JavaScript Cloud Native Development Cookbook

Deliver serverless cloud-native solutions on AWS, Azure, and GCP



Packt
www.packt.com

John Gilbert

JavaScript Cloud Native Development Cookbook

Deliver serverless cloud-native solutions on AWS, Azure, and GCP

John Gilbert

Packt

BIRMINGHAM - MUMBAI

JavaScript Cloud Native Development Cookbook

Copyright © 2018 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Commissioning Editor: Merint Mathew
Acquisition Editor: Alok Dhuri
Content Development Editor: Akshada Iyer
Technical Editor: Adhithya Haridas
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Jisha Chirayil
Production Coordinator: Aparna Bhagat

First published: September 2018

Production reference: 1260918

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

ISBN 978-1-78847-041-4

www.packtpub.com

To my wife, Sarah, and our families, for their endless love and support on this journey.



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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

At www.packt.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.

Foreword

I have known John for a little over two years through his involvement in cloud technologies, especially in serverless architectures and building applications using the Serverless Framework. He took us on a cloud-native journey, thinking and reasoning about new paradigms in building software on the cloud in his previous book, *Cloud Native Development Patterns and Best Practices*.

Continuing on his journey to explore cloud-native systems, he brings us his new book, *JavaScript Cloud Native Development Cookbook*, to showcase recipes in a cookbook format that deliver lean and autonomous cloud-native services. In a nutshell, the recipes in the book demonstrate how to build cloud-native software at a global scale, utilize event-driven architectures, and build an autonomous development environment, starting with a developer committing code to deploying the application using a continuous deployment pipeline.

Serverless computing is the way to go when building cloud-native applications. With no servers to manage or patch, pay-per-execution billing, no paying for idling, auto-scaling, and a microservices/event-driven architecture, there is really no reason not to adopt serverless computing.

In this book, John presents practical Node.js recipes for building serverless cloud-native applications on AWS. The recipes are battle-tested and work around the pitfalls that present themselves in real-life scenarios. The recipes are built with best practices and practical development workflows in mind.

John takes the traditional cloud-native principles and shows you how to implement them with serverless technologies to give you the ultimate edge in building and deploying modern cloud-native applications.

The book covers building a stack from scratch and deploying it to AWS using the Serverless Framework, which automates a lot of the mundane work, letting you focus on building the business functionality. It goes on to incorporate event sourcing, CQRS patterns, and data lakes, and shows you how to implement autonomous cloud-native services. The recipes cover leveraging CDN to execute code on the edge of the cloud and implementing security best practices. It walks you through techniques that optimize for performance and observability while designing applications for managing failures. It showcases deployment at scale, using multiple regions to tackle latency-based routing, regional failovers, and regional database replication.

You will find extensive explanations on core concepts, code snippets with how-it-works details, and a full source code repository of these recipes, for easy use in your own projects.

This book has a special place on my bookshelf, and I hope you will enjoy it as much as I did.

Rupak Ganguly
Enterprise Relations and Advocacy, Serverless Inc.

Contributors

About the author

John Gilbert is a CTO with over 25 years of experience of architecting and delivering distributed, event-driven systems. His cloud journey started more than five years ago and has spanned all the levels of cloud maturity—through lift and shift, software-defined infrastructure, microservices, and continuous deployment. He is the author of *Cloud Native Development Patterns and Best Practices*. He finds delivering cloud-native solutions to be by far the most fun and satisfying, as they force us to rewire how we reason about systems and enable us to accomplish far more with much less effort.

I want to thank Pierre Malko and the whole Dante team, past and present, for the roles everyone played in our cloud-native journey. I want to thank the team at Packt, particularly Alok Dhuri and Akshada Iyer, who made this book possible. And I want to thank Joseph Staley and Max Brinegar for their efforts as technical reviewers.

About the reviewers

Max Brinegar is a principal software engineer with credentials that include a B.S. degree in Computer Science from University of Maryland, College Park, experience in cloud native development at Dante Consulting, Inc., and an AWS developer certification. Experience and expertise include web services development and deployment, modern programming techniques, information processing, and serverless architecture for software on AWS and Azure.

Joseph Staley is a senior software developer who currently specializes in JavaScript and cloud computing architecture. With over 20 years of experience in software design and development, he has helped create solutions for many companies across many industries. Originally building solutions utilizing on-premise Java servers, he has transitioned to implementing cloud-based solutions with JavaScript.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface 1

Chapter 1: Getting Started with Cloud-Native 8
- **Introduction** 8
- **Creating a stack** 9
  - Getting ready 9
  - How to do it... 9
  - How it works... 12
- **Creating a function and working with metrics and logs** 14
  - How to do it... 14
  - How it works... 18
- **Creating an event stream and publishing an event** 19
  - How to do it... 19
  - How it works... 22
- **Creating a stream processor** 24
  - Getting ready 24
  - How to do it... 25
  - How it works... 27
- **Creating an API Gateway** 29
  - How to do it... 29
  - How it works... 32
- **Deploying a single-page application** 33
  - How to do it... 33
  - How it works... 36

Chapter 2: Applying the Event Sourcing and CQRS Patterns 38
- **Introduction** 39
- **Creating a data lake** 39
  - Getting ready 40
  - How to do it... 40
  - How it works... 42
- **Applying the event-first variant of the Event Sourcing pattern** 43
  - How to do it... 44
  - How it works... 46
- **Creating a micro event store** 47
  - How to do it... 47
  - How it works... 51
- **Applying the database-first variant of the Event Sourcing pattern with DynamoDB** 52
  - How to do it... 53

How it works... 57
Applying the database-first variant of the Event Sourcing pattern with Cognito datasets 58
How to do it... 58
How it works... 62
Creating a materialized view in DynamoDB 63
How to do it... 63
How it works... 66
Creating a materialized view in S3 67
How to do it... 67
How it works... 70
Creating a materialized view in Elasticsearch 71
How to do it... 71
How it works... 75
Creating a materialized view in a Cognito dataset 76
How to do it... 76
How it works... 80
Replaying events 81
Getting ready 81
How to do it... 81
How it works... 85
Indexing the data lake 86
How to do it... 86
How it works... 90
Implementing bi-directional synchronization 91
How to do it... 91
How it works... 95

Chapter 3: Implementing Autonomous Services 97
Introduction 97
Implementing a GraphQL CRUD BFF 98
Getting ready 98
How to do it... 98
How it works... 106
Implementing a search BFF 106
How to do it... 107
How it works... 111
Implementing an analytics BFF 112
How to do it... 112
How it works... 117
Implementing an inbound External Service Gateway 118
How to do it... 118
How it works... 121
Implementing an outbound External Service Gateway 122
Getting ready 122

How to do it... 122
How it works... 125
Orchestrating collaboration between services 125
How to do it... 126
How it works... 129
Implementing a Saga 130
How to do it... 130
How it works... 134

Chapter 4: Leveraging the Edge of the Cloud 135
Introduction 135
Serving a single-page application from a CDN 136
How to do it... 136
How it works... 139
Associating a custom domain name with a CDN 140
Getting ready 140
How to do it... 141
How it works... 143
Serving a website from the CDN 143
How to do it... 144
How it works... 148
Deploying a service behind a CDN 149
How to do it... 149
How it works... 152
Serving static JSON from a CDN 153
How to do it... 153
How it works... 157
Triggering the invalidation of content in a CDN 157
How to do it... 157
How it works... 161
Executing code at the edge of the cloud 162
How to do it... 162
How it works... 166

Chapter 5: Securing Cloud-Native Systems 167
Introduction 167
Securing your cloud account 168
How to do it... 168
How it works... 170
Creating a federated identity pool 171
How to do it... 171
How it works... 173
Implementing sign up, sign in, and sign out 174
How to do it... 174
How it works... 176

Securing an API Gateway with OpenID Connect 176
Getting ready 176
How to do it... 177
How it works... 179
Implementing a custom authorizer 179
Getting ready 180
How to do it... 180
How it works... 184
Authorizing a GraphQL-based service 184
Getting ready 184
How to do it... 185
How it works... 188
Implementing a JWT filter 188
Getting ready 189
How to do it... 189
How it works... 192
Using envelope encryption 193
How to do it... 193
How it works... 198
Creating an SSL certificate for encryption in transit 198
Getting ready 199
How to do it... 199
How it works... 201
Configuring a web application firewall 201
How to do it... 202
How it works... 204
Replicating the data lake for disaster recovery 205
How to do it... 205
How it works... 209

Chapter 6: Building a Continuous Deployment Pipeline 210
Introduction 210
Creating the CI/CD pipeline 211
Getting ready 211
How to do it... 211
How it works... 214
Writing unit tests 216
How to do it... 216
How it works... 217
Writing integration tests 219
Getting ready 219
How to do it... 219
How it works... 222
Writing contract tests for a synchronous API 223
How to do it... 223

How it works... 224
Writing contract tests for an asynchronous API 225
How to do it... 225
How it works... 227
Assembling transitive end-to-end tests 227
How to do it... 227
How it works... 229
Leveraging feature flags 230
Getting ready 230
How to do it... 230
How it works... 232

Chapter 7: Optimizing Observability 233
Introduction 233
Monitoring a cloud-native system 234
Getting ready 234
How to do it... 234
How it works... 235
Implementing custom metrics 236
How to do it... 236
How it works... 238
Monitoring domain events 239
Getting ready 239
How to do it... 239
How it works... 241
Creating alerts 241
How to do it... 241
How it works... 243
Creating synthetic transaction tests 244
Getting ready 244
How to do it... 245
How it works... 247

Chapter 8: Designing for Failure 248
Introduction 248
Employing proper timeouts and retries 248
How to do it... 249
How it works... 250
Implementing backpressure and rate limiting 250
Getting ready 251
How to do it... 251
How it works... 254
Handling faults 255
How to do it... 255
How it works... 257

Resubmitting fault events 258
How to do it... 258
How it works... 260
Implementing idempotence with an inverse OpLock 261
How to do it... 261
How it works... 263
Implementing idempotence with Event Sourcing 264
How to do it... 264
How it works... 266

Chapter 9: Optimizing Performance 267
Introduction 267
Tuning Function as a Service 268
How to do it... 268
How it works... 270
Batching requests 271
Getting ready 271
How to do it... 272
How it works... 274
Leveraging asynchronous non-blocking IO 275
How to do it... 275
How it works... 277
Grouping events in stream processors 277
How to do it... 277
How it works... 279
Autoscaling DynamoDB 280
How to do it... 280
How it works... 282
Utilizing cache-control 282
How to do it... 283
How it works... 284
Leveraging session consistency 285
How to do it... 285
How it works... 287

Chapter 10: Deploying to Multiple Regions 289
Introduction 289
Implementing latency-based routing 290
Getting ready 290
How to do it... 290
How it works... 292
Creating a regional health check 294
Getting ready 294
How to do it... 294
How it works... 297

Triggering regional failover 298
Getting ready 298
How to do it... 298
How it works... 302
Implementing regional replication with DynamoDB 302
Getting ready 303
How to do it... 303
How it works... 305
Implementing round-robin replication 307
How to do it... 307
How it works... 311
Chapter 11: Welcoming Polycloud 313
Introduction 313
Creating a service with Google Cloud Functions 314
Getting ready 314
How to do it... 314
How it works... 316
Creating a service with Azure Functions 316
Getting ready 317
How to do it... 317
How it works... 319
Other Books You May Enjoy 320
Index 323

Preface

Welcome to the *JavaScript Cloud Native Development Cookbook*. This cookbook is packed full of recipes to help you along your cloud-native journey. It is intended to be a companion to another of my books, *Cloud Native Development Patterns and Best Practices*. I have personally found delivering cloud-native solutions to be, by far, the most fun and satisfying development practice. This is because cloud-native is more than just optimizing for the cloud. It is an entirely different way of thinking and reasoning about software systems.

In a nutshell, cloud-native is lean and autonomous. Powered by disposable infrastructure, leveraging fully managed cloud services and embracing disposable architecture, cloud-native empowers everyday, self-sufficient, full-stack teams to rapidly and continuously experiment with innovations, while simultaneously building global-scale systems with much less effort than ever before. Following this serverless-first approach allows teams to move fast, but this rapid pace also opens the opportunity for honest human error to wreak havoc across the system. To guard against this, cloud-native systems are composed of autonomous services, which creates bulkheads between the services to reduce the blast radius during a disruption.

In this cookbook, you will learn how to build autonomous services by eliminating all synchronous inter-service communication. You will turn the database inside out and ultimately turn the cloud into the database by implementing the event sourcing and CQRS patterns with event streaming and materialized views. Your team will build confidence in its ability to deliver because asynchronous inter-service communication and data replication remove the downstream and upstream dependencies that make systems brittle. You will also learn how to continuously deploy, test, observe, optimize, and secure your autonomous services across multiple regions.

To get the most out of this book, be prepared with an open mind to discover why cloud-native is different. Cloud-native forces us to rewire how we reason about systems. It tests all our preconceived notions of software architecture. So, be prepared to have a lot of fun building cloud-native systems.

Who this book is for

This book is intended to help create self-sufficient, full-stack, cloud-native development teams. Some cloud experience is helpful, but not required. Basic knowledge of the JavaScript language is assumed. The book serves as a reference for experienced cloud-native developers and as a quick start for entry-level cloud-native developers. Most of all, this book is for anyone who is ready to rewire their engineering brain for cloud-native development.

What this book covers

`Chapter 1`, *Getting Started with Cloud-Native*, showcases how the ease of defining and deploying serverless, cloud-native resources, such as functions, streams, and databases, empowers self-sufficient, full-stack teams to continuously deliver with confidence.

`Chapter 2`, *Applying the Event Sourcing and CQRS Patterns*, demonstrates how to use these patterns to create fully autonomous services, by eliminating inter-service synchronous communication through the use of event streaming and materialized views.

`Chapter 3`, *Implementing Autonomous Services*, explores the boundary and control patterns for creating autonomous services, such as Backend for Frontend, External Service Gateway, and Event Orchestration.

`Chapter 4`, *Leveraging the Edge of the Cloud*, provides concrete examples of using a cloud provider's content delivery network to increase the performance and security of autonomous services.

`Chapter 5`, *Securing Cloud-Native Systems*, looks at leveraging the shared responsibility model of cloud-native security so that you can focus your efforts on securing the domain-specific layers of your cloud-native systems.

`Chapter 6`, *Building a Continuous Deployment Pipeline*, showcases techniques, such as task branch workflow, transitive end-to-end testing, and feature flags, that help teams continuously deploy changes to production with confidence by shifting deployment and testing all the way to the left, controlling batch sizes, and decoupling deployment from release.

`Chapter 7`, *Optimizing Observability*, demonstrates how to instill team confidence by continuously testing in production to assert the health of a cloud-native system and by placing our focus on the mean time to recovery.

`Chapter 8`, *Designing for Failure*, deals with techniques, such as backpressure, idempotency, and the Stream Circuit Breaker pattern, for increasing the robustness and resilience of autonomous services.

`Chapter 9`, *Optimizing Performance*, explores techniques, such as asynchronous non-blocking IO, session consistency, and function tuning, for boosting the responsiveness of autonomous services.

`Chapter 10`, *Deploying to Multiple Regions*, demonstrates how to deploy global autonomous services that maximize availability and minimize latency by creating fully replicated, active-active deployments across regions and implementing regional failover.

`Chapter 11`, *Welcoming Polycloud*, explores the freedom provided by choosing the right cloud provider one service at a time while maintaining a consistent development pipeline experience.

To get the most out of this book

To follow along with the recipes in this cookbook, you will need to configure your development environment according to these steps:

1. Install Node Version Manager (`https://github.com/creationix/nvm` or `https://github.com/coreybutler/nvm-windows`)
2. Install Node.js with `nvm install 8`
3. Install the Serverless Framework with `npm install serverless -g`
4. Create a `MY_STAGE` environment variable with `export MY_STAGE=<your-name>`
5. Create an AWS account (`https://aws.amazon.com/free`) and configure your credentials for the Serverless Framework (`https://serverless.com/framework/docs/providers/aws/guide/credentials`)

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781788470414_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first thing to note is that we must define `runtime: nodejs8.10` in the `serverless.yml` file."

A block of code is set as follows:

```
module.exports.hello = (event, context, callback) => {
  console.log('event: %j', event);
  console.log('env: %j', process.env);
  callback(null, 'success');
};
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
service: cncb-create-function
...
functions:
  hello:
    handler: handler.hello
```

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

```
$ npm run dp:lcl -- -s $MY_STAGE
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **Monitors** | **New Monitor** | **Event**"



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

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

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

Getting ready

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

How to do it…

This section contains the steps required to follow the recipe.

How it works…

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

There's more…

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

See also

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

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, please email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Getting Started with Cloud-Native

In this chapter, the following recipes will be covered:

- Creating a stack
- Creating a function and working with metrics and logs
- Creating an event stream and publishing an event
- Creating a stream processor
- Creating an API Gateway
- Deploying a single-page application

Introduction

Cloud-native is lean. Companies today must continuously experiment with new product ideas so that they can adapt to changing market demands; otherwise, they risk falling behind their competition. To operate at this pace, they must leverage fully managed cloud services and fully-automated deployments to minimize time to market, mitigate operating risks, and empower self-sufficient, full-stack teams to accomplish far more with much less effort.

The recipes in this cookbook demonstrate how to use fully managed, serverless cloud services to develop and deploy lean and autonomous services. This chapter contains bare-boned recipes with no clutter in order to focus on the core aspects of deploying cloud-native components and to establish a solid foundation for the remainder of this cookbook.

Creating a stack

Each autonomous cloud-native service and all its resources are provisioned as a cohesive and self-contained group called a **stack**. On AWS, these are **CloudFormation** stacks. In this recipe, we will use the Serverless Framework to create and manage a bare-bones stack to highlight the steps involved in deploying a cloud-native service.

Getting ready

Before starting this recipe, you will need to follow the instructions in the *Preface* for configuring your development environment with Node.js, the Serverless Framework, and AWS account credentials.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/cr
eate-stack --path cncb-create-stack
```

2. Navigate to the `cncb-create-stack` directory with `cd cncb-create-stack`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-create-stack

provider:
  name: aws
```

4. Review the file named `package.json` with the following content:

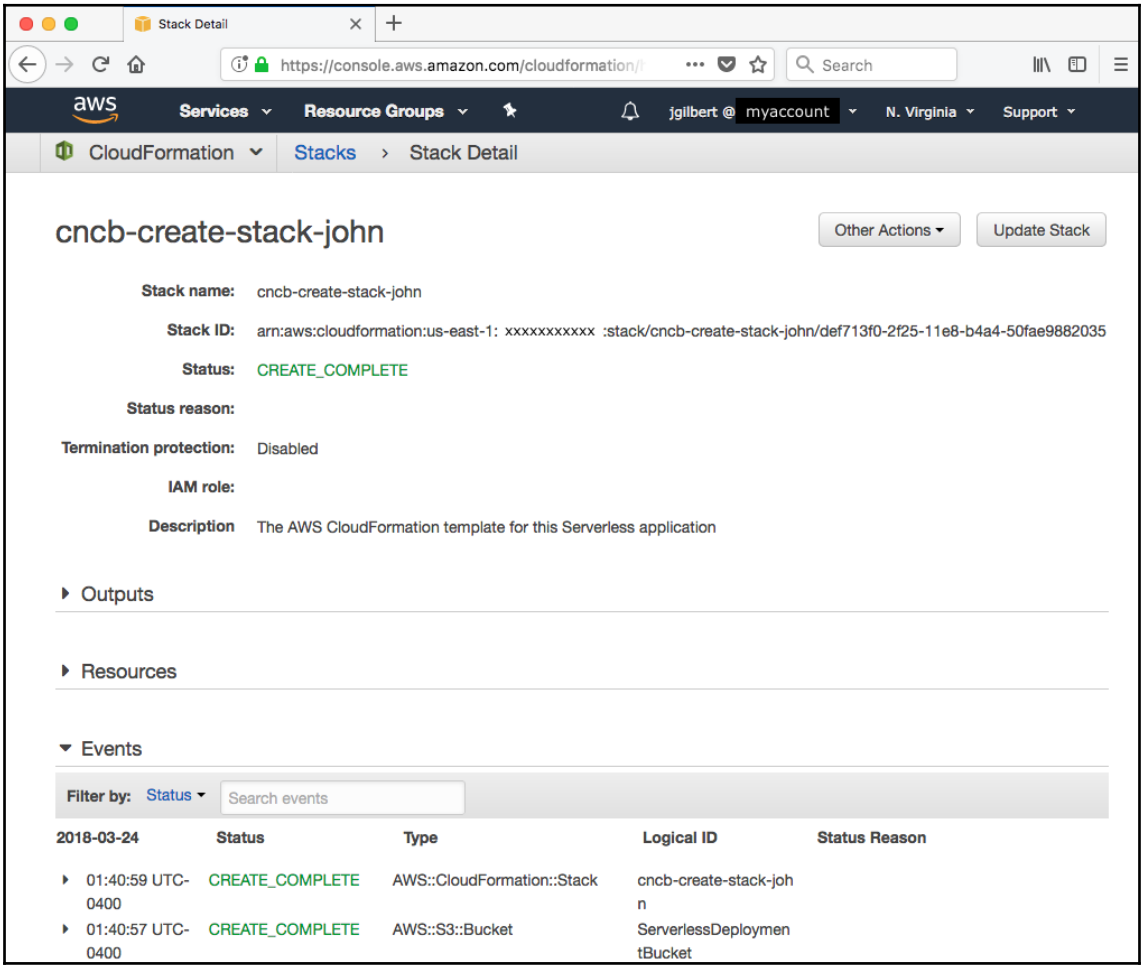
```
{
  "name": "cncb-create-stack",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "test": "sls package -r us-east-1 -s test",
    "dp:lcl": "sls deploy -r us-east-1",
    "rm:lcl": "sls remove -r us-east-1"
  },
  "devDependencies": {
    "serverless": "1.26.0"
```

```
  }
}
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-create-stack@1.0.0 dp:lcl <path-to-your-workspace>/cncb-create-
stack
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
Serverless: Creating Stack...
Serverless: Checking Stack create progress...
.....
Serverless: Stack create finished...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Validating template...
Serverless: Updating Stack...
Service Information
service: cncb-create-stack
stage: john
region: us-east-1
stack: cncb-create-stack-john
api keys:
 None
endpoints:
 None
functions:
 None
```

9. Review the stack in the AWS Console:



10. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The **Serverless Framework** (**SLS**) (`https://serverless.com/framework/docs`) is my tool of choice for deploying cloud resources, regardless of whether or not I am deploying serverless resources, such as functions. SLS is essentially an abstraction layer on top of **infrastructure as code** tools, such as AWS CloudFormation, with extensibility features such as plugins and dynamic variables. We will use SLS in all of our recipes. Each recipe starts by using the SLS feature to `create` a new project by cloning a `template`. You will ultimately want to create your own templates for jump-starting your own projects.

This first project is as bare bones as we can get. It essentially creates an empty CloudFormation stack. In the `serverless.yml` file, we define the `service` name and the `provider`. The `service` name will be combined with the `stage`, which we will discuss shortly, to create a unique stack name within your account and `region`. I have prefixed all the stacks in our recipes with `cncb` to make it easy to filter for these stacks in the AWS Console if you are using a shared account, such as your development or sandbox account at work.

Our next most important tool is **Node Package Manager** (**NPM**) (`https://docs.npmjs.com/`). We will not be packaging any Node modules (also known as libraries), but we will be leveraging NPM's dependency management and scripting features. In the `package.json` file, we declared a development dependency on the Serverless Framework and three custom scripts to test, deploy, and remove our stack. The first command we execute is `npm install`, which will install all the declared dependencies into the project's `node_modules` directory.

Next, we execute the `npm test` script. This is one of several standard scripts for which NPM provides a shortcut alias. We have defined the `test` script to invoke the `sls package` command to assert that everything is configured properly and help us see what is going on under the covers. This command processes the `serverless.yml` file and generates a CloudFormation template in the `.serverless` directory. One of the advantages of the Serverless Framework is that it embodies best practices and uses a **configuration by exception** approach to take a small amount of declaration in the `serverless.yml` files and expand it into a much more verbose CloudFormation template.

Now, we are ready to deploy the stack. As developers, we need to be able to deploy a stack and work on it in isolation from other developers and other environments, such as production. To support this requirement, SLS uses the concept of a **stage**. Stage (`-s $MY_STAGE`) and region (`-r us-east-1`) are two required command-line options when invoking an SLS command. A stack is deployed into a specific region and the stage is used as a prefix in the stack name to make it unique within an account and region. Using this feature, each developer can *deploy* (`dp`) what I refer to as a *local* (`lcl`) stack with their name as the stage with `npm run dp:lcl -- -s $MY_STAGE`. In the examples, I use my name for the stage. We declared the `$MY_STAGE` environment variable in the *Getting ready* section. The double dash (`--`) is NPM's way of letting us pass additional options to a custom script. In `Chapter 6`, *Building a Continuous Deployment Pipeline*, we will discuss deploying stacks to shared environments, such as **staging** and **production**.

CloudFormation has a limit regarding the template body size in a request to the API. Typical templates easily surpass this limit and must be uploaded to S3 instead. The Serverless Framework handles this complexity for us. In the `.serverless` directory, you will notice that there is a `cloudformation-template-create-stack.json` file that declares a `ServerlessDeploymentBucket`. In the `sls deploy` output, you can see that SLS uses this template first and then it uploads the `cloudformation-template-update-stack.json` file to the bucket and updates the stack. It's nice to have this problem already solved for us because it is typical to learn about this limit the hard way.

At first glance, creating an empty stack may seem like a silly idea, but in practice it is actually quite useful. In a sense, you can think of CloudFormation as a CRUD tool for cloud resources. CloudFormation keeps track of the state of all the resources in a stack. It knows when a resource is new to a stack and must be created, when a resource has been removed from a stack and must be deleted, and when a resource has changed and must be updated. It also manages the dependencies and ordering between resources. Furthermore, when an update to a stack fails, it rolls back all the changes.

Unfortunately, when deploying a large number of changes, these rollbacks can be very time-consuming and painful when the error is in one of the last resources to be changed. Therefore, it is best to make changes to a stack in small increments. In `Chapter 6`, *Building a Continuous Deployment Pipeline*, we will discuss the practices of small batch sizes, task branch workflow, and decoupling deployment from release. For now, if you are creating a new service from a proven template, then initialize the new project and deploy the stack with all the template defaults all the way to production with your first pull request. Then, create a new branch for each incremental change. However, if you are working on an experimental service with no proven starting point, then an empty stack is perfectly reasonable for your first deployment to production.

In your daily development routine, it is important to clean up your local stacks when you have completed work on a task or story. The cost of a development account can creep surprisingly high when orphaned stacks accumulate and are rarely removed. The `npm run rm:lcl -- -s $MY_STAGE` script serves this purpose.

Creating a function and working with metrics and logs

Function-as-a-Service is the cornerstone of cloud-native architecture. Functions enable self-sufficient, full-stack teams to focus on delivering lean business solutions without being weighed down by the complexity of running cloud infrastructure. There are no servers to manage, and functions implicitly scale to meet demand. They are integrated with other value-added cloud services, such as streams, databases, API gateways, logging, and metrics, to further accelerate development. Functions are disposable architecture, which empower teams to experiment with different solutions. This recipe demonstrates how straightforward it is to deploy a function.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/cr
eate-function --path cncb-create-function
```

2. Navigate to the `cncb-create-function` directory with `cd cncb-create-function`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-create-function

provider:
  name: aws
  runtime: nodejs8.10
  environment:
    V1: value1

functions:
  hello:
    handler: handler.hello
```

4. Review the file named `handler.js` with the following content:

```
module.exports.hello = (event, context, callback) => {
  console.log('event: %j', event);
  console.log('context: %j', context);
  console.log('env: %j', process.env);
  callback(null, 'success');
};
```

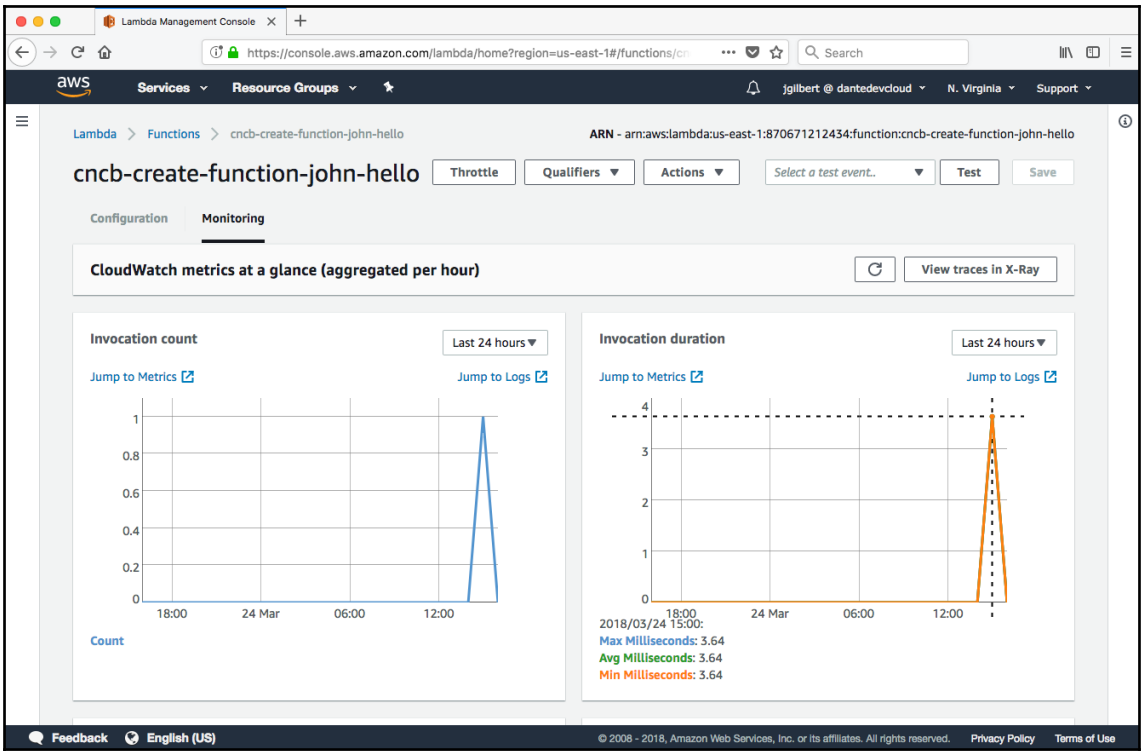
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-create-function@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
create-function
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Creating Stack...
Serverless: Checking Stack create progress...
.....
Serverless: Stack create finished...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service .zip file to S3 (881 B)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.................
Serverless: Stack update finished...
Service Information
service: cncb-create-function
stage: john
region: us-east-1
stack: cncb-create-function-john
api keys:
 None
endpoints:
 None
functions:
 hello: cncb-create-function-john-hello
```

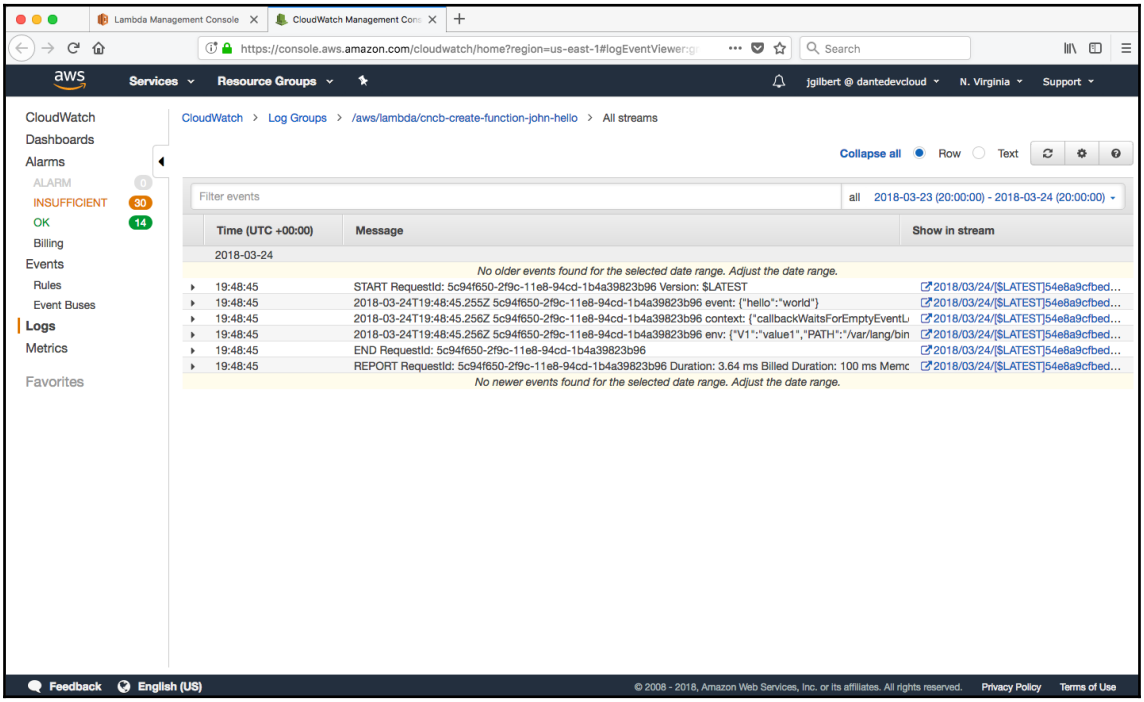
9. Review the stack and function in the AWS Console.
10. Invoke the function with the following command:

```
$ sls invoke -r us-east-1 -f hello -s $MY_STAGE -d '{"hello":"world"}'
"success"
```

11. Review the function metrics in the AWS Console:



12. Review the function logs in the AWS Console:



13. Take a look at the logs locally:

```
$ sls logs -f hello -r us-east-1 -s $MY_STAGE
START RequestId: ... Version: $LATEST
2018-03-24 15:48:45 ... event: {"hello":"world"}
2018-03-24 15:48:45 ... context: {"functionName":"cncb-create-function-
john-hello","memoryLimitInMB":"1024", ...}
2018-03-24 15:48:45 ... env:
{"V1":"value1","TZ":":UTC","AWS_REGION":"us-east-1",
"AWS_ACCESS_KEY_ID":"...", ...}
END RequestId: ...
REPORT ... Duration: 3.64 ms Billed Duration: 100 ms ... Max Memory
Used: 20 MB
```

14. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The Serverless Framework handles the heavy lifting, which allows us to focus on writing the actual function code. The first thing to note is that we must define the `runtime: nodejs8.10` in the `serverless.yml` file. Next, we define a function in the `functions` section with a name and a handler. All other settings have defaulted, following the **configuration by exception** approach. When you look at the generated CloudFormation template, you will see that over 100 lines were generated from just a handful of lines declared in the `serverless.yml` file. A large portion of the generated template is dedicated to defining boilerplate security policies. Dig into the `.serverless/cloudformation-template-update-stack.json` file to see the details.

We also define `environment` variables in the serverless.yml. This allows the functions to be parameterized per deployment stage. We will cover this in more detail in `Chapter 6`, *Building a Continuous Deployment Pipeline*. This also allows settings, such as the debug level, to be temporarily tweaked without redeploying the function.

When we deploy the project, the Serverless Framework packages the function along with its runtime dependencies, as specified in the `package.json` file, into a ZIP file. Then, it uploads the ZIP file to the `ServerlessDeploymentBucket` so that it can be accessed by CloudFormation. The output of the deployment command shows when this is happening. You can look at the content of the ZIP file in the `.serverless` directory or download it from the deployment bucket. We will cover advanced packaging options in `Chapter 9`, *Optimizing Performance*.

The signature of an AWS Lambda function is straightforward. It must export a function that accepts three arguments: an event object, a context object, and a callback function. Our first function will just log the event, content, and the environment variables so that we can peer into the execution environment a little bit. Finally, we must invoke the callback. It is a standard JavaScript callback. We pass an error to the first argument or the successful result to the second argument.

Logging is an important standard feature of **Function as a Service** (**FaaS**). Due to the ephemeral nature of cloud resources, logging in the cloud can be tedious, to put it lightly. In AWS Lambda, console logging is performed asynchronously and recorded in CloudWatch logs. It's a fully-managed logging solution built right in. Take the time to look at the details in the log statements that this function writes. The environment variables are particularly interesting. For example, we can see that each invocation of a function gets a new temporary access key.

Functions also provide a standard set of metrics out-of-the-box, such as invocation count, duration, errors, throttling, and so forth. We will cover this in detail in `Chapter 7`, *Optimizing Observability*.

Creating an event stream and publishing an event

Cloud-native services are autonomous. Each service is completely self-sufficient and runs in isolation to minimize the blast radius when any given service experiences a failure. To achieve this isolation, *bulkheads* must be established between the services. **Event streaming** is one mechanism that is used to create these bulkheads. Autonomous cloud-native services perform all inter-service communication asynchronously via streams to decouple upstream services from downstream services. In `Chapter 2`, *Applying The Event Sourcing and CQRS Patterns*, we will dive deeper into how we create bounded, isolated, and autonomous cloud-native services. This recipe creates the event stream that we will use throughout this cookbook and provides a function for publishing events to the stream.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/ev
ent-stream --path cncb-event-stream
```

2. Navigate to the `cncb-event-stream` directory with `cd cncb-event-stream`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-event-stream

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    - Effect: Allow
      Action:
        - kinesis:PutRecord
      Resource:
        Fn::GetAtt: [ Stream, Arn ]
```

```
functions:
  publish:
    handler: handler.publish
    environment:
      STREAM_NAME:
        Ref: Stream

resources:
  Resources:
    Stream:
      Type: AWS::Kinesis::Stream
      Properties:
        Name: ${opt:stage}-${self:service}-s1
        RetentionPeriodHours: 24
        ShardCount: 1
  Outputs:
    streamName:
      Value:
        Ref: Stream
    streamArn:
      Value:
        Fn::GetAtt: [ Stream, Arn ]
```

4. Review the file named `handler.js` with the following content:

```
const aws = require('aws-sdk');
const uuid = require('uuid');

module.exports.publish = (event, context, callback) => {
  const e = {
    id: uuid.v1(),
    partitionKey: event.partitionKey || uuid.v4(),
    timestamp: Date.now(),
    tags: {
      region: process.env.AWS_REGION,
    },
    ...event,
  }

  const params = {
    StreamName: process.env.STREAM_NAME,
    PartitionKey: e.partitionKey,
    Data: Buffer.from(JSON.stringify(e)),
  };

  const kinesis = new aws.Kinesis();
```

```
  kinesis.putRecord(params, callback);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-create-stream@1.0.0 dp:lcl <path-to-your-workspace>/cncb-create-
stream
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
Service Information
service: cncb-event-stream
stage: john
region: us-east-1
stack: cncb-event-stream-john
...
functions:
 publish: cncb-event-stream-john-publish

Stack Outputs
PublishLambdaFunctionQualifiedArn: arn:aws:lambda:us-
east-1:999999999999:function:cncb-event-stream-john-publish:3
streamArn: arn:aws:kinesis:us-east-1:999999999999:stream/john-cncb-
event-stream-s1
streamName: john-cncb-event-stream-s1
...
```

9. Review the stack, stream, and function in the AWS Console.
10. Invoke the function with the following command:

```
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created"}'
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49582906351415672136958521359460531895314381358803976194"
}
```

11. Take a look at the logs:

```
$ sls logs -f publish -r us-east-1 -s $MY_STAGE
START ...
2018-03-24 23:20:46 ... event: {"type":"thing-created"}
2018-03-24 23:20:46 ... event:
{
  "type":"thing-created",
  "id":"81fd8920-2fdb-11e8-b749-0d2c43ec73d0",
  "partitionKey":"6f4f9a38-61f7-41c9-a3ad-b8c16e42db7c",
  "timestamp":1521948046003,
  "tags":{
    "region":"us-east-1"
  }
}
2018-03-24 23:20:46 ... params: {"StreamName":"john-cncb-event-stream-
s1","PartitionKey":"6f4f9a38-61f7-41c9-a3ad-
b8c16e42db7c","Data":{"type":"Buffer","data":[...]}}
END ...
REPORT ... Duration: 153.47 ms    Billed Duration: 200 ms ... Max
Memory Used: 39 MB
```

12. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.



This stack is a prerequisite for other recipes, as indicated in the *Getting ready* section of each recipe. If you are continuing with related recipes, then you can leave this stack running until you complete the related recipes. However, the stream in this stack is not included in the AWS free tier, so you may want to go ahead and remove this stack and recreate it when needed.

How it works...

The `resources` section of the `serverless.yml` file is used to create cloud resources that are used by services. These resources are defined using standard AWS CloudFormation resource types. In this recipe, we are creating an AWS *Kinesis* stream. We give the stream a name, define the retention period, and specify the number of shards. The Serverless Framework provides a robust mechanism for dynamically replacing variables.

Here, we use the `${opt:stage}` option passed in on the command line and the `${self:service}` name defined in the `serverless.yml` file to create a unique stream name. The standard retention period is 24 hours and the maximum is seven days. For our recipes, one shard will be more than sufficient. We will discuss shards shortly and again in `Chapter 7`, *Optimizing Observability*, and `Chapter 9`, *Optimizing Performance*.

The `Outputs` section of the `serverless.yml` file is where we define values, such as generated IDs and names, that we want to use outside of the stack. We output the **Amazon Resource Names** (**ARNs**) `streamName` and `streamArn` so that we can reference them with Serverless Framework variables in other projects. These values are also displayed on the Terminal when a deployment is complete.

The `publish` function defined in the `serverless.yml` file is used to demonstrate how to publish an event to the stream. We are passing the `STREAM_NAME` to the function as an environment variable. In the `iamRoleStatements` section, we give the function `kinesis:PutRecord` permission to allow it to publish events to this specific stream.

The function `handler.js` file has runtime dependencies on two external libraries—`aws-sdk` and `uuid`. The Serverless Framework will automatically include the runtime dependencies, as defined in the `package.json` file. Take a look inside the generated `.serverless/cncb-event-stream.zip` file. The `aws-sdk` is a special case. It is already available in the AWS Lambda `Node.js` runtime, and therefore is not included. This is important because `aws-sdk` is a large library and the ZIP file size impacts cold start times. We will discuss this in more detail in `Chapter 9`, *Optimizing Performance*.

The `publish` function expects to receive an event object as input, such as `{"type":"thing-created"}`. We then adorn the event with additional information to conform to our standard event format, which we will discuss shortly. Finally, the function creates the required `params` object and then calls `kinesis.putRecord` from the `aws-sdk`. We will be using this function in this and other recipes to simulate event traffic.

All events in our cloud-native systems will conform to the following **Event structure** to allow for consistent handling across all services. Additional fields are event-type-specific:

```
interface Event {
    id: string;
    type: string;
    timestamp: number;
    partitionKey: string;
    tags: { [key: string]: string };
}
```

- `type` describes the event, such as `thing-created`
- `timestamp` is an epoch value, as returned from `Date.now()`
- `id` should be a V1 UUID, which is time-based
- `partitionKey` should be a V4 UUID, which is random number-based
- `tags` is a hashmap of useful data values that are leveraged for content-based routing and aggregating event metrics

It is important to use a *V4 UUID* for the `partitionKey` to avoid hot shards and maximize concurrency. If a *V1 UUID* were used, then all events produced at the same time would go to the same shard. The `partitionKey` will typically be the ID of the domain entity that produced the event, which should use a V4 UUID for the same reason. This has the added benefit of ensuring that all events for the same domain entity are processed through the same shard in the order received.

Creating a stream processor

Stream processors do most of the heavy lifting in cloud-native services. Autonomous cloud-native services perform all inter-service communication asynchronously via event streaming to decouple upstream services from **downstream services**. **Upstream services** publish events to a stream, with no knowledge of the specific downstream services that will eventually consume the events. Downstream services deploy stream-processing functions to consume events of interest. Stream processors will be covered extensively throughout this cookbook. This recipe demonstrates how to create a function that listens for events from an **AWS Kinesis** stream and provides a quick introduction to using the functional reactive programming paradigm for implementing stream processing.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/cr
eate-stream-processor --path cncb-create-stream-processor
```

2. Navigate to the `cncb-create-stream-processor` directory with `cd cncb-create-stream-processor`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-create-stream-processor

provider:
  name: aws
  runtime: nodejs8.10
functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          batchSize: 100
          startingPosition: TRIM_HORIZON
```

4. Review the file named `handler.js` with the following content:

```
const _ = require('highland');

module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .tap(printEvent)
    .filter(forThingCreated)
    .collect()
    .tap(printCount)
    .toCallback(cb);
};

const recordToEvent = r => JSON.parse(Buffer.from(r.kinesis.data,
'base64'));
const forThingCreated = e => e.type === 'thing-created';

const printEvent = e => console.log('event: %j', e);
const printCount = events => console.log('count: %d', events.length);
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-create-stream-processor@1.0.0 dp:lcl <path-to-your-
workspace>/cncb-create-stream-processor
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
Service Information
service: cncb-create-stream-processor
stage: john
region: us-east-1
stack: cncb-create-stream-processor-john
...
functions:
  listener: cncb-create-stream-processor-john-listener
```

9. Review the stack and function in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created"}'
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49582906351415672136958521360120605392824155736450793474"
}
```

11. Take a look at the logs from the original Terminal:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
START ...
2018-03-25 00:16:32 ... event:
{
  "type":"thing-created",
  "id":"81fd8920-2fdb-11e8-b749-0d2c43ec73d0",
  "partitionKey":"6f4f9a38-61f7-41c9-a3ad-b8c16e42db7c",
  "timestamp":1521948046003,
  "tags":{
    "region":"us-east-1"
```

```
  }
}
2018-03-25 00:16:32 ... event:
{
  "type":"thing-created",
  "id":"c6f60550-2fdd-11e8-b749-0d2c43ec73d0",
  ...
}
2018-03-25 00:16:32 ... count: 2
END ...
REPORT ... Duration: 7.73 ms    Billed Duration: 100 ms ... Max Memory
Used: 22 MB

START ...
2018-03-25 00:22:22 ... event:
{
  "type":"thing-created",
  "id":"1c2b5150-2fe4-11e8-b749-0d2c43ec73d0",
  ...
}
2018-03-25 00:22:22 ... count: 1
END ...
REPORT ... Duration: 1.34 ms    Billed Duration: 100 ms ... Max Memory
Used: 22 MB
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Stream processors listen for data from a streaming service such as **Kinesis** or **DynamoDB Streams**. Deploying a stream processor is completely declarative. We configure a function with the `stream` event type and the pertinent settings, such as the `type`, `arn`, `batchSize`, and `startingPosition`. The `arn` is set dynamically using a CloudFormation variable, `${cf:cncb-event-stream-${opt:stage}.streamArn}`, that references the output value of the `cnbc-event-stream` stack.



Streams are the only resources that are shared between autonomous cloud-native services.

We will discuss batch size and starting position in detail in both `Chapter 8`, *Designing for Failure*, and `Chapter 9`, *Optimizing Performance*. For now, you may have noticed that the new stream processor logged all the events that were published to the stream in the last 24 hours. This is because the `startingPosition` is set to `TRIM_HORIZON`. If it was set to `LATEST`, then it would only receive events that were published after the function was created.

Stream processing is a perfect match for functional reactive programming with Node.js streams. The terminology can be a little confusing because the word *stream* is overloaded. I like to think of streams as either *macro* or *micro*. For example, Kinesis is the *macro* stream and the code in our stream processor function is the *micro* stream. My favorite library for implementing the *micro* stream is **Highland.js** (`https://highlandjs.org`). A popular alternative is **RxJS** (`https://rxjs-dev.firebaseapp.com`). As you can see in this recipe, functional reactive programming is very descriptive and readable. One of the reasons for this is that there are no loops. If you try to implement a stream processor with *imperative programming*, you will find that it quickly gets very messy. You also lose backpressure, which we will discuss in `Chapter 8`, *Designing for Failure*.

The code in the `listener` function creates a pipeline of steps that the data from the Kinesis stream will ultimately flow through. The first step, `_(event.Records)`, converts the array of Kinesis records into a Highland.js stream object that will allow each element in the array to be pulled through the stream in turn as the downstream steps are ready to receive the next element. The `.map(recordToEvent)` step decodes the Base64 encoded data from the Kinesis record and parses the JSON into an event object. The next step, `.tap(printEvent)`, simply logs the event so that we can see what is happening in the recipe.

Kinesis and event streaming, in general, is a member of the high performance, dumb-pipe-smart-endpoints generation of messaging middleware. This means that Kinesis, the dumb pipe, does not waste its processing power on filtering data for the endpoints. Instead, all that logic is spread out across the processing power of the smart endpoints. Our stream processor function is the smart endpoint. To that end, the `.filter(forThingCreated)` step is responsible for filtering out the events that the processor is not interested in. All the remaining steps can assume that they are receiving the expected event types.

Our bare-boned stream processor needs something somewhat interesting but simple to do. So, we count and print the number of `thing-created` events in the batch. We have filtered out all other event types, so the `.collect()` step collects all the remaining events into an array. Then, the `.tap(printCount)` step logs the length of the array. Finally, the `.toCallback(cb)` step will invoke the callback function once all the data in the batch has been processed. At this point, the Kinesis checkpoint is advanced and the next batch of events is processed. We will cover error handling and how it relates to batches and checkpoints in `Chapter 8`, *Designing for Failure*.

Creating an API Gateway

An API Gateway is an essential element of cloud-native architecture. It provides a secure and performant perimeter at the boundaries of our cloud-native systems. The boundaries are where the system interacts with everything that is external to the system, including humans and other systems. We will leverage an API Gateway in the recipes that create boundary components such as a **Backend For Frontend** (**BFF**) or an External Service Gateway. This recipe demonstrates how straightforward it is to deploy an API Gateway.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/cr
eate-api-gateway --path cncb-create-api-gateway
```

2. Navigate to the `cncb-create-api-gateway` directory with `cd cncb-create-api-gateway`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-create-api-gateway

provider:
  name: aws
  runtime: nodejs8.10

functions:
  hello:
    handler: handler.hello
    events:
      - http:
```

```
          path: hello
          method: get
          cors: true
```

4. Review the file named `handler.js` with the following content:

```
module.exports.hello = (event, context, callback) => {
  console.log('event: %j', event);

  const response = {
    statusCode: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
    body: JSON.stringify({
      message: 'JavaScript Cloud Native Development Cookbook! Your
function executed successfully!',
      input: event,
    }),
  };

  callback(null, response);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-create-api-gateway@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
create-api-gateway
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
.....
Serverless: Stack update finished...
Service Information
service: cncb-create-api-gateway
stage: john
region: us-east-1
stack: cncb-create-api-gateway-john
api keys:
  None
endpoints:
  GET -
```

```
https://k1ro5oasm6.execute-api.us-east-1.amazonaws.com/john/hello
functions:
  hello: cncb-create-api-gateway-john-hello
```

9. Review the stack, API, and function in the AWS Console.
10. Invoke the `endpoint` shown in the stack output in the following commands:

```
$ curl -v
https://k1ro5oasm6.execute-api.us-east-1.amazonaws.com/john/hello |
json_pp

{
   "input" : {
      "body" : null,
      "pathParameters" : null,
      "requestContext" : { ... },
      "resource" : "/hello",
      "headers" : { ... },
      "queryStringParameters" : null,
      "httpMethod" : "GET",
      "stageVariables" : null,
      "isBase64Encoded" : false,
      "path" : "/hello"
   },
   "message" : "JavaScript Cloud Native Development Cookbook! Your
function executed successfully!"
}
```

11. Take a look at the logs:

```
$ sls logs -f hello -r us-east-1 -s $MY_STAGE

START ...
2018-03-25 01:04:47 ... event:
{"resource":"/hello","path":"/hello","httpMethod":"GET","headers":{ ...
},"requestContext":{ ... },"body":null,"isBase64Encoded":false}
END
REPORT ... Duration: 2.82 ms    Billed Duration: 100 ms ... Max Memory
Used: 20 MB
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Creating an API Gateway is completely declarative. We just configure a function with the `http` event type and the pertinent settings, such as the `path` and `method`. All other settings have defaulted following the **configuration by exception** approach. When you look at the generated `.serverless/cloudformation-template-update-stack.json` file, you will see that over 100 lines were generated from just a handful of lines declared in the `serverless.yml` file. The API name is calculated based on the combination of the service name declared at the top of the `serverless.yml` file and the specified stage. There is a one-to-one mapping between a serverless project and an API Gateway. All the functions in the project declared with an `http` event are included in the API.

The signature of the function is the same as all others; however, the contents of the event and the expected response format are specific to the API Gateway service. The `event` contains the full contents of the HTTP request including the path, parameters, header, body, and more. The `response` requires a `statusCode` and options headers and body. The body must be a string, and the header must be an object. I declared the function with the `cors: true` setting so that the recipe could include a legitimate set of response headers. We will cover security in detail in `Chapter 5`, *Securing Cloud-Native Systems*. For now, know that security features such as SSL, throttling, and DDoS protection are default features of the AWS API Gateway.

The `endpoint` for the API Gateway is declared as a stack output and displayed after the stack is deployed. We will see ways to customize the endpoint in `Chapter 4`, *Leveraging the Edge of the Cloud*, and in `Chapter 10`, *Deploying to Multiple Regions*. Once you invoke the service, you will be able to see the details of the inputs and outputs, both in the HTTP response as it was coded and then in the function's logs. Take a look at the API Gateway in the AWS Console as well. However, the goal of automation and the *Serverless Framework* is to eliminate the need to make changes in the console. I looked at the API in the console while writing this book, but other than that I can't remember the last time I actually needed to go into the API Gateway console.

Deploying a single-page application

The cloud-native light bulb first turned on in my head when I realized I could deploy a single page application, such as Angular, to an S3 bucket and serve it up globally with no need for servers and load balancers whatsoever. This was my first cloud-native W*ow!* moment. It was the moment when I began to understand that cloud-native plays by an entirely different set of rules. The combination of S3 and a JavaScript-based UI delivers a web presentation tier with virtually limitless scalability, virtually no cost, and essentially no operation headaches. This recipe demonstrates how straightforward it is to deploy a single-page application.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch1/de
ploy-spa --path cncb-deploy-spa
```

2. Navigate to the `cncb-deploy-spa` directory with `cd cncb-deploy-spa`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-deploy-spa

provider:
  name: aws

plugins:
  - serverless-spa-deploy

custom:
  spa:
    files:
      - source: ./build
        globs: '**/*'
        headers:
          CacheControl: max-age=300

resources:
  Resources:
    WebsiteBucket:
      Type: AWS::S3::Bucket
      Properties:
        AccessControl: PublicRead
```

```
        WebsiteConfiguration:
          IndexDocument: index.html
          ErrorDocument: index.html

  Outputs:
    WebsiteBucketName:
      Value:
        Ref: WebsiteBucket
    WebsiteURL:
      Value:
        Fn::GetAtt: [ WebsiteBucket, WebsiteURL ]
```

4. Review the file named `package.json` with the following content:

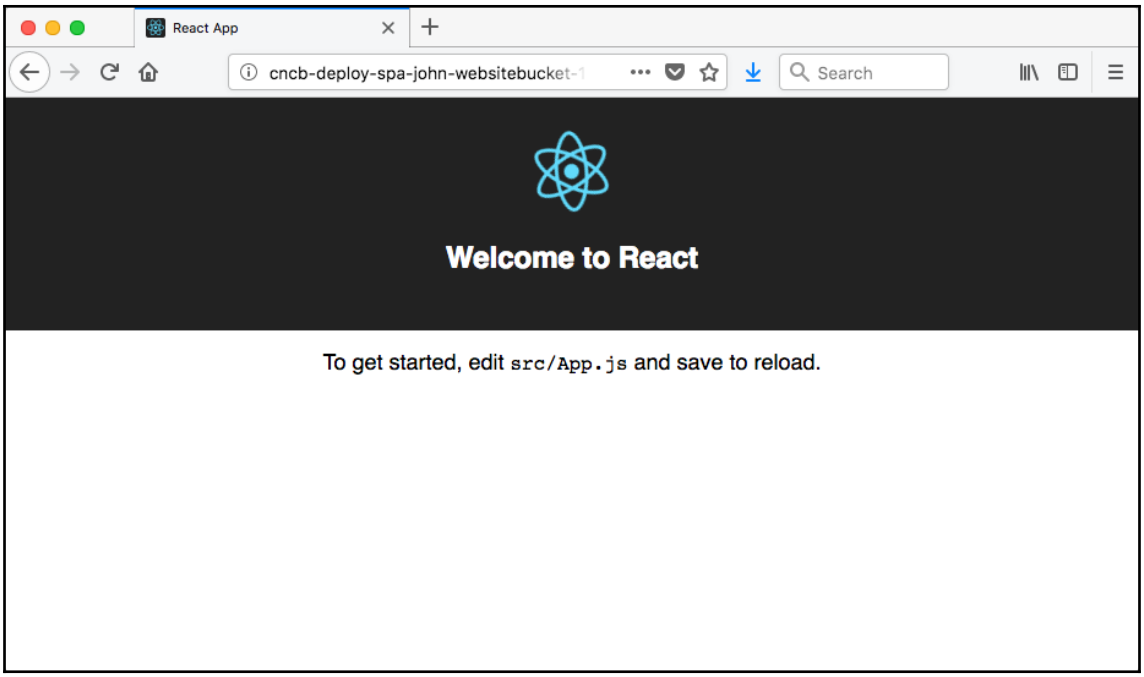
```
{
  "name": "cncb-deploy-spa",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "sls package -r us-east-1 -s test",
    "dp:lcl": "sls deploy -v -r us-east-1",
    "rm:lcl": "sls remove -r us-east-1"
  },
  "dependencies": {
    "react": "16.2.0",
    "react-dom": "16.2.0"
  },
  "devDependencies": {
    "react-scripts": "1.1.1",
    "serverless": "1.26.0",
    "serverless-spa-deploy": "^1.0.0"
  }
}
```

5. Install the dependencies with `npm install`.
6. Run the app locally with `npm start`.
7. Run the tests with `npm test`.

8. Review the contents generated in the `.serverless` directory.
9. Build the app with `npm run build`.
10. Review the contents generated in the `build` directory.
11. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-deploy-spa@1.0.0 dp:lcl <path-to-your-workspace>/cncb-deploy-spa
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
Stack Outputs
WebsiteBucketName: cncb-deploy-spa-john-websitebucket-1s8hgqtof7la7
WebsiteURL:
http://cncb-deploy-spa-john-websitebucket-1s8hgqtof7la7.s3-website-us-e
ast-1.amazonaws.com
...
Serverless: Path: ./build
Serverless: File: asset-manifest.json (application/json)
Serverless: File: favicon.ico (image/x-icon)
Serverless: File: index.html (text/html)
Serverless: File: manifest.json (application/json)
Serverless: File: service-worker.js (application/javascript)
Serverless: File: static/css/main.c17080f1.css (text/css)
Serverless: File: static/css/main.c17080f1.css.map (application/json)
Serverless: File: static/js/main.ee7b2412.js (application/javascript)
Serverless: File: static/js/main.ee7b2412.js.map (application/json)
Serverless: File: static/media/logo.5d5d9eef.svg (image/svg+xml)
```

12. Review the stack and bucket in the AWS Console
13. Browse to the `WebsiteURL` shown in the stack output:



14. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`

How it works...

The first thing to notice is that we are using all the same development tools for the full stack. This is one of many advantages of using JavaScript for backend development. A single, self-sufficient, full-stack team can develop the frontend application and the BFF service with the same programming language. This can allow for more efficient utilization of team resources.

There are two new standard scripts—`start` and `build`. `npm start` will run the frontend app locally using Node.js as the web server. `npm run build` will prepare the application for deployment. I used the `react-scripts` library so as not to clutter the example with a detailed ReactJS build process. This recipe uses a small, canned ReactJS example for the same reason. I wanted an app that was just large enough to have something to deploy. ReactJS is not the focus of this recipe or cookbook. There are volumes already written on ReactJS and similar frameworks.

We are creating an S3 bucket, `WebsiteBucket`, and configuring it as a website. The stack output displays the `WebsiteUrl` used to access the SPA. The SPA will be served from a bucket with no need for servers whatsoever. In this context, we can think of S3 as a global web server.

We are using a Serverless plugin for the first time in this recipe. The `serverless-spa-deploy` plugin will upload the SPA files from the `./build` directory after the stack is deployed. Note that we are not explicitly naming the bucket. CloudFormation will generate the name with a random suffix. This is important because bucket names must be globally unique. The plugin infers the generated bucket name. The plugin has sensible defaults that can be customized, such as to change the `CacheControl` headers for different files. The plugin also empties the bucket, before stack removal.



We will build on this architecture in `Chapter 4`, *Leveraging the Edge of the Cloud*.

2
Applying the Event Sourcing and CQRS Patterns

In this chapter, the following recipes will be covered:

- Creating a data lake
- Applying the event-first variant of the Event Sourcing pattern
- Creating a micro event store
- Applying the database-first variant of the Event Sourcing pattern with DynamoDB
- Applying the database-first variant of the Event Sourcing pattern with Cognito datasets
- Creating a materialized view in DynamoDB
- Creating a materialized view in S3
- Creating a materialized view in Elasticsearch
- Creating a materialized view in a Cognito dataset
- Replaying events
- Indexing the data lake
- Implementing bi-directional synchronization

Introduction

Cloud-native is autonomous. It empowers self-sufficient, full-stack teams to rapidly perform lean experiments and continuously deliver innovation with confidence. The operative word here is *confidence*. We leverage fully managed cloud services, such as function-as-a-service, cloud-native databases, and event streaming to decrease the risk of running these advanced technologies. However, at this rapid pace of change, we cannot completely eliminate the potential for human error. To remain stable despite the pace of change, cloud-native systems are composed of bounded, isolated, and autonomous services that are separated by bulkheads to minimize the blast radius when any given service experiences a failure. Each service is completely self-sufficient and stands on its own, even when related services are unavailable.

Following reactive principles, these autonomous services leverage event streaming for all inter-service communication. Event streaming turns the database inside out by replicating data across services in the form of materialized views stored in cloud-native databases. This cloud-native data forms a bulkhead between services and effectively turns the cloud into the database to maximize responsiveness, resilience, and elasticity. The **Event Sourcing** and **Command Query Responsibility Segregation** (**CQRS**) patterns are fundamental to creating autonomous services. This chapter contains recipes that demonstrate how to use fully managed, serverless cloud services to apply these patterns.

Creating a data lake

One of the major benefits of the Event Sourcing pattern is that it results in an audit trail of all the state changes within a system. This store of events can also be leveraged to replay events to repair broken services and seed new components. A cloud-native event stream, such as **AWS Kinesis**, only stores events for a short period of time, ranging from 24 hours to 7 days. An **event stream** can be thought of as a temporary or temporal event store that is used for normal, near real-time operation. In the *Creating a micro event store* recipe, we will discuss how to create specialized event stores that are dedicated to a single service. In this recipe, we will create a data lake in S3. A **data lake** is a perpetual event store that collects and stores all events in their raw format in perpetuity with complete fidelity and high durability to support auditing and replay.

Getting ready

Before starting this recipe, you will need an **AWS Kinesis Stream**, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/da
ta-lake-s3 --path cncb-data-lake-s3
```

2. Navigate to the `cncb-data-lake-s3` directory with `cd cncb-data-lake-s3`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-data-lake-s3

provider:
  name: aws
  runtime: nodejs8.10

functions:
  transformer:
    handler: handler.transform
    timeout: 120

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket
      DeletionPolicy: Retain
    DeliveryStream:
      Type: AWS::KinesisFirehose::DeliveryStream
      Properties:
        DeliveryStreamType: KinesisStreamAsSource
        KinesisStreamSourceConfiguration:
          KinesisStreamARN: ${cf:cncb-event-stream-
${opt:stage}.streamArn}
          ...
        ExtendedS3DestinationConfiguration:
          BucketARN:
            Fn::GetAtt: [ Bucket, Arn ]
          Prefix: ${cf:cncb-event-stream-${opt:stage}.streamName}/
          ...
```

```
  Outputs:
    DataLakeBucketName:
      Value:
        Ref: Bucket
```

4. Review the file named `handler.js` with the following content:

```
exports.transform = (event, context, callback) => {
  const output = event.records.map((record, i) => {
    // store all available data
    const uow = {
      event: JSON.parse((Buffer.from(record.data,
'base64')).toString('utf8')),
      kinesisRecordMetadata: record.kinesisRecordMetadata,
      firehoseRecordMetadata: {
        deliveryStreamArn: event.deliveryStreamArn,
        region: event.region,
        invocationId: event.invocationId,
        recordId: record.recordId,
        approximateArrivalTimestamp:
record.approximateArrivalTimestamp,
      }
    };

    return {
      recordId: record.recordId,
      result: 'Ok',
      data: Buffer.from(JSON.stringify(uow) + '\n',
'utf-8').toString('base64'),
    };
  });

  callback(null, { records: output });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-data-lake-s3@1.0.0 dp:lcl <path-to-your-workspace>/cncb-data-
lake-s3
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
```

```
...
Serverless: Stack update finished...
...
Stack Outputs
DataLakeBucketName: cncb-data-lake-s3-john-bucket-1851i1c16lnha
...
```

9. Review the stack, data lake bucket, and Firehose delivery stream in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created"}'
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49582906351415672136958521360120605392824155736450793474"
}
```

11. Allow the Firehose buffer time to process and then review the data lake contents created in the S3 bucket.
12. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.



Remove the data lake stack after you have worked through all the other recipes. This will allow you to watch the data lake accumulating all the other events.

How it works...

The most important characteristic of a data lake is that it stores data in perpetuity. The only way to really meet this requirement is to use object storage, such as AWS S3. S3 provides 11 nines of durability. Said another way, S3 provides 99.999999999% durability of objects over a given year. It is also fully managed and provides life cycle management features to age objects into cold storage. Note that the bucket is defined with the `DeletionPolicy` set to `Retain`. This highlights that even if the stack is deleted, we still want to ensure that we are not inappropriately deleting this valuable data.

We are using Kinesis Firehose because it performs the heavy lifting of writing the events to the bucket. It provides a buffer based on the time and size, compression, encryption, and error handling. To simplify this recipe, I did not use compression or encryption, but it is recommended that you use these features.

This recipe defines one delivery stream, because in this cookbook, our stream topology consists of only one stream with `${cf:cncb-event-stream-${opt:stage}.streamArn}`. In practice, your topology will consist of multiple streams, and you will define one Firehose delivery stream per Kinesis stream to ensure that the data lake is capturing all events. We set `prefix` to `${cf:cncb-event-stream-${opt:stage}.streamName}/` so that we can easily distinguish the events in the data lake by their stream.

Another important characteristic of a data lake is that the data is stored in its raw format, without modification. To this end, the `transformer` function adorns all available metadata about the specific Kinesis stream and Firehose delivery stream, to ensure that all available information is collected. In the *Replaying events* recipe, we will see how this metadata can be leveraged. Also, note that `transformer` adds the end-of-line character (`\n`) to facilitate future processing of the data.

Applying the event-first variant of the Event Sourcing pattern

Event sourcing is a key pattern for designing eventually consistent cloud-native systems. Upstream services produce events as their state changes, and downstream services consume these events and produce their own events as needed. This results in a chain of events whereby services collaborate to produce a business process that results in an eventual consistency solution. Each step in this chain must be implemented as an atomic unit of work. Cloud-native systems do not support distributed transactions, because they do not scale horizontally in a cost-effective manner. Therefore, each step must update one, and only one, system. If multiple systems must be updated, then each is updated in a series of steps. In this recipe, we leverage the event-first variant of the Event Sourcing pattern where the atomic unit of work is writing to the event stream. The ultimate persistence of the data is delegated to downstream components.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/ev
ent-first --path cncb-event-first
```

2. Navigate to the `cncb-event-first` directory with `cd cncb-event-first`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-event-first

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    - Effect: Allow
      Action:
        - kinesis:PutRecord
      Resource: ${cf:cncb-event-stream-${opt:stage}.streamArn}

functions:
  submit:
    handler: handler.submit
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}
```

4. Review the file named `handler.js` with the following content:

```
module.exports.submit = (thing, context, callback) => {
  thing.id = thing.id || uuid.v4();

  const event = {
    type: 'thing-submitted',
    id: uuid.v1(),
    partitionKey: thing.id,
    timestamp: Date.now(),
    tags: {
      region: process.env.AWS_REGION,
      kind: thing.kind,
    },
    thing: thing,
  };

  const params = {
    StreamName: process.env.STREAM_NAME,
```

```
    PartitionKey: event.partitionKey,
    Data: Buffer.from(JSON.stringify(event)),
  };

  const kinesis = new aws.Kinesis();

  kinesis.putRecord(params, (err, resp) => {
    callback(err, event);
  });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-event-first@1.0.0 dp:lcl <path-to-your-workspace>/cncb-event-
first
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  submit: cncb-event-first-john-submit
...
```

9. Review the stack and function in the AWS Console.
10. Invoke the `submit` function with the following command:

```
$ sls invoke -f submit -r us-east-1 -s $MY_STAGE -d
'{"id":"11111111-1111-1111-1111-111111111111","name":"thing
one","kind":"other"}'

{
    "type": "thing-submitted",
    "id": "2a1f5290-42c0-11e8-a06b-33908b837f8c",
    "partitionKey": "11111111-1111-1111-1111-111111111111",
    "timestamp": 1524025374265,
    "tags": {
        "region": "us-east-1",
        "kind": "other"
    },
    "thing": {
```

```
            "id": "11111111-1111-1111-1111-111111111111",
            "name": "thing one",
            "kind": "other"
        }
}
```

11. Take a look at the logs:

```
$ sls logs -f submit -r us-east-1 -s $MY_STAGE

START ...
2018-04-18 00:22:54 ... params: {"StreamName":"john-cncb-event-stream-
s1","PartitionKey":"11111111-1111-1111-1111-111111111111","Data":{"type
":"Buffer","data":[...]}}
2018-04-18 00:22:54 ... response:
{"ShardId":"shardId-000000000000","SequenceNumber":"4958...2466"}
END ...
REPORT ... Duration: 381.21 ms    Billed Duration: 400 ms ... Max
Memory Used: 34 MB
```

12. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a command function called `submit` that would be part of a Backend For Frontend service. Following the Event Sourcing pattern, we make this command atomic by only writing to a single resource. In some scenarios, such as initiating a long-lived business process or tracking user clicks, we only need to **fire-and-forget**. In these cases, the event-first variant is most appropriate. The command just needs to execute quickly and leave as little to chance as possible. We write the event to the highly available, fully-managed cloud-native event stream and trust that the downstream services will eventually consume the event.

The logic wraps the domain object in the standard event format, as discussed in the *Creating an event stream and publishing an event* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The event `type` is specified, the domain object ID is used as the `partitionKey`, and useful `tags` are adorned. Finally, the event is written to the stream specified by the `STREAM_NAME` environment variable.

Creating a micro event store

In the *Creating a data lake* recipe, we will discuss how the Event Sourcing pattern provides the system with an audit trail of all the state-change events in the system. An event stream essentially provides a temporal event store that feeds downstream event processors in near real-time. The data lake provides a high durability, perpetual event store that is the official source of record. However, we have a need for a middle ground. Individual stream processors need the ability to source specific events that support their processing requirement. In this recipe, we will implement a micro event store in **AWS DynamoDB** that is owned by and tailored to the needs of a specific service.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/mi
cro-event-store --path cncb-micro-event-store
```

2. Navigate to the `cncb-micro-event-store` directory with `cd cncb-micro-event-store`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-micro-event-store

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:PutItem
        - dynamodb:Query
      Resource:
        Fn::GetAtt: [ Table, Arn ]
  environment:
    TABLE_NAME:
      Ref: Table

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
```

```
          type: kinesis
          ...
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [ Table, StreamArn ]
          batchSize: 100
          startingPosition: TRIM_HORIZON

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${opt:stage}-${self:service}-events
        AttributeDefinitions:
          ...
        KeySchema:
          - AttributeName: partitionKey
            KeyType: HASH
          - AttributeName: eventId
            KeyType: RANGE
        ...
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(byType)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...
const byType = event => event.type.matches(/thing-.+/);

const put = event => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Item: {
      partitionKey: event.partitionKey,
```

```
      eventId: event.id,
      event: event,
    }
  };

  const db = new aws.DynamoDB.DocumentClient();
  return _(db.put(params).promise());
};

module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .flatMap(getMicroEventStore)
    .tap(events => console.log('events: %j', events))
    .collect().toCallback(cb);
};

const getMicroEventStore = (record) => {
  const params = {
    TableName: process.env.TABLE_NAME,
    KeyConditionExpression: '#partitionKey = :partitionKey',
    ExpressionAttributeNames: {
      '#partitionKey': 'partitionKey'
    },
    ExpressionAttributeValues: {
      ':partitionKey': record.dynamodb.Keys.partitionKey.S
    }
  };

  const db = new aws.DynamoDB.DocumentClient();
  return _(db.query(params).promise());
}
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-micro-event-store@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
micro-event-store
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
```

```
functions:
  listener: cncb-micro-event-store-john-listener
  trigger: cncb-micro-event-store-john-trigger
...
```

9. Review the stack, functions, and table in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
updated","partitionKey":"11111111-1111-1111-1111-111111111111","thing":
{"new":{"name":"thing
one","id":"11111111-1111-1111-1111-111111111111"}}}'

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583553996455686705785668952922460091805481438885707778"
}
```

11. Take a look at the logs for the `listener` function:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE

START ...
2018-04-18 01:18:55... {"type":"thing-
updated","partitionKey":"11111111-1111-1111-1111-111111111111","thing":
{"new":{"name":"thing
one","id":"11111111-1111-1111-1111-111111111111"}},"id":"fcc03460-42c7-
11e8-8756-f75e650b2731","timestamp":1524028734118,"tags":{"region":"us-
east-1"}}
2018-04-18 01:18:55.394 (-04:00)    b42aaa92-8a9a-418d-8e22-
ecc54e9966f6    params: {"TableName":"john-cncb-micro-event-store-
events","Item":{"partitionKey":"11111111-1111-1111-1111-111111111111","
eventId":"fcc03460-42c7-11e8-8756-f75e650b2731","event":{"type":"thing-
updated","partitionKey":"11111111-1111-1111-1111-111111111111","thing":
{"new":{"name":"thing
one","id":"11111111-1111-1111-1111-111111111111"}},"id":"fcc03460-42c7-
11e8-8756-f75e650b2731","timestamp":1524028734118,"tags":{"region":"us-
east-1"}}}}
END ...
REPORT ... Duration: 149.24 ms    Billed Duration: 200 ms ... Max
Memory Used: 35 MB
```

12. Take a look at the logs for the `trigger` function:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE

START ...
2018-04-18 01:18:56 ... event:
{"Records":[{"eventID":"b8dbee2d9f49ee05609a7e930ac204e7","eventName":"
INSERT",...,"Keys":{"eventId":{"S":"fcc03460-42c7-11e8-8756-
f75e650b2731"},"partitionKey":{"S":"11111111-1111-1111-1111-11111111111
1"}},...}]}
2018-04-18 01:18:56 ... params: {"TableName":"john-cncb-micro-event-
store-
events",...,"ExpressionAttributeValues":{":partitionKey":"11111111-1111
-1111-1111-111111111111"}}
2018-04-18 01:18:56 ... events:
{"Items":[{"eventId":"2a1f5290-42c0-11e8-
a06b-33908b837f8c","partitionKey":"11111111-1111-1111-1111-111111111111
","event":{"id":"2a1f5290-42c0-11e8-a06b-33908b837f8c","type":"thing-
submitted","partitionKey":"11111111-1111-1111-1111-111111111111","thing
":{"name":"thing
one","kind":"other","id":"11111111-1111-1111-1111-111111111111"},"times
tamp":1524025374265,"tags":{"region":"us-
east-1","kind":"other"}}},{"eventId":"fcc03460-42c7-11e8-8756-
f75e650b2731","partitionKey":"11111111-1111-1111-1111-111111111111","ev
ent":{"id":"fcc03460-42c7-11e8-8756-f75e650b2731","type":"thing-
updated","partitionKey":"11111111-1111-1111-1111-111111111111","thing":
{"new":{"name":"thing
one","id":"11111111-1111-1111-1111-111111111111"}},"timestamp":15240287
34118,"tags":{"region":"us-east-1"}}}],"Count":2,"ScannedCount":2}
END ...
REPORT ... Duration: 70.88 ms    Billed Duration: 100 ms     Memory
Size: 1024 MB    Max Memory Used: 42 MB
```

13. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

When implementing a stream processor function, we often need more information than is available in the current event object. It is a best practice when publishing events to include all the relevant data that is available in the publishing context so that each event represents a micro snapshot of the system at the time of publishing. When this data is not enough, we need to retrieve more data; however, in cloud-native systems, we strive to eliminate all synchronous inter-service communication because it reduces the *autonomy* of the services. Instead, we create a micro event store that is tailored to the needs of the specific service.

First, we implement a `listener` function and `filter` for the desired events from the stream. Each event is stored in a DynamoDB table. You can store the entire event or just the information that is needed. When storing these events, we need to collate related events by carefully defining the `HASH` and `RANGE` keys. For example, we might want to collate all events for a specific domain object ID or all events from a specific user ID. In this example, we use `event.partitionKey` as the hash key, but you can calculate the hash key from any of the available data. For the range key, we need a value that is unique within the hash key. The `event.id` is a good choice if it is implemented with a V1 UUID because they are time-based. The Kinesis sequence number is another good choice. The `event.timestamp` is another alternative, but there could be a potential that events are created at the exact same time within a hash key.

The `trigger` function, which is attached to the DynamoDB stream, takes over after the `listener` has saved an event. The trigger calls `getMicroEventStore` to retrieve the micro event store based on the hash key calculated for the current event. At this point, the stream processor has all the relevant data available in memory. The events in the micro event store are in historical order, based on the value used for the range key. The stream processor can use this data however it sees fit to implement its business logic.



Use the DynamoDB TTL feature to keep the micro event store from growing unbounded.

Applying the database-first variant of the Event Sourcing pattern with DynamoDB

In the previous recipe, *Applying the event-first variant of the Event Sourcing pattern*, we discussed how the Event Sourcing pattern allows us to design eventually consistent systems that are composed of a chain of atomic steps. Distributed transactions are not supported in cloud-native systems, because they do not scale effectively. Therefore, each step must update one, and only one, system. In this recipe, we will leverage the **database-first** variant of the Event Sourcing pattern, where the atomic unit of work is writing to a single cloud-native database. A cloud-native database provides a **change data capture** mechanism that allows further logic to be atomically triggered that publishes an appropriate **domain event** to the event stream for further downstream processing. In this recipe, we will demonstrate implementing this pattern with **AWS DynamoDB** and **DynamoDB Streams**.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/db
-first-dynamodb --path cncb-db-first-dynamodb
```

2. Navigate to the `cncb-db-first-dynamodb` directory with `cd cncb-db-first-dynamodb`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-db-first-dynamodb

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:PutItem
      Resource:
        Fn::GetAtt: [ Table, Arn ]
    - Effect: Allow
      Action:
        - kinesis:PutRecord
      Resource: ${cf:cncb-event-stream-${opt:stage}.streamArn}

functions:
  command:
    handler: handler.command
    environment:
      TABLE_NAME:
        Ref: Table
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [ Table, StreamArn ]
          ...
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}

resources:
  Resources:
```

```
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${opt:stage}-${self:service}-things
        AttributeDefinitions:
          ...
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        ...
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
```

4. Review the file named `handler.js` with the following content:

```
module.exports.command = (request, context, callback) => {
  const thing = {
    id: uuid.v4(),
    ...request,
  };

  const params = {
    TableName: process.env.TABLE_NAME,
    Item: thing,
  };

  const db = new aws.DynamoDB.DocumentClient();
  db.put(params, callback);
};

module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .map(toEvent)
    .flatMap(publish)
    .collect()
    .toCallback(cb);
};

const toEvent = record => ({
  id: record.eventID,
  type: `thing-${EVENT_NAME_MAPPING[record.eventName]}`,
  timestamp: record.dynamodb.ApproximateCreationDateTime * 1000,
  partitionKey: record.dynamodb.Keys.id.S,
  tags: {
    region: record.awsRegion,
  },
  thing: {
    old: record.dynamodb.OldImage ?
```

```
      aws.DynamoDB.Converter.unmarshall(record.dynamodb.OldImage) :
      undefined,
    new: record.dynamodb.NewImage ?
      aws.DynamoDB.Converter.unmarshall(record.dynamodb.NewImage) :
      undefined,
  },
});

const EVENT_NAME_MAPPING = {
  INSERT: 'created',
  MODIFY: 'updated',
  REMOVE: 'deleted',
};

const publish = event => {
  const params = {
    StreamName: process.env.STREAM_NAME,
    PartitionKey: event.partitionKey,
    Data: Buffer.from(JSON.stringify(event)),
  };

  const kinesis = new aws.Kinesis();
  return _(kinesis.putRecord(params).promise());
}
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-db-first-dynamodb@1.0.0 dp:lcl <path-to-your-workspace>/cncb-db-
first-dynamodb
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  command: cncb-db-first-dynamodb-john-command
  trigger: cncb-db-first-dynamodb-john-trigger
```

9. Review the stack, functions, and table in the AWS Console.
10. Invoke the function with the following command:

```
$ sls invoke -r us-east-1 -f command -s $MY_STAGE -d '{"name":"thing
one"}'
```

11. Take a look at the `command` function logs:

```
$ sls logs -f command -r us-east-1 -s $MY_STAGE
START ...
2018-04-17 00:29:13 ... request: {"name":"thing one"}
2018-04-17 00:29:13 ... params: {"TableName":"john-cncb-db-first-
dynamodb-things","Item":{"id":"4297c253-f512-443d-
baaf-65f0a36aaaa3","name":"thing one"}}
END ...
REPORT ... Duration: 136.99 ms    Billed Duration: 200 ms     Memory
Size: 1024 MB    Max Memory Used: 35 MB
```

12. Take a look at the trigger function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
START ...
2018-04-17 00:29:15 ... event:
{"Records":[{"eventID":"39070dc13de0eb76548506a977d4134c","eventName":"
INSERT",...,"dynamodb":{"ApproximateCreationDateTime":1523939340,"Keys"
:{"id":{"S":"4297c253-f512-443d-
baaf-65f0a36aaaa3"}},"NewImage":{"name":{"S":"thing
one"},"id":{"S":"4297c253-f512-443d-
baaf-65f0a36aaaa3"}},"SequenceNumber":"10000000006513931753",...},...}
]}
2018-04-17 00:29:15 ...
{"id":"39070dc13de0eb76548506a977d4134c","type":"thing-
created","timestamp":1523939340000,"partitionKey":"4297c253-f512-443d-
baaf-65f0a36aaaa3","tags":{"region":"us-
east-1"},"thing":{"new":{"name":"thing one","id":"4297c253-f512-443d-
baaf-65f0a36aaaa3"}}}
2018-04-17 00:29:15 ... params: {"StreamName":"john-cncb-event-stream-
s1","PartitionKey":"4297c253-f512-443d-
baaf-65f0a36aaaa3","Data":{"type":"Buffer","data":[...]}}
2018-04-17 00:29:15 ...
{"ShardId":"shardId-000000000000","SequenceNumber":"4958...3778"}
END ...
REPORT ... Duration: 326.99 ms    Billed Duration: 400 ms ... Max
Memory Used: 40 MB
```

13. Review the events collected in the data lake bucket.
14. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a `command` function that would be part of a Backend For Frontend service. Following the Event Sourcing pattern, we make this command atomic by only writing to a single resource. In many scenarios, such as the authoring of data, we need to write data and make sure it's immediately available for reading. In these cases, the *database-first* variant is most appropriate. The command just needs to execute quickly and leave as little to chance as possible. We write the **domain object** to the highly available, fully-managed cloud-native database and trust that the database's **change data capture** mechanism will handle the next step.

In this recipe, the database is DynamoDB and the change data capture mechanism is DynamoDB Streams. The `trigger` function is a stream processor that is consuming events from the specified DynamoDB stream. We enable the stream by adding the `StreamSpecification` to the definition of the table.

The stream processor logic wraps the domain object in the standard event format, as discussed in the *Creating an event stream and publishing an event* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The `record.eventID` generated by DynamoDB is reused as the domain event ID, the database trigger's `record.eventName` is translated into the domain event type, the domain object ID is used as `partitionKey`, and useful `tags` are adorned. The `old` and `new` values of the domain object are included in the event so that downstream services can calculate a delta however they see fit.

Finally, the event is written to the stream specified by the `STREAM_NAME` environment variable. Note that the trigger function is similar to the *event-first* variant. It just needs to execute quickly and leave as little to chance as possible. We write the event to a single resource, the highly available, fully-managed cloud-native event stream, and trust that the downstream services will eventually consume the event.

Applying the database-first variant of the Event Sourcing pattern with Cognito datasets

In the *Applying the event-first variant of the Event Sourcing pattern* recipe, we discussed how the Event Sourcing pattern allows us to design eventually consistent systems that are composed of a chain of atomic steps. Distributed transactions are not supported in cloud-native systems, because they do not scale effectively. Therefore, each step must update one, and only one, system. In this recipe, we leverage the *database-first* variant of the Event Sourcing pattern, where the atomic unit of work is writing to a single cloud-native database. A cloud-native database provides a change data capture mechanism that allows further logic to be atomically triggered that publishes an appropriate domain event to the event stream for further downstream processing. In the recipe, we demonstrate an **offline-first** implementation of this pattern with **AWS Cognito datasets**.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/db
-first-cognito --path cncb-db-first-cognito
```

2. Navigate to the `cncb-db-first-cognito` directory with `cd cncb-db-first-cognito`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-db-first-cognito

provider:
  name: aws
  runtime: nodejs8.10
  ...

functions:
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: kinesis
          arn:
```

```
            Fn::GetAtt: [ CognitoStream, Arn ]
          ...
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}

resources:
  Resources:
    CognitoStream:
      Type: AWS::Kinesis::Stream
      Properties:
          ShardCount: 1

    IdentityPool:
      Type: AWS::Cognito::IdentityPool
      Properties:
        CognitoStreams:
          StreamName:
            Ref: CognitoStream
    ...

  Outputs:
    identityPoolId:
      Value:
        Ref: IdentityPool
    identityPoolName:
      Value:
        Fn::GetAtt: [ IdentityPool, Name ]
```

4. Review the file named `handler.js` with the following content:

```
module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .flatMap(recordToSync)
    .map(toEvent)
    .flatMap(publish)
    .collect().toCallback(cb);
};

const recordToSync = r => {
  const data = JSON.parse(Buffer.from(r.kinesis.data, 'base64'));
  return _(data.kinesisSyncRecords.map(sync => ({
    record: r,
    data: data,
    sync: sync,
    thing: JSON.parse(sync.value)
  })));
}
```

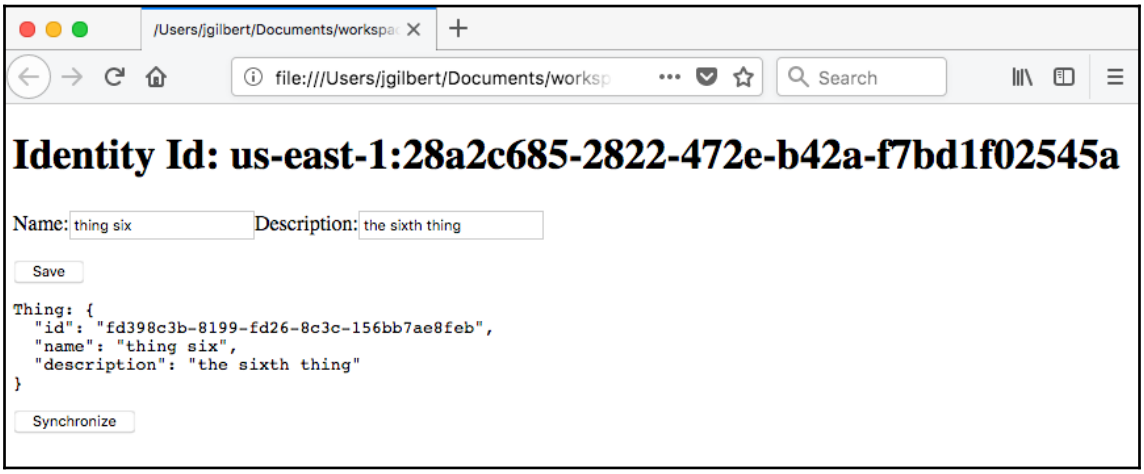
```
const toEvent = uow => ({
  id: uuid.v1(),
  type: `thing-created`,
  timestamp: uow.sync.lastModifiedDate,
  partitionKey: uow.thing.id,
  tags: {
    region: uow.record.awsRegion,
    identityPoolId: uow.data.identityPoolId,
    datasetName: uow.data.datasetName
  },
  thing: {
      identityId: uow.data.identityId, // the end user
      ...uow.thing,
  },
  raw: uow.sync
});

...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-db-first-cognito@1.0.0 dp:lcl <path-to-your-workspace>/cncb-db-
first-cognito
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  trigger: cncb-db-first-cognito-john-trigger

Stack Outputs
identityPoolName: IdentityPool_P6awUWzjQH0y
identityPoolId: us-east-1:e51ba12c-75c2-4548-868d-2d023eb9398b
...
```

9. Review the stack, function, and identity pool in the AWS Console.
10. Update the file named `./index.html` with the `identityPoolId` from the previous output.

11. Open the file named `./index.html` in a browser, enter a `name` and `description`, and press `Save` and then `Synchronize`, as shown in the following screenshot:



12. Take a look at the logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE

START ...
2018-04-19 00:50:15 ... {"id":"2714e290-438d-11e8-
b3de-2bf7e0b964a2","type":"thing-
created","timestamp":1524113413268,"partitionKey":"fd398c3b-8199-
fd26-8c3c-156bb7ae8feb","tags":{"region":"us-
east-1","identityPoolId":"us-
east-1:e51ba12c-75c2-4548-868d-2d023eb9398b","datasetName":"things"},"t
hing":{"identityId":"us-east-1:28a2c685-2822-472e-b42a-
f7bd1f02545a","id":"fd398c3b-8199-fd26-8c3c-156bb7ae8feb","name":"thing
six","description":"the sixth
thing"},"raw":{"key":"thing","value":"{\"id\":\"fd398c3b-8199-
fd26-8c3c-156bb7ae8feb\",\"name\":\"thing six\",\"description\":\"the
sixth
thing\"}","syncCount":1,"lastModifiedDate":1524113413268,"deviceLastMod
ifiedDate":1524113410528,"op":"replace"}}
2018-04-19 00:50:15 ... params: {"StreamName":"john-cncb-event-stream-
s1","PartitionKey":"fd398c3b-8199-
fd26-8c3c-156bb7ae8feb","Data":{"type":"Buffer","data":[...]}}
END ...
REPORT ... Duration: 217.22 ms    Billed Duration: 300 ms ... Max
Memory Used: 40 MB
```

13. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`

How it works...

In this recipe, we are implementing an offline-first solution where the ReactJS client application stores all changes in local storage and then synchronizes the data to a Cognito dataset in the cloud when connectivity is available. This scenario is very common in mobile applications where the mobile application may not always be connected. An AWS Cognito dataset is associated with a specific user in an **AWS Identity Pool**. In this recipe, the identity pool supports unauthenticated users. Anonymous access is another common characteristic of mobile applications.

The bare bones ReactJS application is implemented in the `./index.html` file. It contains a form for the user to enter data. The `Save` button saves the form's data to local storage via the Cognito SDK. The `Synchronize` button uses the SDK to send the local data to the cloud. In a typical application, this synchronization would be triggered behind the scenes by events in the normal flow of the application, such as on save, on load, and before exit. In the *Creating a materialized view in a Cognito Dataset* recipe, we show how synchronizing will also retrieve data from the cloud.

Cognito's change data capture feature is implemented via **AWS Kinesis**. Therefore, we create a Kinesis stream called `CognitoStream` that is dedicated to our Cognito datasets. The `trigger` function is a stream processor that is consuming sync records from this stream. The stream processor's `recordToSync` step extracts the domain object from each sync record, where it is stored as a JSON string. The `toEvent` step wraps the domain object in the standard event format, as discussed in the *Creating an event stream and publishing an event* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. Finally, the event is written to the stream specified by the `STREAM_NAME` environment variable. Note that the trigger function is similar to the event-first variant. It just needs to execute quickly and leave as little to chance as possible. We write the event to a single resource, the highly available, fully managed cloud-native event stream, and trust that the downstream services will eventually consume the event.

Creating a materialized view in DynamoDB

The **Command Query Responsibility Segregation** (**CQRS**) pattern is critical for designing cloud-native systems that are composed of bounded, isolated, and autonomous services with appropriate bulkheads to limit the blast radius when a service experiences an outage. These bulkheads are implemented by creating materialized views in downstream services.

Upstream services are responsible for the commands that write data using the Event Sourcing pattern. Downstream services take responsibility for their own queries by creating materialized views that are specifically tailored to their needs. This **replication** of data increases scalability, reduces latency, and allows services to be completely **autonomous** and function even when upstream source services are unavailable. In this recipe, we will implement a materialized view in **AWS DynamoDB**.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/ma
terialized-view-dynamodb --path cncb-materialized-view-dynamodb
```

2. Navigate to the `cncb-materialized-view-dynamodb` directory with `cd cncb-materialized-view-dynamodb`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-materialized-view-dynamodb

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    TABLE_NAME:
      Ref: Table

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
```

```
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
  query:
    handler: handler.query

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${opt:stage}-${self:service}-things
        AttributeDefinitions:
          ...
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        ...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forThingCreated)
    .map(toThing)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...
const forThingCreated = e => e.type === 'thing-created';

const toThing = event => ({
  id: event.thing.new.id,
  name: event.thing.new.name,
  description: event.thing.new.description,
  asOf: event.timestamp,
});

const put = thing => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Item: thing,
  };

  const db = new aws.DynamoDB.DocumentClient();
  return _(db.put(params).promise());
```

```
};

module.exports.query = (id, context, callback) => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Key: {
      id: id,
    },
  };

  const db = new aws.DynamoDB.DocumentClient();
  db.get(params, callback);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-materialized-view-dynamodb@1.0.0 dp:lcl <path-to-your-
workspace>/cncb-materialized-view-dynamodb
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-materialized-view-dynamodb-john-listener
  query: cncb-materialized-view-dynamodb-john-query
```

9. Review the stack, functions, and table in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
two","id":"22222222-2222-2222-2222-222222222222"}}}'
{
 "ShardId": "shardId-000000000000",
 "SequenceNumber":
"49583553996455686705785668952916415462701426537440215042"
}
```

11. Take a look at the `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
START ...
2018-04-17 00:54:48 ... event: {"Records":[...]}
2018-04-17 00:54:48 ...
{"id":"39070dc13de0eb76548506a977d4134c","type":"thing-
created","timestamp":1523939340000,"tags":{"region":"us-
east-1"},"thing":{"new":{"name":"thing
two","id":"22222222-2222-2222-2222-222222222222"}}}
2018-04-17 00:54:48 ... params: {"TableName":"john-cncb-materialized-
view-dynamodb-
things","Item":{"id":"22222222-2222-2222-2222-222222222222","name":"thi
ng two","asOf":1523939340000}}
END ...
REPORT ... Duration: 306.17 ms    Billed Duration: 400 ms ... Max
Memory Used: 36 MB
```

12. Invoke the `query` command:

```
$ sls invoke -r us-east-1 -f query -s $MY_STAGE -d
22222222-2222-2222-2222-222222222222
{
    "Item": {
        "id": "22222222-2222-2222-2222-222222222222",
        "name": "thing two",
        "asOf": 1523939340000
    }
}
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implemented a `listener` function that consumes upstream events and populates a materialized view that is used by a **Backend For Frontend** (**BFF**) service. This function is a *stream processor,* such as the one we discussed in the *Creating a stream processor* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The function performs a `filter` for the desired events and then transforms the data in a `map` step to the desired materialized view. The materialized view is optimized to support the requirements of the query needed by the BFF. Only the minimum necessary data is stored and the optimal database type is used. In this recipe, the database type is DynamoDB. DynamoDB is a good choice for a materialized view when the data changes frequently.

Note that the `asOf` timestamp is included in the record. In an eventually consistent system, it is important to provide the user with the `asOf` value so that he or she can access the latency of the data. Finally, the data is stored in the highly available, fully managed, cloud-native database.

Creating a materialized view in S3

In the *Creating a materialized view in DynamoDB* recipe, we discussed how the CQRS pattern allows us to design services that are bounded, isolated, and autonomous. This allows services to operate, even when their upstream dependencies are unavailable, because we have eliminated all synchronous inter-service communication in favor of replicating and caching the required data locally in dedicated materialized views. In this recipe, we will implement a materialized view in AWS S3.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/ma
terialized-view-s3 --path cncb-materialized-view-s3
```

2. Navigate to the `cncb-materialized-view-s3` directory with `cd cncb-materialized-view-s3`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-materialized-view-s3

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
```

```
    environment:
      BUCKET_NAME:
        Ref: Bucket

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket

  Outputs:
    BucketName:
      Value:
        Ref: Bucket
    BucketDomainName:
      Value:
        Fn::GetAtt: [ Bucket, DomainName ]
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forThingCreated)
    .map(toThing)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...
const forThingCreated = e => e.type === 'thing-created';

const toThing = event => ({
  id: event.thing.new.id,
  name: event.thing.new.name,
  description: event.thing.new.description,
  asOf: event.timestamp,
});

const put = thing => {
  const params = {
    Bucket: process.env.BUCKET_NAME,
    Key: `things/${thing.id}`,
    ACL: 'public-read',
    ContentType: 'application/json',
    CacheControl: 'max-age=300',
    Body: JSON.stringify(thing),
  };
```

```
  const s3 = new aws.S3();
  return _(s3.putObject(params).promise());
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-materialized-view-s3@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
materialized-view-s3
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-materialized-view-s3-john-listener

Stack Outputs
BucketName: cncb-materialized-view-s3-john-bucket-1pp3d4c2z99kt
BucketDomainName: cncb-materialized-view-s3-john-
bucket-1pp3d4c2z99kt.s3.amazonaws.com
...
```

9. Review the stack, function, and bucket from the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
three","id":"33333333-3333-3333-3333-333333333333"}}}'
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583553996455686705785668952918833314346020725338406914"
}
```

11. Take a look at the logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
START ...
2018-04-17 22:49:20 ... event: {"Records":[...]}
2018-04-17 22:49:20 ... {"type":"thing-
```

```
created","thing":{"new":{"name":"thing
three","id":"33333333-3333-3333-3333-333333333333"}},"id":"16a7b930-42b
3-11e8-8700-a918e007d88a","partitionKey":"3de89e9d-
c48d-4255-84fc-6c1b7e3f8b90","timestamp":1524019758148,"tags":{"region"
:"us-east-1"}}
2018-04-17 22:49:20 ... params: {"Bucket":"cncb-materialized-view-s3-
john-
bucket-1pp3d4c2z99kt","Key":"things/33333333-3333-3333-3333-33333333333
3","ACL":"public-
read","ContentType":"application/json","CacheControl":"max-
age=300","Body":"{\"id\":\"33333333-3333-3333-3333-333333333333\",\"nam
e\":\"thing three\",\"asOf\":1524019758148}"}
2018-04-17 22:49:20 ... {"ETag":"\"edfee997659a520994ed18b82255be2a\""}
END ...
REPORT ... Duration: 167.66 ms    Billed Duration: 200 ms ... Max
Memory Used: 36 MB
```

12. Invoke the following command, after updating the `bucket-suffix`, to get the data from S3:

```
$ curl
https://s3.amazonaws.com/cncb-materialized-view-s3-$MY_STAGE-bucket-<bu
cket-suffix>/things/33333333-3333-3333-3333-333333333333 | json_pp
{
 "asOf" : 1524019758148,
 "name" : "thing three",
 "id" : "33333333-3333-3333-3333-333333333333"
}
```

13. Use the console to delete the objects from the bucket before removing the stack.
14. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a `listener` function that consumes upstream events and populates a materialized view that is used by a Backend For Frontend service. This function is a stream processor, such as the one we discussed in the *Creating a stream processor* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The function performs a `filter` for the desired events and then transforms the data in a `map` step to the desired materialized view. The materialized view is optimized to support the requirements of the query needed by the BFF. Only the minimum necessary data is stored, and the optimal database type is used.

In this recipe, the database type is S3. S3 is a good choice for a materialized view when the data changes infrequently, and it can be cached in the CDN. Note that the `asOf` timestamp is included in the record so that the user can access the latency of the data.

Creating a materialized view in Elasticsearch

In the *Creating a materialized view in DynamoDB* recipe, we discussed how the CQRS pattern allows us to design services that are bounded, isolated, and autonomous. This allows services to operate, even when their upstream dependencies are unavailable, because we have eliminated all synchronous inter-service communication in favor of replicating and caching the required data locally in dedicated materialized views. In this recipe, we will implement a materialized view in AWS Elasticsearch.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/ma
terialized-view-es --path cncb-materialized-view-es
```

2. Navigate to the `cncb-materialized-view-es` directory with `cd cncb-materialized-view-es`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-materialized-view-es

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    DOMAIN_ENDPOINT:
      Fn::GetAtt: [ Domain, DomainEndpoint ]

...

functions:
  listener:
    handler: handler.listener
    events:
```

```
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
  search:
    handler: handler.search

resources:
  Resources:
    Domain:
      Type: AWS::Elasticsearch::Domain
      Properties:
        ...

  Outputs:
    DomainName:
      Value:
        Ref: Domain
    DomainEndpoint:
      Value:
        Fn::GetAtt: [ Domain, DomainEndpoint ]
```

4. Review the file named `handler.js` with the following content:

```
const client = require('elasticsearch').Client({
  hosts: [`https://${process.env.DOMAIN_ENDPOINT}`],
  connectionClass: require('http-aws-es'),
  log: 'trace',
});

module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forThingCreated)
    .map(toThing)
    .flatMap(index)
    .collect()
    .toCallback(cb);
};

...
const forThingCreated = e => e.type === 'thing-created';

const toThing = event => ({
  id: event.thing.new.id,
  name: event.thing.new.name,
  description: event.thing.new.description,
  asOf: event.timestamp,
```

```
});

const index = thing => {
  const params = {
    index: 'things',
    type: 'thing',
    id: thing.id,
    body: thing,
  };

  return _(client.index(params));
};

module.exports.search = (query, context, callback) => {
  const params = {
    index: 'things',
    q: query,
  };

  client.search(params, callback);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:



Deploying an Elasticsearch domain can take upwards of 1-20 minutes.

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-materialized-view-es@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
materialized-view-es
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-materialized-view-es-john-listener
  search: cncb-materialized-view-es-john-search
```

```
Stack Outputs
...
DomainEndpoint: search-cncb-ma-domain-gw419rzj26hz-
p2g37av7sdlltosbqhag3qhwnq.us-east-1.es.amazonaws.com
DomainName: cncb-ma-domain-gw419rzj26hz
...
```

9. Review the stack, function, and Elasticsearch domain in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ $ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
four","id":"44444444-4444-4444-4444-444444444444"}}}'

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583655996852917476267785004768832452002332571160543234"
}
```

11. Take a look at the `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE

START ...
2018-04-19 01:54:33 ... {"type":"thing-
created","thing":{"new":{"name":"thing
four","id":"44444444-4444-4444-4444-444444444444"}},"id":"0e1a68c0-4395
-11e8-b455-8144cebc5972","partitionKey":"8082a69c-00ee-4388-9697-
c590c523c061","timestamp":1524116810060,"tags":{"region":"us-east-1"}}
2018-04-19 01:54:33 ... params:
{"index":"things","type":"thing","id":"44444444-4444-4444-4444-44444444
4444","body":{"id":"44444444-4444-4444-4444-444444444444","name":"thing
four","asOf":1524116810060}}
2018-04-19 01:54:33 ...
{"_index":"things","_type":"thing","_id":"44444444-4444-4444-4444-44444
4444444","_version":1,"result":"created","_shards":{"total":2,"successf
ul":1,"failed":0},"_seq_no":0,"_primary_term":1}
END ...
REPORT ... Duration: 31.00 ms    Billed Duration: 100 ms ... Max Memory
Used: 42 MB
```

12. Invoke the `search` command:

```
$ sls invoke -r us-east-1 -f search -s $MY_STAGE -d four

{
    ...
    "hits": {
        "total": 1,
        "max_score": 0.2876821,
        "hits": [
            {
                "_index": "things",
                "_type": "thing",
                "_id": "44444444-4444-4444-4444-444444444444",
                "_score": 0.2876821,
                "_source": {
                    "id": "44444444-4444-4444-4444-444444444444",
                    "name": "thing four",
                    "asOf": 1524116810060
                }
            }
        ]
    }
}
```

13. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a `listener` function that consumes upstream events and populates a materialized view that is used by a Backend For Frontend service. This function is a stream processor, such as the one we discussed in the *Creating a stream processor* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The function performs a `filter` for the desired events and then transforms the data in a `map` step to the desired materialized view. The materialized view is optimized to support the requirements of the query needed by the BFF. Only the minimum necessary data is stored, and the optimal database type is used. In this recipe, the database type is Elasticsearch. Elasticsearch is a good choice for a materialized view when the data must be searched and filtered. Note that the `asOf` timestamp is included in the record so that the user can access the latency of the data.

Creating a materialized view in a Cognito dataset

In the *Creating a materialized view in DynamoDB* recipe, we discussed how the CQRS pattern allows us to design services that are bounded, isolated, and autonomous. This allows services to operate, even when their upstream dependencies are unavailable, because we have eliminated all synchronous inter-service communication in favor of replicating and caching the required data locally in materialized views. In this recipe, we will implement an offline-first materialized view in an AWS Cognito dataset.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/ma
terialized-view-cognito --path cncb-materialized-view-cognito
```

2. Navigate to the `cncb-materialized-view-cognito` directory with `cd cncb-materialized-view-cognito`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-materialized-view-cognito

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
    environment:
      IDENTITY_POOL_ID:
        Ref: IdentityPool

resources:
```

```
  Resources:
    IdentityPool:
      Type: AWS::Cognito::IdentityPool
      ...

  Outputs:
    identityPoolId:
      Value:
        Ref: IdentityPool
    identityPoolName:
      Value:
        Fn::GetAtt: [ IdentityPool, Name ]
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forThingCreated)
    .map(toThing)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...
const forThingCreated = e => e.type === 'thing-created';

const toThing = event => ({
  id: event.thing.new.id,
  name: event.thing.new.name,
  description: event.thing.new.description,
  identityId: event.thing.new.identityId, // the end user
  asOf: event.timestamp,
});

const put = thing => {
  const params = {
    IdentityPoolId: process.env.IDENTITY_POOL_ID,
    IdentityId: thing.identityId,
    DatasetName: 'things',
  };

  const cognitosync = new aws.CognitoSync();

  return _(
    cognitosync.listRecords(params).promise()
      .then(data => {
```

```
        params.SyncSessionToken = data.SyncSessionToken;
        params.RecordPatches = [{
          Key: 'thing',
          Value: JSON.stringify(thing),
          Op: 'replace',
          SyncCount: data.DatasetSyncCount,
        }];

        return cognitosync.updateRecords(params).promise()
      })
  );
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-materialized-view-cognito@1.0.0 dp:lcl <path-to-your-
workspace>/cncb-materialized-view-cognito
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-materialized-view-cognito-john-listener

Stack Outputs
identityPoolName: IdentityPool_c0GbzyVSh3Ws
identityPoolId: us-east-1:3a07e120-f1d8-4c85-9c34-0f908f2a21a1
...
```

9. Review the stack, function, and identity pool in the AWS Console.
10. Update the file named `index.html` file with the `identityPoolId` from previous output.
11. Open the file named `index.html` in a browser and copy the identity ID for use in the next step.

12. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
five","id":"55555555-5555-5555-5555-555555555555",
"identityId":"<identityId from previous step>"}}}'

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583655996852917476267784847452524471369889169788633090"
}
```

13. Take a look at the logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE

START ...
2018-04-19 00:18:42 ... {"type":"thing-
created","thing":{"new":{"name":"thing
five","id":"55555555-5555-5555-5555-555555555555","identityId":"us-
east-1:ee319396-fec4-424d-
aa19-71ee751624d1"}},"id":"bda76e80-4388-11e8-
a845-5902692b9264","partitionKey":"c9d4e9e5-d33f-4907-9a7a-
af03710fa50f","timestamp":1524111521129,"tags":{"region":"us-east-1"}}
2018-04-19 00:18:42 ... params: {"IdentityPoolId":"us-east-1:3a07e120-
f1d8-4c85-9c34-0f908f2a21a1","IdentityId":"us-east-1:ee319396-
fec4-424d-aa19-71ee751624d1","DatasetName":"things"}
2018-04-19 00:18:43 ...
{"Records":[{"Key":"thing","Value":"{\"id\":\"55555555-5555-5555-5555-5
55555555555\",\"name\":\"thing
five\",\"asOf\":1524111521129,\"identityId\":\"us-east-1:ee319396-
fec4-424d-
aa19-71ee751624d1\"}","SyncCount":1,"LastModifiedDate":"2018-04-19T04:1
8:42.978Z","LastModifiedBy":"123456789012","DeviceLastModifiedDate":"20
18-04-19T04:18:42.978Z"}]}
END ...
REPORT ... Duration: 340.94 ms    Billed Duration: 400 ms ... Max
Memory Used: 33 MB
```

14. Open the file named `index.html` in a browser and press the **Synchronize** button to retrieve the data from the materialized view:



15. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a `listener` function that consumes upstream events and populates a materialized view that is used by a Backend For Frontend service. This function is a stream processor, such as the one we discussed in the *Creating a stream processor* recipe in `Chapter 1`, *Getting Started with Cloud-Native*. The function performs a `filter` for the desired events and then transforms the data in a `map` step to the desired materialized view. The materialized view is optimized to support the requirements of the query needed by the BFF. Only the minimum necessary data is stored, and the optimal database type is used.

In this recipe, the database type is a Cognito dataset. A Cognito dataset is a good choice for a materialized view when network availability is intermittent, and thus an offline-first approach is needed to synchronize data to a user's devices. The data must also be specific to a user so that it can be targeted to the user based on the user's `identityId`. Due to the intermittent nature of connectivity, the `asOf` timestamp is included in the record so that the user can access the latency of the data.

Replaying events

One of the advantages of the Event Sourcing and data lake patterns is that they allow us to replay events when necessary to repair broken services and seed new services, and even new versions of a service. In this recipe, we will implement a utility that reads selected events from the data lake and applies them to a specified Lambda function.

Getting ready

Before starting this recipe, you will need the data lake that was created in the *Creating a data lake* recipe in this chapter. The data lake should contain events that were generated by working through the other recipes in this chapter.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/re
playing-events --path cncb-replaying-events
```

2. Navigate to the `cncb-replaying-events` directory with `cd cncb-replaying-events`.
3. Review the file named `./lib/replay.js` with the following content:

```
exports.command = 'replay [bucket] [prefix]'
exports.desc = 'Replay the events in [bucket] for [prefix]'

const _ = require('highland');
const lodash = require('lodash');
const aws = require('aws-sdk');
aws.config.setPromisesDependency(require('bluebird'));

exports.builder = {
    bucket: {
        alias: 'b',
    },
    prefix: {
        alias: 'p',
    },
    function: {
        alias: 'f',
```

```
    },
    dry: {
        alias: 'd',
        default: true,
        type: 'boolean'
    },
    region: {
        alias: 'r',
        default: 'us-east-1'
    },
}

exports.handler = (argv) => {
    aws.config.logger = process.stdout;
    aws.config.region = argv.region;

    const s3 = new aws.S3();
    const lambda = new aws.Lambda();

    paginate(s3, argv)
        .flatMap(obj => get(s3, argv, obj))
        .flatMap(event => invoke(lambda, argv, event))
        .collect()
        .each(list => console.log('count:', list.length));
}

const paginate = (s3, options) => {
    let marker = undefined;

    return _((push, next) => {
        const params = {
            Bucket: options.bucket,
            Prefix: options.prefix,
            Marker: marker // paging indicator
        };

        s3.listObjects(params).promise()
            .then(data => {
                if (data.IsTruncated) {
                    marker = lodash.last(data.Contents)['Key'];
                } else {
                    marker = undefined;
                }

                data.Contents.forEach(obj => {
                    push(null, obj);
                })
            })
```

```
            .catch(err => {
                push(err, null);
            })
            .finally(() => {
                if (marker) { // indicates more pages
                    next();
                } else {
                    push(null, _.nil);
                }
            })
    });
}

const get = (s3, options, obj) => {
    const params = {
        Bucket: options.b,
        Key: obj.Key
    };

    return _(
        s3.getObject(params).promise()
            .then(data => Buffer.from(data.Body).toString())
    )
        .split() // EOL we added in data lake recipe transformer
        .filter(line => line.length != 0)
        .map(JSON.parse);
}

const invoke = (lambda, options, event) => {
    let payload = {
        Records: [
            {
                kinesis: {
                    partitionKey:
event.kinesisRecordMetadata.partitionKey,
                    sequenceNumber:
event.kinesisRecordMetadata.sequenceNumber,
                    data:
Buffer.from(JSON.stringify(event.event)).toString('base64'),
                    kinesisSchemaVersion: '1.0',
                },
                eventID:
`${event.kinesisRecordMetadata.shardId}:${event.kinesisRecordMetadata.s
equenceNumber}`,
                eventName: 'aws:kinesis:record',
                eventSourceARN:
event.firehoseRecordMetadata.deliveryStreamArn,
                eventSource: 'aws:kinesis',
```

```
                eventVersion: '1.0',
                awsRegion: event.firehoseRecordMetadata.region,
            }
        ]
    };

    payload = Buffer.from(JSON.stringify(payload));

    const params = {
        FunctionName: options.function,
        InvocationType: options.dry ? 'DryRun' :
           payload.length <= 100000 ? 'Event' : 'RequestResponse',
        Payload: payload,
    };

    return _(
        lambda.invoke(params).promise()
    );
}
```

4. Install the dependencies with `npm install`.
5. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
6. Replay events with the following command:

```
$ node index.js replay -b cncb-data-lake-s3-john-bucket-396po814rlai -p
john-cncb-event-stream-s1 -f cncb-replaying-events-john-listener -dry
false
[AWS s3 200 0.288s 0 retries] listObjects({ Bucket: 'cncb-data-lake-s3-
john-bucket-396po814rlai',
  Prefix: 'john-cncb-event-stream-s1',
  Marker: undefined })
[AWS s3 200 0.199s 0 retries] getObject({ Bucket: 'cncb-data-lake-s3-
john-bucket-396po814rlai',
  Key: 'john-cncb-event-stream-s1/2018/04/08/03/cncb-data-lake-s3-john-
DeliveryStream-13N6LEC9XJ6DZ-3-2018-04-08-03-53-28-d79d6893-
aa4c-4845-8964-61256ffc6496' })
[AWS lambda 202 0.199s 0 retries] invoke({ FunctionName: 'cncb-
replaying-events-john-listener',
  InvocationType: 'Event',
  Payload: '***SensitiveInformation***' })
[AWS lambda 202 0.151s 0 retries] invoke({ FunctionName: 'cncb-
replaying-events-john-listener',
  InvocationType: 'Event',
  Payload: '***SensitiveInformation***' })
[AWS lambda 202 0.146s 0 retries] invoke({ FunctionName: 'cncb-
replaying-events-john-listener',
  InvocationType: 'Event',
```

```
  Payload: '***SensitiveInformation***' })
count: 3
```

7. Take a look at the logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
START ...
2018-04-17 23:43:14 ... event:
{"Records":[{"kinesis":{"partitionKey":"ccfd67c3-a266-4dec-9576-
ae5ea228a79c","sequenceNumber":"49583337208235522365774435506752843085880683263405588482","data":"...","kinesisSchemaVersion":"1.0"},"eventID":"shardId-000000000000:49583337208235522365774435506752843085880683263405588482","eventName":"aws:kinesis:record","eventSourceARN":"arn:aws:firehose:us-east-1:123456789012:deliverystream/cncb-data-lake-s3-john-DeliveryStream-13N6LEC9XJ6DZ","eventSource":"aws:kinesis","eventVersion":"1.0","awsRegion":"us-east-1"}]}
END ...
REPORT ... Duration: 10.03 ms    Billed Duration: 100 ms ... Max Memory
Used: 20 MB

...
```

8. Remove the stack once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we implement a **Command-Line Interface** (**CLI**) program that reads events from the data lake S3 bucket and sends them to a specific AWS Lambda function. When replaying events, we do not re-publish the events because this would broadcast the events to all subscribers. Instead, we want to replay events to a specific function to either repair the specific service or seed a new service.

When executing the program, we provide the name of the data lake `bucket` and the specific path `prefix` as arguments. The `prefix` allows us to replay only a portion of the events, such as a specific month, day, or hour. The program uses functional reactive programming with the `Highland.js` library. We use a `generator` function to page through the objects in the bucket and `push` each object down the stream. **Backpressure** is a major advantage of this programming approach, as we will discuss in `Chapter 8`, *Designing for Failure*. If we retrieved all the data from the bucket in a loop, as we would in the imperative programming style, then we would likely run out of memory and/or overwhelm the Lambda function and receive throttling errors.

Instead, we pull data through the stream. When downstream steps are ready for more work they pull the `next` piece of data. This triggers the generator function to paginate data from S3 when the program is ready for more data.

When storing events in the data lake bucket, Kinesis Firehose buffers the events until a maximum amount of time is reached or a maximum file size is reached. This buffering maximizes the write performance when saving the events. When transforming the data for these files, we delimited the events with an EOL character. Therefore, when we `get` a specific file, we leverage the Highland.js `split` function to stream each row in the file one at a time. The split function also supports **backpressure**.

For each event, we `invoke` the `function` specified in the command-line arguments. These functions are designed to listen for events from a Kinesis stream. Therefore, we must wrap each event in the Kinesis input format that these functions are expecting. This is one reason why we included the Kinesis metadata when saving the events to the data lake in the *Creating a data lake* recipe. To maximize throughput, we invoke the Lambda *asynchronously* with the `Event` InvocationType, provided that the payload size is within the limits. Otherwise, we invoke the Lambda *synchronously* with the `RequestReponse` InvocationType. We also leverage the Lambda `DryRun` feature so that we can see what events might be replayed before actually effecting the change.

Indexing the data lake

A data lake is a crucial design pattern for providing cloud-native systems with an audit trail of all the events in a system and for supporting the ability to replay events. In the *Creating a data lake* recipe, we implemented the S3 component of the data lake that provides high durability. However, a data lake is only useful if we can find the relevant data. In this recipe, we will index all the events in Elasticsearch so that we can search events for troubleshooting and business analytics.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/da
ta-lake-es --path cncb-data-lake-es
```

2. Navigate to the `cncb-data-lake-es` directory with `cd cncb-data-lake-es`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-data-lake-es

provider:
  name: aws
  runtime: nodejs8.10

plugins:
  - elasticsearch

functions:
  transformer:
    handler: handler.transform
    timeout: 120

resources:
  Resources:
    Domain:
      Type: AWS::Elasticsearch::Domain
      Properties:
        ...

    DeliveryStream:
      Type: AWS::KinesisFirehose::DeliveryStream
      Properties:
        DeliveryStreamType: KinesisStreamAsSource
        KinesisStreamSourceConfiguration:
          KinesisStreamARN: ${cf:cncb-event-stream-
${opt:stage}.streamArn}
          ...
        ElasticsearchDestinationConfiguration:
          DomainARN:
            Fn::GetAtt: [ Domain, DomainArn ]
          IndexName: events
          IndexRotationPeriod: OneDay
          TypeName: event
          BufferingHints:
            IntervalInSeconds: 60
            SizeInMBs: 50
          RetryOptions:
            DurationInSeconds: 60
          ...
          ProcessingConfiguration:
${file(includes.yml):ProcessingConfiguration}
```

```
    Bucket:
      Type: AWS::S3::Bucket

    ...

  Outputs:
    ...
    DomainEndpoint:
      Value:
        Fn::GetAtt: [ Domain, DomainEndpoint ]
    KibanaEndpoint:
      Value:
        Fn::Join:
          - ''
          - - Fn::GetAtt: [ Domain, DomainEndpoint ]
            - '/_plugin/kibana'
    ...
```

4. Review the file named `handler.js` with the following content:

```
exports.transform = (event, context, callback) => {
  const output = event.records.map((record, i) => {
    // store all available data
    const uow = {
      event: JSON.parse((Buffer.from(record.data,
'base64')).toString('utf8')),
      kinesisRecordMetadata: record.kinesisRecordMetadata,
      firehoseRecordMetadata: {
        deliveryStreamArn: event.deliveryStreamArn,
        region: event.region,
        invocationId: event.invocationId,
        recordId: record.recordId,
        approximateArrivalTimestamp:
record.approximateArrivalTimestamp,
      }
    };

    return {
      recordId: record.recordId,
      result: 'Ok',
      data: Buffer.from(JSON.stringify(uow),
'utf-8').toString('base64'),
    };
  });

  callback(null, { records: output });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:



Deploying an Elasticsearch domain can take upwards of 20 minutes.

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-data-lake-es@1.0.0 dp:lcl <path-to-your-workspace>/cncb-data-
lake-es
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  transformer: cncb-data-lake-es-john-transformer

Stack Outputs
DeliveryStream: cncb-data-lake-es-john-DeliveryStream-1ME9ZI78H3347
DomainEndpoint: search-cncb-da-domain-5qx46izjweyq-
oehy3i3euztbnog4juse3cmrs4.us-east-1.es.amazonaws.com
DeliveryStreamArn: arn:aws:firehose:us-
east-1:123456789012:deliverystream/cncb-data-lake-es-john-
DeliveryStream-1ME9ZI78H3347
KibanaEndpoint: search-cncb-da-domain-5qx46izjweyq-
oehy3i3euztbnog4juse3cmrs4.us-east-1.es.amazonaws.com/_plugin/kibana
DomainArn: arn:aws:es:us-east-1:123456789012:domain/cncb-da-
domain-5qx46izjweyq
...
```

9. Review the stack, function, and Elasticsearch domain in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d '{"type":"thing-
created"}'

{
 "ShardId": "shardId-000000000000",
 "SequenceNumber":
```

```
"49583655996852917476267785049074754815059037929823272962"
}
```



Allow the Firehose buffer time to process, as the buffer interval is 60 seconds.

11. Take a look at the `transformer` function logs:

```
$ sls logs -f transformer -r us-east-1 -s $MY_STAGE
```

12. Open Kibana using the preceding `KibanaEndpoint` output with protocol `https`.
13. Select the `Management` menu, set the index pattern to `events-*`, and press the `Next step` button.
14. Select `timestamp` as the `Time Filter field name`, and press `Create Index pattern`.
15. Select the `Discover` menu to view the current events in the index.
16. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The data lake is a valuable source of information. Elasticsearch is uniquely suited for indexing this coarse-grained time series information. **Kibana** is the data visualization plugin for Elasticsearch. Kibana is a great tool for creating dashboards containing statistics about the events in the data lake and to perform ad hoc searches to troubleshoot system problems based on the contents of the events.

In this recipe, we are using Kinesis Firehose because it performs the heavy lifting of writing the events to Elasticsearch. It provides buffering based on time and size, hides the complexity of the Elasticsearch bulk index API, provides error handling, and supports index rotation. In the custom `elasticsearch` Serverless plugin, we create the index template that defines the `index_patterns` and the `timestamp` field used to affect the index rotation.

This recipe defines one delivery stream, because in this cookbook, our stream topology consists of only one stream with `${cf:cncb-event-stream-${opt:stage}.streamArn}`. In practice, your topology will consist of multiple streams and you will define one Firehose delivery stream per Kinesis stream to ensure that all events are indexed.

Implementing bi-directional synchronization

Cloud-native systems are architectured to support the continuous evolution of the system. Upstream and downstream services are designed to be pluggable. New service implementations can be added without impacting related services. Furthermore, continuous deployment and delivery necessitate the ability to run multiple versions of a service side by side and synchronize data between the different versions. The old version is simply removed when the new version is complete and the feature is flipped on. In this recipe, we will enhance the *database-first* variant of the Event Sourcing pattern with the *latching* pattern to facilitate bi-directional synchronization without causing an infinite loop of events.

How to do it...

1. Create two projects from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/bi
-directional-sync --path cncb-1-bi-directional-sync

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch2/bi
-directional-sync --path cncb-2-bi-directional-sync
```

2. Review the file named `serverless.yml` with the following content in each project:

```
service: cncb-1-bi-directional-sync

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    SERVERLESS_PROJECT: ${self:service}
    ...

functions:
  command:
    handler: handler.command
  trigger:
    handler: handler.trigger
    events:
```

```
      - stream:
          type: dynamodb
          ...
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          ...
  query:
    handler: handler.query

resources:
  Resources:
    Table:
      ...
```

3. Review the file named `handler.js` with the following content:

```
module.exports.command = (request, context, callback) => {
  const thing = {
    id: uuid.v4(),
    latch: 'open',
    ...request,
  };

  ...
  db.put(params, callback);
};

module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .filter(forLatchOpen)
    .map(toEvent)
    .flatMap(publish)
    .collect()
    .toCallback(cb);
};

const forLatchOpen = e => e.dynamodb.NewImage.latch.S === 'open';

const toEvent = record => ({
  id: record.eventID,
  ...
  tags: {
    region: record.awsRegion,
    source: process.env.SERVERLESS_PROJECT
  },
```

```
  thing: ...,
});

...

module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forSourceNotSelf)
    .filter(forThingCrud)
    .map(toThing)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...
const forSourceNotSelf = e => e.tags.source !=
process.env.SERVERLESS_PROJECT;
...

const toThing = event => ({
  id: event.thing.new.id,
  ...
  latch: 'closed',
});

...
```

4. Navigate to the `cncb-1-bi-directional-sync` directory with `cd cncb-1-bi-directional-sync`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-1-bi-directional-sync@1.0.0 dp:lcl <path-to-your-
workspace>/cncb-1-bi-directional-sync
> sls deploy -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
```

```
functions:
  command: cncb-1-bi-directional-sync-john-command
  trigger: cncb-1-bi-directional-sync-john-trigger
  listener: cncb-1-bi-directional-sync-john-listener
  query: cncb-1-bi-directional-sync-john-query
```

9. Review the stack, functions, and table in the AWS Console.
10. Navigate to the `cncb-2-bi-directional-sync` directory with `cd cncb-2-bi-directional-sync`.
11. Repeat steps 5-9 for the `cncb-2-bi-directional-sync` project.
12. Navigate back to the `cncb-1-bi-directional-sync` directory with `cd cncb-1-bi-directional-sync`.
13. Invoke the `command` function to save data to the first service:

```
$ sls invoke -r us-east-1 -f command -s $MY_STAGE -d
'{"id":"77777777-7777-7777-7777-777777777777","name":"thing seven"}'
```

14. Take a look at the logs for the `command` and `trigger` functions:

```
$ sls logs -f command -r us-east-1 -s $MY_STAGE

START ...
2018-04-24 02:02:11 ... event:
{"id":"77777777-7777-7777-7777-777777777777","name":"thing seven"}
2018-04-24 02:02:11 ... params: {"TableName":"john-cncb-1-bi-
directional-sync-
things","Item":{"id":"77777777-7777-7777-7777-777777777777","latch":"op
en","name":"thing seven"}}
END ...
REPORT ... Duration: 146.90 ms    Billed Duration: 200 ms ... Max
Memory Used: 40 MB

$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
START ...
2018-04-24 02:02:13 ... event:
{"Records":[{"eventID":"494ec22686941c0d5ff56dee86df47dd","eventName":"
INSERT",...,"Keys":{"id":{"S":"77777777-7777-7777-7777-777777777777"}},
"NewImage":{"name":{"S":"thing
seven"},"id":{"S":"77777777-7777-7777-7777-777777777777"},"latch":{"S":
"open"}},...},...}]}
2018-04-24 02:02:13 ...
{"id":"494ec22686941c0d5ff56dee86df47dd","type":"thing-
created",...,"tags":{"region":"us-east-1","source":"cncb-1-bi-
directional-sync"},"thing":{"new":{"name":"thing
seven","id":"77777777-7777-7777-7777-777777777777","latch":"open"}}}
...
```

```
END ...
REPORT ... Duration: 140.20 ms    Billed Duration: 200 ms ... Max
Memory Used: 35 MB
```

15. Navigate to the `cncb-2-bi-directional-sync` directory with `cd cncb-2-bi-directional-sync`
16. Take a look at the logs for the `listener` and `trigger` functions:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE

$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
```

17. Invoke the `query` function to retrieve the synchronized data to the second service:

```
$ sls invoke -r us-east-1 -f query -s $MY_STAGE -d
77777777-7777-7777-7777-777777777777
```

18. Remove both stacks once you have finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Cloud-native systems are architected to evolve. Over time, the functional requirements will change and the technology options will improve. However, some changes are not incremental and/or do not support an immediate switch from one implementation to another. In these cases, it is necessary to have multiple versions of the same functionality running simultaneously. If these services produce data, then it is necessary to synchronize data changes between the services. This bi-directional synchronization will produce an infinite messaging loop if an appropriate **latching** mechanism is not employed.

This recipe builds on the database-first variant of the Event Sourcing pattern. A user of service one invokes the command function. The `command` function opens the `latch` by setting the latch on the domain object to `open`. The `trigger` function's `forLatchOpen` filter will only allow publishing an event when the latch is `open`, because the open latch indicates that the change originated in service one. The `listener` function's `forSourceNotSelf` filter in service one ignores the event because the `source` tag indicates that the event originates from service one. The `listener` function in service two closes the `latch` before saving the data by setting the latch on the domain object to `closed`. The `trigger` function in service two does not publish an event, because the `closed` latch indicates that the change did not originate in service two.

This same flow unfolds when the command originates in service two. You can add a third and fourth service and more, and all the services will remain in sync.

3
Implementing Autonomous Services

In this chapter, the following recipes will be covered:

- Implementing a GraphQL CRUD BFF
- Implementing a search BFF
- Implementing an analytics BFF
- Implementing an inbound External Service Gateway
- Implementing an outbound External Service Gateway
- Orchestrating collaboration between services
- Implementing a Saga

Introduction

In `Chapter 1`, *Getting Started with Cloud-Native*, we began our journey to understand why cloud-native is lean and autonomous. We focused on recipes that demonstrate how leveraging fully managed cloud services empower self-sufficient, full-stack teams to rapidly and continuously deliver innovation with confidence. In `Chapter 2`, *Applying the Event Sourcing and CQRS Patterns*, we worked through recipes that showcase how these patterns establish the bulkheads that enable the creation of autonomous services.

In this chapter, we bring all these foundational pieces together with recipes for implementing autonomous service patterns. In my book, `Cloud Native Development Patterns and Best Practices`, I discuss various approaches for decomposing a cloud-native system into bound, isolated, and autonomous services.

Every service should certainly have a bounded context and a single responsibility, but we can decompose services along additional dimensions as well. The life cycle of data is an important consideration for defining services, because the users, requirements, and persistence mechanisms will likely change as data ages. We also decompose services based on boundary and control patterns. Boundary services, such as a **Backend for Frontend** (**BFF**) or an **External Service Gateway** (**ESG**), interact with things that are external to the system, such as humans and other systems. Control services orchestrate the interactions between these decoupled boundary services. The recipes in this chapter demonstrate common permutations of these decomposition strategies.

Implementing a GraphQL CRUD BFF

The BFF pattern accelerates innovation because the team that implements the frontend also owns and implements the backend service that supports the frontend. This enables teams to be self-sufficient and unencumbered by competing demands for a shared backend service. In this recipe, we will create a CRUD BFF service that supports data at the beginning of its life cycle. The single responsibility of this service is authoring data for a specific bounded context. It leverages *database-first* Event Sourcing to publish domain events to downstream services. The service exposes a GraphQL-based API.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/bff-graphql-crud --path cncb-bff-graphql-crud
```

2. Navigate to the `cncb-bff-graphql-crud` directory with `cd cncb-bff-graphql-crud`.

3. Review the file named `serverless.yml` with the following content:

```
service: cncb-bff-graphql-crud

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...

functions:
  graphql:
    handler: handler.graphql
    events:
      - http:
          path: graphql
          method: post
          cors: true
    environment:
      TABLE_NAME:
        Ref: Table
  graphiql:
    handler: handler.graphiql
    ...
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: dynamodb
          ...

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      ...
```

4. Review the file named `./schema/thing/typedefs.js` with the following content:

```
module.exports = `
  type Thing {
    id: String!
    name: String
    description: String
  }

  type ThingConnection {
```

```
    items: [Thing!]!
    cursor: String
  }

  extend type Query {
    thing(id: String!): Thing
    things(name: String, limit: Int, cursor: String): ThingConnection
  }

  input ThingInput {
    id: String
    name: String!
    description: String
  }

  extend type Mutation {
    saveThing(
      input: ThingInput
    ): Thing
    deleteThing(
      id: ID!
    ): Thing
  }
`;
```

5. Review the file named `./schema/thing/resolvers.js` with the following content:

```
module.exports = {
  Query: {
    thing(_, { id }, ctx) {
      return ctx.models.Thing.getById(id);
    },
    things(_, { name, limit, cursor }, ctx) {
      return ctx.models.Thing.queryByName(name, limit, cursor);
    },
  },
  Mutation: {
    saveThing: (_, { input }, ctx) => {
      return ctx.models.Thing.save(input.id, input);
    },
    deleteThing: (_, args, ctx) => {
      return ctx.models.Thing.delete(args.id);
    },
  },
};
```

6. Review the file named `handler.js` with the following content:

```
...
const { graphqlLambda, graphiqlLambda } = require('apollo-server-
lambda');
const schema = require('./schema');
const Connector = require('./lib/connector');
const { Thing } = require('./schema/thing');

module.exports.graphql = (event, context, cb) => {
  graphqlLambda(
    (event, context) => {
      return {
        schema, context: { models: {
            Thing: new Thing( new Connector(process.env.TABLE_NAME) )
          } }
      };
    }
  )(event, context, (error, output) => {
    cb(error, ...);
  });
};
. . .
```

7. Install the dependencies with `npm install`.
8. Run the tests with `npm test -- -s $MY_STAGE`.
9. Review the contents generated in the `.serverless` directory.
10. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-bff-graphql-crud@1.0.0 dp:lcl <path-to-your-workspace>/cncb-bff-
graphql-crud
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  POST -
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/john/graphql
  GET -
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/john/graphiql
functions:
  graphql: cncb-bff-graphql-crud-john-graphql
  graphiql: cncb-bff-graphql-crud-john-graphiql
```

```
  trigger: cncb-bff-graphql-crud-john-trigger

Stack Outputs
...
ServiceEndpoint:
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/john
...
```

11. Review the stack in the AWS Console.
12. Invoke the function with the following `curl` commands:



Make sure to replace the API Gateway ID (that is, `ac0n4oyzm6`) in the endpoints with the value output during deployment.

```
$ curl -X POST -H 'Content-Type: application/json' -d
'{"query":"mutation { saveThing(input: { id:
\"33333333-1111-1111-1111-000000000000\", name: \"thing1\",
description: \"This is thing one of two.\" }) { id } }"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "saveThing" : {
         "id" : "33333333-1111-1111-1111-000000000000"
      }
   }
}

$ curl -X POST -H 'Content-Type: application/json' -d
'{"query":"mutation { saveThing(input: { id:
\"33333333-1111-1111-2222-000000000000\", name: \"thing2\",
description: \"This is thing two of two.\" }) { id } }"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "saveThing" : {
         "id" : "33333333-1111-1111-2222-000000000000"
      }
   }
}

$ curl -X POST -H 'Content-Type: application/json' -d '{"query":"query
```

```
{ thing(id: \"33333333-1111-1111-1111-000000000000\") { id name
description }}"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "thing" : {
         "description" : "This is thing one of two.",
         "id" : "33333333-1111-1111-1111-000000000000",
         "name" : "thing1"
      }
   }
}

$ curl -X POST -H 'Content-Type: application/json' -d '{"query":"query
{ things(name: \"thing\") { items { id name } cursor }}"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "things" : {
         "items" : [
            {
               "id" : "33333333-1111-1111-1111-000000000000",
               "name" : "thing1"
            },
            {
               "name" : "thing2",
               "id" : "33333333-1111-1111-2222-000000000000"
            }
         ],
         "cursor" : null
      }
   }
}

$ curl -X POST -H 'Content-Type: application/json' -d '{"query":"query
{ things(name: \"thing\", limit: 1) { items { id name } cursor }}"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "things" : {
         "items" : [
            {
```

```
                "id" : "33333333-1111-1111-1111-000000000000",
                "name" : "thing1"
             }
          ],
          "cursor" :
"eyJpZCI6IjMzMzMzMzMzLTExMTEtMTExMS0xMTExLTAwMDAwMDAwMDAwMCJ9"
       }
    }
}
```

```
$ curl -X POST -H 'Content-Type: application/json' -d '{"query":"query
{ things(name: \"thing\", limit: 1, cursor:\"CURSOR VALUE FROM PREVIOUS
RESPONSE\") { items { id name } cursor }}"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp
```

```
{
   "data" : {
      "things" : {
         "items" : [
            {
               "id" : "33333333-1111-1111-2222-000000000000",
               "name" : "thing2"
            }
         ],
         "cursor" :
"eyJpZCI6IjMzMzMzMzMzLTExMTEtMTExMS0yMjIyLTAwMDAwMDAwMDAwMCJ9"
       }
    }
}
```

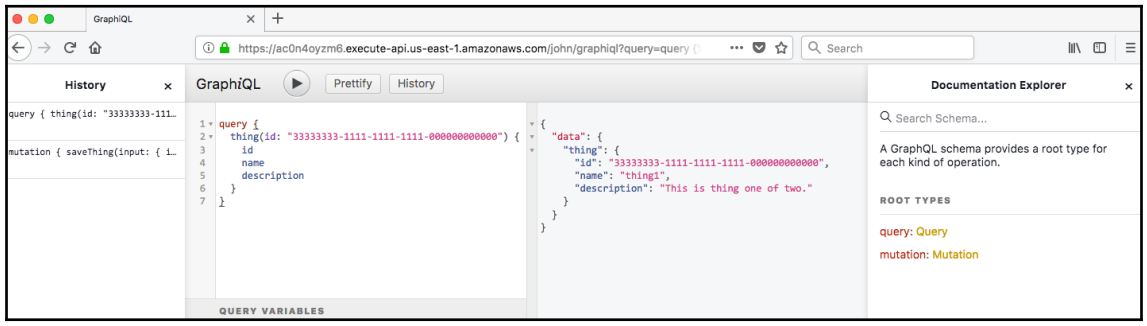
```
$ curl -X POST -H 'Content-Type: application/json' -d
'{"query":"mutation { deleteThing( id:
\"33333333-1111-1111-1111-000000000000\" ) { id } }"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp
```

```
{
   "data" : {
      "deleteThing" : {
         "id" : "33333333-1111-1111-1111-000000000000"
      }
   }
}
```

```
$ curl -X POST -H 'Content-Type: application/json' -d
'{"query":"mutation { deleteThing( id:
\"33333333-1111-1111-2222-000000000000\" ) { id } }"}'
```

```
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "deleteThing" : {
         "id" : "33333333-1111-1111-2222-000000000000"
      }
   }
}

$ curl -X POST -H 'Content-Type: application/json' -d '{"query":"query
{ things(name: \"thing\") { items { id } }}"}'
https://ac0n4oyzm6.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "things" : {
         "items" : []
      }
   }
}
```

13. Perform the same mutations and queries using GraphiQL by using the endpoint output during deployment:



14. Take a look at the `trigger` function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
```

15. Review the events collected in the data lake bucket.
16. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe builds on the *Applying the database-first variant of the Event Sourcing pattern with DynamoDB* recipe in `Chapter 2`, *Applying the Event Sourcing and CQRS Patterns* by exposing the ability to author data in a bounded context through a GraphQL API. *GraphQL* is becoming increasingly popular because of the flexibility of the resulting API and the power of client libraries, such as the Apollo Client. We implement a single `graphql` function to support our API and then add the necessary functionality through the `schema`, `resolvers`, `models`, and `connectors`.

The GraphQL schema is where we define our `types`, `queries`, and `mutations`. In this recipe, we can query `thing` types by ID and by name, and `save` and `delete`. The `resolvers` map the GraphQL requests to `model` objects that encapsulate the business logic. The `models`, in turn, talk to `connectors` that encapsulate the details of the database API. The `models` and `connectors` are registered with the `schema` in the `handler` function with a very simple but effective form of constructor-based dependency injection. We don't use dependency injection very often in cloud-native, because functions are so small and focused that it is overkill and can impede performance. With GraphQL, this simple form is very effective for facilitating testing. The `Graphiql` tool is very useful for exposing the self-documenting nature of APIs.

The single responsibility of this service is authoring data and publishing the events, using database-first Event Sourcing, for a specific bounded context. The code within the service follows a very repeatable coding convention of `types`, `resolvers`, `models`, `connectors`, and `triggers`. As such, it is very easy to reason about the correctness of the code, even as the number of business domains in the service increases. Therefore, it is reasonable to have a larger number of domains in a single authoring BFF services, so long as the domains are cohesive, part of the same bounded context, and authored by a consistent group of users.

Implementing a search BFF

In the *Implementing a GraphQL CRUD BFF* recipe, we discussed how the *BFF* pattern accelerates innovation. We have also discussed how different user groups interact with data at different stages in the data life cycle, and how different persistent mechanisms are more appropriate at the different stages. In this recipe, we will create a BFF service that supports the read-only consumption of data. The single responsibility of this service is *indexing* and retrieving data for a specific bounded context. It applies the *CQRS* pattern to create two *materialized views* that work in tandem, one in Elasticsearch and another in S3. The service exposes a RESTful API.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/bff-rest-search --path cncb-bff-rest-search
```

2. Navigate to the `cncb-bff-rest-search` directory with `cd cncb-bff-rest-search`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-bff-rest-search

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    BUCKET_NAME:
      Ref: Bucket
    DOMAIN_ENDPOINT:
      Fn::GetAtt: [ Domain, DomainEndpoint ]
...

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
  trigger:
    handler: handler.trigger
    events:
      - sns:
          arn:
            Ref: BucketTopic
          topicName: ${self:service}-${opt:stage}-trigger
  search:
    handler: handler.search
    events:
      - http:
          path: search
          method: get
```

```
          cors: true

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket
      ...
      Properties:
        NotificationConfiguration:
          TopicConfigurations:
            - Event: s3:ObjectCreated:Put
              Topic:
                Ref: BucketTopic
    BucketTopic:
      Type: AWS::SNS::Topic
    ...

    Domain:
      Type: AWS::Elasticsearch::Domain
  ...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forThingCreated)
    .map(toThing)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

...

module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .flatMap(messagesToTriggers)
    .flatMap(get)
    .map(toSearchRecord)
    .flatMap(index)
    .collect()
    .toCallback(cb);
};

const messagesToTriggers = r => _(JSON.parse(r.Sns.Message).Records);

const get = (trigger) => {
```

```
  const params = {
    Bucket: trigger.s3.bucket.name,
    Key: trigger.s3.object.key,
  };

  const s3 = new aws.S3();
  return _(
    s3.getObject(params).promise()
      .then(data => ({
        trigger: trigger,
        thing: JSON.parse(Buffer.from(data.Body)),
      }))
  );
};

const toSearchRecord = uow => ({
  id: uow.thing.id,
  name: uow.thing.name,
  description: uow.thing.description,
  url:
`https://s3.amazonaws.com/${uow.trigger.s3.bucket.name}/${uow.trigger.s
3.object.key}`,
});

...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-bff-rest-search@1.0.0 dp:lcl <path-to-your-workspace>/cncb-bff-
rest-search
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
https://n31t5dsei8.execute-api.us-east-1.amazonaws.com/john/search
functions:
  listener: cncb-bff-rest-search-john-listener
  trigger: cncb-bff-rest-search-john-trigger
```

```
  search: cncb-bff-rest-search-john-search

Stack Outputs
...
BucketArn: arn:aws:s3:::cncb-bff-rest-search-john-bucket-1xjkvimbjtfj2
BucketName: cncb-bff-rest-search-john-bucket-1xjkvimbjtfj2
TopicArn: arn:aws:sns:us-east-1:123456789012:cncb-bff-rest-search-john-
trigger
DomainEndpoint: search-cncb-bf-domain-xavolfersvjd-
uotz6ggdqhhwk7irxhnkjl26ay.us-east-1.es.amazonaws.com
DomainName: cncb-bf-domain-xavolfersvjd
...
ServiceEndpoint:
https://n31t5dsei8.execute-api.us-east-1.amazonaws.com/john
...
```

9. Review the stack and resources in the AWS Console.
10. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
three","id":"33333333-2222-0000-1111-111111111111"}}}'

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583553996455686705785668952918833314346020725338406914"
}
```

11. Invoke the following `curl` commands, after updating the `API-ID` and `BUCKET-SUFFIX`, to search the data and retrieve the detailed data from `S3`:

```
$ curl
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/search?q
=three | json_pp

[
   {
      "id" : "33333333-2222-0000-1111-111111111111",
      "url" :
"https://s3.amazonaws.com/cncb-bff-rest-search-john-bucket-1xjkvimbjtfj
2/things/33333333-2222-0000-1111-111111111111",
      "name" : "thing three"
   }
]

$ curl
```

```
https://s3.amazonaws.com/cncb-bff-rest-search-$MY_STAGE-bucket-<BUCKET-
SUFFIX>/things/33333333-2222-0000-1111-111111111111 | json_pp

{
   "asOf" : 1526026359761,
   "name" : "thing three",
   "id" : "33333333-2222-0000-1111-111111111111"
}
```

12. Take a look at the trigger function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe combines and builds on the *Creating a materialized view in S3* and the *Creating a materialized view in Elasticsearch* recipes to create a highly scalable, efficient, and cost-effective read-only view of the data in a bounded context. First, the `listener` function atomically creates the materialized view in *S3*. The S3 `Bucket` is configured to send events to a **Simple Notification Service** (**SNS**) topic called `BucketTopic`. We use SNS to deliver the S3 events because only a single observer can consume S3 events, while SNS, in turn, can deliver to any number of observers. Next, the `trigger` function atomically indexes the data in the *Elasticsearch* `Domain` and includes the `url` to the materialized view in S3.

The RESTful search service exposed by the API Gateway can explicitly scale to meet demand and efficiently search a large amount of indexed data. The detailed data can then be cost-effectively retrieved from S3, based on the returned URL, without the need to go through the API Gateway, a function, and the database. We create the data in S3 first and then index the data in Elasticsearch to ensure that the search results do not include data that has not been successfully stored in S3.

Implementing an analytics BFF

In the *Implementing a GraphQL CRUD BFF* recipe, we discussed how the *BFF* pattern accelerates innovation. We have also discussed how different user groups interact with data at different stages in the data life cycle, and how different persistent mechanisms are more appropriate at the different stages. In this recipe, we will create a BFF service that provides statistics about the life cycle of data. The single responsibility of this service is accumulating and aggregating metrics about data in a specific bounded context. It applies the Event Sourcing pattern to create a *micro event store* that is used to continuously calculate a *materialized view* of the metrics. The service exposes a RESTful API.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/bff-rest-analytics --path cncb-bff-rest-analytics
```

2. Navigate to the `cncb-bff-rest-analytics` directory with `cd cncb-bff-rest-analytics`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-bff-rest-analytics

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    EVENTS_TABLE_NAME:
      Ref: Events
    VIEW_TABLE_NAME:
      Ref: View

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
```

```
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [ Events, StreamArn ]
          ...
  query:
    handler: handler.query
    events:
      - http:
          ...

resources:
  Resources:
    Events:
      Type: AWS::DynamoDB::Table
      Properties:
        ...
        KeySchema:
          - AttributeName: partitionKey
            KeyType: HASH
          - AttributeName: timestamp
            KeyType: RANGE
        TimeToLiveSpecification:
          AttributeName: ttl
          Enabled: true
        ...
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
    View:
      Type: AWS::DynamoDB::Table
      Properties:
        ...
        KeySchema:
          - AttributeName: userId
            KeyType: HASH
          - AttributeName: yearmonth
            KeyType: RANGE
        ...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(byType)
```

```
    .flatMap(putEvent)
    .collect()
    .toCallback(cb);
};

...
const byType = event => event.type.match(/.+/); // any

const putEvent = (event) => {
  const params = {
    TableName: process.env.EVENTS_TABLE_NAME,
    Item: {
      partitionKey: event.partitionKey,
      timestamp: event.timestamp,
      event: event,
      ttl: moment(event.timestamp).add(1, 'h').unix()
    }
  };

  const db = new aws.DynamoDB.DocumentClient();
  return _(db.put(params).promise());
};

module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .flatMap(getMicroEventStore)
    .flatMap(store => _(store) // sub-stream
      .reduce({}, count)
      .flatMap(putCounters)
    )
    .collect()
    .toCallback(cb);
};

const getMicroEventStore = (record) => {
  ...
}

const count = (counters, cur) => {
  return Object.assign(
    {
      userId: cur.partitionKey,
      yearmonth: moment(cur.timestamp).format('YYYY-MM'),
    },
    counters,
    {
      total: counters.total ? counters.total + 1 : 1,
      [cur.event.type]: counters[cur.event.type] ?
```

```
counters[cur.event.type] + 1 : 1,
    }
  );
  ;
}

const putCounters = counters => {
  ...
};

module.exports.query = (event, context, cb) => {
  ...
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-bff-rest-analytics@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
bff-rest-analytics
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
https://efbildhw0h.execute-api.us-east-1.amazonaws.com/john/query
functions:
  listener: cncb-bff-rest-analytics-john-listener
  trigger: cncb-bff-rest-analytics-john-trigger
  query: cncb-bff-rest-analytics-john-query

Stack Outputs
...
ServiceEndpoint:
https://efbildhw0h.execute-api.us-east-1.amazonaws.com/john
...
```

9. Review the stack and resources in the AWS Console.
10. Publish several events from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d
'{"type":"purple","partitionKey":"33333333-3333-1111-1111-111111111111"
}'

$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d
'{"type":"orange","partitionKey":"33333333-3333-1111-1111-111111111111"
}'

$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d
'{"type":"green","partitionKey":"33333333-3333-1111-2222-111111111111"}
'

$ sls invoke -r us-east-1 -f publish -s $MY_STAGE -d
'{"type":"green","partitionKey":"33333333-3333-1111-2222-111111111111"}
'
```

11. Invoke the following `curl` command, after updating the `<API-ID>`, to query the analytics:

```
$ curl
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/query |
json_pp

[
   {
      "userId" : "33333333-3333-1111-1111-111111111111",
      "yearmonth" : "2018-05",
      "purple" : 1,
      "orange" : 1,
      "total" : 2
   },
   {
      "userId" : "33333333-3333-1111-2222-111111111111",
      "yearmonth" : "2018-05",
      "green" : 2,
      "total" : 2
   }
]
```

12. Take a look at the `trigger` function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe combines and builds on the *Creating a micro event store* and the *Creating a materialized view in DynamoDB* recipes in `Chapter 2`, *Applying the Event Sourcing and CQRS Patterns* to create an advanced materialized view that counts events by `type`, `user`, `month`, and `year`. The service employs two DynamoDB tables, the micro event store, and the materialized view. The `HASH` key for the event store is the `partitionKey`, which contains the `userId` so that we can correlate events by the user. The range key is the `timestamp` so that we can collate the events and query by month. The hash key for the view table is also `userId`, and the range key is `monthyear`, so that we can retrieve the statistics by user, month, and year. In this example, we are counting all events, but in a typical solution, you would be filtering `byType` for a specific set of event types.

The `listener` function performs the crucial job of *filtering*, *correlating,* and *collating* the events into the micro event store, but the real interesting logic in this recipe is in the `trigger` function. The logic is based on the concepts of the *ACID 2.0* transaction model. **ACID** 2.0 stands for **Associative, Commutative, Idempotent, and Distributed**. In essence, this model allows us to arrive at the same, correct answer, regardless of whether or not the events arrive in the correct order or even if we receive the same events multiple times. Our hash and range key in the micro event store handles the idempotency. For each new key, we recalculate the materialized view by querying the event store based on the context of the new event, and performing the calculation based on the latest known data. If an event arrives out of order, it simply triggers a recalculation. In this specific example, the end user would expect the statistics to eventually become consistent by the end of the month or shortly thereafter.

The calculations can be arbitrarily complex. The calculation is performed in memory and the results of the micro event store query can be sliced and diced in many different ways. For this recipe, the `reduce` method on the stream is perfect for counting. It is important to note that the `sub-stream` ensures that the count is performed by `userId`, because that was the hash key of the results returned from the event store. The results are stored in the materialized view as a JSON document so that they can be retrieved efficiently.

The **TimeToLive** (**TTL**) feature is set up on the events table. This feature can be used to keep the event store from growing unbounded, but it can also be used to trigger periodic rollup calculations. I set TTL to one hour so that you can see it execute if you wait long enough, but you would typically set this to a value suitable for your calculations, on the order of a month, quarter, or year.

Implementing an inbound External Service Gateway

The **External Service Gateway** (**ESG**) pattern provides an anti-corruption layer between a cloud-native system and any external services that it interacts with. Each gateway acts as a bridge to exchange events between the system and a specific external system. In this recipe, we will create an ESG service that allows events to flow inbound from an external service. The single responsibility of this service is to encapsulate the details of the external system. The service exposes a RESTful webhook to the external system. The external events are transformed into an internal format and published using *event-first* Event Sourcing.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/es
g-inbound --path cncb-esg-inbound
```

2. Navigate to the `cncb-esg-inbound` directory with `cd cncb-esg-inbound`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-esg-inbound

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...

functions:
  webhook:
    handler: handler.webhook
    events:
```

```
      - http:
          path: webhook
          method: post
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}
```

4. Review the file named `handler.js` with the following content:

```
module.exports.webhook = (request, context, callback) => {
  const body = JSON.parse(request.body);

  const event = {
    type: `issue-${body.action}`,
    id: request.headers['X-GitHub-Delivery'],
    partitionKey: String(body.issue.id),
    timestamp: Date.parse(body.issue['updated_at']),
    tags: {
      region: process.env.AWS_REGION,
      repository: body.repository.name,
    },
    issue: body, // canonical
    raw: request
  };

  ...

  kinesis.putRecord(params, (err, resp) => {
    const response = {
      statusCode: err ? 500 : 200,
    };
    callback(null, response);
  });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.

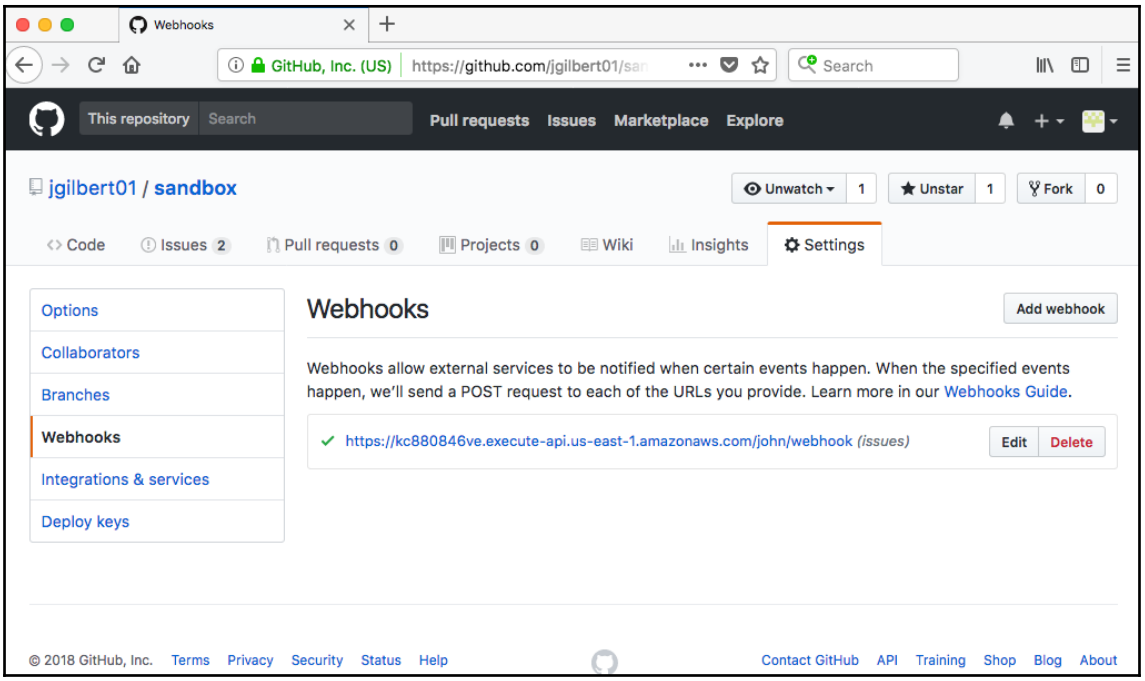
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-esg-inbound@1.0.0 dp:lcl <path-to-your-workspace>/js-cloud-
native-cookbook/workspace/cncb-esg-inbound
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  POST -
https://kc880846ve.execute-api.us-east-1.amazonaws.com/john/webhook
functions:
  webhook: cncb-esg-inbound-john-webhook

Stack Outputs
...
ServiceEndpoint:
https://kc880846ve.execute-api.us-east-1.amazonaws.com/john
...
```

9. Review the stack and resources in the AWS Console.
10. Set up a webhook in your GitHub project:
    1. Set the Payload URL to the endpoint of your webhook
    2. Set the content type to `application/json`
    3. Set the secret to a random value
    4. Select just the `Issues` events checkbox



For detailed instructions on creating a GitHub webhook, see `https://developer.github.com/webhooks/creating`.



11. Create and/or update one or more issues in your GitHub project to trigger the webhook.
12. Take a look at the `webhook` function logs:

```
$ sls logs -f webhook -r us-east-1 -s $MY_STAGE
```

13. Review the events in the data lake.
14. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

I chose to use GitHub as the external system in this recipe because it is freely available to everyone and representative of typical requirements. In this recipe, our inbound ESG service needs to provide an API that will be invoked by the external system and conforms to the signature of the external system's webhook. We implement this webhook using the API Gateway and a `webhook` function. The single responsibility of this function is to transform the external event to an internal event and atomically publish it using event-first Event Sourcing.

Note that the external event ID is used as the internal event ID to provide for idempotency. The external event data is included in the internal event in its raw format so that it can be recorded as an audit in the data lake. The external format is also transformed in an internal canonical format to support the pluggability of different external systems. The logic in an inbound ESG service is intentionally kept simple to minimize the chance of errors and help ensure the atomic exchange of the events between the systems.

Implementing an outbound External Service Gateway

In the *Implementing an inbound External Service Gateway* recipe, we discussed how the ESG pattern provides an anti-corruption layer between the cloud-native system and its external dependencies. In this recipe, we will create an ESG service that allows events to flow outbound to an external service. The single responsibility of this service is to encapsulate the details of the external system. The service applies the CQRS pattern. The internal events are transformed to the external format and forwarded to the external system via its API.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

You will need a GitHub account and a repository. I recommend creating a repository named `sandbox`. Use the following command to create a GitHub personal access token, or follow the instructions in the GitHub UI:

```
curl https://api.github.com/authorizations \
--user "your-github-id" \
--data '{"scopes":["repo"],"note":"recipe"}'
```

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/esg-outbound --path cncb-esg-outbound
```

2. Navigate to the `cncb-esg-outbound` directory with `cd cncb-esg-outbound`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-esg-outbound

provider:
  name: aws
  runtime: nodejs8.10
  environment:
    REPO: enter-your-github-project
    OWNER: enter-your-github-id
    TOKEN: enter-your-github-token

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
```

4. Update the `REPO`, `OWNER`, and `TOKEN` environment variables in the `serverless.yml` file.
5. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(byType)
    .flatMap(post)
    .collect()
    .toCallback(cb);
};

...
const byType = event => event.type === 'issue-created';

const post = event => {
  // transform internal to external
  const body = {
    title: event.issue.new.title,
    body: event.issue.new.description,
  };

  return _(
fetch(`https://api.github.com/repos/${process.env.OWNER}/${process.env.
```

```
REPO}/issues`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${process.env.TOKEN}`
      },
      body: JSON.stringify(body)
    })
  );
};
```

6. Install the dependencies with `npm install`.
7. Run the tests with `npm test -- -s $MY_STAGE`.
8. Review the contents generated in the `.serverless` directory.
9. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-esg-outbound@1.0.0 dp:lcl <path-to-your-workspace>/cncb-esg-
outbound
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-esg-outbound-john-listener
...
```

10. Review the stack and resources in the AWS Console.
11. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -d '{"type":"issue-
created","issue":{"new":{"title":"issue one","description":"this is
issue one.","id":"33333333-55555-1111-1111-111111111111"}}}'

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583655996852917476267887119095157508589012871374962690"
}
```

12. Confirm that the issue was created in your GitHub project.
13. Take a look at the `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
```

14. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

I chose to use GitHub as the external system in this recipe because it is freely available to everyone and its API is representative of typical requirements. One of the major details that are encapsulated by an ESG service is the security credentials needed to access the external API. In this recipe, we must create and secure a long-lived *personal access token* and include it as an authorization header in every API request. The details of how to secure a token are out of scope for this recipe, however, a service such as AWS Secret Manager is typically employed. For this recipe, the token is stored as an environment variable.

The `listener` function consumes the desired events, transforms them into the external format, and atomically invokes the external API. That is the limit of the responsibility of an ESG service. These services effectively make external services look like any other service in the system, while also encapsulating the details so that these external dependencies can be easily switched in the future. The transformation logic can become complex. The latching technique, discussed in the *Implementing bi-directional synchronization* recipe, may come into play, as well as the need to cross-reference external IDs to internal IDs. In many cases, the external data can be thought of as a materialized view, in which case the *micro event store* techniques may be useful. In a system that is offered as a service, an ESG service would provide your own outbound webhook feature.

Orchestrating collaboration between services

Autonomous cloud-native services perform all inter-service communication asynchronously via streams to decouple upstream services from downstream services. Although the upstream and downstream services are not directly coupled to each other, they are coupled to the event types that they produce and consume. The *Event Orchestration* control pattern acts as a *mediator* to completely decouple event producers from event consumers by translating between event types.

In this recipe, we will create a control service that orchestrates the interaction between two boundary services. The single responsibility of this service is to encapsulate the details of the collaboration. The upstream events are transformed to the event types' expected downstream, and published using *event-first* Event Sourcing.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/ev
ent-orchestration --path cncb-event-orchestration
```

2. Navigate to the `cncb-event-orchestration` directory with `cd cncb-event-orchestration`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-event-orchestration

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...

functions:
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}
```

4. Review the file named `handler.js` with the following content:

```
const transitions = [
  {
    filter: 'order-submitted',
    emit: (uow) => ({
      id: uuid.v1(),
      type: 'make-reservation',
      timestamp: Date.now(),
```

```
      partitionKey: uow.event.partitionKey,
      reservation: {
        sku: uow.event.order.sku,
        quantity: uow.event.order.quantity,
      },
      context: {
        order: uow.event.order,
        trigger: uow.event.id
      }
    })
  },
  {
    filter: 'reservation-confirmed',
    emit: (uow) => ({
      id: uuid.v1(),
      type: 'update-order-status',
      timestamp: Date.now(),
      partitionKey: uow.event.partitionKey,
      order: {
        status: 'reserved',
      },
      context: {
        reservation: uow.event.reservation,
        order: uow.event.context.order,
        trigger: uow.event.id
      }
    })
  },
];

module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(onTransitions)
    .flatMap(toEvents)
    .flatMap(publish)
    .collect()
    .toCallback(cb);
};

const recordToUow = r => ({
  record: r,
  event: JSON.parse(Buffer.from(r.kinesis.data, 'base64')),
});

const onTransitions = uow => {
  // find matching transitions
  uow.transitions = transitions.filter(trans => trans.filter ===
```

```
uow.event.type);

  // proceed forward if there are any matches
  return uow.transitions.length > 0;
};

const toEvents = uow => {
  // create the event to emit for each matching transition
  return _(uow.transitions.map(t => t.emit(uow)));
};

const publish = event => {
  . . .
}
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-event-orchestration@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
event-orchestration
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  listener: cncb-event-orchestration-john-listener
...
```

9. Review the stack and resources in the AWS Console.
10. Publish these events from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -p ../cncb-event-
orchestration/data/order.json

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583655996852917476267896339723499436825420846818394114"
}
```

```
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -p ../cncb-event-
orchestration/data/reservation.json

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49583655996852917476267896340117609254019790713686851586"
}
```

11. Take a look at the `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
```

12. Review the events in the data lake.
13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This *control* service has a single *stream processor* function that listens for specific events and reacts by emitting more events using event-first Event Sourcing. The events it listens for are described in the `transitions` metadata, which essentially defines the state machine of a long-lived business process. Each business process is implemented as an autonomous control service that orchestrates the collaboration between a set of completely decoupled boundary services. Each *boundary* service involved in the collaboration defines the set of events it produces and consumes independently of the other services. The control service provides the glue that brings these services together to deliver a higher value outcome.

The *downstream* services that are triggered by the emitted events do have one requirement that they must support when defining their incoming and outgoing event types. The incoming event types must accept the `context` element as an opaque set of data and pass the context data along in the outgoing event. A downstream service can leverage the context data, but should not explicitly change the context data. The context data allows the control service to correlate the events in a specific collaboration instance without needing to explicitly store and retrieve data. However, a control service could maintain its own *micro event store* to facilitate complex transition logic, such as joining multiple parallel flows back together before advancing.

Implementing a Saga

The Saga pattern is a solution to long-lived transactions that is based on eventual consistency and compensating transactions. It was first discussed in a paper by Hector Garcia-Molina and Kenneth Salem (`https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf`). Each step in a long-lived transaction is atomic. When a downstream step fails, it produces a violation event. The upstream services react to the violation event by performing a compensating action. In this recipe, we will create a service that submits data for downstream processing. The service also listens for a violation event and takes corrective action.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch3/saga --path cncb-saga
```

2. Navigate to the `cncb-saga` directory with `cd cncb-sage`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-saga

provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    ...
  environment:
    TABLE_NAME:
      Ref: Table

functions:
  submit:
    handler: handler.submit
  trigger:
    handler: handler.trigger
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [ Table, StreamArn ]
          ...
```

```
    environment:
      STREAM_NAME: ${cf:cncb-event-stream-${opt:stage}.streamName}
  listener:
    handler: handler.listener
    events:
      - stream:
          type: kinesis
          arn: ${cf:cncb-event-stream-${opt:stage}.streamArn}
          ...
  query:
    handler: handler.query

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${opt:stage}-${self:service}-orders
        ...
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
```

4. Review the file named `handler.js` with the following content:

```
...

module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToEvent)
    .filter(forReservationViolation)
    .flatMap(compensate)
    .collect()
    .toCallback(cb);
};

const forReservationViolation = e => e.type === 'reservation-
violation';

const compensate = event => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Key: {
      id: event.context.order.id
    },
    AttributeUpdates: {
      status: { Action: 'PUT', Value: 'cancelled' }
    },
  };
```

```
  const db = new aws.DynamoDB.DocumentClient();
  return _(db.update(params).promise());
};

...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-saga@1.0.0 dp:lcl <path-to-your-workspace>/cncb-saga
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  submit: cncb-saga-john-submit
  trigger: cncb-saga-john-trigger
  listener: cncb-saga-john-listener
  query: cncb-saga-john-query
...
```

9. Review the stack and resources in the AWS Console.
10. Invoke the `submit` and `query` functions with the following commands:

```
$ sls invoke -f submit -r us-east-1 -s $MY_STAGE -p data/order.json

$ sls invoke -f query -r us-east-1 -s $MY_STAGE -d
33333333-7777-1111-1111-111111111111
{
    "Item": {
        "quantity": 1,
        "id": "33333333-7777-1111-1111-111111111111",
        "sku": "1",
        "status": "submitted"
    }
}
```

11. Publish a violation event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -p ../cncb-
saga/data/reservation.json

{
    "ShardId": "shardId-000000000000",
    "SequenceNumber":
"49584174522005480245492626573048465901488330636951289858"
}
```

12. Invoke the `query` function again with the following command, and note the updated `status`:

```
$ sls invoke -r us-east-1 -f query -s $MY_STAGE -d
33333333-7777-1111-1111-111111111111

{
    "Item": {
        "quantity": 1,
        "id": "33333333-7777-1111-1111-111111111111",
        "sku": "1",
        "status": "cancelled"
    }
}
```

13. Take a look at the `trigger` function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
```

14. Take a look at the `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
```

15. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe builds on recipes that we have already covered. We submit an order domain object using *database-first* Event Sourcing, and we have a `query` function to retrieve the current status of the order. The `listener` function is the most interesting part of this recipe. It listens for the `reservation-violation` events and performs a compensating action. In this case, the `compensation` is simply to change the `status` to `cancelled`. Compensating actions can be arbitrarily complex and are specific to a given service. For example, a service may need to reverse a complex calculation, while accounting for intermediate changes and triggering cascading changes as well. The audit trail provided by Event Sourcing and a *micro event store* may be useful for recalculating the new state.

Another thing to note in this example is that the collaboration is implemented using *event choreography* instead of *Event Orchestration*. In other words, this service is explicitly coupled to the `reservation-violation` event type. Event choreography is typically used in smaller and/or younger systems, or between highly related services. As a system matures and grows, the flexibility of *Event Orchestration* becomes more valuable. We employed Event Orchestration in the *Orchestrating collaboration between services* recipe.

4
Leveraging the Edge of the Cloud

In this chapter, the following recipes will be covered:

- Serving a single-page application from the CDN
- Associating a custom domain name with a CDN
- Serving a website from the CDN
- Deploying a service behind a CDN
- Serving static JSON from a CDN
- Triggering the invalidation of content in a CDN
- Executing code at the edge of the cloud

Introduction

The edge of the cloud is likely the single most underrated layer of cloud-native systems. However, as I have previously mentioned, my first cloud-native *wow* moment was when I realized I could run a **single-page application** (**SPA**) presentation layer from the edge without the complexity, risk, and cost of running an elastic load balancer and multiple EC2 instances. Furthermore, leveraging the edge brings global scale to certain aspects of cloud-native systems without expending additional effort on multi-regional deployments. Our end users enjoy reduced latency, while we reduce the load on the internals of our system, reduce cost, and increase security. The recipes in this chapter cover a multitude of ways we can leverage the edge of the cloud to advance the quality of our cloud-native system, with minimum effort.

Serving a single-page application from a CDN

In the *Deploying a single-page application* recipe, we covered the steps required to serve a single-page application from an S3 bucket. It is great fun to watch the light bulb turn on in an architect's mind when he or she realizes that such a simple deployment process can deliver so much scalability. Here is a recent quote from one of my customers—*That's it? No really, that's it*? In this recipe, we take this process one step further and demonstrate how easily we can add a CloudFront **Content Delivery Network** (**CDN**) layer in front of S3 to avail ourselves of even more valuable features.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-spa --path cncb-cdn-spa
```

2. Navigate to the `cncb-cdn-spa` directory with `cd cncb-cdn-spa`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-spa

provider:
  name: aws

plugins:
  - serverless-spa-deploy
  - serverless-spa-config

custom:
  spa:
    files:
      - source: ./build
        globs: '**/*'
        headers:
          CacheControl: max-age=31536000 # 1 year
      - source: ./build
        globs: 'index.html'
        headers:
          CacheControl: max-age=300 # 5 minutes
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory:

```
{
 "Resources": {
    ...
    "WebsiteBucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "AccessControl": "PublicRead",
        "WebsiteConfiguration": {
          "IndexDocument": "index.html",
          "ErrorDocument": "index.html"
        }
      }
    },
    "WebsiteDistribution": {
      "Type": "AWS::CloudFront::Distribution",
      "Properties": {
        "DistributionConfig": {
          "Comment": "Website: test-cncb-cdn-spa (us-east-1)",
          "Origins": [
            {
              "Id": "S3Origin",
              "DomainName": {
                "Fn::Select": [
                  1,
                  {
                    "Fn::Split": [
                      "//",
                      {
                        "Fn::GetAtt": [
                          "WebsiteBucket",
                          "WebsiteURL"
                        ]
                      }
                    ]
                  }
                ]
              },
              "CustomOriginConfig": {
                "OriginProtocolPolicy": "http-only"
              }
            }
          ],
          "Enabled": true,
```

```
            "PriceClass": "PriceClass_100",
            "DefaultRootObject": "index.html",
            "CustomErrorResponses": [
              {
                "ErrorCachingMinTTL": 0,
                "ErrorCode": 404,
                "ResponseCode": 200,
                "ResponsePagePath": "/index.html"
              }
            ],
            "DefaultCacheBehavior": {
              "TargetOriginId": "S3Origin",
              "AllowedMethods": [
                "GET",
                "HEAD"
              ],
              "CachedMethods": [
                "HEAD",
                "GET"
              ],
              "Compress": true,
              "ForwardedValues": {
                "QueryString": false,
                "Cookies": {
                  "Forward": "none"
                }
              },
              "MinTTL": 0,
              "DefaultTTL": 0,
              "ViewerProtocolPolicy": "allow-all"
            }
          }
        }
      }
    },
    ...
}
```

7. Build the app with `npm run build`.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-spa@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-spa
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
```

```
Serverless: Stack update finished...
...
Stack Outputs
WebsiteDistributionURL: https://dqvo8ga8z7ao3.cloudfront.net
WebsiteS3URL:
http://cncb-cdn-spa-john-websitebucket-1huxcgjseaili.s3-website-us-east
-1.amazonaws.com
WebsiteBucketName: cncb-cdn-spa-john-websitebucket-1huxcgjseaili
WebsiteDistributionId: E3JF634XQF4PE9
...
Serverless: Path: ./build
Serverless: File: asset-manifest.json (application/json)
...
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

9. Review the stack and CloudFront distribution in the AWS Console.
10. Browse to `WebsiteS3URL` and `WebsiteDistributionURL`, shown in the stack output, and note the performance difference between the two in the network tab of your browser's inspect tool, with the cache enabled and disabled:

```
http://<see WebsiteS3URL output>.s3-website-us-east-1.amazonaws.com
http://<see WebsiteDistributionURL output>.cloudfront.net
```

11. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The configuration of a CloudFront distribution is verbose and boilerplate. The `serverless-spa-config` plugin simplifies the effort, encapsulates the best practices, and allows for configuration by exception. In this recipe, we use all the defaults. In the generated `.serverless/cloudformation-template-update-stack.json` template, we can see that the `WebsiteBucket` is defined and configured as the default `S3Origin`, with the `index.html` file as the `DefaultRootObject`. The `PriceClass` defaults to North America and Europe, to minimize the amount of time it takes to provision the distribution. The error handling of the bucket (`ErrorDocument`) and the distribution (`CustomErrorResponses`) is configured to delegate error handling to the logic of the SPA.

The main purpose of the distribution is to cache the SPA resources at the edge. This logic is handled by two pieces. First, the `DefaultCacheBehavior` is set up with a `DefaultTTL` of zero, to ensure that the cache-control headers of the individual resources in the bucket are used to control the TTL. Second, the `serverless-spa-deploy` plugin is configured with two different `CacheControl` settings. Everything other than the `index.html` file is deployed with a max-age of one year because Webpack names these resources with a hash value that is generated for each build to implicitly bust the cache. The `index.html` file must have a constant name, because it is the `DefaultRootObject`, so we set its max-age to 5 minutes. This means that within approximately five minutes of a deployment, we can expect end users to start receiving the latest code changes. After five minutes, the browser will ask the CDN for the `index.html` file and the CDN will return an error 304 if the ETag is unchanged. This strikes a balance between minimizing data transfer and allowing changes to propagate quickly. You can increase or decrease the max-age as you see fit.

At this point, we are using the CDN to improve download performance for the user by pushing the resources to the edge to reduce latency, and compressing the resources to reduce bandwidth. This alone is reason enough to leverage the edge of the cloud. Additional features include custom domain names, SSL certificates, logging, and integration with a **web application firewall** (**WAF**). We will cover these topics in further recipes.

Associating a custom domain name with a CDN

In the *Serving a single-page application from a CDN* recipe, we covered the steps required to add a CloudFront CDN layer in front of S3 and avail ourselves of more valuable features. One of these features is the ability to associate a custom domain name with the resources served from the CDN, such as a single-page application or a cloud-native service. In this recipe, we will take the deployment process another step further and demonstrate how to add a Route53 record set and a CloudFront alias.

Getting ready

You will need a registered domain name and a Route53 hosted zone, which you can use in this recipe to create a subdomain for the SPA that will be deployed.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-dns --path cncb-cdn-dns
```

2. Navigate to the `cncb-cdn-dns` directory with `cd cncb-cdn-dns`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-dns

provider:
  name: aws

plugins:
  - serverless-spa-deploy
  - serverless-spa-config

custom:
  spa:
    files:
      ...
  dns:
    hostedZoneId:  Z1234567890123
    domainName: example.com
    endpoint: app.${self:custom.dns.domainName}
```

4. Update the `serverless.yml` file with your `hostedZoneId` and `domainName`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory:

```
{
  "Resources": {
    ...
    "WebsiteDistribution": {
      "Type": "AWS::CloudFront::Distribution",
      "Properties": {
        "DistributionConfig": {
          ...
          "Aliases": [
            "app.example.com"
          ],
          ...
```

```
            }
          }
        }
      },
      "WebsiteEndpointRecord": {
        "Type": "AWS::Route53::RecordSet",
        "Properties": {
          "HostedZoneId": "Z1234567890123",
          "Name": "app.example.com.",
          "Type": "A",
          "AliasTarget": {
            "HostedZoneId": "Z2FDTNDATAQYW2",
            "DNSName": {
              "Fn::GetAtt": [
                "WebsiteDistribution",
                "DomainName"
              ]
            }
          }
        }
      }
    },
    ...
}
```

8. Build the app with `npm run build`.
9. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-dns@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-dns
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
Stack Outputs
WebsiteDistributionURL: https://dwrdvnqsnetay.cloudfront.net
WebsiteS3URL:
http://cncb-cdn-dns-john-websitebucket-1dwzb5bfkv34s.s3-website-us-east
-1.amazonaws.com
WebsiteBucketName: cncb-cdn-dns-john-websitebucket-1dwzb5bfkv34s
WebsiteURL: http://app.example.com
WebsiteDistributionId: ED6YKAFJDF2ND
...
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

10. Review the stack, Route53, and CloudFront distribution in the AWS Console.
11. Browse to `WebsiteURL`, shown in the stack output:

```
http://app.example.com <see WebsiteURL output>
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we improve on the *Serving a single-page application from a CDN* recipe by adding a custom domain name for the SPA. In `serverless.yml`, we added the `hostedZoneId`, `domainName`, and `endpoint` values. These values trigger the `serverless-spa-config` plugin to configure the `WebsiteEndpointRecord` in Route53, and set the `Aliases` on the CloudFront distribution.

Serving a website from the CDN

As an industry, we have a tendency to create very dynamic websites, even when the content does not change frequently. Some **Content Management Systems** (**CMS**) recalculate content for every request, even when the content has not changed. These requests pass through multiple layers, read from the database, and then calculate and return the response. It is not uncommon to see average response times in the range of five seconds. It is said that doing the same thing over and over again and expecting a different result is the definition of insanity.

Cloud-native systems take an entirely different approach to creating websites. JAMstack (`https://jamstack.org`) is a modern, cloud-native approach based on client-side JavaScript, reusable APIs, and Markup. These static sites are managed by Git workflows, generated by CI/CD pipelines, and deployed to the edge of the cloud many times a day. This is yet another example of cloud-native challenging us to rewire our software engineering brains. This recipe demonstrates the generation and deployment of these static websites.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-site --path cncb-cdn-site
```

2. Navigate to the `cncb-cdn-site` directory with `cd cncb-cdn-site`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-site

provider:
  name: aws

plugins:
  - serverless-spa-deploy
  - serverless-spa-config

custom:
  spa:
    files:
      - source: ./dist
        globs: '**/*'
        headers:
          CacheControl: max-age=31536000 # 1 year
    redirect: true
  dns:
    hostedZoneId:  Z1234567890123
    domainName: example.com
    endpoint: www.${self:custom.dns.domainName}
```

4. Update the `serverless.yml` file with your `hostedZoneId` and `domainName`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory:

```
{
  "Resources": {
    ...
    "RedirectBucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "example.com",
        "WebsiteConfiguration": {
```

```
            "RedirectAllRequestsTo": {
              "HostName": "www.example.com"
            }
          }
        }
      },
      "WebsiteDistribution": {
        "Type": "AWS::CloudFront::Distribution",
        "Properties": {
          "DistributionConfig": {
"Comment": "Website: test-cncb-cdn-site (us-east-1)",
          ...
            "Aliases": [
              "www.example.com"
            ],
          ...
          }
        }
      },
      "RedirectDistribution": {
        "Type": "AWS::CloudFront::Distribution",
        "Properties": {
          "DistributionConfig": {
            "Comment": "Redirect: test-cncb-cdn-site (us-east-1)",
          ...
            "Aliases": [
              "example.com"
            ],
          ...
          }
        }
      },
      "WebsiteEndpointRecord": {
        "Type": "AWS::Route53::RecordSet",
        "Properties": {
          "HostedZoneId": "Z1234567890123",
          "Name": "www.example.com.",
          "Type": "A",
          "AliasTarget": {
            "HostedZoneId": "Z2FDTNDATAQYW2",
            "DNSName": {
              "Fn::GetAtt": [
                "WebsiteDistribution",
                "DomainName"
              ]
            }
          }
        }
```

```
    },
    "RedirectEndpointRecord": {
      "Type": "AWS::Route53::RecordSet",
      "Properties": {
        "HostedZoneId": "Z1234567890123",
        "Name": "example.com.",
        "Type": "A",
        "AliasTarget": {
          "HostedZoneId": "Z2FDTNDATAQYW2",
          "DNSName": {
            "Fn::GetAtt": [
              "RedirectDistribution",
              "DomainName"
            ]
          }
        }
      }
    }
  },
  ...
}
```

8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-site@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-site
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
Stack Outputs
WebsiteDistributionURL: https://d2ooxtd49ayyfd.cloudfront.net
WebsiteS3URL:
http://cncb-cdn-site-john-websitebucket-mrn9ntyltxim.s3-website-us-east
-1.amazonaws.com
WebsiteBucketName: cncb-cdn-site-john-websitebucket-mrn9ntyltxim
WebsiteURL: http://www.example.com
WebsiteDistributionId: E10ZLN9USZTDSO
...
Serverless: Path: ./dist
Serverless: File: after/c2Vjb25kLXBvc3Q=/index.html (text/html)
Serverless: File: after/dGhpcmQtcG9zdA==/index.html (text/html)
Serverless: File: after/Zm91cnRoLXBvc3Q=/index.html (text/html)
Serverless: File: after/ZmlmdGgtcG9zdA==/index.html (text/html)
Serverless: File: after/Zmlyc3QtcG9zdA==/index.html (text/html)
```

```
Serverless: File: blog/fifth-post/index.html (text/html)
Serverless: File: blog/first-post/index.html (text/html)
Serverless: File: blog/fourth-post/index.html (text/html)
Serverless: File: blog/second-post/index.html (text/html)
Serverless: File: blog/third-post/index.html (text/html)
Serverless: File: favicon.ico (image/x-icon)
Serverless: File: index.html (text/html)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2.json (application/json)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2/after-c2Vjb25kLXBvc3Q=.json
(application/json)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2/after-dGhpcmQtcG9zdA==.json
(application/json)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2/after-Zm91cnRoLXBvc3Q=.json
(application/json)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2/after-ZmlmdGgtcG9zdA==.json
(application/json)
Serverless: File: phenomic/content/posts/by-
default/1/desc/date/limit-2/after-Zmlyc3QtcG9zdA==.json
(application/json)
Serverless: File: phenomic/content/posts/item/fifth-post.json
(application/json)
Serverless: File: phenomic/content/posts/item/first-post.json
(application/json)
Serverless: File: phenomic/content/posts/item/fourth-post.json
(application/json)
Serverless: File: phenomic/content/posts/item/second-post.json
(application/json)
Serverless: File: phenomic/content/posts/item/third-post.json
(application/json)
Serverless: File: phenomic/phenomic.main.9a7f8f5f.js
(application/javascript)
Serverless: File: robots.txt (text/plain)
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

9. Review the stack, Route53, and CloudFront distribution in the AWS Console.
10. Browse to `WebsiteURL`, shown in the stack output, and again without the `www.`, and observe the redirect to `www`:

```
http://www.example.com <see WebsiteURL output>
http://example.com <see WebsiteURL output>
```

11. Browse through the website pages.
12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this recipe, we use a static site generator called **Phenomic** (`https://www.staticgen.com/phenomic`). There is a seemingly endless array of tools enumerated on the **StaticGen** site (`https://www.staticgen.com`). With Phenomic, we write the website in a combination of ReactJS and Markdown. Then, we isomorphically generate the JavaScript and Markdown into a set of static resources that we deploy to S3, as can be seen in the deployment output. I have pre-built the website and included the generated resources in the repository. A typical SPA downloads the JavaScript and generates the site in the browser. Depending on the website, this process can be lengthy and noticeable. Runtime-based isomorphic generation performs this process on the server-side before returning the website to the browser. A JAMstack website, on the other hand, is statically generated at deployment time, which results in the best runtime performance. Although the website is static, the pages contain dynamic JavaScript that is written just like in any ReactJS SPA. These websites are frequently updated and re-deployed multiple times per day, which adds another dynamic dimension without having to ask the server on every request if something has changed.

These sites are typically deployed to a `www` subdomain, and support redirecting the root domain to this subdomain. Building on the *Serving a single-page application from a CDN* and *Associating a custom domain name with CDN* recipes, this recipe leverages the `serverless-spa-deploy` and `serverless-spa-config` plugins and adds some additional settings. We specify the `endpoint` as `www.${self:custom.dns.domainName}` and set the redirect flag to `true`. This, in turn, creates the `RedirectBucket`, which performs the redirects, along with an additional `RedirectDistribution` and `RedirectEndpointRecord` to support the root domain.

Deploying a service behind a CDN

When we think of a CDN, it is typical to think that they are only useful for serving up static content. However, it is an AWS best practice to place CloudFront in front of all resources, even dynamic services, to improve security and performance. From a security perspective, CloudFront minimizes the attack surfaces and handles DDOS attacks at the edge to reduce the load on internal components. With regard to performance, CloudFront optimizes the pipe between the edge and availability zones, which improves performance even for `POST` and `PUT` operations. Furthermore, even a cache-control header of just a few seconds can have a significant impact on `GET` operations. In this recipe, we will demonstrate how to add CloudFront in front of the AWS API Gateway.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-service --path cncb-cdn-service
```

2. Navigate to the `cncb-cdn-service` directory with `cd cncb-cdn-service`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-service

provider:
  name: aws
  runtime: nodejs8.10
  endpointType: REGIONAL

functions:
  hello:
    handler: handler.hello
    events:
      - http:
          path: hello
          method: get
          cors: true

resources:
  Resources:
    ApiDistribution:
      Type: AWS::CloudFront::Distribution
```

```
      Properties:
        DistributionConfig:
          ...
          Origins:
            - Id: ApiGateway
              DomainName:
                Fn::Join:
                  - "."
                  - - Ref: ApiGatewayRestApi
                    - execute-api.${opt:region}.amazonaws.com
              OriginPath: /${opt:stage}
              CustomOriginConfig:
                OriginProtocolPolicy: https-only
                OriginSSLProtocols: [ TLSv1.2 ]
          ...
          DefaultCacheBehavior:
            TargetOriginId: ApiGateway
            AllowedMethods: [ DELETE, GET, HEAD, OPTIONS, PATCH, POST,
PUT ]
            CachedMethods: [ GET, HEAD, OPTIONS ]
            Compress: true
            ForwardedValues:
              QueryString: true
              Headers: [ Accept, Authorization ]
              Cookies:
                Forward: all
            MinTTL: 0
            DefaultTTL: 0
            ViewerProtocolPolicy: https-only

  Outputs:
    ApiDistributionId:
      Value:
        Ref: ApiDistribution
    ApiDistributionEndpoint:
      Value:
        Fn::Join:
          - ''
          - - https://
            - Fn::GetAtt: [ ApiDistribution, DomainName ]
```

4. Review the file named `handler.js` with the following content:

```
module.exports.hello = (request, context, callback) => {
  const response = {
    statusCode: 200,
    headers: {
      'Cache-Control': 'max-age=5',
```

```
    },
    body: ...,
  };

  callback(null, response);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-service@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-
service
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
https://5clnzj3knc.execute-api.us-east-1.amazonaws.com/john/hello
functions:
  hello: cncb-cdn-service-john-hello

Stack Outputs
ApiDistributionEndpoint: https://d1vrpoljefctug.cloudfront.net
ServiceEndpoint:
https://5clnzj3knc.execute-api.us-east-1.amazonaws.com/john
...
ApiDistributionId: E2X1H9ZQ1B9U0R
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

9. Review the stack and CloudFront distribution in the AWS Console.
10. Invoke the endpoint shown in the stack output, with the following `curl` command, multiple times to see the difference in performance for the cached results:

```
$ curl -s -D - -w "Time: %{time_total}" -o /dev/null  https://<see
stack output>.cloudfront.net/hello | egrep -i 'X-Cache|Time'

X-Cache: Miss from cloudfront
Time: 0.324
$ curl ...
X-Cache: Hit from cloudfront
Time: 0.168
$ curl ...
X-Cache: Hit from cloudfront
Time: 0.170
$ curl ...
X-Cache: Miss from cloudfront
Time: 0.319
$ curl ...
X-Cache: Hit from cloudfront
Time: 0.167
```

11. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The first thing to note in the `serverless.yml` file is that the `endpointType` is set to `REGIONAL`. By default, AWS API Gateway will provision a CloudFront distribution. However, we do not have access to this distribution and cannot take advantage of all its features. Plus, the default does not support multi-regional deployments. Therefore, we specify `REGIONAL` so that we can manage the CDN ourselves. Next, we need to configure the `Origins`. We specify the `DomainName` to point to the regional API Gateway endpoint, and we specify the stage as the `OriginPath`, so that we no longer need to include it in the URL.

Next, we configure the `DefaultCacheBehavior`. We allow both read and write methods, and we cache the read methods. We set `DefaultTTL` to zero to ensure that the `Cache-Control` headers set in the service code are used to control the TTL. In this recipe, the code sets the `max-age` to 5 seconds, and we can see that our cache hits respond approximately twice as fast. We also set `Compress` to `true` to minimize data transfer for both increased performance and to help reduced cost. It is important to forward all the `Headers` that are expected by the backend. For example, the authorization header is crucial for securing a service with OAuth 2.0, as we will discuss in the *Securing an API gateway with OAuth 2.0* recipe in `Chapter 5`, *Securing Cloud-Native Systems*.



Note that `ApiGatewayRestApi` is the logical resource ID that is created and controlled by the Serverless Framework.

Serving static JSON from a CDN

In the *Implementing a search BFF* recipe, we created a service that serves some content from a materialized view in Elasticsearch via an API Gateway, and other content directly from a materialized view in S3. This is a great approach that can cost-effectively deliver under extremely heavy loads. In this recipe, we will demonstrate how to add a single CloudFront distribution in front of both the API Gateway and S3 to encapsulate these internal design decisions behind a single domain name.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-json --path cncb-cdn-json
```

2. Navigate to the `cncb-cdn-json` directory with `cd cncb-cdn-json`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-json

provider:
  name: aws
  runtime: nodejs8.10
```

```
    endpointType: REGIONAL
    ...

functions:
  search:
    handler: handler.search
    ...

resources:
  Resources:
    ApiDistribution:
      Type: AWS::CloudFront::Distribution
      Properties:
        DistributionConfig:
          ...
          Origins:
            - Id: S3Origin
              DomainName:
                Fn::Join:
                  - "."
                  - - Ref: Bucket
                    - s3.amazonaws.com
              ...
            - Id: ApiGateway
              DomainName:
                Fn::Join:
                  - "."
                  - - Ref: ApiGatewayRestApi
                    - execute-api.${opt:region}.amazonaws.com
              OriginPath: /${opt:stage}
              ...
          ...
          DefaultCacheBehavior:
            TargetOriginId: S3Origin
            AllowedMethods:
              - GET
              - HEAD
            CachedMethods:
              - HEAD
              - GET
            ...
            MinTTL: 0
            DefaultTTL: 0
            ...
          CacheBehaviors:
            - TargetOriginId: ApiGateway
              PathPattern: /search
              AllowedMethods: [ GET, HEAD, OPTIONS ]
```

```
                CachedMethods: [ GET, HEAD, OPTIONS ]
                ...
                MinTTL: 0
                DefaultTTL: 0
                ...

    Bucket:
      Type: AWS::S3::Bucket
...
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory.
7. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-json@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-json
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
https://hvk3o94wij.execute-api.us-east-1.amazonaws.com/john/search
functions:
  search: cncb-cdn-json-john-search
  load: cncb-cdn-json-john-load

Stack Outputs
...
ApiDistributionEndpoint: https://dfktdq2w7ht2p.cloudfront.net
BucketName: cncb-cdn-json-john-bucket-ls21fzjp2qs2
ServiceEndpoint:
https://hvk3o94wij.execute-api.us-east-1.amazonaws.com/john
ApiDistributionId: E3JJREB1B4TIGL
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

8. Review the stack and CloudFront distribution in the AWS Console.

9. Invoke the following command to load data:

```
$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-5555-1111-1111-000000000000","name":"thing one"}'

{
    "ETag": "\"3f4ca01316d6f88052a940ab198b2dc7\""
}

$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-5555-1111-2222-000000000000","name":"thing two"}'

{
    "ETag": "\"926f41091e2f47208f90f9b9848dffd0\""
}

$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-5555-1111-3333-000000000000","name":"thing three"}'

{
    "ETag": "\"2763ac570ff0969bd42182506ba24dfa\""
}
```

10. Invoke the endpoint shown in the stack output with the following `curl` commands:

```
$ curl https://<see stack output>.cloudfront.net/search | json_pp

[
"https://dfktdq2w7ht2p.cloudfront.net/things/44444444-5555-1111-1111-00
0000000000",
"https://dfktdq2w7ht2p.cloudfront.net/things/44444444-5555-1111-2222-00
0000000000",
"https://dfktdq2w7ht2p.cloudfront.net/things/44444444-5555-1111-3333-00
0000000000"
]

$ curl -s -D - -w "Time: %{time_total}" -o /dev/null https://<see stack
output>.cloudfront.net/things/44444444-5555-1111-1111-000000000000 |
egrep -i 'X-Cache|Time'

X-Cache: Miss from cloudfront
Time: 0.576
$ curl ...
X-Cache: Hit from cloudfront
Time: 0.248
```

11. Remove the stack with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe builds on the *Deploying a service behind a CDN* recipe. The main difference in this recipe is that we have multiple `Origins` and multiple `CacheBehaviors`, one each for our `Bucket` and our `ApiGatewayRestApi`. We use the `DefaultCacheBehavior` for our `S3Origin`, because we could store many different business domains in the bucket with different paths. Conversely, there is a single `PathPattern` (`/search`) that needs to be directed to our `APIGateway` origin, therefore we define this under `CacheBehaviors`. Again, in all cases, we set the `DefaultTTL` to zero to ensure our cache-control headers control the TTL. The end result is that our multiple origins now look like one from the outside.



Note that `ApiGatewayRestApi` is the logical resource ID that is created and controlled by the Serverless Framework.

Triggering the invalidation of content in a CDN

In the *Implementing a search BFF* recipe, we statically serve dynamic JSON content from S3, and in the *Serving static JSON from a CDN* recipe, we add a cache-control header with a long max-age to further improve the performance. This technique works great for content that is dynamic, yet changes at a slow and unpredictable rate. In this recipe, we will demonstrate how to improve on this design by responding to data change events and invalidating the cache so that the latest data is retrieved.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-invalidate --path cncb-cdn-invalidate
```

2. Navigate to the `cncb-cdn-invalidate` directory with `cd cncb-cdn-invalidate`.

3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-invalidate

provider:
  name: aws
  runtime: nodejs8.10
  ...

functions:
  load:
    ...
  trigger:
    handler: handler.trigger
    events:
      - sns:
          arn:
            Ref: BucketTopic
          topicName: ${self:service}-${opt:stage}-trigger
    environment:
      DISABLED: false
      DISTRIBUTION_ID:
        Ref: ApiDistribution

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket
      DependsOn: [ BucketTopic, BucketTopicPolicy ]
      Properties:
        NotificationConfiguration:
          TopicConfigurations:
            - Event: s3:ObjectCreated:Put
              Topic:
                Ref: BucketTopic
    BucketTopic:
      Type: AWS::SNS::Topic
      Properties:
        TopicName: ${self:service}-${opt:stage}-trigger
    BucketTopicPolicy: ${file(includes.yml):BucketTopicPolicy}
    ApiDistribution:
      Type: AWS::CloudFront::Distribution
      Properties:
        DistributionConfig:
          ...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.trigger = (event, context, cb) => {
  _(process.env.DISABLED === 'true' ? [] : event.Records)
    .flatMap(messagesToTriggers)
    .flatMap(invalidate)
    .collect()
    .toCallback(cb);
};

const messagesToTriggers = r => _(JSON.parse(r.Sns.Message).Records);

const invalidate = (trigger) => {
  const params = {
    DistributionId: process.env.DISTRIBUTION_ID,
    InvalidationBatch: {
      CallerReference: uuid.v1(),
      Paths: {
        Quantity: 1,
        Items: [`/${trigger.s3.object.key}`]
      }
    }
  };

  const cf = new aws.CloudFront();
  return _(cf.createInvalidation(params).promise());
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-invalidate@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-
invalidate
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  load: cncb-cdn-invalidate-john-load
  trigger: cncb-cdn-invalidate-john-trigger
```

```
Stack Outputs
BucketArn: arn:aws:s3:::cncb-cdn-invalidate-john-bucket-z3u7jc60piub
BucketName: cncb-cdn-invalidate-john-bucket-z3u7jc60piub
ApiDistributionEndpoint: https://dgmob5bgqpnpi.cloudfront.net
TopicArn: arn:aws:sns:us-east-1:123456789012:cncb-cdn-invalidate-john-
trigger
...
ApiDistributionId: EV1QUKWEQV6XN
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more.

9. Review the stack in the AWS Console.
10. Invoke the following command to load data:

```
$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-6666-1111-1111-000000000000","name":"thing one"}'

{
    "ETag": "\"3f4ca01316d6f88052a940ab198b2dc7\""
}
```

11. Invoke the endpoint shown in the stack output with the following `curl` commands:

```
$ curl https://<see stack
output>.cloudfront.net/things/44444444-6666-1111-1111-000000000000 |
json_pp

{
   "name" : "thing one",
   "id" : "44444444-6666-1111-1111-000000000000"
}
```

12. Invoke the following command to load data:

```
$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-6666-1111-1111-000000000000","name":"thing one
again"}'

{
    "ETag": "\"edf9676ddcc150f722f0f74b7a41bd7f\""
}
```

13. Take a look at the logs for the `trigger` function:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE

2018-05-20 02:43:16 ... params:
{"DistributionId":"EV1QUKWEQV6XN","InvalidationBatch":{"CallerReference
":"13a8e420-5bf9-11e8-818d-9de0acd70c96","Paths":{"Quantity":1,"Items":
["/things/44444444-6666-1111-1111-000000000000"]}}}
2018-05-20 02:43:17 ...
{"Location":"https://cloudfront.amazonaws.com/2017-10-30/distribution/E
V1QUKWEQV6XN/invalidation/I33URVVAO02X7I","Invalidation":{"Id":"I33URVV
AO02X7I","Status":"InProgress","CreateTime":"2018-05-20T06:43:17.102Z",
"InvalidationBatch":{"Paths":{"Quantity":1,"Items":["/things/44444444-6
666-1111-1111-000000000000"]},"CallerReference":"13a8e420-5bf9-11e8-818
d-9de0acd70c96"}}}
```

14. Invoke the endpoint shown in the stack output with the following `curl` commands until the invalidation completes and the output changes:

```
$ curl https://<see stack
output>.cloudfront.net/things/44444444-6666-1111-1111-000000000000 |
json_pp

{
   "name" : "thing one again",
   "id" : "44444444-6666-1111-1111-000000000000"
}
```

15. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe builds on the *Implementing a materialized view in S3* and *Implementing a search BFF* recipes. We define the bucket's `NotificationConfiguration` to send events to the SNS `BucketTopic` so that we can trigger more than one consumer. In the *Implementing a search BFF* recipe, we triggered indexing in Elasticsearch. In this recipe, we demonstrate how we can also trigger the invalidation of the cache in the CloudFront distribution. The `trigger` function creates an invalidation request for each `trigger.s3.object.key`. These invalidations will force the CDN to retrieve these resources from the origin the next time they are requested.



A slow flow of invalidations does not incur significant cost. However, large batches of individual invalidations can be costly. This could occur during a data conversion process when enhancing a service. In these cases, the `DISABLED` environment variable should be set to `true` before executing the conversion. Then, manually invalidate the distribution with a wildcard when the conversion is complete.

Executing code at the edge of the cloud

AWS Lambda@Edge is a feature that allows functions to execute at the edge of the cloud in response to CloudFront events. This capability opens some interesting opportunities to control, modify, and generate content at the edge of the cloud with extremely low latency. I think we are only just beginning to uncover the potential use cases that can be implemented with this new feature. This recipe demonstrates the association of a bare-bones function with a CloudFront distribution. The function responds with an Unauthorized (403) status code when an authorization header is not present in the request.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch4/cd
n-lambda --path cncb-cdn-lambda
```

2. Navigate to the `cncb-cdn-lambda` directory with `cd cncb-cdn-lambda`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cdn-lambda

plugins:
  - serverless-plugin-cloudfront-lambda-edge

provider:
  name: aws
  runtime: nodejs8.10
  ...

functions:
  authorize:
    handler: handler.authorize
```

```
    memorySize: 128
    timeout: 1
    lambdaAtEdge:
      distribution: 'ApiDistribution'
      eventType: 'viewer-request'
  ...

resources:
  Resources:
    Bucket:
      Type: AWS::S3::Bucket

    ApiDistribution:
      Type: AWS::CloudFront::Distribution
      Properties:
        DistributionConfig:
          ...
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory:

```
...
"LambdaFunctionAssociations": [
  {
    "EventType": "viewer-request",
    "LambdaFunctionARN": {
      "Ref":
"AuthorizeLambdaVersionSticfT7s2DCStJsDgzhTCrXZB1CjlEXXc3bR4YS1owM"
    }
  }
]
...
```

7. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cdn-lambda@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cdn-lambda
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
functions:
  authorize: cncb-cdn-lambda-john-authorize
```

```
  load: cncb-cdn-lambda-john-load

Stack Outputs
ApiDistributionEndpoint: https://d1kktmeew2xtn2.cloudfront.net
BucketName: cncb-cdn-lambda-john-bucket-olzhip1qvzqi
...
ApiDistributionId: E2VL0VWTW5IXUA
```



Deploying a CloudFront distribution can take upwards of 20 minutes or more. Any changes to these functions will cause a redeploy of the distribution, because the distribution is linked to the function version.

8. Review the stack, function, and CloudFront distribution in the AWS Console.
9. Invoke the following command to load the data:

```
$ sls invoke -f load -r us-east-1 -s $MY_STAGE -d
'{"id":"44444444-7777-1111-1111-000000000000","name":"thing one"}'

{
    "ETag": "\"3f4ca01316d6f88052a940ab198b2dc7\""
}
```

10. Invoke the endpoint shown in the stack output with the following `curl` commands:

```
$ curl -v -H "Authorization: Bearer 1234567890" https://<see stack
output>.cloudfront.net/things/44444444-7777-1111-1111-000000000000 |
json_pp

...
> Authorization: Bearer 1234567890
>
< HTTP/1.1 200 OK
...
{
   "name" : "thing one",
   "id" : "44444444-7777-1111-1111-000000000000"
}

$ curl -v https://<see stack
output>.cloudfront.net/things/44444444-7777-1111-1111-000000000000 |
json_pp

...
< HTTP/1.1 401 Unauthorized
...
```

```
[
   {
      "Records" : [
         {
            "cf" : {
               "config" : {
                  "eventType" : "viewer-request",
                  "requestId" :
"zDpj1UckJLTIln8dwak2ZL1SX0LtWfGPhg3mC1EGIgfRd6gzJFVeqg==",
                  "distributionId" : "E2VL0VWTW5IXUA",
                  "distributionDomainName" :
"d1kktmeew2xtn2.cloudfront.net"
               },
            ...
             }
         }
      ]
   },
   {
    ...
      "logGroupName" : "/aws/lambda/us-east-1.cncb-cdn-lambda-john-
authorize",
    ...
   },
   {
      "AWS_REGION" : "us-east-1",
    ...
   }
]
```

11. Review the logs in the AWS Console.
12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.



Lambda@Edge replicates functions to multiple regions per the `PriceClass` of the distribution. A function cannot be deleted until all the replicas have been deleted. This should happen within several hours of deleting the distribution. Therefore, removing the stack will initially fail, and can be repeated after the replicas have been deleted.

How it works...

In this recipe, we use the `serverless-plugin-cloudfront-lambda-edge` plugin to associate the `authorize` function with the CloudFront `ApiDistribution`. We specify the `distribution` and the `eventType` under the `lambdaAtEdge` element. The plugin then uses this information to create the `LambdaFunctionAssociations` element on the distribution in the generated CloudFormation template. The `eventType` can be set to `viewer-request`, `origin-request`, `origin-response`, or `view-response`. This recipe uses the `viewer-request`, because it needs access to the authorization header sent by the viewer. We explicitly set the `memorySize` and `timeout` for the function, because Lambda@Edge imposes restrictions on these values.



Lambda@Edge records `console.log` statements in the region associated with the specific edge location that was accessed. This recipe returns `logGroupName` and `AWS_REGION` in the response to help demonstrate where to find the logs.

5
Securing Cloud-Native Systems

In this chapter, the following recipes will be covered:

- Securing your cloud account
- Creating a federated identity pool
- Implementing sign up, sign in, and sign out
- Securing an API Gateway with OpenID Connect
- Implementing a custom authorizer
- Authorizing a GraphQL-based service
- Implementing a JWT filter
- Using envelope encryption
- Creating an SSL certificate for encryption in transit
- Configuring a web application firewall
- Replicating the data lake for disaster recovery

Introduction

Security in the cloud is based on the shared responsibility model. Below a certain line in the stack is the responsibility of the cloud provider and above that line is the responsibility of the cloud consumer. Cloud-native and serverless computing push that line higher and higher. This allows teams to focus their efforts on what they know best—their business domains. With the security mechanisms provided by the cloud, teams can practice *security-by-design* and concentrate on *defense-in-depth* techniques to secure their data. In each recipe, so far, we have seen how serverless computing requires us to define security policies between components at each layer in the stack. The recipes in this chapter will cover securing our cloud accounts, securing our applications with OAuth 2.0/Open ID Connect, securing our data at rest, and creating a perimeter around our cloud-native systems by delegating aspects of security to the edge of the cloud.

Securing your cloud account

Everything we do to secure our cloud-native systems is all for nothing if we do not endeavor to secure our cloud accounts as well. There is a set of best practices that we must put in place for every cloud account we create. As we strive to create autonomous services, we should leverage the natural bulkhead between cloud accounts by grouping related services into more, fine-grained accounts instead of fewer, coarse-grained accounts. In this recipe, we will see how treating accounts as code enables us to manage many accounts easily by applying the same infrastructure-as-code practices we employ to manage our many autonomous services.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/ac
count-as-code --path cncb-account-as-code
```

2. Navigate to the `cncb-account-as-code` directory with `cd cncb-account-as-code`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-account-as-code

provider:
  name: aws
  # cfnRole:
arn:aws:iam::${self:custom.accountNumber}:role/${opt:stage}-cfnRole

custom:
  accountNumber: 123456789012

resources:
  Resources:
    AuditBucket:
      Type: AWS::S3::Bucket
      DeletionPolicy: Retain
      ...
    CloudTrail:
      Type: AWS::CloudTrail::Trail
      ...

    CloudFormationServiceRole:
```

```
      Type: AWS::IAM::Role
      Properties:
        RoleName: ${opt:stage}-cfnRole
        ...
    ExecuteCloudFormationPolicy:
      Type: AWS::IAM::ManagedPolicy
      ...
    CiCdUser:
      Type: AWS::IAM::User
      Properties:
        ManagedPolicyArns:
          - Ref: ExecuteCloudFormationPolicy

    AdminUserGroup:
      Type: AWS::IAM::Group
      ...
    ReadOnlyUserGroup:
      Type: AWS::IAM::Group
      ...
    PowerUserGroup:
      Type: AWS::IAM::Group
      ...
    ManageAccessKey:
      Condition: IsDev
      Type: AWS::IAM::ManagedPolicy
      ...
    MfaOrHqRequired:
      Condition: Exclude
      Type: AWS::IAM::ManagedPolicy
      ...
  ...
```

4. Update `accountNumber` in `serverless.yml`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-account-as-code@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
account-as-code
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
```

```
Serverless: Stack update finished...
...
```

9. Review the stack and resources in the AWS Console.
10. Uncomment `cfnRole` and then deploy again with the `force` flag with `npm run dp:lcl -- -s $MY_STAGE --force`.
11. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Every account I create starts off with the same essential security settings using a `serverless.yml`, such as the one in this recipe. I create no other stacks in the account until this account-scoped stack is created. All further changes, other than creating users, are delivered as changes to this stack. The first responsibility of this stack is to turn on `CloudTrail`. In `Chapter 7`, *Optimizing Observability*, we will see how we can use this audit trail to monitor and alert about unexpected changes to security policies. `AuditBucket` is also a candidate for replicating to the recovery account as discussed in the *Replicating the data lake for disaster recovery* recipe.

Next, the stack creates the user groups that will be used for granting permissions to all users of the account. The `AdminUserGroup`, `PowerUserGroup`, and `ReadOnlyUserGroup` groups are a good starting point, along with using the managed policies provided by AWS. As the usage of the account matures, these groups will evolve using the same approach discussed in `Chapter 6`, *Building a Continuous Deployment Pipeline*. However, only the security policies are codified. The assignment of users to groups is a manual process that should follow an appropriate approval process. The stack includes the `MfaOrHqRequired`, policy to require **Multi-Factor Authentication** (**MFA**) and whitelist the corporate IP addresses, but it is disabled initially. It should certainly be enabled for all production accounts. In a development account, most developers are assigned to the power user group, so that they can freely experiment with cloud services. The power user group has no IAM permissions, so an optional `ManageAccessKey` policy is included to allow power users to manage their access keys. Note, it is very important to control the usage of access keys and frequently rotate them.

When executing a `serverless.yml` file, we need an access key. As an added security measure, CloudFormation supports the use of a service role that allows CloudFormation to assume a specific role with temporary credentials. Using the `cfnRole` attribute in a `serverless.yml` file enables this feature. This stack creates an initial `CloudFormationServiceRole` that should be used by all stacks. As the account matures, this role should be tuned to the least possible privileges. The `ExecuteCloudFormationPolicy` included only has enough permissions to execute a `serverless.yml` file. This policy will be used by `CiCdUser`, which we will use in `Chapter 6`, *Building a Continuous Deployment Pipeline*.

Creating a federated identity pool

Managing users is a requirement of virtually every system. Over a long career, I can certainly attest to creating identity management functionality over and over again. Fortunately, we can now get this functionality as a service from many providers, including our cloud providers. And because there are so many options available, we need a federated solution that delegates to many other identity management systems while presenting a single, unified model to our cloud-native system. In this recipe, we will show how to create an *AWS Cognito user pool*, which we will then use in other recipes to secure our services.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/co
gnito-pool --path cncb-cognito-pool
```

2. Navigate to the `cncb-cognito-pool` directory with `cd cncb-cognito-pool`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cognito-pool

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole

resources:
  Resources:
    CognitoUserPoolCncb:
      Type: AWS::Cognito::UserPool
```

```
      Properties:
        UserPoolName: cncb-${opt:stage}
        ...
        Schema:
          - AttributeDataType: 'String'
            DeveloperOnlyAttribute: false
            Mutable: true
            Name: 'email'
            Required: true
        ...
    CognitoUserPoolCncbClient:
      Type: AWS::Cognito::UserPoolClient
      Properties:
        UserPoolId:
          Ref: CognitoUserPoolCncb

  Outputs:
    ...

plugins:
  - cognito-plugin

custom:
  pool:
    domain: cncb-${opt:stage}
    allowedOAuthFlows: ['implicit']
    allowedOAuthFlowsUserPoolClient: true
    allowedOAuthScopes: ...
    callbackURLs: ['http://localhost:3000/implicit/callback']
    logoutURLs: ['http://localhost:3000']
    refreshTokenValidity: 30
    supportedIdentityProviders: ['COGNITO']
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory.
7. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cognito-pool@1.0.0 dp:lcl <path-to-your-workspace>/cncb-cognito-
pool
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
```

```
...

Stack Outputs
userPoolArn: arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-
east-1_tDXH8JAky
userPoolClientId: 4p5p08njgmq9sph130l3du2b7q
loginURL: https://cncb-john.auth.us-
east-1.amazoncognito.com/login?redirect_uri=http://localhost:3000/impli
cit/callback&response_type=token&client_id=4p5p08njgmq9sph130l3du2b7q
userPoolProviderName: cognito-idp.us-east-1.amazonaws.com/us-
east-1_tDXH8JAky
userPoolId: us-east-1_tDXH8JAky
userPoolProviderURL:
https://cognito-idp.us-east-1.amazonaws.com/us-east-1_tDXH8JAky
```

8. Review the stack and resources in the AWS Console.
9. Access `loginURL` from the stack output to verify that the user pool was created successfully.
10. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.



We will be referencing this user pool in other recipes in this chapter, so remove this stack after you are finished with this chapter.

How it works...

This recipe simply codifies the creation of an AWS Cognito user pool. Like any other infrastructure resource, we want to treat it as code and manage it in a continuous delivery pipeline. For this recipe, we define the bare essentials. You will want to define any additional attributes that you want the pool to manage. For this recipe, we just specify the `email` attribute.



It is important to understand that these attributes cannot be changed via CloudFormation without creating a new user pool, which has a ripple effect on any resources that are dependent on the user pool. So, expect to focus some effort on experimenting upfront with the right combination of attributes.

For each application that will rely on this user pool, we need to define a `UserPoolClient`. Each application would typically define the client in a stack that it manages. However, it is important not to overuse a single user pool. Once again this is a question of autonomy. If the pool of users and the applications used by those users are truly independent, then they should be managed in separate Cognito user pools, even if that requires some duplication of effort. For example, if you find yourself writing complicated logic using Cognito user groups to segregate users unnaturally, then you may be better off with multiple user pools. An example of misuse is mixing employees and customers in the same user pool.

CloudFormation, as of the writing of this chapter, does not have full support for the Cognito API. Therefore a plugin is used for the additional settings on the `UserPoolClient`, such as `domain`, `callbackUrls`, and `allowedOAuthFlows`.

Implementing sign up, sign in, and sign out

Implementing sign up, sign in, and sign out over and over again is not very lean. Implementing this logic in a single-page application is also not desirable. In this recipe, we will see how to implement the *OpenID Connect Implicit Flow* in a single-page application to authenticate users with the *AWS Cognito Hosted UI*.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/co
gnito-signin --path cncb-cognito-signin
```

2. Navigate to the `cncb-cognito-signin` directory with `cd cncb-cognito-signin`.
3. Review the file named `src/App.js` with the following content and update the `clientId` and `domain` fields with the values from the `userPool` stack output in the previous recipe:

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';
import { CognitoSecurity, ImplicitCallback, SecureRoute } from
'./authenticate';
import Home from './Home';

class App extends Component {
```

```
  render() {
    return (
      <Router>
        <CognitoSecurity
          domain='cncb-<stage>.auth.us-east-1.amazoncognito.com'
          clientId='a1b2c3d4e5f6g7h8i9j0k1l2m3'
          ...
redirectSignIn={`${window.location.origin}/implicit/callback`}
          redirectSignOut={window.location.origin}
        >
          <SecureRoute path='/' exact component={Home} />
          <Route path='/implicit/callback' component={ImplicitCallback}
/>
        </CognitoSecurity>
      </Router>
    );
  }
}

export default App;
```

4. Review the file named `src/Home.js` with the following content:

```
import React from 'react';
import { withAuth } from './authenticate';
...

const Home = ({ auth }) => (
  <div ...>
    ...
    <button onClick={auth.logout}>Logout</button>
    <pre ...>{JSON.stringify(auth.getSession(), null, 2)}</pre>
  </div>
);

export default withAuth(Home);
```

5. Install the dependencies with `npm install`.
6. Run the app locally with `npm start`.
7. Browse to `http://localhost:3000`.
8. Click the `Sign Up` link and follow the instructions.
9. Click the `Sign Out` link and then `Sign In` again.
10. Review the `idToken` displayed on the page.



We will use these screens to create a valid `idToken` in the authorization recipes.

How it works...

Adding sign up, sign in, and sign out to a single-page app is very straightforward. We include some additional libraries, initialize them with the proper configurations, and decorate the existing code. In this recipe, we make these changes to our simple example React application. AWS provides the `amazon-cognito-auth-js` library to simplify this task and we wrap it in some React components in `src/authenticate` folder. First, we initialize the `CognitoSecurity` component in the `src/App.js`. Next, we set up `SecureRoute` for the `Home` component that will redirect to the *Cognito hosted UI* if the user is not authenticated. The `ImpicitCallback` component will handle the redirect after the user logs in. Finally, we add the `withAuth` decorator to the `Home` component. In more elaborate applications, we would just be decorating more routes and components. The framework handles everything else, such as saving the **JSON Web Token** (**JWT**) to local storage and making it available for use in the `auth` property. For example, the `Home` component displays the tokens (`auth.getSession()`) and provides a logout button (`auth.logout`).

Securing an API Gateway with OpenID Connect

One of the advantages of using an API Gateway is that we are pushing security concerns, such as authorization, to the perimeter of our system and away from our internal resources. This simplifies the internal code and improves scalability. In this recipe, we will configure an *AWS API Gateway* to authorize against an *AWS Cognito user pool*.

Getting ready

You will need the Cognito user pool created in the *Creating a federated identity pool* recipe and the sample application created in the *Implementing sign up, sign in, and sign out* recipe to create the identity token used in this recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/co
gnito-authorizer --path cncb-cognito-authorizer
```

2. Navigate to the `cncb-cognito-authorizer` directory with `cd cncb-cognito-authorizer`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-cognito-authorizer

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole
  ...

functions:
  hello:
    handler: handler.hello
    events:
      - http:
          ...
          authorizer:
            arn: ${cf:cncb-cognito-pool-${opt:stage}.userPoolArn}
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test -- -s $MY_STAGE`.
6. Review the contents generated in the `.serverless` directory.
7. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-cognito-authorizer@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
cognito-authorizer
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
```

```
https://ff70szvc44.execute-api.us-east-1.amazonaws.com/john/hello
functions:
  hello: cncb-cognito-authorizer-john-hello
```

8. Review the stack and resources in the AWS Console.
9. Invoke the following `curl` command, after updating `<API-ID>`, to attempt to access the service without an `Authorization` token:

```
$ curl
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/hello |
json_pp

{
   "message" : "Unauthorized"
}
```

10. Use the app in the previous recipe to sign in and generate `idToken`, and then export `idToken` with the following command:

```
$ export CNCB_TOKEN=<idToken value>
```

11. Invoke the following `curl` command, after updating `<API-ID>`, to access the service successfully with an `Authorization` token:

```
$ curl -v -H "Authorization: Bearer $CNCB_TOKEN"
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/hello |
json_pp

{
   "message" : "JavaScript Cloud Native Development Cookbook! Your
function executed successfully!",
   "input" : {
      "headers" : {
         "Authorization" : "...",
         ...
      },
      ...
      "requestContext" : {
         ...
         "authorizer" : {
            "claims" : {
               "email_verified" : "true",
               "auth_time" : "1528264383",
               "cognito:username" : "john",
               "event_id" : "e091cd96-694d-11e8-897c-e3fe55ba3d67",
               "iss" :
"https://cognito-idp.us-east-1.amazonaws.com/us-east-1_tDXH8JAky",
```

```
                "exp" : "Wed Jun 06 06:53:03 UTC 2018",
                "sub" : "e4bdd021-a160-4aff-bce2-e652f9469e3e",
                "aud" : "4p5p08njgmq9sph130l3du2b7q",
                "email" : "john@example.com",
                "iat" : "Wed Jun 06 05:53:03 UTC 2018",
                "token_use" : "id"
            }
        }
    },
    ...
  }
}
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The combination of the AWS API Gateway, AWS Cognito, and the Serverless Framework make securing a service with OpenID Connect extremely straightforward. The AWS API Gateway can use authorizer functions to control access to a service. These functions verify the JWT passed in the `Authorization` header and return an IAM policy. We will delve into these details in the *Implementing a custom authorizer* recipe. AWS Cognito provides an authorizer function that verifies the JWTs generated by a specific user pool. In the `serverless.yml` file, we simply need to set `authorizer` to the `userPoolArn` of the specific Cognito user pool. Once authorized, the API Gateway passes the decode `claims` from the JWT along to the lambda function in the `requestContext`, so this data can be used in the business logic, if needs be.

Implementing a custom authorizer

In the *Securing an API Gateway with OpenID Connect* recipe, we leveraged the Cognito authorizer that is provided by AWS. This is one of the advantages of using Cognito. However, this is not the only option. Sometimes we may want more control over the policy that is returned. In other cases, we may need to use a third-party tool such as *Auth0* or *Okta*. In this recipe, we will show how to support these scenarios by implementing a custom authorizer.

Getting ready

You will need the Cognito user pool created in the *Creating a federated identity pool* recipe and the sample application created in the *Implementing sign up, sign in, and sign out* recipe to create the identity token used in this recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/cu
stom-authorizer --path cncb-custom-authorizer
```

2. Navigate to the `cncb-custom-authorizer` directory with `cd cncb-custom-authorizer`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-custom-authorizer

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole
  ...

functions:
  authorizer:
    handler: handler.authorize
    environment:
      AUD: ${cf:cncb-cognito-pool-${opt:stage}.userPoolClientId}
      ISS: ${cf:cncb-cognito-pool-${opt:stage}.userPoolProviderURL}
      JWKS: ${self:functions.authorizer.environment.ISS}/.well-
known/jwks.json
      DEBUG: '*'

  hello:
    handler: handler.hello
    events:
      - http:
          ...
          authorizer: authorizer
```

4. Review the file named `handler.js` with the following content:

```
...
module.exports.authorize = (event, context, cb) => {
  decode(event)
    .then(fetchKey)
    .then(verify)
    .then(generatePolicy)
    ...
};

const decode = ({ authorizationToken, methodArn }) => {
  ...
  return Promise.resolve({
    ...
    token: match[1],
    decoded: jwt.decode(match[1], { complete: true }),
  });
};

const fetchKey = (uow) => {
  ...
  return client.getSigningKeyAsync(kid)
    .then(key => ({
      key: key.publicKey || key.rsaPublicKey,
      ...uow,
    }));
};

const verify = (uow) => {
  ...
  return verifyAsync(token, key, {
    audience: process.env.AUD,
    issuer: process.env.ISS
  })
    ...
};

const generatePolicy = (uow) => {
  ...
  return {
    policy: {
      principalId: claims.sub,
      policyDocument: {
        Statement: [{ Action: 'execute-api:Invoke', Effect, Resource
}],
      },
      context: claims,
```

```
    },
    ...uow,
  };
};
...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-custom-authorizer@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
custom-authorizer
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  GET -
https://8iznazkhr0.execute-api.us-east-1.amazonaws.com/john/hello
functions:
  authorizer: cncb-custom-authorizer-john-authorizer
  hello: cncb-custom-authorizer-john-hello

Stack Outputs
...
ServiceEndpoint:
https://8iznazkhr0.execute-api.us-east-1.amazonaws.com/john
```

9. Review the stack and resources in the AWS Console.
10. Invoke the following `curl` command, after updating `<API-ID>`, to attempt to access the service without an `Authorization` token:

```
$ curl
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/hello |
json_pp

{
   "message" : "Unauthorized"
}
```

11. Use the app in the *Implementing sign up, sign in, and sign out* recipe to sign in and generate `idToken`, and then export `idToken` with the following command:

```
$ export CNCB_TOKEN=<idToken value>
```

12. Invoke the following `curl` command, after updating `<API-ID>`, to access the service successfully with an `Authorization` token:

```
$ curl -v -H "Authorization: Bearer $CNCB_TOKEN"
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/hello |
json_pp

{
   "message" : "JavaScript Cloud Native Development Cookbook! Your
function executed successfully!",
   "input" : {
      "headers" : {
         "Authorization" : "...",
         ...
      },
      ...
      "requestContext" : {
         ...
         "authorizer" : {
            "exp" : "1528342765",
            "cognito:username" : "john",
            "iss" :
"https://cognito-idp.us-east-1.amazonaws.com/us-east-1_tDXH8JAky",
            "sub" : "e4bdd021-a160-4aff-bce2-e652f9469e3e",
            "iat" : "1528339165",
            "email_verified" : "true",
            "auth_time" : "1528339165",
            "email" : "john@example.com",
            "aud" : "4p5p08njgmq9sph130l3du2b7q",
            "event_id" : "fdcdf125-69fb-11e8-a6ef-ab31871bed60",
            "token_use" : "id",
            "principalId" : "e4bdd021-a160-4aff-bce2-e652f9469e3e"
         }
      },
      ...
   }
}
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The process for connecting a custom authorizer to the API Gateway is the same as with the Cognito authorizer. We can implement the `authorizer` function in the same project, as we have in this recipe, or it can be implemented in another stack so that it can be shared across services. The `jwks-rsa` and `jsonwebtoken` open source libraries implement the bulk of the logic. First, we assert the presence of the token and `decode` it. Next, we use the key ID (`kid`) that is present in the decoded token to retrieve the `.well-known/jwks.json` public key for the issuer. Then, we verify the signature of the token against the key and assert that the audience (`aud`) and issuer (`iss`) are as expected. Finally, the function returns an IAM policy that grants access to the service based on the path. The `claims` of the token are also returned in the `context` field so that they can be forwarded to the backend function. If any validations fail then we return an `Unauthorized` error.



The claims object must be flattened, otherwise `AuthorizerConfigurationException` is thrown.

Authorizing a GraphQL-based service

We have seen how to use a *JWT* to authorize access to services. In addition to this coarse-grained access control, we can also leverage the claims in the JWT to perform fine-grained, role-based access control. In this recipe, we will show how to use directives to create annotations that are used to define role-based permissions declaratively in a GraphQL schema.

Getting ready

You will need the Cognito user pool created in the *Creating a federated identity pool* recipe and the sample application created in the *Implementing sign up, sign in, and sign out* recipe to create the identity token used in this recipe. You will need to assign the `Author` group, via the Cognito Console, to the user that you will use in this recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/gr
aphql-jwt --path cncb-graphql-jwt
```

2. Navigate to the `cncb-graphql-jwt` directory with `cd cncb-graphql-jwt`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-graphql-jwt

provider:
  name: aws
  runtime: nodejs8.10
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole

functions:
  graphql:
    handler: handler.graphql
    events:
      - http:
          ...
          authorizer:
            arn: ${cf:cncb-cognito-pool-${opt:stage}.userPoolArn}
```

4. Review the `index.js`, `schema/thing/typedefs.js`, and `directives.js` files with the following content:

index.js

```
...
const { directiveResolvers } = require('./directives');

const directives = `
  directive @hasRole(roles: [String]) on QUERY | FIELD | MUTATION
`;
...
module.exports = {
  typeDefs: [directives, schemaDefinition, query, mutation,
thingTypeDefs],
  resolvers: merge({}, thingResolvers),
  directiveResolvers,
};
```

schema/thing/typedefs.js

```
module.exports = `
  ...
  extend type Mutation {
    saveThing(
      input: ThingInput
    ): Thing @hasRole(roles: ["Author"])
    deleteThing(
      id: ID!
    ): Thing @hasRole(roles: ["Manager"])
  }
`;
```

directives.js

```
const getGroups = ctx => get(ctx.event,
'requestContext.authorizer.claims.cognito:groups', '');

const directiveResolvers = {
  hasRole: (next, source, { roles }, ctx) => {
    const groups = getGroups(ctx).split(',');
    if (intersection(groups, roles).length > 0) {
      return next();
    }
    throw new UserError('Access Denied');
  },
}

module.exports = { directiveResolvers };
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-graphql-jwt@1.0.0 dp:lcl <path-to-your-workspace>/cncb-graphql-
jwt
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  POST -
https://b33zxnw20b.execute-api.us-east-1.amazonaws.com/john/graphql
```

```
functions:
  graphql: cncb-graphql-jwt-john-graphql

Stack Outputs
...
ServiceEndpoint:
https://b33zxnw20b.execute-api.us-east-1.amazonaws.com/john
```

9. Review the stack and resources in the AWS Console.
10. Use the app in the *Implementing sign up, sign in, and sign out* recipe to sign in and generate `idToken`, and then export `idToken` with the following command:

```
$ export CNCB_TOKEN=<idToken value>
```

11. Invoke the following `curl` commands, after updating `<API-ID>`, to successfully access the service with an `Authorization` token:

```
$ curl -v -X POST -H "Authorization: Bearer $CNCB_TOKEN" -H 'Content-
Type: application/json' -d '{"query":"mutation { saveThing(input: { id:
\"55555555-6666-1111-1111-000000000000\", name: \"thing1\",
description: \"This is thing one of two.\" }) { id } }"}'
https://b33zxnw20b.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp

{
   "data" : {
      "saveThing" : {
         "id" : "55555555-6666-1111-1111-000000000000"
      }
   }
}

$ curl -v -X POST -H "Authorization: Bearer $CNCB_TOKEN" -H 'Content-
Type: application/json' -d '{"query":"query { thing(id:
\"55555555-6666-1111-1111-000000000000\") { id name description }}"}'
https://b33zxnw20b.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp
{
   "data" : {
      "thing" : {
         "name" : "thing1",
         "id" : "55555555-6666-1111-1111-000000000000",
         "description" : "This is thing one of two."
      }
   }
}
```

```
$ curl -v -X POST -H "Authorization: Bearer $CNCB_TOKEN" -H 'Content-
Type: application/json' -d '{"query":"mutation { deleteThing( id:
\"55555555-6666-1111-1111-000000000000\" ) { id } }"}'
https://b33zxnw20b.execute-api.us-east-1.amazonaws.com/$MY_STAGE/graphq
l | json_pp
{
   "data" : {
      "deleteThing" : null
   },
   "errors" : [
      {
         "message" : "Access Denied"
      }
   ]
}
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The service is configured with a Cognito `authorizer` that verifies the token and forwards `claims`. These `claims` include the `groups` that the user is a member of. At design time, we want to define the roles declaratively that are required to access privileged actions. In GraphQL, we can annotate a `schema` using `directives`. In this recipe, we define a `hasRole` directive and implement a `resolver` that checks the allowed `roles` defined in the annotation against `groups` present in the claims, and then it either allows or denies access. The `resolver` logic is decoupled from `schema` and the annotations in `schema` are straightforward and clean.

Implementing a JWT filter

We have seen how to use a JWT to authorize access to services and how we can use the claims in the token to perform fine-grained, role-based authorization on actions within a service. We usually need to control access at the data instance level as well. For example, a customer should only have access to his or her data, or an employee should only have access to the data for a specific division. To accomplish this, we typically adorn filters to queries based on the user's entitlements. In a RESTful API, this information is usually included in the URL as path parameters as well. It is typical to use path parameters to perform queries.

However, we want to use the claims in the JWT to perform filters instead, because the values in the token are asserted by the authenticity of the token signature. In this recipe, we will demonstrate how to use the claims in the JWT to create query filters.

Getting ready

You will need the Cognito user pool created in the *Creating a federated identity pool* recipe and the sample application created in the *Implementing sign up, sign in, and sign out* recipe to create the identity token used in this recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/jw
t-filter --path cncb-jwt-filter
```

2. Navigate to the `cncb-jwt-filter` directory with `cd cncb-jwt-filter`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-jwt-filter

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole
  ...

functions:
  save:
    handler: handler.save
    events:
      - http:
          ...
          authorizer:
            arn: ${cf:cncb-cognito-pool-${opt:stage}.userPoolArn}
  get:
    handler: handler.get
    events:
      - http:
          path: things/{sub}/{id}
          ...
          authorizer:
```

```
            arn: ${cf:cncb-cognito-pool-${opt:stage}.userPoolArn}

resources:
  Resources:
    Table:
      ...
        KeySchema:
          - AttributeName: sub
            KeyType: HASH
          - AttributeName: id
            KeyType: RANG
```

4. Review the file named `handler.js` with the following content:

```
module.exports.save = (request, context, callback) => {
  const body = JSON.parse(request.body);
  const sub = request.requestContext.authorizer.claims.sub;
  const id = body.id || uuid.v4();

  const params = {
    TableName: process.env.TABLE_NAME,
    Item: {
      sub,
      id,
      ...body
    }
  };

  ...
  db.put(params, (err, resp) => {
    ...
  });
};

module.exports.get = (request, context, callback) => {
  const sub = request.requestContext.authorizer.claims.sub;
  const id = request.pathParameters.id;

  if (sub !== request.pathParameters.sub) {
    callback(null, { statusCode: 401 });
    return;
  }

  const params = {
    TableName: process.env.TABLE_NAME,
    Key: {
      sub,
      id,
```

```
    },
  };

  ...
  db.get(params, (err, resp) => {
    ...
  });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-jwt-filter@1.0.0 dp:lcl <path-to-your-workspace>/cncb-jwt-filter
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  POST -
https://ho1q4u5hp6.execute-api.us-east-1.amazonaws.com/john/things
  GET -
https://ho1q4u5hp6.execute-api.us-east-1.amazonaws.com/john/things/{sub
}/{id}
functions:
  save: cncb-jwt-filter-john-save
  get: cncb-jwt-filter-john-get
```

9. Review the stack and resources in the AWS Console.
10. Use the app in the *Implementing sign up, sign in, and sign out* recipe to sign in and generate `idToken`, and then export `idToken` with the following command:

```
$ export CNCB_TOKEN=<idToken value>
```

11. Invoke the following `curl` commands, after updating `<API-ID>`, to access data successfully within the restrictions of the `Authorization` token:

```
$ curl -v -X POST -H "Authorization: Bearer $CNCB_TOKEN" -H 'Content-
Type: application/json' -d '{ "id":
"55555555-7777-1111-1111-000000000000", "name": "thing1",
"description": "This is thing one of two." }'
```

```
https://<API-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/things

< HTTP/1.1 201 Created
< location:
https://ho1q4u5hp6.execute-api.us-east-1.amazonaws.com/john/things/e4bd
d021-a160-4aff-bce2-e652f9469e3e/55555555-7777-1111-1111-000000000000

$ curl -v -H "Authorization: Bearer $CNCB_TOKEN" <Location response
header from POST> | json_pp

{
   "description" : "This is thing one of two.",
   "id" : "55555555-7777-1111-1111-000000000000",
   "name" : "thing1",
   "sub" : "e4bdd021-a160-4aff-bce2-e652f9469e3e"
}

$ curl -v -H "Authorization: Bearer $CNCB_TOKEN"
https://<API-ID>.execute-api.us-
east-1.amazonaws.com/$MY_STAGE/things/<An invalid
value>/55555555-7777-1111-1111-000000000000 | json_pp

< HTTP/1.1 401 Unauthorized
```

12. Take a look at the logs:

```
$ sls logs -f save -r us-east-1 -s $MY_STAGE

$ sls logs -f get -r us-east-1 -s $MY_STAGE
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The client of an API can use any values to formulate a URL, but it cannot tamper with the content of the JWT token because the issuer has signed the token. Therefore, we need to override any request values with the values from the token. In this recipe, we are saving and retrieving data for a specific user as determined by the subject or subclaim in the user's token. The service is configured with an `authorizer` that verifies the token and forwards `claims`.

To simplify the example, the subject is used as the `HASH` key and the data `uuid` as the `RANGE` key. When the data is retrieved, we assert the query parameter against the value in the token and return a `401 statusCode` if they do not match. If they match, we use the value from the token in the actual query to guard against any bugs in the assertion logic that could inadvertently return unauthorized data.

Using envelope encryption

Encrypting data at rest is critical for most systems. We must ensure the privacy of our customer's data and of our corporate data. Unfortunately, we all too often turn on disk-based encryption and then check off the requirement as complete. However, this only protects the data when the disk is disconnected from the system. When the disk is connected, then the data is automatically decrypted when it is read from disk. For example, create a DynamoDB table with server-side encryption enabled and then create some data and view it in the console. So long as you have permission, you will be able to see the data in clear text. To truly ensure the privacy of data at rest, we must encrypt data at the application level and effectively redact all sensitive information. In this recipe, we use the AWS **Key Management Service** (**KMS**) and a technique called *envelope encryption* to secure data at rest in a DynamoDB table.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/en
velope-encryption --path cncb-envelope-encryption
```

2. Navigate to the `cncb-envelope-encryption` directory with `cd cncb-envelope-encryption`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-envelope-encryption

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole
  runtime: nodejs8.10
  endpointType: REGIONAL
  iamRoleStatements:
    ...
```

```
    environment:
      TABLE_NAME:
        Ref: Table
      MASTER_KEY_ALIAS:
        Ref: MasterKeyAlias

  functions:
    save:
      ...
    get:
      ...

  resources:
    Resources:
      Table:
        ...

      MasterKey:
        Type: AWS::KMS::Key
        Properties:
          KeyPolicy:
            Version: '2012-10-17'
            ...

      MasterKeyAlias:
        Type: AWS::KMS::Alias
        Properties:
          AliasName: alias/${self:service}-${opt:stage}
          TargetKeyId:
            Ref: MasterKey
    ...
```

4. Review the file named `handler.js` with the following content:

```
const encrypt = (thing) => {
  const params = {
    KeyId: process.env.MASTER_KEY_ALIAS,
    KeySpec: 'AES_256',
  };

  ...
  return kms.generateDataKey(params).promise()
    .then((dataKey) => {
      const encryptedThing = Object.keys(thing).reduce((encryptedThing,
key) => {
        if (key !== 'id')
          encryptedThing[key] =
            CryptoJS.AES.encrypt(thing[key], dataKey.Plaintext);
```

```
        return encryptedThing;
      }, {});

      return {
        id: thing.id,
        dataKey: dataKey.CiphertextBlob.toString('base64'),
        ...encryptedThing,
      };
    });
};

const decrypt = (thing) => {
  const params = {
    CiphertextBlob: Buffer.from(thing.dataKey, 'base64'),
  };

  ...
  return kms.decrypt(params).promise()
    .then((dataKey) => {
      const decryptedThing = Object.keys(thing).reduce((decryptedThing,
key) => {
        if (key !== 'id' && key !== 'dataKey')
          decryptedThing[key] =
            CryptoJS.AES.decrypt(thing[key], dataKey.Plaintext);
        return decryptedThing;
      }, {});

      return {
        id: thing.id,
        ...decryptedThing,
      };
    });
};

module.exports.save = (request, context, callback) => {
  const thing = JSON.parse(request.body);
  const id = thing.id || uuid.v4();

  encrypt(thing)
    .then((encryptedThing) => {
      const params = {
        TableName: process.env.TABLE_NAME,
        Item: {
          id,
          ...encryptedThing,
        }
      };
```

```
      ...
      return db.put(params).promise();
    })
    ...
};

module.exports.get = (request, context, callback) => {
  const id = request.pathParameters.id;
  ...
  db.get(params).promise()
    .then((resp) => {
      return resp.Item ? decrypt(resp.Item) : null;
    })
    ...
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-envelope-encryption@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
envelope-encryption
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
endpoints:
  POST -
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/john/things
  GET -
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/john/things/{id}
functions:
  save: cncb-envelope-encryption-john-save
  get: cncb-envelope-encryption-john-get

Stack Outputs
...
MasterKeyId: c83de811-bda8-4bdc-83e1-32e8491849e5
MasterKeyArn: arn:aws:kms:us-east-1:123456789012:key/c83de811-
bda8-4bdc-83e1-32e8491849e5
```

```
ServiceEndpoint:
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/john
MasterKeyAlias: alias/cncb-envelope-encryption-john
```

9. Review the stack and resources in the AWS Console.
10. Invoke the following `curl` commands, after updating `<API-ID>`, to encrypt and decrypt data:

```
$ curl -v -X POST -d '{ "id": "55555555-8888-1111-1111-000000000000",
"name": "thing1", "description": "This is thing one of two." }'
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/$MY_STAGE/things

< HTTP/1.1 201 Created
< location:
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/john/things/5555
5555-8888-1111-1111-000000000000

$ curl -v
https://7wpqdcsoad.execute-api.us-east-1.amazonaws.com/$MY_STAGE/things
/55555555-8888-1111-1111-000000000000 | json_pp

{
   "name" : "thing1",
   "id" : "55555555-8888-1111-1111-000000000000",
   "description" : "This is thing one of two."
}
```

11. Review the encrypted data in the DynamoDB Console.
12. Take a look at the logs:

```
$ sls logs -f save -r us-east-1 -s $MY_STAGE

$ sls logs -f get -r us-east-1 -s $MY_STAGE
```

13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.



KMS is not included in the AWS free tier.

How it works...

Envelope encryption is, in essence, the practice of encrypting one key with another key; sensitive data is encrypted with a data key and then the data key is encrypted with a master key. In this recipe, the `save` function encrypts the data before saving it to DynamoDB, and the `get` function decrypts the data after retrieving it from DynamoDB and before returning the data to the caller. In the `serverless.yml` file, we define a KMS `MasterKey` and a `MasterKeyAlias`. The alias facilitates the rotation of the master key. The `save` function calls `kms.generateDataKey` to create a data key for the object. Each object gets its own data key and a new key is generated each time the object is saved. Again, this practice facilitates key rotation. Following the *security-by-design* practice, we identify which fields are sensitive as we design and develop a service. In this recipe, we encrypt all of the fields individually. The data key is used to encrypt each field locally using an `AES` encryption library. The data key was also encrypted by the master key and returned to the `CyphertextBlob` field when the data key was generated. The encrypted data key is stored alongside the data so that it can be decrypted by the `get` function. The `get` function has direct access to the encrypted data key once the data has been retrieved from the database. The `get` function has been granted permission to call `kms.decrypt` to decrypt the data key. This piece of the puzzle is crucial. Access to the master key must be restricted as per the least privilege principle. The fields are decrypted locally using an AES encryption library so that they can be returned to the caller.

Creating an SSL certificate for encryption in transit

Encrypting data in transit is critical for systems with sensitive data, which accounts for most systems these days. Fully managed cloud services, such as function-as-a-service, cloud-native databases, and API Gateways, encrypt data in transit as a matter of course. This helps ensure that our data in motion is secured across the full stack, with little to no effort on our part. However, we ultimately want to expose our cloud-native resources via custom domain names. To do this and support SSL, we must provide our own SSL certificates. This process can be tedious and we must ensure that we rotate certificates in a timely manner before they expire and cause a system outage. Fortunately, more and more cloud providers are offering fully managed certificates that are automatically rotated. In the recipe, we will use the *AWS Certificate Manager* to create a certificate and associate it with a *CloudFront* distribution.

Getting ready

You will need a registered domain name and a Route 53 hosted zone that you can use in this recipe to create a subdomain for the site that will be deployed, and you will need authority to approve the creation of the certificate or access to someone with the authority.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/ss
l-cert --path cncb-ssl-cert
```

2. Navigate to the `cncb-ssl-cert` directory with `cd cncb-ssl-cert`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-ssl-cert

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole

plugins:
 - serverless-spa-deploy
 - serverless-spa-config

custom:
  spa:
    ...
  dns:
    hostedZoneId: ZXXXXXXXXXXXX
    validationDomain: example.com
    domainName: ${self:custom.dns.validationDomain}
    wildcard: '*.${self:custom.dns.domainName}'
    endpoint: ${opt:stage}-
${self:service}.${self:custom.dns.domainName}
  cdn:
    acmCertificateArn:
      Ref: WildcardCertificate

resources:
  Resources:
    WildcardCertificate:
      Type: AWS::CertificateManager::Certificate
```

```
      Properties:
        DomainName: ${self:custom.dns.wildcard}
        DomainValidationOptions:
          - DomainName: ${self:custom.dns.wildcard}
            ValidationDomain: ${self:custom.dns.validationDomain}
        SubjectAlternativeNames:
          - ${self:custom.dns.domainName}

  ...
```

4. Update the `serverless.yml` file with your `hostedZoneId` and `validationDomain`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-ssl-cert@1.0.0 dp:lcl <path-to-your-workspace>/cncb-ssl-cert
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...

Stack Outputs
...
WebsiteDistributionURL: https://d19i44112h4l3r.cloudfront.net
...
WebsiteURL: https://john-cncb-ssl-cert.example.com
WebsiteDistributionId: EQSJSWLD0F1JI
WildcardCertificateArn: arn:aws:acm:us-
east-1:870671212434:certificate/72807b5b-fe37-4d5c-8f92-25ffcccb6f79

Serverless: Path: ./build
Serverless: File: index.html (text/html)
```



To complete this step, an authorized person is needed to receive the certificate approval request email and approve the creation of the certificate. The CloudFormation stack will appear to pause until the certificate is approved.

9. Review the stack and resources in the AWS Console.
10. Invoke the following `curl` command with the `WebsiteURL` from the stack output:

```
$ curl -v -https://john-cncb-ssl-cert.example.com
```

11. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In the resources section of the `serverless.yml` file, we define the `WildcardCertificate` resource to instruct the AWS Certificate Manager to create the desired certificate. We create a wildcard certificate so that it can be used by many services. It is important to specify the correct `validationDomain`, because this will be used to request approval before creating the certificate. You must also provide `hostedZoneId` for the top-level domain name. Everything else is handled by the `serverless-spa-config` plugin. The recipe deploys a simple site consisting of an `index.html` page. The `endpoint` for the site is used to create a Route 53 record set in the hosted zone and is assigned as the alias of the site's CloudFront distribution. The wildcard certificate matches the endpoint and its `acmCertificateArn` is assigned to the distribution. Ultimately, we can access `WebsiteURL` with the `https` protocol and the custom domain name.

Configuring a web application firewall

A **web application firewall** (**WAF**) is an important tool for controlling the traffic of a cloud-native system. A WAF protects the system by blocking traffic from common exploits such as bad bots, SQL injection, **Cross-Site Scripting** (**XSS**), HTTP floods, and known attackers. We effectively create a perimeter around the system that blocks traffic at the edge of the cloud before it can impact system resources. We can use many of the techniques in this book to monitor internal and external resources and dynamically update the firewall rules as the flow of traffic changes. Managed rules are also available on the market, so we can leverage the extensive security expertise of third parties. In this recipe, we will demonstrate how the pieces fit together by creating a rule to block traffic from outside a corporate network and associate the WAF with a *CloudFront* distribution.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/waf --path cncb-waf
```

2. Navigate to the `cncb-waf` directory with `cd cncb-waf`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-waf

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole

plugins:
 - serverless-spa-deploy
 - serverless-spa-config

custom:
  spa:
    ...
  cdn:
    webACLId:
      Ref: WebACL
    # logging:
    #   bucketName: ${cf:cncb-account-as-code-
${opt:stage}.AuditBucketName}

resources:
  Resources:
    WhitelistIPSet:
      Type: AWS::WAF::IPSet
      Properties:
        Name: IPSet for whitelisted IP adresses
        IPSetDescriptors:
          - Type: IPV4
            Value: 0.0.0.1/32

    WhitelistRule:
      Type: AWS::WAF::Rule
      Properties:
        Name: WhitelistRule
        MetricName: WhitelistRule
        Predicates:
```

```
          - DataId:
              Ref: WhitelistIPSet
            Negated: false
            Type: IPMatch
    WebACL:
      Type: AWS::WAF::WebACL
      Properties:
        Name: Master WebACL
        DefaultAction:
          Type: BLOCK
        MetricName: MasterWebACL
        Rules:
          - Action:
              Type: ALLOW
            Priority: 1
            RuleId:
              Ref: WhitelistRule

Outputs:
  WebACLId:
    Value:
      Ref: WebACL
```

4. Determine your corporate external public IP address by logging in to your VPN and then executing the following command:

```
$ curl ipecho.net/plain ; echo
```

5. Update `WhitelistIPSet` in the `serverless.yml` file with your IP address followed by `/32`.
6. Install the dependencies with `npm install`.
7. Run the tests with `npm test`.
8. Review the contents generated in the `.serverless` directory.
9. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-waf@1.0.0 dp:lcl <path-to-your-workspace>/cncb-waf
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...

Stack Outputs
```

```
WebsiteDistributionURL: https://d3a9sc88i7431l.cloudfront.net
WebsiteS3URL:
http://cncb-waf-john-websitebucket-1c13hrzslok5s.s3-website-us-east-1.a
mazonaws.com
WebsiteBucketName: cncb-waf-john-websitebucket-1c13hrzslok5s
WebsiteDistributionId: E3OLRBU9LRZBUE
WebACLId: 68af80b5-8eda-43d0-be25-6c65d5cc691e

Serverless: Path: ./build
Serverless: File: index.html (text/html)
```

10. Review the stack and resources in the AWS Console.
11. Browse to the `WebsiteDistributionURL` listed in the stack outputs while connecting to your VPN and you will be able to access the page.
12. Now, disconnect from the VPN and you will be denied access to the page once the cache has cleared.
13. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The first thing to note is that usage of the AWS WAF is predicated on using CloudFront. The `serverless-spa-config` plugin creates the CloudFront distribution and assigns `webACLId`. For this recipe, we are securing a simple `index.html` page. We create a static `WebACL` to block everything except a single IP address. We define the `WhitelistIPSet` for the single address and associate it with `WhitelistRule`. Then, we associate the rule with an **access control list** (**ACL**) and define the default action to `BLOCK` all access, and then an action to `ALLOW` access based on `WhitelistRule`.

This example is useful, but it only scratches the surface of what is possible. For example, we could uncomment the logging configuration for the CloudFront distribution and then process the access logs and dynamically create rules based on suspicious activity. On a daily basis, we can retrieve public reputation lists and update the rule sets as well. Defining and maintaining effective rules can be a full-time effort, therefore the fully managed rules that are available on the AWS marketplace are an attractive alternative for, or companion to, custom rules.

Replicating the data lake for disaster recovery

When I first read this story about Code Spaces (`https://www.infoworld.com/article/2608076/data-center/murder-in-the-amazon-cloud.html`), I was a bit horrified until I realized that this company perished so that we could all learn from its experience. Code Spaces was a company that used AWS and their account was hijacked and held to ransom. They fought back and the entire contents of their account were deleted, including their backups, and they simply went out of business. Proper use of MFA and proper access key hygiene is critical to ward off such attacks. It is also crucial to maintain backups as an entirely separate and disconnected account, so that the breach of a single account does not spell disaster for an entire system and company. In this recipe, we will use the S3 replication feature to replicate buckets to a dedicated recovery account. At a minimum, this technique should be used to replicate the contents of the data lake, since the events in the data lake can be replayed to recreate the system.

How to do it...

1. Create the source and recovery projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/dr
/recovery-account --path cncb-dr-recovery-account

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch5/dr
/src1-account --path cncb-dr-src1-account
```

2. Navigate to the `cncb-dr-recovery-account` directory with `cd cncb-dr-recovery-account`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-dr-recovery-account

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole

custom:
  accounts:
    src1:
      accountNumber: '#{AWS::AccountId}' # using same account to
```

```
simplify recipe
...
resources:
  Resources:
    DrSrc1Bucket1:
      Type: AWS::S3::Bucket
      DeletionPolicy: Retain
      Properties:
        BucketName: cncb-${opt:stage}-us-west-1-src1-bucket1-dr
        VersioningConfiguration:
          Status: Enabled
    DrSrc1Bucket1Policy:
      Type: AWS::S3::BucketPolicy
      Properties:
        Bucket:
          Ref: DrSrc1Bucket1
        PolicyDocument:
          Statement:
            - Effect: Allow
              Principal:
                AWS:
arn:aws:iam::${self:custom.accounts.src1.accountNumber}:root
              Action:
                - s3:ReplicateDelete
                - s3:ReplicateObject
                - s3:ObjectOwnerOverrideToBucketOwner
              Resource:
                ...
  ...
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory.
7. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-dr@1.0.0 dp:lcl <path-to-your-workspace>/cncb-dr
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
Stack Outputs
DrSrc1Bucket1Name: cncb-john-us-west-1-src1-bucket1-dr
```

8. Navigate to the `cncb-dr-src1-account` directory with `cd cncb-dr-src1-account`.
9. Review the file named `serverless.yml` with the following content:

```
service: cncb-dr-src1-account

provider:
  name: aws
  # cfnRole: arn:aws:iam::<account-number>:role/${opt:stage}-cfnRole
  ...

custom:
  replicationBucketArn: arn:aws:s3:::cncb-${opt:stage}-us-west-1-src1-
bucket1-dr
  recovery:
    accountNumber: '#{AWS::AccountId}' # using same account to simplify
recipe
...
resources:
  Resources:
    Src1Bucket1:
      Type: AWS::S3::Bucket
      DeletionPolicy: Retain
      Properties:
        BucketName: cncb-${opt:stage}-us-east-1-src1-bucket1
        VersioningConfiguration:
          Status: Enabled
        ReplicationConfiguration:
          Role: arn:aws:iam::#{AWS::AccountId}:role/${self:service}-
${opt:stage}-${opt:region}-replicate
          Rules:
            - Destination:
                Bucket: ${self:custom.replicationBucketArn}
                StorageClass: STANDARD_IA
                Account: ${self:custom.recovery.accountNumber}
                AccessControlTranslation:
                  Owner: Destination
              Status: Enabled
              Prefix: ''

    Src1Bucket1ReplicationRole:
      DependsOn: Src1Bucket1
      Type: AWS::IAM::Role
      Properties:
        RoleName: ${self:service}-${opt:stage}-${opt:region}-replicate
        AssumeRolePolicyDocument:
          Statement:
```

```
            - Effect: Allow
              Principal:
                Service:
                  - s3.amazonaws.com
              Action:
                - sts:AssumeRole
        Policies:
          - PolicyName: replicate
            PolicyDocument:
              Statement:
                ...
                - Effect: Allow
                  Action:
                    - s3:ReplicateObject
                    - s3:ReplicateDelete
                    - s3:ObjectOwnerOverrideToBucketOwner
                  Resource: ${self:custom.replicationBucketArn}/*
...
```

10. Install the dependencies with `npm install`.
11. Run the tests with `npm test`.
12. Review the contents generated in the `.serverless` directory.
13. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-dr@1.0.0 dp:lcl <path-to-your-workspace>/cncb-dr
> sls deploy -v -r us-east-1 "-s" "john"

Serverless: Packaging service...
...
Serverless: Stack update finished...
...
Stack Outputs
Src1Bucket1Name: cncb-john-us-east-1-src1-bucket1
```

14. Review the stacks and buckets in the AWS Console.
15. Invoke the following command to load data:

```
$ sls invoke -r us-east-1 -f load -s $MY_STAGE
```

16. Review the contents of both buckets to see that the data has replicated.
17. Remove each stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`

How it works...

AWS S3 does all the heavy lifting of replicating the contents of one bucket to another. We just have to define what and where we want to replicate and set up all the permissions correctly. It is important to replicate to a separate account that is completely disconnected from any other accounts to ensure that a breach in another account does not impact the recovery account. To simplify this recipe, we will use a single account, but this does not change the CloudFormation templates other than the account number value. It is also a good idea to replicate into a region that you are not otherwise using, so that an outage in one of your regions does not also mean there is an outage in your recovery region. Next, we need to create a corresponding bucket in the recovery account for every source bucket that we want to replicate. It is a good idea to use a naming convention with prefixes and/or suffixes to differentiate the source bucket easily from the destination bucket. All of the buckets must turn on versioning to support replication. The source buckets grant permission to the S3 service to perform the replication, and the destination buckets grant permission to the source account to write to the recovery account. We also transfer ownership of the replicated contents to the recovery account. Ultimately, if the contents are completely deleted from the source bucket, the destination bucket will still contain the contents along with the deletion marker.

6
Building a Continuous Deployment Pipeline

In this chapter, the following recipes will be covered:

- Creating the CI/CD pipeline
- Writing unit tests
- Writing integration tests
- Writing contract tests for a synchronous API
- Writing contract tests for an asynchronous API
- Assembling transitive end-to-end tests
- Leveraging feature flags

Introduction

Throughout the preceding chapters, we have seen how cloud-native is lean and autonomous. Leveraging fully-managed cloud services and establishing proper bulkheads empowers self-sufficient, full-stack teams to rapidly and continuously deliver autonomous services with the confidence that a failure in any one service will not cripple the upstream and downstream services that depend on it. This architecture is a major advancement because these safeguards protect us from inevitable human errors. However, we must still endeavor to minimize human error and increase our confidence in our systems.

To minimize and control potential mistakes, we need to minimize and control our batch sizes. We accomplish this by following the practice of *decoupling deployment from release*. A **deployment** is just the act of deploying a piece of software into an environment, whereas a **release** is just the act of making that software available to a set of users. Following **lean** methods, we release functionality to users in a series of small, focused experiments that determine whether or not the solution is on the right track, so that timely course corrections can be made. Each experiment consists of a set of stories, and each story consists of a set of small focused tasks. These tasks are our unit of deployment. For each story, we plan a roadmap that will continuously deploy these tasks in an order that accounts for all inter-dependencies, so that there is zero downtime. The practices that govern each individual task are collectively referred to as a **task branch workflow**. The recipes in this chapter demonstrate the inner-working of a task branch workflow and how, ultimately, we enable these features for users with feature flags.

Creating the CI/CD pipeline

Small batch sizes reduce deployment risk because it is much easier to reason about their correctness and much easier to correct them when they are in error. Task branch workflow is a Git workflow that is focused on extremely short-lived branches, in the range of just hours rather than days. It is similar to an *issue branch workflow*, in that each task is tracked as an issue in the project management tool. The length of an issue is ambiguous, however, because an issue can be used to track an entire feature. This recipe demonstrates how issue tracking, Git branches, pull requests, testing, code review, and the CI/CD pipeline work together in a task branch workflow to govern small focused units of deployment.

Getting ready

Before starting this recipe, you will need to have an account on GitLab (`https://about.gitlab.com/`).

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/pi
peline --path cncb-pipeline
```

2. Navigate to the `cncb-pipeline` directory, `cd cncb-pipeline`.
3. Initialize the Git repository locally and remotely in `gitlab.com`, as follows:

```
$ git init
$ git remote add origin git@gitlab.com:<username>/cncb-pipeline.git
$ git add .gitignore
$ git commit -m "initial commit"
$ git push -u origin master
```

4. Confirm that the project was created in your Gitlab account.
5. Create a new issue in the project named `intialize-project`.
6. Press the **Create merge request** button and note the name of the branch, such as `1-initialize-project`.
7. Check out the branch locally at with `git pull && git checkout 1-initialize-project`.
8. Review the file named `.gitlab-ci.yml` with the following content:

```
image: node:8

before_script:
  - cp .npmrc-conf .npmrc
  - npm install --unsafe-perm

test:
  stage: test
  script:
    - npm test
    - npm run test:int

stg-east:
  stage: deploy
  variables:
    AWS_ACCESS_KEY_ID: $DEV_AWS_ACCESS_KEY_ID
    AWS_SECRET_ACCESS_KEY: $DEV_AWS_SECRET_ACCESS_KEY
  script:
    - npm run dp:stg:e
  except:
    - master

production-east:
  stage: deploy
  variables:
    AWS_ACCESS_KEY_ID: $PROD_AWS_ACCESS_KEY_ID
    AWS_SECRET_ACCESS_KEY: $PROD_AWS_SECRET_ACCESS_KEY
  script:
```

```
    - npm run dp:prd:e
  only:
    - master
```

9. Review the file named `package.json` with the following content:

```
...
  "scripts": {
    "test": "echo running unit tests...",
    "test:int": "echo running integration tests...",
    ...
    "dp:stg:e": "sls deploy -v -r us-east-1 -s stg --acct dev",
    "dp:prd:e": "sls deploy -v -r us-east-1 -s prd --acct prod"
  },
...
```

10. Review the file named `serverless.yml` with the following content:

```
service: cncb-pipeline

provider:
  name: aws
  # cfnRole:
arn:aws:iam::${self:custom.accounts.${opt:acct}.accountNumber}:role/${o
pt:stage}-cfnRole

custom:
  accounts:
    dev:
      accountNumber: 123456789012
    prod:
      accountNumber: 123456789012
```

11. Configure the `DEV_AWS_ACCESS_KEY_ID`, `DEV_AWS_SECRET_ACCESS_KEY`, `PROD_AWS_ACCESS_KEY_ID`, `PROD_AWS_SECRET_ACCESS_KEY`, and `NPM_TOKEN` environment variables in the GitLab project (`https://gitlab.com/`) under **Settings | CI/CD | Variables.**
12. Push the project files to the remote repository, as follows:

```
$ git pull
$ git add .
$ git commit -m "initialize project"
$ git push origin 1-initialize-project
```



Before pushing your changes, you should always execute `git pull` to keep your task branch in sync with the master branch.

13. Review the code in the merge request.
14. Review the progress of the branch pipeline.
15. Review the `cncb-pipeline-stg` stack in the AWS Console.
16. Remove the `WIP:` prefix from the name of the merge request and accept the merge request.



It is best to start a pull request as early as possible to receive feedback as early as possible. The `WIP:` prefix indicates to reviewers that work is still progressing. The prefix is purely procedural, but GitLab will not allow a merge request with a WIP prefix to be accidentally accepted.

17. Review the progress of the master pipeline.
18. Review the `cncb-pipeline-prd` stack in the AWS Console.
19. Remove both stacks once you are finished.

How it works...

We are using `GitLab.com` simply because it is a freely-available and hosted toolset that is well-integrated. There are other alternatives, such as Bitbucket Pipelines, that require a little more elbow grease to stand up but they still offer comparable features. A `bitbucket-pipelines.yml` file is included in the recipes for comparison.

As we have seen throughout this cookbook, our unit of deployment is a stack, as defined by a `serverless.yml` file in the root of a project directory. As we see in this recipe, each project is managed in its own Git repository and has its own CI/CD pipeline. The pipeline is defined by a configuration file that lives in the root of the project as well, such as a `.gitlab-ci.yml` or `bitbucket-pipelines.yml` file. These pipelines are integrated with the Git branching strategy and are governed by pull requests.



Note that GitLab uses the term *merge request*, whereas other tools use the term *pull request*. The two terms can be used interchangeably.

A task branch workflow begins when an issue or task is pulled in from the backlog and used to create a branch in the repository. A pull request is created to govern the branch. The pipeline executes all tests on the branch and its progress is displayed in the pull request. Once the tests are considered successful, the pipeline deploys the stack to the staging environment in the development account. A code review is performed in the pull request and discussion is recorded with comments. Once everything is in order, the pull request can be accepted to merge the changes to the master branch and trigger deployment of the stack to the production environment in the production account.

The first line of the pipeline definition denotes that the `node:8` Docker image will be used to execute the pipeline. The rest of the pipeline definition orchestrates the steps we have been executing manually throughout this cookbook. First, `npm install` installs all the dependencies as defined in the `package.json` file. Then, we execute all the tests on the given branch. Finally, we deploy the stack to the specific environment and region with `npm`; in this case, `npm run dp:stg:e` or `npm run dp:prd:e`. The details of each step are encapsulated in the `npm` scripts.



Note that, throughout this cookbook, we have been using the `npm run dp:lcl` script to perform our deployments. These allow each developer to perform development and testing in a personal stack (that is, local or `lcl`) to help ensure that the staging (`stg`) environment stays stable and therefore the production (`prd`) environment as well.

Environment variables, such as `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, are securely stored by the pipeline and never logged. We define a set of variables per account, as identified by the `DEV_` and `PROD_` prefix, and then map them in the pipeline definition. In the *Securing your cloud account* recipe, we created `CiCdUser` to grant permissions to the pipelines. Here, we need to manually create an access key for those users and securely store them as pipeline variables. The keys and pipeline variables are then periodically rotated and updated.

The pipeline deploys the stack to the staging environment in the development account for each task branch, and to the production environment in the production account for the master branch. The access key determines which account is used and the Serverless Framework `-s` option fully qualifies the name of the stack. We then add an additional option called `--acct` to allow us to index into account-scoped custom variables, such as `${self:custom.accounts.${opt:acct}.accountNumber}`. To help avoid confusion between the production stage and the production account, we need to use use slightly different abbreviations, such as `prd` and `prod`.

Writing unit tests

Unit testing is arguably the most important type of testing and should certainly account for the majority of test cases in the test pyramid. Testing should follow a scientific method where we hold some variables constant, adjust the input, and measure the output. Unit testing accomplishes this by testing individual units in isolation. This allows unit tests to focus on functionality and maximize coverage.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/un
it-testing --path cncb-unit-testing
```

2. Navigate to the `cncb-unit-testing` directory with `cd cncb-unit-testing`.
3. Install the dependencies with `npm install`.
4. Review the file named `package.json` with the following content:

```
"scripts": {
  ...
  "pretest": "npm run clean && npm run lint",
  "test": "nyc mocha ... ./test/unit/**/*.test.js",
  ...
},
```

5. Run the unit tests, as follows:

```
$ npm test
...
  14 passing (76ms)
...
===== Coverage summary =====
Statements   : 100% ( 57/57 )
Branches     : 100% ( 4/4 )
Functions    : 100% ( 28/28 )
Lines        : 100% ( 51/51 )
```

6. Review the file named `test/unit/connector/db.test.js` with the following content:

```
it('should get by id', async () => {
  const spy = sinon.spy((params, cb) => cb(null, {
    Item: ...
  }));

  AWS.mock('DynamoDB.DocumentClient', 'get', spy);

  const data = await new Connector('t1').getById(ID);
  expect(spy).to.have.been.calledOnce;
  ...
});
```

7. Review the file named `test/unit/get/index.test.js` with the following content:

```
it('should get by id', async () => {
  ...
  const stub = sinon.stub(Connector.prototype, 'getById')
    .returns(Promise.resolve(THING));

  const data = await new Handler(TABLE_NAME).handle(REQUEST);
  expect(stub).to.have.been.calledWith(ID);
  ...
});
```

8. (Optional) Repeat the steps from the *Creating the CI/CD pipeline* recipe with this project.

How it works...

In previous recipes, we purposefully simplified the examples to a reasonable degree to highlight the specific topics. The code was correct but the recipes did not have any unit tests, because the topic had not yet been addressed. The first thing you are likely to notice in the recipes in this chapter is that we are adding additional structure to the code; for example, each function has its own directory and files and we have also added some lightweight layering to the code. This structure is intended to facilitate the testing process by making it easier to isolate the unit that is under test. So, let's now dig deeper into the tools and structure that have been added.

The first tool that is called in the `npm test` script is `nyc`, which is the command line interface for the `istanbul` code coverage tool. The `.nycrc` file configures the code coverage process. Here, we require 100% coverage. This is perfectly reasonable for the scope of our bounded, isolated, and autonomous services. It is also reasonable because we are writing the unit tests incrementally as we also incrementally build the services in a series of task branch workflows. Furthermore, without keeping the coverage at 100%, it would be too easy to skip the testing until later on in the development process, which is dangerous in a continuous deployment pipeline and defeats the purpose. Fortunately, the structure of the code makes it much easier to identify which features are lacking tests.

The `npm pretest` script runs the linting process. `eslint` is a very valuable tool. It enforces best practices, automatically fixes many violations, and identifies common problems. In essence, linting helps to teach developers how to write better code. The linting process can be tuned with the `.eslintignore` and `.eslintrc.js` files.

Isolating external dependencies is an essential part of unit testing. Our testing tools of choice are `mocha`, `chai`, `sinon`, and `aws-sdk-mock`. There are many tools available, but this combination is extremely popular. mocha is the overarching testing framework; `chai` is the assertion library; `sinon` provides test spies, stubs, and mocks; and `aws-sdk-mock` builds on sinon to simplify testing against the `aws-sdk`. To further facilitate this process, we isolate our `aws-sdk` calls inside Connector classes. This does have the added benefit of reusing code throughout the service, but its primary benefit is to simplify testing. We write unit tests specifically for the classes that use `aws-sdk-mock`. Throughout the rest of the unit test code, we stub out the connector layer, which greatly simplifies the setup of each test, because we have isolated the complexities of the `aws-sdk`.

The `Handler` classes account for the bulk of the testing. These classes encapsulate and orchestrate the business logic, and therefore will require the most testing permutations. To facilitate this effort, we decouple the `Handler` classes from the `callback` function. The `handle` method either returns a *promise* or a *stream* and the top-level function then adapts these to the callback. This allows tests to easily tap into the processing flow to assert the outputs.

Our tests are inherently asynchronous; therefore, it is important to guard against evergreen tests that do not fail when the code is broken. For handlers that return promises, the best approach is to use `async/await` to guard against swallowed exceptions. For handlers that return a stream, the best approach is to use the collect/tap/done pattern to protect against scenarios where the data does not flow all the way through the stream.

Writing integration tests

Integration tests focus on testing the API calls between dependent services. In our cloud-native systems, these are concentrated on intra-service interactions with fully-managed cloud services. They ensure that the interactions are properly coded to send and receive proper payloads. These calls require the network, but networks are notoriously unreliable. This is the major cause of flaky tests that randomly and haphazardly fail. Flaky tests, in turn, are a major cause of poor team morale. This recipe demonstrates how to use a VCR library to create test doubles that allow integration testing to be executed in isolation without a dependency on the network or the deployment of external services.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/integration-testing --path cncb-integration-testing
```

2. Navigate to the `cncb-integration-testing` directory with `cd cncb-integration-testing`.
3. Install the dependencies with `npm install`.
4. Review the file named `package.json` with the following content:

```
  "scripts": {
    ...
    "test:int": "npm start -- --exec \"mocha ...
./test/int/**/*.test.js\"",
    "start": "sls offline start --port 3001 -r us-east-1 -s stg --acct
dev",
    ...
  },
```

5. Run the unit test with `npm test`.
6. Run the integration tests in `replay` mode, as follows:

```
$ npm run test:int
...
Serverless: Replay mode = replay
Serverless: Starting Offline: stg/us-east-1.
...
  get/index.js
    ✓ should get by id
...
  3 passing (358ms)
...
Serverless: Halting offline server
```

7. Review the files in the `./fixtures` directory.
8. Review the files related to the synchronous integration tests, as follows:

serverless.yml

```
...
plugins:
  - serverless-webpack
  - baton-vcr-serverless-plugin
  - serverless-offline
...
```

test/int/get/index.test.js

```
...
const supertest = require('supertest');
const client = supertest('http://localhost:3001');
...
describe('get/index.js', () => {
  it('should get by id', () => client
    .get('/things/00000000-0000-0000-0000-000000000000')
    .expect(200)
    .expect((res) => {
      expect(JSON.parse(res.text)).to.deep.equal(THING);
    }));
});
```

webpack.config.js

```
...
const injectMocks = (entries) =>
  Object.keys(entries).reduce((e, key) => {
```

```
    e[key] = ['./test/int/mocks.js', entries[key]];
    return e;
  }, {});

const includeMocks = () => slsw.lib.webpack.isLocal &&
process.env.REPLAY != 'bloody';

module.exports = {
  entry: includeMocks() ? injectMocks(slsw.lib.entries) :
slsw.lib.entries,
...
```

9. Review the files related to the asynchronous integration tests, as follows:

test/int/trigger/index.test.js

```
describe('trigger/index.js', () => {
  before(() => {
    require('baton-vcr-replay-for-aws-sdk');
    process.env.STREAM_NAME = 'stg-cncb-event-stream-s1';
    aws.config.update({ region: 'us-east-1' });
  });

  it('should trigger', (done) => {
    new Handler(process.env.STREAM_NAME).handle(TRIGGER)
      .collect()
      .tap((data) => {
        expect(data).to.deep.equal([{
          response: RESPONSE,
          event: EVENT,
        }]);
      })
      .done(done);
  });
})
```

10. Deploy the stack with `npm run dp:stg:e`.
11. Delete the `./fixtures` directory and run the integration tests again in `record` mode, as follows:

```
$ DEBUG=replay REPLAY=record npm run test:int

...
Serverless: GET /things/00000000-0000-0000-0000-000000000000 (λ: get)
  replay Requesting POST https://dynamodb.us-east-1.amazonaws.com:443/
+0ms
  replay Creating ./fixtures/dynamodb.us-east-1.amazonaws.com-443
+203ms
```

```
  replay Received 200 https://dynamodb.us-east-1.amazonaws.com:443/
+5ms
...
```

12. (Optional) Repeat the steps from the *Creating the CI/CD pipeline* recipe with this project.

How it works...

Integration testing uses all of the same tools that are used for unit testing, plus some additional tools. This is an advantage in that the learning curve is incremental. The first new tool of interest is the `serverless-offline` plugin. This plugin reads the `serverless.yml` file and simulates the API Gateway locally to facilitate the testing of synchronous APIs. Next, we use `supertest` to make HTTP calls to the locally-running service and assert the responses. Inevitably, these services make calls to AWS services using the default or specified access key. These are the calls that we want to record and playback in the CI/CD pipeline. `baton-vcr-serverless-plugin` initializes the `Replay` VCR library in the `serverless-offline` process. By default, the VCR runs in `replay` mode and will fail if a recording is not found under the fixtures directory. When writing a new test, the developer runs the tests in `record` mode by setting the `REPLAY` environment variable, `REPLAY=record npm run test:int`. To ensure that a recording is found, we must hold constant all dynamically-generated values that are used in the requests; to accomplish this, we inject mocks into the webpack configuration. In this example, the `./test/int/mocks.js` file uses `sinon` to mock UUID.

Writing integration tests for asynchronous functions, such as `triggers` and `listeners`, is mostly similar. First, we need to manually capture the events in the log files, such as a DynamoDB Stream event, and include them in test cases. Then, we initialize the `baton-vcr-replay-for-aws-sdk` library when the tests begin. From here, the process of recording requests is the same. In the case of a `trigger` function, it will record a call to publish an event to Kinesis, where a `listener` function will typically record calls to the service's tables.

When writing integration tests, it is important to keep the test pyramid in mind. Integration tests are focused on interactions within the specific API calls; they do not need to cover all of the different functional scenarios, as that is the job of unit tests. The integration tests just need to focus on the structure of the messages to ensure they are compatible with what is expected by and returned from the external service. To support the creation of the recordings for these tests, the external system may need to be initialized with a specific dataset. This initialization should be coded as part of the tests and recorded as well.

Writing contract tests for a synchronous API

Contract testing and integration testing are two sides of the same coin. Integration tests ensure that a consumer is calling a provider service correctly, whereas contract tests ensure that the provider service continues to meet its obligations to its consumers and that any changes are backward-compatible. These tests are also consumer driven. This means that the consumer submits a pull request to the provider's project to add these additional tests. The provider is not supposed to change these tests. If a contract test breaks, it implies that a backwards-incompatible change has been made. The provider has to make the change compatible and then work with the consumer team to create an upgrade roadmap.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/co
ntract-testing-sync --path cncb-contract-testing-sync
```

2. Navigate to the `bff` directory with `cd cncb-contract-testing-sync/bff`
3. Install the dependencies with `npm install`.
4. Run the unit tests with `npm test`.
5. Run the integration tests in `replay` mode with `npm run test:int`.
6. Review the files in the `./fixtures` directory.
7. Review the file named `./test/int/frontend/contract.test.js` with the following content:

```
import 'mocha';

const supertest = require('supertest');
const relay = require('baton-request-relay');
const client = supertest('http://localhost:3001');

describe('contract/frontend', () => {
  it('should relay the frontend save request',
    () => run('./fixtures/frontend/save'));

  it('should relay the frontend get request',
    () => run('./fixtures/frontend/get'));
});

const run = fixture => {
```

```
  const rec = relay(fixture);

  return client[rec.request.method](rec.request.path)
    .set(rec.request.headers)
    .send(rec.request.body)

    .expect(rec.response.statusCode)
    .expect(rec.response.body);
};
```

8. Review the consumer's `test` and `fixture` files under `./frontend`.
9. (Optional) Repeat the steps from the *Creating the CI/CD pipeline* recipe with this project.

How it works...

When it comes to integration and contract testing, the devil is in the detail. Specifically, the detail of the individual fields, their data types, and their valid values. It is all too easy for a provider to change a seemingly mundane detail that violates a consumer's understanding of the contract. These changes then go unnoticed until the worst possible moment. In this regard, handcrafted contract tests are unreliable, because the same misunderstanding about the contract usually translates into the test as well. Instead, we need to use the same recordings on both sides of the interaction.

The consumer creates an integration test in its project that records the interaction with the provider's service. These same recordings are then copied to the provider's project and used to drive the contract tests. A consumer-specific `fixtures` subdirectory is created for the recordings. The contract tests use the `baton-request-relay` library to read the recordings so that they can be used to drive `supertest` to execute the same requests in the provider's project.

In our cloud-native systems, these synchronous requests are usually between a frontend and its **Backend for Frontend** (**BFF**) service, which is owned by the same team. The fact that the same team owns the consumer and provider does not negate the value of these tests, because even a subtle change, such as changing a short integer to a long integer, can have a drastic impact if all the wrong assumptions were made on either side.

Writing contract tests for an asynchronous API

The objective of contract testing for asynchronous APIs is to ensure backwards-compatibility between the provider and the consumer—just as it is with synchronous APIs. Testing asynchronous communication is naturally flaky, as there is the unreliability of networks plus the unreliable latency of asynchronous messaging. All too often these tests fail because messages are slow to arrive and the tests timeout. To solve this problem, we isolate the tests from the messaging system; we record the send message request on one end, then relay the message and assert the contract on the receiving side.

How to do it...

1. Create the upstream and downstream projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/co
ntract-testing-async/upstream --path cncb-contract-testing-async-
upstream

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/co
ntract-testing-async/downstream --path cncb-contract-testing-async-
downstream
```

2. Navigate to the `upstream` directory with `cd cncb-contract-testing-async-upstream`.
3. Install the dependencies with `npm install`.
4. Run the unit test with `npm test`.
5. Run the integration tests in `replay` mode with `npm run test:int`.
6. Review the files in the `./fixtures/downstream-consumer-x` directory.
7. Review the file named `./test/int/downstream-consumer-x/contract.test.js` with the following content:

```
...
const EVENT = require('../../../fixtures/downstream-consumer-x/thing-
created.json');

describe('contract/downstream-consumer-x', () => {
  before(() => {
```

```
    const replay = require('baton-vcr-replay-for-aws-sdk');
    replay.fixtures = './fixtures/downstream-consumer-x';
    process.env.STREAM_NAME = 'stg-cncb-event-stream-s1';
    aws.config.update({ region: 'us-east-1' });
  });

  afterEach(() => {
    sinon.restore();
  });

  it('should publish thing-created', (done) => {
    sinon.stub(utils,
'uuidv4').returns('00000000-0000-0000-0000-000000000001');
    handle(EVENT, {}, done);
  });
});
```

8. Navigate to the `downstream` directory with `cd ../cncb-contract-testing-async-downstream`.
9. Repeat the same steps to run the tests.
10. Review the downstream consumer's `./test/int/upstream-provider-y` and `./fixture/upstream-provider-y` files, as follows:

```
...
const relay = require('baton-event-relay');

describe('contract/upstream-provider-y', () => {
  before(() => {
    ...
    const replay = require('baton-vcr-replay-for-aws-sdk');
    replay.fixtures = './fixtures/upstream-provider-y';
  });

  it('should process the thing-created event', (done) => {
    const rec = relay('./fixtures/upstream-provider-y/thing-created');
    handle(rec.event, {}, done);
  });
});
```

11. (Optional) Repeat the steps from the *Creating the CI/CD pipeline* recipe with this project.

How it works...

The process of recording and creating a contract test for an asynchronous API is the inverse of a synchronous API. With a synchronous API, the consumer initiates the interaction, whereas with an asynchronous API, the provider initiates the publishing of the event. However, asynchronous providers are unaware of their consumers, so these tests still need to be consumer-driven.

First, the consumer project creates a test in the upstream project and records the call to publish an event to the event stream. Then, the consumer relays the recording in its own project, using the `baton-event-relay` library, to assert that the content of the event is as it expects. Once again, the provider project does not own these tests and should not fix the test if it breaks due to a backwards-incompatible change.

Assembling transitive end-to-end tests

Traditional end-to-end testing is labor-intensive and expensive. As a result, traditional batch sizes are large and end-to-end testing is performed infrequently or skipped altogether. This is the exact opposite of our objective with continuous deployment. We want small batch sizes and we want full testing on every deployment, multiple times per day, which is when we typically hear cries of heresy. This recipe demonstrates how this can be achieved with the tools and techniques already covered in this chapter.

How to do it...

1. Create the `author-frontend`, `author-bff`, `customer-bff`, and `customer-frontend` projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/tr
ansitive-testing/author-frontend --path cncb-transitive-testing-author-
frontend

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/tr
ansitive-testing/author-bff --path cncb-transitive-testing-author-bff

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/tr
ansitive-testing/customer-bff --path cncb-transitive-testing-customer-
bff
```

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/tr
ansitive-testing/customer-frontend --path cncb-transitive-testing-
customer-frontend
```

2. Install the dependencies in each project with `npm install`.
3. Run the end-to-end tests in `replay` mode for each project, as follows:

```
$ cd ./author-frontend
$ DEBUG=replay npm run test:int

$ cd ../author-bff
$ npm test
$ DEBUG=replay npm run test:int

$ cd ../customer-bff
$ npm test
$ DEBUG=replay npm run test:int

$ cd ../customer-frontend
$ DEBUG=replay npm run test:int
```

4. Review all of the following fixture files:

```
./author-frontend/fixtures/0.0.0.0-3001/save-thing0
./author-frontend/fixtures/0.0.0.0-3001/get-thing0

./author-bff/fixtures/author-frontend/save-thing0
./author-bff/fixtures/dynamodb.us-east-1.amazonaws.com-443/save-thing0

./author-bff/fixtures/author-frontend/get-thing0
./author-bff/fixtures/dynamodb.us-east-1.amazonaws.com-443/get-thing0

./author-bff/fixtures/downstream-customer-bff/thing0-INSERT.json
./author-bff/fixtures/kinesis.us-east-1.amazonaws.com-443/thing0-
created

./customer-bff/fixtures/upstream-author-bff/thing0-created
./customer-bff/fixtures/dynamodb.us-east-1.amazonaws.com-443/save-
thing0

./customer-bff/fixtures/customer-frontend/get-thing0
./customer-bff/fixtures/dynamodb.us-east-1.amazonaws.com-443/get-thing0

./customer-frontend/fixtures/0.0.0.0-3001/get-thing0
```

5. Review all of the following end-to-end test case files:

```
./author-frontend/test/int/e2e.test.js

./author-bff/test/int/author-frontend/e2e.test.js
./author-bff/test/int/downstream-customer-bff/e2e.test.js

./customer-bff/test/int/upstream-author-bff/e2e.test.js
./customer-bff/test/int/customer-frontend/e2e.test.js

./customer-frontend/test/int/e2e.test.js
```

How it works...

Integration testing and contract testing are two sides of the same coin. We have already seen how these tests can be implemented with test doubles that replay previously recorded request and response pairs. This allows each service to be tested in isolation without the need to deploy any other service. A service provider team creates integration tests to ensure their service works as they expect, and service consumer teams create contract tests to ensure that the provider's service works as they expect. We then build on this so that the test engineers from all teams work together to define sufficient end-to-end test scenarios to ensure that all of the services are working together. For each scenario, we string together a series of integration and contract tests across all projects, where the recordings from one project are used to drive the tests in the next project, and so forth. Borrowing from the transitive property of equality, if Service *A* produces Payload 1, which works with Service *B* to produce Payload 2, which works with Service *C* to produce the expected Payload 3, then we can assert that when Service *A* produces Payload 1 then ultimately Service *C* will produce Payload 3.

In this recipe, we have an authoring application and a customer application. Each consists of a frontend project and a BFF project. The `author-frontend` project made the `save-thing0` recording. This recording was copied to the `author-bff` project and ultimately resulted in the `thing0-created` recording. The `thing0-created` recording was then copied to the `customer-bff` project and ultimately resulted in the `get-thing0` recording. The `get-thing0` recording was then copied to the `customer-frontend` project to support its testing.

The end result is a set of completely autonomous test suites. The test suite of a specific service is asserted each time the service is modified, without the need to rerun a test suite in each project. Only a backwards-incompatible change requires dependent projects to update their test cases and recordings; so, we no longer need to maintain an end-to-end testing environment. Database scripts are no longer needed to reset databases to a known state, as the data is embodied in test cases and recordings. These transitive end-to-end tests form the tip of a comprehensive testing pyramid that increases our confidence that we have minimized the chance of human error in our continuous deployment pipeline.

Leveraging feature flags

The practice of decoupling deployment from release is predicated on the use of feature flags. We are continuously deploying small batches of change to mitigate the risks of each deployment. These changes are deployed all the way to production, so we need a feature flag mechanism to disable these capabilities until we are ready to release them and make them generally available. We also need the ability to enable these capabilities for a subset of users, such as beta users and internal testers. It is also preferable to leverage the natural feature flags of a system, such as permissions and preferences, to minimize the technical debt that results from littering code with custom feature flags. This recipe will show you how to leverage the claims in a JWT token to enable and disable features.

Getting ready

Before starting this recipe, you will need an AWS Cognito user pool, such as the one created in the *Creating a federated identity pool* recipe. The user pool should have the following two groups defined: `Author` and `BetaUser`.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch6/fe
ature-flag --path cncb-feature-flag
```

2. Navigate to the `cncb-feature-flag` directory with `cd cncb-feature-flag`.

3. Review the file named `src/Authorize.js`, as follows:

```
...

const getGroups = props => get(props,
  'auth.cognito.signInUserSession.idToken.payload.cognito:groups', '');

const check = (allowedRoles, props) => {
  const groups = getGroups(props);
  return intersection(groups, allowedRoles).length > 0;
};

const HasRole = allowedRoles =>
  props => check(allowedRoles, props) ?
    props.children :
    null;

export const HasAuthorRole = HasRole(['Author']);
export const HasBetaUserRole = HasRole(['BetaUser']);
```

4. Review the file named `src/Home.js`, as follows:

```
const Home = ({ auth }) => (
  <div>
    ...
    <HasAuthorRole>
      This is the Author Feature!
    </HasAuthorRole>
    <HasBetaUserRole>
      This is a New Beta Feature...
    </HasBetaUserRole>
    ...
  </div>
);
```

5. Review the file named `src/App.js` and update the `clientId` and `domain` fields with the values for the user pool stack.
6. Install the dependencies with `npm install`.
7. Run the app locally with `npm start`.
8. Click the `Sign Up` link and follow the instructions.
9. Click the `Sign Out` link and then `Sign Up` another user.
10. Assign each user to a different group in the AWS user pool console—one to the `Author` group and the other to the `BetaUser` group.
11. `Sign In` as each user and notice how the screen renders differently for each one.

How it works...

First and foremost, zero-downtime deployment has to be treated as a first-class requirement and must be accounted for in the task roadmap and system design. If an enhancement is just adding a new optional field on an existing domain entity then a feature flag isn't required at all. If a field is just being removed, the order of deployment is important, from most dependent to least dependent. If an entirely new feature is being added, then access to the whole feature can be restricted. The most interesting scenarios are when a significant change is being made to an existing and popular feature. If the change is significant then it may be best to support two versions of the feature simultaneously, such as two versions of a page—in which case the scenario is essentially the same as the entirely new feature scenario. If a new version is not warranted, then care should be taken not to tie the code in knots.

In this simplified ReactJS example, the JWT token is accessible after sign-in through the `auth` property. Instances of the `HasRole` component are configured with `allowedRoles`. The component checks if the JWT token has a matching group in the `cognito:groups` field. If a match is found then the `children` components are rendered; otherwise, `null` is returned and nothing is rendered.

7
Optimizing Observability

In this chapter, the following recipes will be covered:

- Monitoring a cloud-native system
- Implementing custom metrics
- Monitoring domain events
- Creating alerts
- Creating continuous synthetic transaction tests

Introduction

Confidence is crucial to maximize the potential of our lean and autonomous cloud-native services, because a crisis of confidence will stifle progress. Leveraging fully managed cloud services and following cloud-native design patterns to create autonomous services significantly increases team confidence. Decoupling deployment from release and shifting testing to the left, to create a streamlined continuous deployment pipeline, further increases team confidence. Yet, this is not enough. We need to shift testing to the right as well, all the way into production, so that we can monitor and alert the team about the status of the system. This gives teams confidence that they will have timely information so that they can minimize the mean time to recovery when errors do happen. The recipes in this chapter demonstrate how to optimize the observability of cloud-native services, alert about what matters, and continuously test in production to increase team confidence.

Monitoring a cloud-native system

Leveraging fully managed cloud services is key to creating lean, cloud-native services, because embracing this disposable architecture empowers self-sufficient, full-stack teams to rapidly deliver with confidence based on the foundation provided by those cloud services. Team confidence is further increased because this foundation comes with good observability. This recipe demonstrates how to tap into cloud-provider metrics using a cloud-provider-agnostic, third-party monitoring service.

Getting ready

Before starting this recipe you will need a Datadog account (`https://www.datadoghq.com`).

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch7/da
tadog-account --path cncb-datadog-account
```

2. Navigate to the `cncb-datadog-account` directory with `cd cncb-datadog-account`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-datadog-account
...
resources:
  Resources:
    DatadogAWSIntegrationPolicy:
      Type: AWS::IAM::ManagedPolicy
      Properties:
        PolicyDocument: ${file(includes.yml):PolicyDocument}
    DatadogAWSIntegrationRole:
      Type: AWS::IAM::Role
      Properties:
        RoleName: DatadogAWSIntegrationRole
        AssumeRolePolicyDocument:
          Statement:
            Effect: Allow
            Principal:
              AWS: arn:aws:iam::464622532012:root
```

```
            Action: sts:AssumeRole
            Condition:
              StringEquals:
                sts:ExternalId: <copy value from datadog aws
integration dialog>
        ManagedPolicyArns:
          - Ref: DatadogAWSIntegrationPolicy
```

4. Install the dependencies with `npm install`.
5. Run the tests with `npm test`.
6. Review the contents generated in the `.serverless` directory.
7. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
8. Log in to **Datadog** and go to the **Integrations** page and select the AWS integration tile.
9. Select **Role Delegation**, enter your AWS account ID and set the AWS role name to `DatadogAWSIntegrationRole`.
10. Copy the AWS external ID value and use it to update `sts:ExternalId` in `serverless.yml`.
11. Set tags to `account:cncb` and press **Install Integration.**
12. Update the stack with `npm run dp:lcl -- -s $MY_STAGE`.
13. Review the stack and resources in the AWS Console.
14. Invoke the sample function multiple times with the following command:

```
$ sls invoke -f hello -r us-east-1 -s $MY_STAGE
```

15. Review the preset Datadog Lambda dashboard with **Dashboards | Dashboard List | All Integrations | AWS Lambda.**
16. Remove the stack once you are finished with this chapter with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Cloud providers collect valuable metrics for their cloud services. However, they do not necessarily retain this data for extended periods, and the ability to slice and dice this data can be limited. Therefore, it is recommended to employ a third-party monitoring service to fill in the gaps and provide more comprehensive monitoring capabilities. Furthermore, in the eventuality of utilizing a polyglot cloud, a cloud-provider-agnostic monitoring service offers a unified monitoring experience. My monitoring service of choice is Datadog. This recipe shows how easily and quickly we can connect Datadog to an AWS account and start aggregating metrics to increase the observability of our cloud-native systems.

To allow Datadog to start collecting metrics from an AWS account, we must grant it permission to do so. As the *How to do it* section shows, this requires steps on the AWS side and the Datadog side. First, we deploy a stack to create `DatadogAWSIntegrationPolicy` with all the necessary permissions, and `DatadogAWSIntegrationRole` connects the AWS account with Datadog's AWS account. This last bit is important. Datadog runs in AWS as well. This means that we can use *Role Delegation* to connect the accounts instead of sharing access keys. Once `DatadogAWSIntegrationRole` is created, we can configure the AWS integration on the Datadog side, which has a prerequisite for the existence of the role. Datadog generates `ExternalId`, which we need to add to `DatadogAWSIntegrationRole` as a condition for assuming the role. With the integration in place, Datadog consumes the requested metrics from CloudWatch in your AWS account, so that they can be aggregated into meaningful dashboards, retained for historical analysis, and monitored to alert about conditions of interest.

Implementing custom metrics

The metrics provided by value-added cloud services, such as *Function-as-a-service*, are a great starting point. Teams can put their cloud-native services into production with just these metrics with a reasonable level of confidence. However, more observability is almost always better. We need fine-grained details about the inner workings of our functions. This recipe demonstrates how to collect additional metrics, such as cold starts, memory, CPU utilization, and the latency of HTTP resources.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch7/cu
stom-metrics --path cncb-custom-metrics
```

2. Navigate to the `cncb-custom-metrics` directory with `cd cncb-custom-metrics`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-custom-metrics
...
functions:
  hello:
    ...
```

```
    environment:
      ACCOUNT_NAME: ${opt:account}
      SERVERLESS_STAGE: ${opt:stage}
      SERVERLESS_PROJECT: ${self:service}
      MONITOR_ADVANCED: false
      DEBUG: '*'
```

4. Review the file named `handler.js` with the following content:

```
const { monitor, count } = require('serverless-datadog-metrics');
const debug = require('debug')('handler');

module.exports.hello = monitor((request, context, callback) => {
  debug('request: %j', request);
  count('hello.count', 1);
  const response = { ... };
  callback(null, response);
});
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-custom-metrics@1.0.0 dp:lcl <path-to-your-workspace>/cncb-
custom-metrics
> sls deploy -r us-east-1 --account cncb "-s" "john"
...
endpoints:
  GET -
https://h865txqjqj.execute-api.us-east-1.amazonaws.com/john/hello
```

9. Review the stack and resources in the AWS Console.
10. Invoke the endpoint shown in the stack output in the following commands:

```
$ curl
https://<APP-ID>.execute-api.us-east-1.amazonaws.com/$MY_STAGE/hello |
json_pp
{
   "message" : "JavaScript Cloud Native Development Cookbook! ..."
}
```

11. Take a look at the logs:

```
$ sls logs -f hello -r us-east-1 -s $MY_STAGE

MONITORING|1530339912|1|count|aws.lambda.coldstart.count|#account:cncb,
...
MONITORING|1530339912|0.259|gauge|node.process.uptime|#account:cncb,...
MONITORING|1530339912|1|count|hello.count|#account:cncb,...
MONITORING|1530339912|0|check|aws.lambda.check|#account:cncb,...
MONITORING|1530339912|0.498...|gauge|node.mem.heap.utilization|#account
:cncb,...
MONITORING|1530339912|1.740238|histogram|aws.lambda.handler|#account:cn
cb,...
```

12. Execute the service several more times and then review the Lambda dashboard in Datadog under **Dashboards** | **Dashboard List** | **All Integrations** | **AWS Lambda**.
13. Explore the custom metrics in Datadog under **Metrics** | **Explore with Graph**: `hello.count` and `aws.lambda.handler.avg`.
14. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Adding custom metrics to a function works differently than traditional monitoring. The traditional approach involves adding an agent to each machine that collects metrics and periodically sends the data to the monitoring system. But with *Function-as-a-service,* there is no machine for us to deploy an agent on. An alternative is simply to send the collected metrics at the end of each function invocation. However, this adds significant latency to each function invocation. Datadog offers a unique alternative based on structured log statements. Counts, gauges, histograms, and checks are simply logged as they are collected and Datadog automatically consumes these statements from CloudWatch Logs.

The `serverless-datadog-metrics` library (`https://www.npmjs.com/package/serverless-datadog-metrics`) facilitates using this approach. We simply wrap the handler function with the `monitor` function and it will collect useful metrics, such as cold starts, errors, execution time, memory, and CPU utilization as well as the latency of HTTP resources. The HTTP metrics are very valuable. All HTTP calls to resources, such as DynamoDB, S3, and Kinesis, are automatically recorded so that we can see how much time a function spends waiting on its external resources.

This library also exports low-level functions, such as `count`, `gauge`, and `histogram`, to support additional custom metrics. The environment valuables, such as `ACCOUNT_NAME` and `SERVERLESS_PROJECT`, are used as tags for filtering metrics in dashboards and alerts.

Monitoring domain events

In traditional systems, we typically focus on observing the behavior of synchronous requests. However, our cloud-native systems are highly asynchronous and event-driven. Therefore, we need to place equal or greater attention on the flow of domain events through the system so that we can determine when these flows deviate from the norm. This recipe demonstrates how to collect domain event metrics.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch7/ev
ent-metrics --path cncb-event-metrics
```

2. Navigate to the `cncb-event-metrics` directory with `cd cncb-event-metrics`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .tap(count)
    ...
    .collect().toCallback(cb);
};

const count = (uow) => {
  const tags = [
```

```
    `account:${process.env.ACCOUNT_NAME}`,
    `region:${uow.record.awsRegion}`,
    `stream:${uow.record.eventSourceARN.split('/')[1]}`,
    `shard:${uow.record.eventID.split('-')[1].split(':')[0]}`,
    `source:${uow.event.tags && uow.event.tags.source || 'not-
specified'}`,
    `type:${uow.event.type}`,
  ];

console.log(`MONITORING|${uow.event.timestamp}|1|count|domain.event|#${
tags.join()}`);
};

...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack: `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE

[
    {
        "total": 4500,
        "purple": 1151,
        "green": 1132,
        "blue": 1069,
        "orange": 1148
    }
]
```

11. Take a look at the logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
...
...
MONITORING|1531020609200|1|count|domain.event|#account:cncb,...,type:pu
rple
```

12. Explore the event metrics in Datadog under **Metrics | Explore with Graph**: `domain.event` and one graph per `type`.
13. Remove the stack once you are finished: `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Monitoring events works similarly to collecting events in the data lake. A single stream processor observes all events from all streams and simply counts the domain events by event `type`, along with additional tags, such as `region`, `stream`, and `source`. Again, these counts are recorded as structured log statements and Datadog consumes these statements from CloudWatch Logs. Graphing the domain event metrics in a dashboard can provide great insight into the behavior of a system. We will see how to alert about the flow of domain events in the *Creating alerts* recipe. We also perform special handling for `fault` events. For these events, we invoke the Datadog Event API, which provides for sending additional contextual information, such as a stack trace. We will discuss `fault` events in `Chapter 8`, *Designing for Failure*.

Creating alerts

To maximize our confidence in our cloud-native services, we need to be alerted about issues ahead of the end users so that we can respond quickly and minimize the mean time to recovery. This also means that we need to eliminate alert fatigue and only alert on what really matters, otherwise important alerts will be lost in the noise. This recipe demonstrates how to create alerts on key metrics.

How to do it...

1. Log in to your Datadog account.
2. Create an IAM alert with the following settings:
    - Select **Monitors** | **New Monitor** | **Event**
    - Match events from `Amazon Cloudtrail` over `aws-service:iam`
    - **Multi Alert**—`aws_account`
    - **Alert threshold**—`above 0`
    - **Title**—`aws.iam`
    - **Include trigger tags**—`true`
    - **Notify**—`yourself`

3. Create an iterator age alert with the following settings:
    - Select **Monitors** | **New Monitor** | **Integration** | **AWS Lambda**
    - **Metric**—`aws.lambda.iterator_age`
    - **Max by**—`functionname`, `region` and `account`
    - **Multi Alert**—`functionname`, `region` and `account`
    - **Alert threshold**—`7200000` (2 hrs)
    - **Warning threshold**—`1800000` (0.5 hrs)
    - **Title**—`aws.lambda.iterator_age`
    - **Include trigger tags**—`true`
    - **Notify**—`yourself`
4. Create a request rate alert with the following settings:
    - Select **Monitors** | **New Monitor** | **Anomaly**
    - **Metric**—`aws.apigateway.count`
    - **Average by**—`apiname`, `region`, and `account` as `rate`
    - **Multi Alert**—`apiname`, `region`, and `account`
    - **Alert conditions**—start with the defaults and tune to your data
    - **Title**—`aws.apigateway.rate`
    - **Include trigger tags**—`true`
    - **Notify**—`yourself`
5. Create a domain event rate alert with the following settings:
    - Select **Monitors** | **New Monitor** | **Anomaly**
    - **Metric**—`domain.event`
    - **Average by**—`type`, `region`, and `account` as `rate`
    - **Multi Alert**—`type`, `region`, and `account`
    - **Alert conditions**—Start with the defaults and tune to your data
    - **Title**—`domain.event.rate`
    - **Include trigger tags**—`true`
    - **Notify**—`yourself`
6. Create a Fault Event alert with the following settings:
    - Select **Monitors** | **New Monitor** | **Event**
    - **Match events containing**—`Fault Event` with status `error` from: `My Apps`
    - **Multi Alert**—`functionname`, `region`, and `account`
    - **Alert threshold**—`above 0`

- **Title**—`fault.event`
- **Include trigger tags**—`true`
- **Notify**—`yourself`



Datadog autocomplete menus are populated based on the metrics that have been recently collected.

How it works...

Now that our system is observable, we need to do something proactive and useful with all this data. There is simply too much data to process manually and our confidence will only be increased if we can turn this data into valuable, timely information. We will certainly use this data for root-cause and post-mortem analysis, but our confidence is increased by our focus on mean time to recovery. Therefore, we need to create monitors that are constantly *testing* the data, turning it into information and alerting on what matters. However, we must be careful to avoid alert fatigue. The best practice is to alert liberally, but page judiciously on symptoms rather than causes. For example, we should create many monitors that only record that a threshold was crossed so that this additional information can be used in the root-cause analysis. Other monitors will email the team to warn of a potential problem, but a few select monitors will page the team to jump into immediate action.

To know the difference between a symptom and a cause, we categorize our metrics into work metrics and resource metrics. Work metrics represent the user-visible output of the system. Resource metrics represent the internal workings of the system. Our resource monitors will usually record and send warnings, and our work monitors will page the team. The RED method (`https://dzone.com/articles/red-method-for-prometheus-3-key-metrics-for-micros`) and the USE method (`https://dzone.com/articles/red-method-for-prometheus-3-key-metrics-for-micros`) break these categories down further. **RED** stands for **Rate, Error, and Duration**. When a critical service has a significant decrease in the rate of requests or events or a significant increase in errors, or latency significantly increases, then these may warrant paging the team. **USE** stands for **Utilization**, **Saturation**, and **Errors**. When the utilization of a resource, such as DynamoDB or Kinesis, reaches a certain level then it is probably good to warn the team. However, saturation and/or errors, such as throttling, may just warrant recording, because they may quickly subside or, if prolonged, they will trigger the work monitors.

This recipe demonstrated a few possible monitors. The `fault` monitor represents work that is failing and must be addressed. The stream `iterator age` monitor straddles the line, because it could represent temporary resource saturation, or it could represent an error that is causing work to back up. Therefore, it has both a warning and an alert at different thresholds. The `anomaly detection` monitors should focus on work metrics, such as the rate of critical requests or domain events. It is also a good idea to monitor CloudTrail for any IAM changes, such as to roles and permissions.

The `notification` step is optional if you only need to record the condition. To warn the team, send the notification to chat and/or group email. To page the team, send the notification to an SNS topic. It is best to use the **Multi Alert** feature, triggered on pertinent tags, and include these in the notification title so that this information is available at a glance.

Ultimately, to be valuable and to avoid fatigue, these monitors need care and feeding. These monitors are your tests in production. As your team's understanding of your system increases, then you will uncover better tests/monitors. When your monitors produce false positives, they will need to be tuned or eliminated. Your level of confidence is the true measure of successful monitoring.

Creating synthetic transaction tests

If a tree falls in a forest and no one is around to hear it, does it make a sound? Or more on topic, if a deployment is broken in a region and no one is awake to use it, does it make a difference? Of course, the answer is yes. We want to be alerted about the broken deployment so that it can be fixed before normal traffic begins. To enable this, we need to continuously pump synthetic traffic through the system so that there is a continuous signal to test. This recipe demonstrates how to generate synthetic traffic using cloud-provider-agnostic, third-party services.

Getting ready

Before starting this recipe, you will need a Pingdom account (`https://www.pingdom.com`). You will also need an AWS Cognito user pool, such as the one created in the *Creating a federated identity pool* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch7/sy
nthetics --path cncb-synthetics
```

2. Navigate to the `cncb-synthetics` directory `cd cncb-synthetics`.
3. Install the dependencies with `npm install`.
4. Review the file named `src/App.js` and update the `clientId` and `domain` fields with the values for the user pool stack.
5. Build the app with `npm run build`.
6. Deploy the stack:



Deploying a CloudFront distribution can take upward of 20 minutes.

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-synthetics@1.0.0 dp:lcl <path-to-your-workspace>/cncb-synthetics
> sls deploy -r us-east-1 --account cncb "-s" "john"
...
WebsiteDistributionURL: https://dqvo8ga8z7ao3.cloudfront.net
```

7. Update the `callbackURLs` and `logoutURLs` of the user pool stack to include `WebsiteDistributionURL` and then redeploy it.
8. Browse to `WebsiteDistributionURL` provided in the stack output to test the site configuration.
9. Log in to your Pingdom account.
10. Create an uptime check with the following settings:
    - Select **Experience Monitoring** | **Uptime** | **Add check**.
    - Name—`cncb-synthetics`.
    - Check interval—`1 minute`
    - URL—`WebsiteDistributionURL from your deploy output`

11. Review the **Experience Monitoring** | **Uptime** dashboard periodically.
12. Create a real user monitor with the following settings:
    - Select **Experience Monitoring** | **Visitor Insights (RUM)** | **Add site**.
    - Name—`cncb-synthetics`
    - URL—`WebsiteDistributionURL from your deploy output`
13. Review the file named `public/index.html`, uncomment the following code, and replace the ID with the value from the generated code snippet:

```
<!--
<script src="//rum-static.pingdom.net/ID.js" async></script>
-->
```

14. Build and redeploy the app:

```
$ npm run build
$ npm run dp:lcl -- -s $MY_STAGE
```

15. Review the **Experience Monitoring** | **Visitor Insights (RUM)** dashboard periodically.
16. Create a Synthetic Transaction test with the following settings:
    - Select **Experience Monitoring** | **Transactions** | **Add check**
    - Name—`cncb-synthetics`
    - Test interval—`10 minutes`
    - Go to the URL `WebsiteDistributionURL from deploy output`
    - Fill in the `username` field with `your-username`
    - Fill in the `password` field with `your-password`
    - Click **Sign In**
    - Wait for the element `.App-title` to contain `Welcome to React`
17. Review the **Experience Monitoring** | **Transactions** dashboard periodically.
18. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.
19. Cancel your Pingdom free trial, under **Account** | **Subscriptions** | **Manage Plan**, before the 14 days expire, to avoid costs.

How it works...

Testing in production is different than traditional preproduction testing. As demonstrated in the *Creating alerts* recipe, our production tests are implemented as monitors that are constantly testing the signals emitted by the system. However, if there is no traffic, then there is no signal, and so no tests to alert us to problems in any deployments that happen during these periods. This, in turn, decreases our confidence in these deployments. The solution is to generate steady synthetic traffic to fill in the gaps when there is no natural traffic.

Uptime checks are the simplest to put in place because they only make a single request. These should be included at a minimum, because they can be put in place quickly with little effort and because they have the highest frequency. **Real User Monitoring** (**RUM**) should be included because it only requires a simple code modification and because a significant amount of the user performance experience in cloud-native systems is executed in the browser by single-page applications. Finally, a small but strategic set of synthetic transaction scripts needs to be implemented to smoke test crucial features continuously. These scripts resemble traditional test scripts, but their focus is on continuously exercising these critical happy paths to ensure that the crucial features are unaffected by the continuous deployment of new features.

8
Designing for Failure

In this chapter, the following recipes will be covered:

- Employing proper timeouts and retries
- Implementing backpressure and rate limiting
- Handling faults
- Resubmitting fault events
- Implementing idempotence with an inverse OpLock
- Implementing idempotence with Event Sourcing

Introduction

Managing failure is the cornerstone of cloud-native. We build autonomous services that limit the blast radius when they do fail and continue to operate when other services fail. We decouple deployment from release, and we control the batch size of each deployment so that we can easily identify the problem when a deployment does go wrong. We shift testing to the left into the continuous deployment pipeline to catch issues before a deployment, as well as all the way to the right into production, where we continuously test the system and alert on anomalies to minimize the meantime to recovery. The recipes in this chapter demonstrate how to design a service to be resilient and forgiving in the face of failure so that transient failures are properly handled, their impact is minimized, and the service can self-heal.

Employing proper timeouts and retries

The reality of computer networks is that they are unreliable. The reality of cloud computing is that it relies on computer networks, therefore it is imperative that we implement services to properly handle network anomalies. This recipe demonstrates how to properly configure functions and SDK calls with the appropriate timeouts and retries.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/ti
meout-retry --path cncb-timeout-retry
```

2. Navigate to the `cncb-timeout-retry` directory with `cd cncb-timeout-retry`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-timeout-retry
...
functions:
  command:
    handler: handler.command
    timeout: 6
    memorySize: 1024
...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.command = (request, context, callback) => {
  const db = new aws.DynamoDB.DocumentClient({
    httpOptions: { timeout: 1000 },
    logger: console,
  });
...
  db.put(params, callback);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the function with the following command:

```
$ sls invoke -f command -r us-east-1 -s $MY_STAGE -d '{"name":"thing
one"}'
```

11. Take a look at the following command function logs:

```
$ sls logs -f command -r us-east-1 -s $MY_STAGE

2018-07-14 23:41:14.229 (-04:00) ... [AWS dynamodb 200 0.063s 0
retries] putItem({ TableName: 'john-cncb-timeout-retry-things',
  Item:
   { id: { S: 'c22f3ce3-551a-4999-b750-a20e33c3d53b' },
     name: { S: 'thing one' } } })
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Network hiccups can happen at any time. Where one request may not go through, the next request may go through just fine; therefore, we should have short timeouts so that we can retry the process as soon as possible, but not so short that a good request times out before it has a chance to complete normally. We must also ensure that our timeout and retry cycle has enough time to complete before the function times out. The `aws-sdk` is configured by default to time out after two minutes and performs three retries with an increasing delay time. Of course, two minutes is too long. Setting the `timeout` to 1000 (1 second) will typically be long enough for a request to complete and allow for three retries to complete before a function timeout of 6 seconds.

If requests time out too frequently then this could be an indication that the function has been allocated with too few resources. For example, there is a correlation between the `memorySize` allocated to a function and the machine instance size that is used. Smaller machine instances also have less network I/O capacity, which could lead to more frequent network hiccups. Thus, increasing `memorySize` will decrease network volatility.

Implementing backpressure and rate limiting

The various services in a cloud-native system must be able to handle the ebb and flow of traffic through the system. Upstream services should never overload downstream services, and downstream services must be able to handle peak loads without falling behind or overloading services further downstream. This recipe shows how to leverage the natural backpressure of stream processors and implement additional rate limiting to manage throttling.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/ba
ckpressure-ratelimit --path cncb-backpressure-ratelimit
```

2. Navigate to the `cncb-backpressure-ratelimit` directory with `cd cncb-backpressure-ratelimit`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-backpressure-ratelimit
...
functions:
  listener:
    handler: handler.listener
    timeout: 240 # headroom for retries
    events:
      - stream:
          batchSize: 1000 # / (timeout / 2) < write capacity
          ...
    environment:
      WRITE_CAPACITY_UNITS: 10
      SHARD_COUNT: 1
...
resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        ...
        ProvisionedThroughput:
          ...
          WriteCapacityUnits:
${self:functions.listener.environment.WRITE_CAPACITY_UNITS}
```

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forPurple)
    .ratelimit(Number(process.env.WRITE_CAPACITY) /
      Number(process.env.SHARD_COUNT) / 10, 100)
    .flatMap(put)
    .collect()
    .toCallback(cb);
};

const put = uow => {
  const params = { ... };
  const db = new aws.DynamoDB.DocumentClient({
    httpOptions: { timeout: 1000 },
    // default values:
    // maxRetries: 10,
    // retryDelayOptions: {
    //   base: 50,
    // },
    logger: console,
  });

  return _(db.put(params).promise()
    .then(() => uow)
  );
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
[
    {
        "total": 4775,
        "blue": 1221,
        "green": 1190,
        "purple": 1202,
```

```
        "orange": 1162
    }
]
```

11. Take a look at the following `trigger` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'event count'
2018-07-15 22:53:29 ... event count: 1000
2018-07-15 22:54:00 ... event count: 1000
2018-07-15 22:54:33 ... event count: 1000
2018-07-15 22:55:05 ... event count: 425
2018-07-15 22:55:19 ... event count: 1000
2018-07-15 22:55:51 ... event count: 350

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'Duration'
REPORT ... Duration: 31011.59 ms ...
REPORT ... Duration: 33038.58 ms ...
REPORT ... Duration: 32399.91 ms ...
REPORT ... Duration: 13999.56 ms ...
REPORT ... Duration: 31670.86 ms ...
REPORT ... Duration: 12856.77 ms ...

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'retries'
...
2018-07-15 22:55:49 ... [AWS dynamodb 200 0.026s 0 retries] putItem({
TableName: '...',
  Item:
   { id: { S: '686dc03a-88a3-11e8-829c-67d049599dd2' },
     type: { S: 'purple' },
     timestamp: { N: '1531709604787' },
     partitionKey: { S: '3' },
     tags: { M: { region: { S: 'us-east-1' } } } },
  ReturnConsumedCapacity: 'TOTAL' })
2018-07-15 22:55:50 ... [AWS dynamodb 200 0.013s 0 retries] putItem({
TableName: '...',
  Item:
   { id: { S: '686d4b14-88a3-11e8-829c-67d049599dd2' },
     type: { S: 'purple' },
     timestamp: { N: '1531709604784' },
     partitionKey: { S: '4' },
     tags: { M: { region: { S: 'us-east-1' } } } },
  ReturnConsumedCapacity: 'TOTAL' })
...
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Backpressure is a critical characteristic of a well-implemented stream processor. When using the *imperative programming paradigm*, such as looping over the array of records in a batch, the downstream target system could easily be overwhelmed because the loop will process the records as fast as it can without regard for the throughput capacity of the target system. Alternatively, the **Functional Reactive Programming** (**FRP**) paradigm, with a library such as Highland.js (`https://highlandjs.org`) or RxJS (`https://github.com/ReactiveX/rxjs`), provides a natural backpressure, because data is pulled downstream only as fast as the downstream steps are able to complete their tasks. For example, an external system that has low throughput will only pull data as quickly as its capacity will allow.

On the other hand, highly-scalable systems, such as DynamoDB or Kinesis, that are able to process data with extremely high throughput rely on throttling to restrict capacity. In this case, the natural backpressure of FRP is not enough; additional use of the `ratelimit` feature is required. As an example, when I was writing this recipe I ran a simulation without rate limiting and then went out to dinner. When I came back several hours later, the events generated by the simulation were still trying to process. This is because DynamoDB was throttling the requests and the exponential backoff and retry built into the `aws-sdk` was taking up too much time, leading the function to `timeout` and retry the whole batch again. This demonstrates that while retries, as discussed in the *Employing proper timeouts and retries* recipe, are important for synchronous requests, they cannot be solely relied upon for asynchronous stream processing. Instead, we need to proactively limit the rate of flow to avoid throttling and exponential backoff to help ensure a batch completes within the function timeout.

In this recipe, we use a simple algorithm to calculate the rate of flow—`WRITE_CAPACITY / SHARD_COUNT / 10` per every `100` milliseconds. This ensures that DynamoDB will not receive more requests per second than have been allocated. I also used a simple algorithm to determine the batch size—`batchSize / (timeout / 2) < WRITE_CAPACITY`. This ensures that there should be plenty of time to complete the batch under normal conditions, but there will be twice the necessary time available in case there is throttling. Note that this is just a logical starting point; this is the area where performance tuning in cloud-native systems should be focused. The characteristics of your data and target systems will dictate the most effective settings. As we will see in the *Autoscaling DynamoDB* recipe, autoscaling adds yet another dimension to backpressure, rate limiting, and performance tuning. Regardless, you can start with these simple and safe algorithms and tune them over time.

Handling faults

Stream processors are naturally resilient and forgiving of transient errors because they employ backpressure and automatically retry failing batches. However, hard errors, if not handled properly, can cause a traffic jam that results in dropped messages. This recipe will show you how to delegate these errors as fault events so that good messages can continue to flow.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/ha
ndling-faults --path cncb-handling-faults
```

2. Navigate to the `cncb-handling-faults` directory with `cd cncb-handling-faults`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forThingCreated)
    .tap(validate)
    .tap(randomError)
    .flatMap(save)
    .errors(errors)
    .collect().toCallback(cb);
};

const validate = uow => {
  if (uow.event.thing.name === undefined) {
    const err = new Error('Validation Error: name is required');
    // handled
    err.uow = uow;
    throw err;
  }
};

const randomError = () => {
  if (Math.floor((Math.random() * 5) + 1) === 3) {
    // unhandled
```

```
    throw new Error('Random Error');
  }
};

const save = uow => {
 ...
 return _(db.put(uow.params).promise()
   .catch(err => {
     // handled
     err.uow = uow;
     throw err;
   }));
};

const errors = (err, push) => {
  if (err.uow) {
    // handled exceptions are adorned with the uow in error
    push(null, publish({
      type: 'fault',
      timestamp: Date.now(),
      tags: {
        functionName: process.env.AWS_LAMBDA_FUNCTION_NAME,
      },
      err: {
        name: err.name,
        message: err.message,
        stack: err.stack,
      },
      uow: err.uow,
    }));
  } else {
    // rethrow unhandled errors to stop processing
    push(err);
  }
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
```

11. Take a look at the following `trigger` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'publishing
fault'
... {"type":"fault","timestamp":...,"tags":{"functionName":"cncb-
handling-faults-john-
listener"},"err":{"name":"Error","message":"Validation Error: name is
required" ...

... {"type":"fault","timestamp":...,"tags":{"functionName":"cncb-
handling-faults-john-
listener"},"err":{"name":"ValidationException","message":"...Missing
the key id..." ...
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

This recipe implements the *Stream Circuit Breaker* pattern that I discuss in depth in my book, *Cloud Native Development Patterns and Best Practices* (`https://www.packtpub.com/application-development/cloud-native-development-patterns-and-best-practices`). Stream processors that experience hard errors will continue to reprocess those events until they expire from the stream—unless the stream processor is implemented to set these errors aside by delegating them as fault events for processing elsewhere, such as described in the *Resubmitting fault events* recipe. This alleviates the traffic jam so that other events can continue to flow.

This recipe simulates events that will fail at different stages in the stream processor. Some events simulate upstream bugs that will fail the validation logic that asserts that events are being created properly upstream. Other events will fail when they are inserted into DynamoDB. The logic also randomly fails some events to simulate transient errors that do not produce faults and will automatically be retried. In the logs, you will see two fault events published. When a random error is generated, you will see in the logs that the function retries the batch. If the simulation does not raise a random error then you should rerun it until it does.

To isolate events in a stream, we need to introduce the concept of a **unit of work** (**UOW**) that groups one or more events from the batch into an atomic unit that must succeed or fail together. The UOW contains the original Kinesis record (`uow.record`), the event parsed from the record (`uow.event`), and any intermediate processing results (`uow.params`) that are attached to the UOW as it passes through the stream processor. The UOW is also used to identify errors as handled or unhandled. When an expected error is identified by the validation logic or caught from external calls, the UOW is adorned to the error and the error is re-thrown. The adorned unit of work (`error.uow`) acts as the indicator to the `errors` handler that the error was handled in the stream processor and that it should `publish` the error as a fault event. Unexpected errors, such as randomly-generated errors, are not handled by the stream processor logic and thus will not have an adorned UOW. The error handler will `push` these errors downstream to cause the function to fail and retry. In the *Creating alerts* recipe, we discuss monitoring for fault events, as well as iterator age, so that the team can receive timely notifications about stream processor errors.

Resubmitting fault events

We have designed our stream processors to delegate errors as fault events so that valid events can continue to flow. We monitor for and alert on fault events so that appropriate action can be taken to address the root cause. Once the problem is resolved, it may be necessary to reprocess the failed events. This recipe demonstrates how to resubmit fault events back to the stream processor that raised the fault.

How to do it...

1. Create the `monitor`, `simulator`, and `cli` projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/re
submitting-faults/monitor --path cncb-resubmitting-faults-monitor

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/re
submitting-faults/cli --path cncb-resubmitting-faults-cli

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/re
submitting-faults/simulator --path cncb-resubmitting-faults-simulator
```

2. Review the file named `serverless.yml` in the `cncb-resubmitting-faults-monitor` and `cncb-resubmitting-faults-simulator` directories.
3. Deploy the `monitor` and `simulator` stacks, as follows:

```
$ cd ../cncb-resubmitting-faults-monitor
$ npm install
$ npm test -- -s $MY_STAGE
$ npm run dp:lcl -- -s $MY_STAGE

  Stack Outputs
  BucketName: cncb-resubmitting-faults-monitor-john-
bucket-1llq835xdczd8

$ cd ../cncb-resubmitting-faults-simulator
$ npm install
$ npm test -- -s $MY_STAGE
$ npm run dp:lcl -- -s $MY_STAGE
```

4. Review the stacks and resources in the AWS Console.
5. Run the simulator from the `cncb-resubmitting-faults-simulator` directory, as follows:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
```

6. Confirm that a fault file is written to the bucket specified in the monitor stack output `cncb-resubmitting-faults-monitor-*-bucket-*` and note the path. If no fault was generated, run the simulator again until one is generated.
7. Review the file named `./cli/lib/resubmit.js` with the following content:

```
exports.command = 'resubmit [bucket] [prefix]'
exports.desc = 'Resubmit the faults in [bucket] for [prefix]'
...
const invoke = (lambda, options, event) => {
  const Payload = JSON.stringify({
      Records: [event.uow.record],
  });

  const params = {
    FunctionName: event.tags.functionName,
    ...
    Payload: Buffer.from(Payload),
  };

  return _(lambda.invoke(params).promise());
}
```

8. Resubmit the fault with the following command:

```
$ cd ../cncb-resubmitting-faults-cli
$ npm install
$ node index.js resubmit -b cncb-resubmitting-faults-monitor-$MY_STAGE-
bucket-<suffix> -p <s3-path> --dry false
```

9. Empty the `cncb-resubmitting-faults-monitor-*` bucket manually and remove the `monitor` and `simulator` stacks once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In the *Handling faults* recipe, we saw how stream processors delegate hard errors by publishing fault events with all the data pertaining to the unit of work that failed. In this recipe, a fault monitor consumes these fault events and stores them in an S3 bucket. This enables the team to review the specific fault to help determine the root cause of the problem. The fault contains the specific exception that was caught, the event that failed, and all of the contextual information that was attached to the unit of work.

Once the root cause has been addressed, the original event can be submitted back to the stream processor that published the fault. This is possible because the fault event contains the original Kinesis record (`event.uow.record`) and the name of the function to invoke (`event.tags.functionName`). The command line utility reads all the fault events from the bucket for the specified path and invokes the specific functions. From the perspective of the function logic, this direct invocation of the function is no different to it being invoked directly from the Kinesis stream. However, the stream processor must be designed to be idempotent and to be able to handle events out of order, as we will discuss in the *Implementing idempotence with an inverse OpLock* and *Implementing idempotence with Event Sourcing* recipes.

Implementing idempotence with an inverse OpLock

From a business rule standpoint, it is important that events are processed exactly once; otherwise, problems may arise, such as double counting or not counting at all. However, our cloud-native systems must be resilient to failure and proactively retry to ensure no messages are dropped. Unfortunately, this means that messages may be delivered multiple times, for example when a producer re-publishes an event or a stream processor retries a batch that may have been partially processed. The solution to this problem is to implement all actions to be idempotent. This recipe implements idempotency with what I refer to as an inverse OpLock.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/id
empotence-inverse-oplock --path cncb-idempotence-inverse-oplock
```

2. Navigate to the `cncb-idempotence-inverse-oplock` directory with `cd cncb-idempotence-inverse-oplock`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forThingSaved)
    .flatMap(save)
    .collect().toCallback(cb);
};

const save = uow => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Item: {
      ...uow.event.thing,
      oplock: uow.event.timestamp,
    },
    ConditionExpression: 'attribute_not_exists(#oplock) OR #oplock <
:timestamp',
```

```
    ...
  };

  const db = new aws.DynamoDB.DocumentClient({
    httpOptions: { timeout: 1000 },
    logger: console,
  });

  return _(db.put(params).promise()
    .catch(handleConditionalCheckFailedException)
    .then(() => uow)
  );
}

const handleConditionalCheckFailedException = (err) => {
  if (err.code !== 'ConditionalCheckFailedException') {
    err.uow = uow;
    throw err;
  }
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the simulate function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
```

11. Take a look at the following `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE

... [AWS dynamodb 200 0.098s 0 retries] putItem({ TableName: '...',
  Item:
   { id: { S: '3022eaeb-45b7-46d5-b0a1-696c0eb9aa25' },
     oplock: { N: '1531628180237' } },
...
... [AWS dynamodb 400 0.026s 0 retries] putItem({ TableName: '...',
  Item:
   { id: { S: '3022eaeb-45b7-46d5-b0a1-696c0eb9aa25' },
     oplock: { N: '1531628180237' } },
...
... { ConditionalCheckFailedException: The conditionalrequest failed
    at Request.extractError ...
```

```
...
  message: 'The conditional request failed',
  code: 'ConditionalCheckFailedException',
  time: 2018-07-15T04:16:21.202Z,
  requestId: '36BB14IGTPGM8DE8CFQJS0ME3VVV4KQNSO5AEMVJF66Q9ASUAAJG',
  statusCode: 400,
  retryable: false,
  retryDelay: 20.791698133966396 }

... [AWS dynamodb 200 0.025s 0 retries] putItem({ TableName: '...',
  Item:
   { id: { S: '3022eaeb-45b7-46d5-b0a1-696c0eb9aa25' },
     oplock: { N: '1531628181237' } },
...
... [AWS dynamodb 400 0.038s 0 retries] putItem({ TableName: '...',
  Item:
   { id: { S: '3022eaeb-45b7-46d5-b0a1-696c0eb9aa25' },
     oplock: { N: '1531628180237' } },
...
... { ConditionalCheckFailedException: The conditionalrequest failed
    at Request.extractError ...
...
  message: 'The conditional request failed',
...
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Traditional optimistic locking prevents multiple users from updating the same record at the same time. A record is only updated if the `oplock` field has not changed since the user retrieved the data. If the data has changed then an exception is thrown and the user is forced to retrieve the data again before proceeding with the update. This forces the updates to be performed sequentially, and it requires human interaction to resolve any potential conflicts.

The **inverse OpLock** is designed to provide idempotency for asynchronous processing. Instead of forcing the transaction to retry, we simply do the opposite—we drop the older or duplicate event. A traditional OpLock may be used in the upstream Backend For Frontend service to sequence user transactions, where, downstream services implement an inverse OpLock to ensure that older or duplicate events do not overwrite the most recent data. In this recipe, we use the `uow.event.timestamp` as the `oplock` value. In some scenarios, it may be preferential to use the sequence number if multiple events happen at the exact same millisecond. `ConditionalCheckFailedException` is caught and ignored. All other exceptions are re-thrown with the unit of work attached to cause a fault event to be published, as discussed in the *Handling faults* recipe.

The simulation in this recipe publishes a `thing-created` event, publishes it again, and then publishes a `thing-updated` event followed by the `thing-created` event a third time. The logs show that the `thing-created` event is only processed once and the duplicates are ignored.

Implementing idempotence with Event Sourcing

From a business rule standpoint, it is important that events are processed exactly once; otherwise, problems may arise, such as double counting or not counting at all. However, our cloud-native systems must be resilient to failure and proactively retry to ensure no messages are dropped. Unfortunately, this means that messages may be delivered multiple times, such as when a producer re-publishes an event or a stream processor retries a batch that may have been partially processed. The solution to this problem is to implement all actions to be idempotent. This recipe demonstrates how to use Event Sourcing and a micro event store to implement idempotence.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch8/id
empotence-es --path cncb-idempotence-es
```

2. Navigate to the `cncb-idempotence-es` directory with `cd cncb-idempotence-es`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forThingSaved)
    .flatMap(saveEvent)
    .collect().toCallback(cb);
};

const saveEvent = uow => {
  const params = {
    TableName: process.env.EVENTS_TABLE_NAME,
    Item: {
      id: uow.event.thing.id,
      sequence: uow.event.id,
      event: uow.event,
    }
  };

  const db = new aws.DynamoDB.DocumentClient({
    httpOptions: { timeout: 1000 },
    logger: console,
  });

  return _(db.put(params).promise()
    .then(() => uow)
  );
}
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack with `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the simulate function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
```

11. Take a look at the following `trigger` function logs:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE

... record: { ...
"Keys":{"sequence":{"S":"3fdb8c10-87ea-11e8-9cf5-0b6c5b83bdcb"},"id":{"
S":"8c083ef9-d180-48b8-a773-db0f61815f38"}}, ...

... record: { ...
"Keys":{"sequence":{"S":"3fdb8c11-87ea-11e8-9cf5-0b6c5b83bdcb"},"id":{"
S":"8c083ef9-d180-48b8-a773-db0f61815f38"}}, ...
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`

How it works...

Event Sourcing facilitates idempotence because events are immutable. The same event with the same unique ID can be published or processed multiple times with the same outcome. The micro event store serves as a buffer that weeds out duplicates. The service consumes desired events and stores them in a micro event store with a **hashkey** that groups related events, such as the `uow.event.thing.id` of the domain object, and a range key based on the `uow.event.id`. This primary key is also immutable. As a result, the same event can be saved multiple times, but only a single event is produced on the database stream. Thus, the business logic, which is discussed in the *Creating a micro event store* or *Implementing an analytics BFF* recipe, is only triggered once.

The simulation in this recipe publishes a `thing-created` event, publishes it again, and then publishes a `thing-updated` event followed by the `thing-created` event a third time. The logs show that the three `thing-created` event instances only result in a single event on the DynamoDB Stream.

9
Optimizing Performance

In this chapter, the following recipes will be covered:

- Tuning Function as a Service
- Batching requests
- Leveraging asynchronous non-blocking IO
- Grouping events in stream processors
- Autoscaling DynamoDB
- Utilizing cache-control
- Leveraging session consistency

Introduction

Cloud-native turns performance testing, tuning, and optimization on their heads. Many of the fully-managed, serverless cloud services that are leveraged have implicit scalability. These services are purchased per request and will automatically scale to meet peak and unexpected demands. For these resources, it is much less necessary to perform upfront performance testing; instead, we optimize for observability, as discussed in `Chapter 7`, *Optimizing Observability*, and continuously tune based on the information gathered from continuous testing in production. We also leverage continuous deployment to push necessary improvements. This worth-based development approach helps ensure that we are focusing our efforts on the highest value improvements.

Still, there are resources, such as some stream processors and data stores, that rely heavily on explicitly defined batch sizes and read/write capacities. For crucial services, these resources must be sufficiently allocated to ensure peak data processing volumes do not overwhelm them. The recipes in this chapter will therefore focus on performance optimization techniques that are worth applying upfront in the design and development process to help ensure services are not working against themselves.

Tuning Function as a Service

Tuning functions is very different from traditional service tuning because there are so few explicit knobs to turn. There are also many implications of the short life cycle of a function that turns traditional techniques and frameworks into anti-patterns. The following recipe explains a common memory mistake, discusses the cold start implications of traditional language and library choices, and will show you how to package a JavaScript function with **webpack** to minimize download time.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/tu
ning-faas --path cncb-tuning-faas
```

2. Navigate to the `cncb-tuning-faas` directory, `cd cncb-tuning-faas`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-tuning-faas

plugins:
  - serverless-webpack

provider:
  memorySize: 1024 # default
  ...

package:
  individually: true
custom:
 webpack:
   includeModules: true

functions:
  save:
    memorySize: 512 # function specific
    ...
```

4. Review the file named `webpack.config.js` with the following content:

```
const slsw = require('serverless-webpack');
const nodeExternals = require("webpack-node-externals");
const path = require('path');

module.exports = {
  entry: slsw.lib.entries,
  output: {
    libraryTarget: 'commonjs',
    path: path.join(__dirname, '.webpack'),
    filename: '[name].js'
  },
  optimization: {
    minimize: false
  },
  target: 'node',
  mode: slsw.lib.webpack.isLocal ? "development" : "production",
  externals: [
    nodeExternals()
  ],
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            loader: 'babel-loader',
          }
        ],
        include: __dirname,
        exclude: /node_modules/
      }
    ]
  }
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory, note the ZIP file per function, and then unzip each to see the contents.
8. Deploy the stack, `npm run dp:lcl -- -s $MY_STAGE`.

9. Review the stack and resources in the AWS Console and note the code size in the Lambda console.
10. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The most obvious knob we have for tuning **Function as a Service** (**FaaS**) is `memorySize` allocation. This setting drives the price calculation as well. Unfortunately, the correlation with price tends to be counter-intuitive and can result in decisions that actually increase costs, while also reducing performance. The way AWS Lambda pricing works is if you double the memory allocation but consequently cut the execution time in half, the price is the same. The corollary is that if you cut memory allocation in half and consequently double the execution time, you are spending the same amount of money for less performance. It works this way because memory allocation actually correlates to the machine instance size that a function is executed on. More memory allocation means that the function will also have a faster CPU and more network IO throughput. In short, *do not skimp on memory allocation*.

A major concern and source of confusion with Function as a Service is cold start times. For asynchronous functions, such as processing a Kinesis Stream, cold start times are not as concerning. This is because cold start frequency is lower as functions tend to be reused for several hours for each shard. However, minimizing cold start times for synchronous functions behind an API Gateway is very important, because multiple functions are started to accommodate concurrent load. As a result, cold start times could impact the end user experience.

The first thing that impacts cold start time is the size of the function package that must be downloaded to the container. This is one reason that allocating more memory and hence more network IO throughput improves the performance of a function. It is also important to minimize the size of the package that must be downloaded. We will discuss how to use webpack to optimize JavaScript functions shortly.

The next thing that impacts cold start times is the choice of language or runtime. Scripting languages, such as JavaScript and Python, do very little at startup and thus have very little impact on cold start times. Conversely, Java must do work at startup to load classes and prepare the JVM. As the number of classes increases, the impact on cold starts also increases. This leads to the next impact on cold start times: the choice of libraries and frameworks, such as **object relation mapping** (**ORM**) and dependency injection frameworks, and connection pooling libraries. These tend to do a lot of work at startup because they were designed to work in long running servers.

A common issue among FaaS developers using Java is the improvement of cold start times for functions written with Spring and Hibernate; however these tools were not designed for FaaS in the first place. I have programmed in Java for over 20 years, from when it first appeared in the 1990s. I was skeptical about changing to JavaScript at first, but this cookbook is testament to its fit with cloud-native and serverless architecture. It is worth noting, however, that polyglot programming is the best policy; use the right programming language for a specific service, but understand the implications of it when using it with Faas.

To minimize a JavaScript function's package size, we leverage webpack for the same reasons we use it to minimize downloads to browsers. Webpack performs tree shaking, which removes unused code to reduce package size. In the `serverless.yml` file, we include the `serverless-webpack` plugin and configure it to package functions individually. Packaging functions individually allows us to maximize the benefits of tree shaking. The `webpack.config.js` file further controls the packaging process. The `serverless-webpack` plugin provides the `slsw.lib.entries` utility so that we do not need to duplicate the function names to define all the `entry` points. We also turn off the `minimize` feature, which uglifies the code. We do this to avoid including source maps for debugging, which significantly increases the package size. We also exclude all of the external libraries in the `node_modules` folder and configure the plugin to `includeModules`, which includes those that are actually used as runtime. One special exception is the `aws-sdk` module, which is never included because it is already available in the function container. The end result is a lean function package that contains only what is necessary.

Batching requests

The design of a stream processor must account for the volume of data it will receive. The data should be processed in real time and the processor should not fall behind. The following recipe demonstrates how to use DynamoDB batch writes to help ensure sufficient throughput.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/fr
p-batching --path cncb-frp-batching
```

2. Navigate to the `cncb-frp-batching` directory, `cd cncb-frp-batching`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forPurple)
    .ratelimit(Number(process.env.WRITE_CAPACITY) /
      Number(process.env.SHARD_COUNT) /
      Number(process.env.WRITE_BATCH_SIZE) / 10, 100)
    .batch(Number(process.env.WRITE_BATCH_SIZE))
    .map(batchUow)
    .flatMap(batchWrite)
    .collect().toCallback(cb);
};

const batchUow = batch => ({ batch });

const batchWrite = batchUow => {
  batchUow.params = {
    RequestItems: {
      [process.env.TABLE_NAME]: batchUow.batch.map(uow =>
        ({
          PutRequest: {
            Item: uow.event
          }
        })
      )
    },
  };

  ...

  return _(db.batchWrite(batchUow.params).promise()
    .then(data => (
      Object.keys(data.UnprocessedItems).length > 0 ?
        Promise.reject(data) :
```

```
          batchUow
      ))
      .catch(err => {
        err.uow = batchUow;
        throw err;
      })
    );
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack, `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
[
    {
        "total": 3850,
        "orange": 942,
        "purple": 952,
        "blue": 1008,
        "green": 948
    }
]
```

11. Take a look at the following `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'event count'
2018-08-04 23:46:53 ... event count: 100
2018-08-04 23:46:57 ... event count: 1000
2018-08-04 23:47:23 ... event count: 250
2018-08-04 23:47:30 ... event count: 1000
2018-08-04 23:47:54 ... event count: 1000
2018-08-04 23:48:18 ... event count: 500

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'Duration'
REPORT ... Duration: 3688.65 ms ...
REPORT ... Duration: 25869.08 ms ...
REPORT ... Duration: 7293.39 ms ...
REPORT ... Duration: 23662.65 ms ...
REPORT ... Duration: 24752.11 ms ...
REPORT ... Duration: 13983.72 ms ...

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'retries'
```

```
...
2018-08-04 23:48:20 ... [AWS dynamodb 200 0.031s 0 retries]
batchWriteItem({ RequestItems:
   { 'john-cncb-frp-batching-things':
      [ { PutRequest:
          { Item:
             { id: { S: '320e6023-9862-11e8-b0f6-01e9feb460f5' },
               type: { S: 'purple' },
               timestamp: { N: '1533440814882' },
               partitionKey: { S: '5' },
               tags: { M: { region: { S: 'us-east-1' } } } } } },
        { PutRequest:
          { Item:
             { id: { S: '320e6025-9862-11e8-b0f6-01e9feb460f5' },
               type: { S: 'purple' },
               timestamp: { N: '1533440814882' },
               partitionKey: { S: '1' },
               tags: { M: { region: { S: 'us-east-1' } } } } } },
        ...
        [length]: 10 ] },
  ReturnConsumedCapacity: 'TOTAL',
  ReturnItemCollectionMetrics: 'SIZE' })
...
```

12. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

If a stream processor receives a batch of 1,000 events, will it execute faster if it has to make 1,000 requests to the database or just 100 requests? The answer of course depends on many variables, but in general, making fewer calls over the network is better because it minimizes the impact of network latency. To this end, services such as DynamoDB and Elasticsearch provide APIs that allow batches of commands to be submitted in a single request. In this recipe, we use DynamoDB's `batchWrite` operation. To prepare a batch, we simply add a `batch` step to the pipeline and specify the `WRITE_BATCH_SIZE`. This performance improvement is very simple to add, but it is important to keep in mind that batching requests increase the rate at which DynamoDB's write capacity is consumed. Therefore, it is necessary to include the `WRITE_BATCH_SIZE` in the `ratelimit` calculation and increase the write capacity accordingly, as discussed in the *Implementing backpressure and rate limiting* recipe.

Another important thing to note is that these batch requests are not treated as a single transaction. Some commands may succeed and others may fail in a single request; it is therefore necessary to inspect the response for `UnprocessedItems` that needs to be resubmitted. In this recipe, we treat each batch as a **unit of work** (**uow**) and raise a fault for the entire batch, as discussed in the *Handling faults* recipe. This is a good, safe place to start before tuning the logic to retry only the commands that fail. Note that, ultimately, you would only raise a fault when the maximum number of retries has been attempted.

Leveraging asynchronous non-blocking IO

The design of a stream processor must account for the volume of data it will receive. The data should be processed in real-time and the processor should not fall behind. The following recipe demonstrates how to leverage asynchronous, non-blocking IO to process data in parallel to help ensure sufficient throughput.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/frp-async-non-blocking-io --path cncb-frp-async-non-blocking-io
```

2. Navigate to the `cncb-frp-async-non-blocking-io` directory, `cd cncb-frp-async-non-blocking-io`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forPurple)
    .ratelimit(Number(process.env.WRITE_CAPACITY) /
      Number(process.env.SHARD_COUNT) /
      Number(process.env.PARALLEL) / 10, 100)
    .map(put)
    .parallel(Number(process.env.PARALLEL))
    .collect().toCallback(cb);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
[
    {
        "total": 4675,
        "blue": 1136,
        "green": 1201,
        "purple": 1167,
        "orange": 1171
    }
]
```

11. Take a look at the following `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'event count'
2018-08-05 00:03:05 ... event count: 1675
2018-08-05 00:03:46 ... event count: 1751
2018-08-05 00:04:34 ... event count: 1249

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'Duration'
REPORT ... Duration: 41104.28 ms ...
REPORT ... Duration: 48312.47 ms ...
REPORT ... Duration: 31450.13 ms ...

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'retries'
...
2018-08-05 00:04:58.034 ... [AWS dynamodb 200 0.024s 0 retries] ...
2018-08-05 00:04:58.136 ... [AWS dynamodb 200 0.022s 0 retries] ...
2018-08-05 00:04:58.254 ... [AWS dynamodb 200 0.034s 0 retries] ...
2018-08-05 00:04:58.329 ... [AWS dynamodb 200 0.007s 0 retries] ...
2018-08-05 00:04:58.430 ... [AWS dynamodb 200 0.007s 0 retries] ...
2018-08-05 00:04:58.540 ... [AWS dynamodb 200 0.015s 0 retries] ...
2018-08-05 00:04:58.661 ... [AWS dynamodb 200 0.035s 0 retries] ...
2018-08-05 00:04:58.744 ... [AWS dynamodb 200 0.016s 0 retries] ...
2018-08-05 00:04:58.843 ... [AWS dynamodb 200 0.014s 0 retries] ...
2018-08-05 00:04:58.953 ... [AWS dynamodb 200 0.023s 0 retries] ...
...
```

12. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

Asynchronous non-blocking IO is extremely valuable for maximizing throughput. Without it, a stream processor will block and do nothing until an external call has completed. This recipe demonstrates how to use a `parallel` step to control the number of concurrent calls that can execute. As an example of the impact this can have, I once had a script that read from S3 and would take well over an hour to process, but once I added a `parallel` step with a setting of 16, the script executed in just five minutes. The improvement was so significant that Datadog contacted me, almost immediately, to see if we had a runaway process.

To allow concurrent calls, we simply add a `parallel` step to the pipeline after an external call step and specify the `PARALLEL` amount. This performance improvement is very simple to add, but it is important to keep in mind that parallel requests increase the rate at which DynamoDB's write capacity is consumed. It is therefore necessary to include the `PARALLEL` amount in the `ratelimit` calculation and increase the write capacity accordingly, as discussed in the *Implementing backpressure and rate limiting* recipe. Further performance improvements may be achieved by combining parallel execution with grouping and batching.

Grouping events in stream processors

The design of a stream processor must account for the volume of data it will receive. The data should be processed in real-time and the processor should not fall behind. The following recipe demonstrates how grouping related data in a stream can help ensure sufficient throughput.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/frp-grouping --path cncb-frp-grouping
```

2. Navigate to the `cncb-frp-grouping` directory, `cd cncb-frp-grouping`.
3. Review the file named `serverless.yml`.

4. Review the file named `handler.js` with the following content:

```
module.exports.listener = (event, context, cb) => {
  _(event.Records)
    .map(recordToUow)
    .filter(forPurple)
    .group(uow => uow.event.partitionKey)
    .flatMap(groupUow)
    .ratelimit(Number(process.env.WRITE_CAPACITY) /
      Number(process.env.SHARD_COUNT) / 10, 100)
    .flatMap(put)
    .collect().toCallback(cb);
};

const groupUow = groups => _(Object.keys(groups).map(key => ({ batch:
groups[key]})));

const put = groupUow => {
  const params = {
    TableName: process.env.TABLE_NAME,
    Item: groupUow.batch[groupUow.batch.length - 1].event, // last
  };
  ...
  return _(db.put(params).promise()
    .then(() => groupUow)
  );
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function with the following command:

```
$ sls invoke -f simulate -r us-east-1 -s $MY_STAGE
[
    {
        "total": 4500,
        "blue": 1134,
        "green": 1114,
        "purple": 1144,
        "orange": 1108
    }
]
```

11. Take a look at the following `listener` function logs:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'event count'
2018-08-05 00:28:19 ... event count: 1000
2018-08-05 00:28:20 ... event count: 1000
2018-08-05 00:28:21 ... event count: 650
2018-08-05 00:28:22 ... event count: 1000
2018-08-05 00:28:23 ... event count: 850

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'Duration'
REPORT ... Duration: 759.50 ms ...
REPORT ... Duration: 611.70 ms ...
REPORT ... Duration: 629.91 ms ...
REPORT ... Duration: 612.90 ms ...
REPORT ... Duration: 623.11 ms ...

$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'retries'
2018-08-05 00:28:20.197 ... [AWS dynamodb 200 0.112s 0 retries] ...
2018-08-05 00:28:20.320 ... [AWS dynamodb 200 0.018s 0 retries] ...
...
2018-08-05 00:28:23.537 ... [AWS dynamodb 200 0.008s 0 retries] ...
2018-08-05 00:28:23.657 ... [AWS dynamodb 200 0.019s 0 retries] ...
```

12. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The *Batching requests* recipe demonstrates how to minimize the overhead of network IO by batching multiple unrelated commands into a single request. Another way to minimize network IO is to simply reduce the number of commands that need to be executed by grouping related events, and only executing a single command per grouping. For example, we might perform a calculation per group or just sample some of the data. In this recipe, we grouped events by the `partitionKey`. We can group events by any data in the events, but the best results are achieved when the grouping is relative to the partition key; this is because the partition key ensures that related events are sent to the same shard.

The `group` step makes it straightforward to reduce related events into groups based on the content of the events. For more complicated logic, a `reduce` step can be used directly. Next, we map each group to a unit of work (`groupUow`) that must succeed or fail together, as discussed in the *Handling faults* recipe. Finally, as shown in the preceding example, we write the `last` event of each group. Note from the logs that grouping results in significantly fewer writes; for this specific run, there were 4,500 events simulated and only 25 writes. Further performance improvements may be achieved by combining grouping with batching and parallel invocations.

Autoscaling DynamoDB

Cloud-native, with FaaS and serverless, minimize the amount of effort that is needed to scale the infrastructure that supports the service layer. However, we now need to focus on tuning the stream processors and minimize any throttling of the target data store. The following recipe demonstrates how to use DynamoDB autoscaling to help ensure that enough capacity is allocated to provide sufficient throughput and avoid throttling.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/dy
namodb-autoscaling --path cncb-dynamodb-autoscaling
```

2. Navigate to the `cncb-dynamodb-autoscaling` directory, `cd cncb-dynamodb-autoscaling`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-dynamodb-autoscaling

...

plugins:
  - serverless-dynamodb-autoscaling-plugin

custom:
  autoscaling:
    - table: Table
      write:
        minimum: 5
```

```
          maximum: 50
          usage: 0.6
          actions:
            - name: morning
              minimum: 5
              maximum: 50
              schedule: cron(0 6 * * ? *)
            - name: night
              minimum: 1
              maximum: 1
              schedule: cron(0 0 * * ? *)
        read:
          ...

resources:
 Resources:
   Table:
     Type: AWS::DynamoDB::Table
     ...
```

4. Review the file named `handler.js`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack `npm run dp:lcl -- -s $MY_STAGE`.
9. Review the stack and resources in the AWS Console.
10. Invoke the `simulate` function multiple times with the following command to trigger autoscaling, `sls invoke -f simulate -r us-east-1 -s $MY_STAGE`.
11. Take a look at the following `listener` function logs with the following commands:

```
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'event count'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter 'Duration'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"1 retries"'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"2 retries"'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"3 retries"'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"4 retries"'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"5 retries"'
$ sls logs -f listener -r us-east-1 -s $MY_STAGE --filter '"6 retries"'
```

12. Watch the **Write capacity** and **Throttled write requests** metrics on the DynamoDB **Metrics** tab in the AWS Console to see the autoscaling increment meet the demand, and then scale down at `night` and back up again in the `morning`.
13. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In the *Implementing backpressure and rate limiting* recipe, we see how it is important for stream processors to minimize throttling to maximize throughput. In this chapter, we have discussed techniques to optimize throughput, such as batching, grouping and asynchronous non-blocking requests, which all increase the data store capacity that must be allocated. However, while we do need to ensure that we have sufficient capacity, we also want to minimize wasted capacity, and autoscaling helps us achieve that. Autoscaling can address demand that grows over time to an expected peak, predictable demand, such as known events, and unpredictable demand.

In this recipe, we use the `serverless-dynamodb-autoscaling-plugin` to define the `autoscaling` policies on a per table basis. For both the `read` and `write` capacity, we specify the `minimum` and `maximum` capacity and the desired `usage` percentage. This `usage` percentage defines the amount of headroom we would like to have so that we can increase capacity early enough to help ensure that additional capacity is allocated before we reach 100 percent utilization and begin to throttle. We can also schedule autoscaling `actions` at specific times. In this recipe, we scale down at `night` to minimize waste and then scale back up in the `morning` before typical demand arrives.

Utilizing cache-control

Autonomous, cloud-native services maintain their own materialized views and store this replicated data in highly-available and extremely performant cloud-native databases. When combined with the performance of an API Gateway and FaaS, it is typically unnecessary to add a traditional caching mechanism to achieve the desired performance for a user-facing, **backend-for-frontend** (**BFF**) service. That being said, this doesn't mean we shouldn't take advantage of the CDN, such as CloudFront, that is already wrapping a service. The following recipe will therefore show you how to utilize cache-control headers and leverage a CDN to improve performance for end users, as well as reduce the load on a service.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/ca
che-control --path cncb-cache-control
```

2. Navigate to the `cncb-cache-control` directory, `cd cncb-cache-control`.
3. Review the file named `serverless.yml`.
4. Review the file named `handler.js` with the following content:

```
module.exports.get = (request, context, callback) => {
  const response = {
    statusCode: 200,
    headers: {
      'Cache-Control': 'max-age=5',
    },
    body: ...,
  };

  callback(null, response);
};

module.exports.save = (request, context, callback) => {
 const response = {
   statusCode: 200,
   headers: {
     'Cache-Control': 'no-cache, no-store, must-revalidate',
   },
 };

 callback(null, response);
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack, as follows:

```
$ npm run dp:lcