslog.File=/var/log/soap-access.log
```

Enabling SQL logging in JIRA

Similar to access logs, another useful piece 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...

In JIRA5, SQL logging can be turned on from the user interface at **Administration** | **System** | **Troubleshooting and Support...** | **Logging & Profiling**, as shown here:



Just like access logs, the changes are temporary and will be reverted back in the next restart. Permanent changes can be made in the `WEB-INF/classes/log4j.properties` file, although it is not recommended. In this case, the logging entry to be modified is as follows:

```
log4j.logger.com.atlassian.jira.ofbiz.LoggingSQLInterceptor = OFF,
sqllog
log4j.additivity.com.atlassian.jira.ofbiz.LoggingSQLInterceptor =
false

log4j.logger.com.atlassian.jira.security.xsrf.
XsrfVulnerabilityDetectionSQLInterceptor = OFF, xsrflog
log4j.additivity.com.atlassian.jira.security.xsrf.
XsrfVulnerabilityDetectionSQLInterceptor = 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 `logs` folder.

You can find details of numerous SQLs executed as follows:

```
2013-02-24 23:08:42,522 http-8080-3 jobinkk 1388x96x3 1kj296 /
secure/EditAction!default.jspa 0ms "SELECT ID, FIELDIDENTIFIER,
SEQUENCE, FIELDSCREENTAB FROM PUBLIC.fieldscreenlayoutitem WHERE
FIELDSCREENTAB='10000'"
2013-02-24 23:08:42,523 http-8080-3 jobinkk 1388x96x3 1kj296 /secure/
EditAction!default.jspa 1ms Connection returned. borrowed : 1
```

As in the case of access logs, the SQL logfile path can be changed by modifying the `log4j.appender.sqllog.file` property as follows:

```
log4j.appender.sqllog.File=/var/log/sql.log
```

Overriding JIRA's default components in plugins

JIRA uses `PicoContainer` as a central object factory. `PicoContainer` is responsible for instantiating objects and resolving their constructor dependencies. Within JIRA a lot of Manager, Service, and Utility classes are already registered with `PicoContainer`. The registration happens in the `ComponentRegistrar` class' `registerComponents()` method and these classes can be retrieved via dependency injection or using the `ComponentManager` class' getter methods or the `getComponentInstanceOfType()` method.

While it is true that most of the plugins can work with these already-registered components and the new ones created using the Component Plugins module, sometimes the need arises to override an existing component registered within JIRA. In this recipe, we will see how to do that.

Getting ready

Create a skeleton plugin using the Atlassian Plugin SDK. The plugin must be v1.

How to do it...

The overriding of existing components in JIRA is also done using the **Component Plugins** module. But you must note that the plugin must be v1 and should be deployed under the `WEB-INF/lib` folder for the override to work. Following are the simple, yet powerful, steps:

1. Identify the component that we need to extend and find out the interface associated with it. For example, let us try to override the default JIRA `PriorityManager` interface. The interface in this case will be `com.atlassian.jira.config.PriorityManager`.
2. Add a component plugin module in the `atlassian-plugin.xml` file with the interface in *Step 1*:

   ```
   <component key="priorityManager " name="My Priority Manager"
   class="com.jtricks.MyPriorityManager ">
     <interface>com.atlassian.jira.config.PriorityManager</interface>
   </component>
   ```

   As usual, the component module has a unique key and an optional name. Here, the `class` attribute points to the new component's implementation class.

3. Create the implementation class used in the component plugin module, `com.jtricks.MyPriorityManager` in this case.

   We will need to implement all the methods in the `PriorityManager` interface but it is entirely up to us how to implement them. It would be a lot easier if we just needed to manipulate only a few methods in them because in that case we could choose to extend the default implementation class in JIRA and override only the methods we are interested in!

   For simplicity, let us assume we need to only modify the `createPriority` operation in `PriorityManager` to do some extra bits. For this, we can create the `MyPriorityManager` class by extending `com.atlassian.jira.config.DefaultPriorityManager`, the default implementation class in JIRA, and override the `createPriority` method:

   ```
   public class MyPriorityManager extends DefaultPriorityManager{

     public MyPriorityManager(ConstantsManager constantsManager,
   TranslationManager translationManager, JiraAuthenticationContext
   jiraAuthenticationContext, OfBizDelegator ofBizDelegator,
   IssueIndexManager issueIndexManager, ApplicationProperties
   applicationProperties) {
       super(constantsManager, translationManager,
   jiraAuthenticationContext, ofBizDelegator, issueIndexManager,
   applicationProperties);
   ```

```
  }

  @Override
  public synchronized Priority createPriority(String arg0, String
arg1, String arg2, String arg3) {
    System.out.println("Creating priority in overriden
component");
    return super.createPriority(arg0, arg1, arg2, arg3);
  }
}
```

In our example, let us just print a line to the logs, but we can do harder things here!

4. Package the plugin and deploy it under `WEB-INF/lib`.

How it works...

When JIRA is started, all the default components that are registered with `PicoContainer` are loaded first. But when the plugins are loaded, if there is a component module that uses the same interface and a different implementation class, this class will be registered for that interface. In our example, `MyPriorityManager` is registered instead of the `DefaultPriorityManager` class.



This method fails in the case of some Manager classes, possibly because of the order in which the classes are loaded. You might want refer to the next section in such scenarios!

After overriding `PriorityManager` as discussed before, we will see the message printed into the server logs every time a priority is created, as shown in the following screenshot:

```
a.bc.dataimport.DefaultExportService] Finished saving the Active Objects Backup
2013-02-25 00:26:35,527 QuartzWorker-0 ERROR ServiceRunner    Service Provider T
oken Remover [sal.jira.scheduling.JiraPluginSchedulerService] Unable to load a s
ervice descriptor for the job 'Service Provider Token Remover'. This is usually
the result of an obsolete service that can removed in the Administration section
.
2013-02-25 00:26:35,528 QuartzWorker-0 ERROR ServiceRunner    upmPluginUpgradeCh
eckJob [sal.jira.scheduling.JiraPluginSchedulerService] Unable to load a service
 descriptor for the job 'upmPluginUpgradeCheckJob'. This is usually the result o
f an obsolete service that can removed in the Administration section.
2013-02-25 00:28:05,494 Modification Check:thread-1 INFO      [atlassian.jira.st
artup.JiraStartupLogger]

___ Modifications ____________________________

     Modified Files                              : entitydefs/entitygroup.xml,
 system-issueoperations-plugin.xml, jira-application.properties, entitydefs/enti
tymodel.xml, com/atlassian/jira/web/action/JiraWebActionSupport.properties
     Removed Files                               : None

Creating priority in overriden component
Creating priority in overriden component
```

There's more...

While overriding a component, using the Component Plugins module is the recommended way. If that doesn't work, modifying the JIRA code is probably the only option.

Overriding by modifying the JIRA code

For people who have modified the JIRA source distribution, overriding the component can be done in a single line. After creating the new component—implementing the interface that we want to override—we can modify the `registerComponents` method in the `com.atlassian.jira.ContainerRegistrar` class to include the new class instead of the default class.

For example, `PriorityManager` can be overridden by replacing:

```
register.implementation(PROVIDED, PriorityManager.class,
DefaultPriorityManager.class);
```

With:

```
register.implementation(PROVIDED, PriorityManager.class,
MyPriorityManager.class);
```

Note that components can either be `INTERNAL`, meaning that they will be available only to JIRA itself, or `PROVIDED`, in which case they will also be available to the `plugins2` plugins.

Internationalization in webwork plugins

We have seen in the 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 last n in internationalization!

How to do it...

Internationalization in a webwork plugin is achieved with the help of a resource bundle with the same name as the action that it is associated with. The following are the steps to enable it in a JIRA webwork plugin:

1. Create a properties file with the same name as that of the action class under the same package structure under the `src/main/resources` folder in the plugin.

   For example, if we consider the `RedirectAction` example in the previous recipes, the property file will be `RedirectAction.properties` residing under the `src/main/resources/com/jtricks` folder.

2. Add the key-value pair of properties that needs to be used in action as follows:

   ```
   good.bye=Good Bye
   ```

   Here, `good.bye` is the key that will be used and will be the same across all language property files. The value here, that is "Good Bye", will be used for the default locale but will have the equivalent translations in the other language property files.

3. Create property files in the same folder for other required languages in the following format: `${actionName}_${languageCode}_${countryCode}.properties`. For example, if we need to personalize the preceding action for UK and US and French users, the following will be the property filenames:

   ```
   RedirectAction_en_US.properties
   RedirectAction_en_UK.properties
   RedirectAction_fr_FR.properties
   ```

4. Add the `good.bye` property in each of the property files with the appropriate translation as the values. For example, a property with the value "Good Bye" in the English property file will have the value "Au revoir" in French!

   ```
   good.bye=Good Bye (in RedirectAction_en_UK.properties)
   good.bye=Au revoir (in RedirectAction_fr_FR.properties)
   ```

5. In the action class, use the `getText(key)` method to retrieve the appropriate message. Keep in mind that the action class extends the `JiraWebActionSupport` class, which implements the `getText` method!

   For example, the value "Good Bye" can be printed to users in different locales in their own language as follows:

   ```
   System.out.println(getText("good.bye"));
   ```

This magic, however, is broken in v2 plugins and there is already an issue reported with Atlassian at `https://jira.atlassian.com/browse/JRA-23720`. The workaround is to override the `getTexts` method as follows in the action class:

```
@Override
public ResourceBundle getTexts(String bundleName) {
  return ResourceBundle.getBundle(bundleName, getLocale(), getClass().
getClassLoader());
}
```

Here we get the resource bundle using the class loader of the action class and that fixes the preceding issue in v2 plugins!

Before we wind up, if you need to get the `i18n` texts in the velocity templates, the following are the steps:

1. Add the property files as before.
2. Add the resource entry in the `atlassian-plugin.xml` file as follows:

   ```
   <resource name="common-18n" type="i18n" location="com.jtricks.
   RedirectAction"/>
   ```

   Here, the resource points to the property file with the package and name (omitting the country or language code).

3. Use the `$i18n` object to retrieve the property values as follows:

   ```
   $i18n.getText("good.bye")
   ```

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 the v1 plugin has got access to all the libraries and classes available in the application class path whereas v2 plugins can't access them.

For example, the v1 plugins can access some common utility classes by dropping the JAR file with those classes in the `WEB-INF/lib` folder or adding those classes under `WB-INF/classes`. But that won't work with v2 plugins as they need the JAR files embedded with them under `MET-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 the Atlassian Plugin SDK.

How to do it...

Let us assume we have a Number utility class that does the 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:

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 classes that need to be shared so that it is visible to other v2 plugins. This step is very important.

   Even though it is a simple utility class, we need the `atlassian-plugin.xml` file for this step. We can use `bundle-instructions` under the `plugin-info` element in the `atlassian-plugin.xml` file for exporting selected packages to other plugins/bundles.

   The `bundle-instructions` element allows child elements.

   - `Export-Package`: To export selected packages from the plugin to be shared across other plugins.
   - `Import-Package`: To import only selected packages into a plugin. By default it imports all the exported packages from other plugins.

   In this case, we need to modify the `atlassian-plugin.xml` file to export our utility class and this can be done as follows:

```
<plugin-info>
  <description>Shared Utilities</description>
  <version>2.0</version>
  <vendor name="JTricks" url="http://www.j-tricks.com/" />
  <bundle-instructions>
    <Export-Package>com.jtricks.utilities</Export-Package>
  </bundle-instructions>
</plugin-info>
```

3. It is possible to export only selected versions and choose not to export certain packages. More details on this can be found at `http://www.aqute.biz/Bnd/Bnd`.

4. Optionally, we can use the `Import-Package` element to import the preceding exported package. By default, it will be imported anyway and hence this step can be omitted. However it will be useful when you want to import only selected packages or make the import mandatory and so on. Again, the details can be found in the preceding link.
5. Package the plugin and deploy it as a v2 plugin.

Now the utility class is available to all the other v2 plugins. When developing, the other plugins should have this class in the classpath, which can be achieved by adding the preceding plugin as a dependency in the `pom.xml` file with scope as provided.

```
<dependency>
  <groupId>com.jtricks</groupId>
  <artifactId>utility-plugin</artifactId>
  <version>1.0</version>
  <scope>provided</scope>
</dependency>
```

When we do that, the `add` method can be invoked as if the class is within the same plugin. For example, the `RedirectAction` class may have the method as follows:

```
@Override
protected String doExecute() throws Exception {
  System.out.println("Action invoked. Doing something important before
redirecting to Dashboard!");
  System.out.println("Sum:"+NumberUtility.add(3, 5));
  return getRedirect("/secure/Dashboard.jspa");
}
```

Operations using direct HTML links

It probably makes sense to wind up this book by giving a little tip on how we can do powerful operations in JIRA by a simple click on a link, either from your e-mail or a from a web form or from within JIRA itself!

Almost all the 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 it will override all the preprocessing, validations, and so on in some cases.

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` is the URL-encoded argument 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 see this in detail.

How to do it...

Let us consider a simple example to start with; that is, creating issues. Creating an issue has four stages.

- Going to the initial create screen
- Selecting the project and `issuetype` and clicking on **Next**
- Entering all the details on the issue
- Clicking on **Submit** with the details

We can execute each of these in single steps provided we know the details in advance. For 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` values are pre-selected when you access this link, because that is the JIRA default behavior. But what if we want to pre-select some other project? All we need is to add the `pid` parameter in the URL as follows:

   ```
   http://localhost:8080/secure/CreateIssue!default.jspa?pid=10100
   ```

2. If we need to go to the second step directly by selecting the project and issuetype, just add the `issuetype` parameter as well into the URL, separated by `&`.

   ```
   http://localhost:8080/secure/CreateIssue!default.
   jspa?pid=10100&issuetype=1
   ```

3. If we need to pre-populate the 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=1010
   0&issuetype=1&priority=1&summary=Emergency+Bug&reporter=jobinkk
   ```

   Note that all the mandatory fields should be populated to avoid validation errors. The preceding example also shows how the URL is encoded to comply with HTML syntax by replacing the space in the summary with a `+`. That is, `Emergency Bug` is written as `Emergency+Bug`, which can also be written as `Emergency%20Bug`.

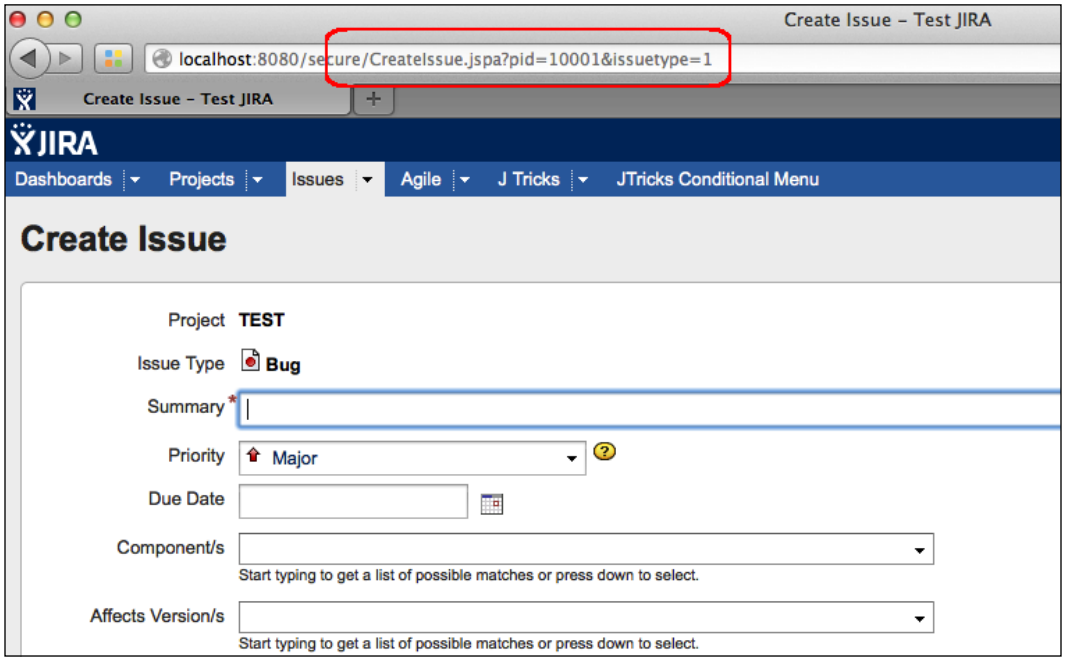
4. And if we want to create the issue in one click with the preceding details, use the `CreateIssueDetails` action instead of `CreateIssueDetails!init`:

   ```
   http://localhost:8080/secure/CreateIssueDetails.jspa?pid=10100&issuetype=1&priority=1&summary=Emergency+Bug&reporter=jobinkk
   ```

   Hopefully, that gives us an idea about how the operation can be executed via direct links. Make sure the user is logged in or anonymous issue creation is turned on when the previous link is clicked.

But how do we find out which action class is involved or what are the parameters to be passed?

This you can do easily from the browser URL if the request uses the `GET` method. Create an issue with the project and issuetype selected (case 2 explained earlier is an example), as shown next:



When the request is POST, as in case 4, we can find out the action name from the URL but the parameters need to be worked out from what is posted when the action is executed. There are multiple ways to do it and an easy way out of them would be to use the browser capabilities. For example, using Firebug with Mozilla Firefox will get you the parameters posted when an action is executed, as shown here:



Here we can see the parameters **pid**, **issuetype**, **priority**, **summary**, and **reporter** getting submitted in the **Params** section. Also, we can see the action name in the URL. Once you get the list of parameters, we can use them in the URL with appropriate values separated by `&` as we saw in *Step 4*.

This technique opens up lot of possibilities. For example, we can easily automate the submission of these URLs that we have constructed using command-line tools such as **wget** or **curl**. Read about these more at `http://confluence.atlassian.com/display/JIRA/Creating+Issues+via+direct+HTML+links` and `http://confluence.atlassian.com/display/JIRACOM/Automating+JIRA+operations+via+wget`.

Index

Symbols

@GET annotation 381

A

AbstractJqlFunction class 223
AbstractNamedRemoteEntity class 385
access logs
enabling, in JIRA 467-469
active objects
used, for data storage 405-407
active workflow
editing 136, 137
addAttachments method 365
addPermissionTo method 374
Administration | Plugins 302
Administration screen 299
advanced searching 220
afterPropertiesSet() method 72, 74
Apache Lucene 34
atlas-integration-test command 29
Atlassian bundled plugins
about 50
modifying 50, 51
Atlassian JIRA 7
Atlassian Marketplace 9
Atlassian plugin software development kit.
See **SDK**
atlassian-plugin.xml, components
atlassian-plugin element 42
plugin-info element 43, 44
plugin modules 44
atlas-unit-test command 29
attachments
AttachmentInput, using 365
browsing 366
creating 266
deleting 267
file, using 365
input stream, using 364
new filename, using 364, 365
reading, on issue 267
working with 265, 266, 364
available workflow actions
ids, naming 146
obtaining programmatically 144, 145

B

Browse Component screen
new tabs , adding 328, 329
Browse Project screen
new tabs, adding 319
Browse Version screen
new tabs , adding 325, 327
bundle dependency system 46

C

cascadingselect property 185
changegroup 424
change history
dealing with, in database 424-426
changeitem table 424
change logs
programming 277, 278
change log value
customizing 94-96
checkbox property 185
comments
creating, from e-mail 287-289
deleting 276
updating 276

working with 368
common transitions
creating, in workflow 153-158
Component Plugins module 472
components
adding, in JIRA 459-462
importing, in JIRA 459-462
managing 373
working 462
conditions
about 114
adding, for web garments 305-307
constructor injection 80
contextInitialized method 456
core APIs 49, 50
createProject method 375
CronEditorBean class 252
Cron expression 249
Crow 36
CrowdService 36
curl 481
custom behavior
adding, for user details 445-449
custom field details
retrieving, from database 414-416
custom field project
importing 108, 109
custom fields
about 78
adding, to notification mails 103-105
dealing with 88-90
displaying, on subtask columns 100, 101
help text, adding 105, 106
making, sortable 99, 100
migrating from one another 410
options, programming 90-92
searchers 83
validation, overriding 93
writing 78-82
custom fields searchers
about 83
customfield-searcher module 84-86
working 87
custom field type
changing 98
migrating from 96-98
custom schema
used, for JIRA DB extension 395-398

D

database
change history, dealing with 424-426
custom field details, retrieving 414-416
groups, retrieving 422-424
issue information, retrieving 411-414
issues permissions, retrieving 416-419
issues, updating 421, 422
users, retrieving 422-424
workflow details, retrieving 419, 420
data entities
exposing, as REST APIs 378-383
data validation
in JIRA report 177-179
date fields 102, 103
date property 184
DB entities
accessing, from plugins 399, 400
new record, writing 401
record, updating 401
debugging 28, 29
decorators
using 300-302
working 302-304
deindexing 244-246
delegator class 399
destroy() method 72
Developers role 121
development environment
about 10
IDEs, configuring 12, 13
local Maven, using 12
Proxy settings for Maven 11, 12
setting up, steps 10, 11
troubleshooting 13
development process, JIRA plugin
Atlassian Marketplace 9
steps 8, 9
troubleshooting 10
direct HTML links
used, for operations 478-481
directory_position column 422
doValidation method 67

drop-down menu
adding, on top navigation bar 310

E

Edit Issue permission 138
e-mail
comments, creating 287, 288
issues, creating 287, 288
e-mail content
custom field information, adding 444
customizing 442-444
Embedded Crowd 422
Entity Engine module 394
Excel format
reports 174-177

F

FastDev
about 24
admin credentials, changing 28
ignored files, adding 27
using, for plugin development 24-27
Favourite Filters gadget 214
field 220
filter
creating 246
deleting 247
subscribing 249-252
updating 247
used, for query interception 456-459
filterpicker property 185
filterprojectpicker property 185
filters
managing, programmatically 246-248
filter subscriptions
about 249
cron expression 249
e-mail on empty 250, 251
group name 250
findListIteratorByCondition method 400
fragments
used, for project tab panel creating 321-325
function 220

G

gadget.getMsg method 209
gadgets
adding, into Gmail 214-217
getCategoryFromId() method 383
getChangelogValue method 94
getComponentInstanceOfType() method 471
getDataType method 222
getIssueObject method 283
getLanguage() method 466
getMinimumNumberOfExpectedArguments method 222
getModule method 464
getProjectCategories() method 386
getProjectsNoSchemes method 359
getRedirect() method 445
getText(key) method 475
getTimePerformed() method 316
getValues() method 170
getVelocityParamsForEdit method 118, 124
getVelocityParamsForInput method 117, 124, 131
getVelocityParamsForView method 117, 124, 131
global transition
about 158
creating, in workflow 159, 160
Gmail | Settings | Gadgets 215
groups
from database, retrieving 422, 423

H

help text
adding, to custom fields 105, 106
hidden property 184
HitCollector 231

I

i18n 19
indexing 244-246
init method 313

internationalization
in webwork plugins 474-476
workflow transitions 142-144
issue
creating, from e-mail 287-289
creating, from plugin 254-256
creating, IssueManager used 256
deleting 259
making, editable 138
subtasks, creating 257
updating 258, 259
working with 360-363
issue cloning
fields, discarding 282, 283
issue comments
creating 275
restricting, to project role 275, 276
working with 275
issueCreated method 437
issue fields
JavaScript tricks 284, 286
IssueIndexManager 245
issue information
retrieving, from database 411-414
issue link renderers
adding 330-337
working 337
issue links
about 279
deleting 280
programming 279
retrieving, on issue 281
Issue Navigator 229
issue operations
adding 260-263
conditions 263-265
reordering, on View Issue page 346, 347
Issue Operations module 263
Issue Operations Plugin Module 260
Issue Security scheme 416
issues permissions
retrieving, from database 416-419
issuetype parameter 479
issue warning/notifications
displaying 342-345

J

Java client
writing, for REST API 354, 355
JavaScript tricks
issue fields 284, 286
Java XML-RPC client
writing 358
JDBC calls
database connection, obtaining for 409
Jelly escalation 160-162
Jelly scripts 35
Jelly Service 160, 161
JIRA
access logs, enabling 467-469
building, from source 59-61
components, adding 459-461
components, exposing to other plugins 461
components, importing 459-461
feature 292
JIRA dependencies, building 62
listeners, writing 435-440
module types, adding 463-465
pie chart, creating 189-194
public components, importing 462
query interception, filters used 456-459
reporting 166
scheduled tasks, writing 433-435
service properties, using in components 462
service, writing 428-430
servlet, deploying 450-453
SOAP service, deploying 384-388
SQL logging, enabling 470, 471
webwork actions, adding to 63-67
XML-RPC service, deploying 388-390
JIRA 4 gadgets
about 194
REST services, invoking 202-206
user preferences, configuring 206-213
working 199-202
writing 194-197
JIRA appearance
modifying 292-294
JIRA architecture
about 32, 34

database 37
diagram 35
JIRA Utility 35
Manager Classes 35
plugins 38, 39
presentation 37
property management 37
scheduled jobs 38
searching 38
third-party components 32
user authentication 36
user management 36
workflow 38
JIRA configuration properties
accessing 408, 409
JIRA DB
extending, custom schema used 395-398
plugin information, persisting 401-404
JIRA default components
JIRA code, modifying 474
JIRA Home directory 37
jiraissue table 412
JIRA plugin
about 8
configuration, reusing 22
deploying 19-21
plugin development process 8
specific version, using 21
troubleshooting 22
JIRA Query Language. *See* **JQL**
JIRA report
data validation 177, 178
Object-configurable parameters 181-189
working 173, 174
writing 167-171
JIRA report access
restricting 179, 181
JIRA reporting
types 166
JIRA's default components
overriding, in plugins 471-474
JIRA system custom fields 47
JIRA system plugins
about 46
system-contentlinkresolvers-plugin.xml 47
system-customfieldtypes-plugin.xml 47
system-footer-plugin.xml 47
system-issueoperations-plugin.xml 47
system-issuetabpanels-plugin.xml 47
system-issueviews-plugin.xml 47
system-jql-function-plugin.xml 48
system-keyboard-shortcuts-plugin.xml 48
system-macros-plugin.xml 48
system-project-plugin.xml 48
system-projectroleactors-plugin.xml 48
system-rendercomponentfactories-plugin.xml 48
system-renderers-plugin.xml 48
system-reports-plugin.xml 48
system-top-navigation-plugin.xml 48
system-user-format-plugin.xml 48
system-user-profile-panels.xml 48
system-webresources-plugin.xml 48
system-webwork1-plugin.xml 48
system-workflow-plugin.xml 48
JQL 166 219
JQL function
about 220-224
sanitizing 226, 228
working 225, 226
JQL function module 221
JQL query
parsing, in plugins 242
J Tricks 311

L

listeners
plugins, disabling 440, 441
plugins, enabling 440, 441
plugins, handling 440, 441
writing, in JIRA 435-440
location key 298
login method 387
long property 182

M

meta tag 302
mode parameter 243
Module Type Plugin module 463
module types
adding, to JIRA 463-465
created modules, using 466
used, for module creating 465

multiselect property 183
mvninstall commands 62

N

new tabs
adding, in Browse Component screen 328, 329
adding, in Browse Project screen 319
adding, in Browse Version screen 325-327
adding, in View Issue screen 315-318
new web item, UI
adding 297-299
new web sections, UI
adding 295, 297
working 296, 297
none option
removing, from select field 107

O

Object-configurable parameters
about 181
cascadingselect 185
checkbox 185
date 184
filterpicker 185
filterprojectpicker 185
hidden 184
long 182
multiselect 183
predefined key/value pairs 183
select 182
string 182
text 184
user 184
working 187, 189
Object Relational Mapping. *See* **ORM**
Open For Business 394
operator 220
options 91
ORM 33
os_historystep table 419
OSUser 32
OSWorkflow 33, 114

P

Permission scheme 416
pie chart
creating, in JIRA 189-194
plugin
debugging, in Eclipse 23
redeploying 23
used, for issue creating 254-256
plugin descriptor 8
plugin install 23
plugin installation events
capturing 71-75
plugin key 9
plugin modules
about 17, 39
actions and components 41
adding 17-19
custom fields functionality type 39
links functionality type 40
remote invocation type 41
reporting functionality type 39
searching functionality type 40
searchingfunctionality type 40
workflow functionality type 39
plugin modules types
ao 42
atlassian-plugin.xml, components 42
keyboard shortcut 42
module type 42
resource 41
servlet 41
servlet-context-listener 41
servlet-context param 41
servlet-filter 41
user-format 42
web-resource 41
web-resource transformer 41
plugins
about 7
JIRA's default components, overriding 471-474
JQL query, parsing 242
searching in 239-241

Plugins1 versions
about 44
converting, to Plugins2 versions 51-53
working with 45, 46
Plugins2 versions
about 44
working with 45, 46
plugin uninstallation events
capturing 71-75
post functions 114
project tab panel
creating, fragments used 321-325
PropertySet 33, 37

Q

Quartz 252
Quartz API 38
quick search
assignee 236
components 238
date fields 237
issue key 236
issue type 237
priority 237
project 236
reporter 236
resolution 237
used, for smart querying 235-238
versions 237

R

read-only custom field 79
RedirectAction class 478
remote administration methods 374-377
RemoteCategory object 390
remote group management 369, 370
remote time tracking
about 366
steps 366, 368
remote user management 369, 370
removeVersion method 372
Report Plugin Module 167
reports
in Excel format 174-177
requiresAsyncLoading method 335
resolutions
including, for specific transitions 139, 140
resources
adding, to plugin 54, 55
REST 354
REST API
Java client, writing 354, 355
REST services
invoking, from JIRA 4 gadgets 202-206
REST/SOAP/XML-RPC 35
RunSearchRequest method 162

S

Scan and Reload button 26
scheduled tasks
writing, in IRA 433-435
schemeissuesecuritylevels 418
SDK 10
search queries
linking directly to 243, 244
search request view
adding 229-235
search request view plugin module 229
Search views 47
select field
none option, removing 107
velocity changes, reloading 108
select property 182
Seraph 32
service
configurable parameters, adding 430-432
exposing, as REST APIs 378-383
writing, in JIRA 428-430
servlet
deploying, in JIRA 450-453
servlet context
shared parameters, adding 453-456
Servlet Context Parameter plugin 453
Servlet Filter Plugin 456
Servlet plugin module 450
setFields method 283
Shared Access Layer (SAL) 435
showPanel method 320, 329
simple link factory 312

simple searching 220
Single issue views 47
skeleton plugin
about 13
creating 13, 14
Eclipse project, creating 16
working 14-16
smart querying
used, for quick search 235-238
SOAP 354
SOAP client
creating 356, 357
SOAP service
deploying, in JIRA 384-388
soy templates 37
special characters
- 249
, 249
? 250
* 249
/ 250
250
L 250
W 250
Spring Dynamic Modules 53
SQL logging
enabling, in JIRA 470, 471
stable APIs 49, 50
steps 114
string property 182
subtasks
creating, on issue 257
System content link resolvers 47
System issue operations 47
System issue tab panels 47
System Webwork plugin 48

T

TDD 28, 29
template function 205
test-driven development. *See* **TDD**
testing
about 28, 29
integration/functional tests, custom data used 30
JIRA/Tomcat versions, testing against 30
Test User 130
text area custom field
size, changing 109, 111
text property 184
third-party components
Active Objects 33
Apache Lucene 34
Embedded Crowd 33
gadget framework 34
OfBiz entity engine 33
OSUser 32
OSWorkflow 33
PropertySet 33
Quartz 34
Seraph 32
webwork 32
time tracking
about 267, 268
steps 268
top navigation bar
new drop-down menu, adding 310
transitions 114
TransitionWorkflow method 162

U

UI
new web sections, adding 295
Universal Plugin Manager. *See* **UPM**
UPM 20, 430
Use Common Transition option 154
user details
custom behavior, adding 445-449
user fields 102, 103
User Format plugin module 446
User Picker 130
user property 184
users
from database, retrieving 422, 423

V

v2 plugins
common libraries, sharing 476-478
validate method 222
validators 114
value 220

Value Generator class 170
Velocity context
creating, for web garments 308, 309
Velocity templates 37
versions
managing 371, 372
View Issue page
about 350
fields, reordering 348-352
issue operations, reordering 346, 347
View Issue screen
issue tab panel, loading asynchronously 318
new tabs, adding 315-318
View Workflow Steps page 153

W

web fragments
conditions, adding 305-307
new Velocity context, adding 308, 309
Web Item plugin 295
Web Item Plugin Module 260
web items
about 297
creating dynamically 311-314
web resources
adding, to plugin 56, 57
batch mode, turning off 58
contexts 58
web-section module 295
Web Section plugin 295
Webwork 32
webwork action
extending, for UI element addition 338-342
webwork actions
adding, to JIRA 63-67
commands, adding to 68
different page, redirecting 444
extending, in JIRA 68-71
webwork plugins
about 300
internationalization 474-476
wget 481
workflow
common transitions, creating 153-158
global transition, creating in 159, 160
issue, progressing 370
workflow actions
reordering, in JIRA 151, 152
workflow condition
writing 115-121
workflow history
obtaining, from database 148-150
WorkflowManager 145
workflow post function
writing 129-135
workflow properties 138
workflows
about 114, 115
progressing programmatically 146-148
workflow status 114
workflow status based permissions
about 140-142
working 142
workflow transitions
internationalization 142, 143
workflow validator
writing 122-128
worklog management
about 267, 268
remaining estimate, auto adjusting 269
remaining estimate, retaining 270
steps 268
work, logging 270-272
worklogs, deleting 274
worklogs, updating 273
worklogs
deleting 274
remaining estimate, adjusting 275
remaining estimate, auto adjusting 274
remaining estimate, retaining 274

X

XML/RPC 354
XML-RPC service
deploying, in JIRA 388-390
XmlRpcServiceImpl class 389



Thank you for buying

JIRA 5.x Development Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

About Packt Enterprise

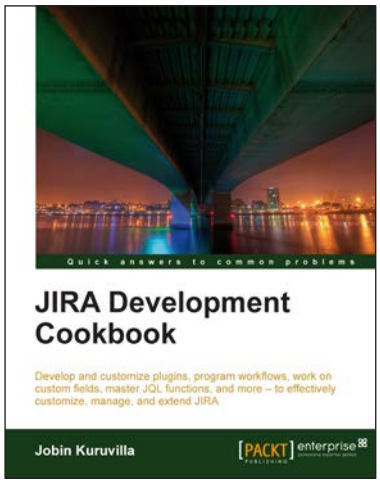
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



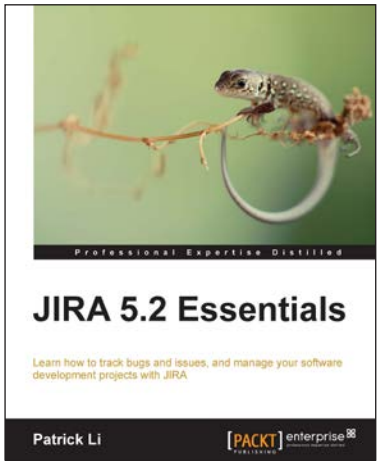
JIRA Development Cookbook

ISBN: 978-1-84968-180-3 Paperback: 476 pages

Develop and customize plugins, program workflows, work on custom fields, master JQL functions, and more – to effectively customize, manage, and extend JIRA

1. Extend and Customize JIRA--Work with custom fields, workflows, Reports & Gadgets, JQL functions, plugins, and more
2. Customize the look and feel of your JIRA User Interface by adding new tabs, web items and sections, drop down menus, and more
3. Master JQL - JIRA Query Language that enables advanced searching capabilities through which users can search for issues in their JIRA instance and then exploit all the capabilities of issue navigator

---



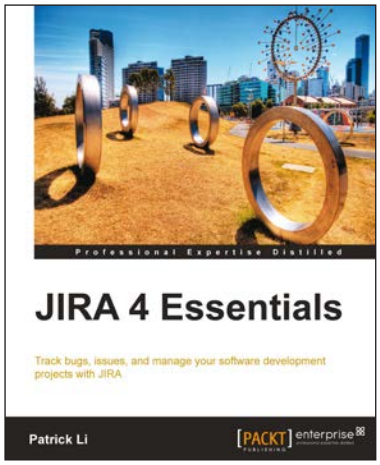
JIRA 5.2 Essentials

ISBN: 978-1-78217-999-3 Paperback: 340 pages

Learn how to track bugs and issues, and manage your software development projects with JIRA

1. Learn how to set up JIRA for software development
2. Effectively manage and handle software bugs and issues
3. Includes updated JIRA content as well as coverage of the popular GreenHopper plugin

[PACKT] enterprise
professional expertise distilled
PUBLISHING



JIRA 4 Essentials

ISBN: 978-1-84968-172-8 Paperback: 352 pages

Track bugs, issues, and manage your software development projects with JIRA

1. Successfully manage issues and track your projects using JIRA
2. Model business processes using JIRA Workflows
3. Ensure only the right people get access to your data, by using user management and access control in JIRA
4. Packed with step-by-step instruction, screenshots, and practical examples

---



Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 488 pages

Over 110 easy-to-follow recipes backend up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips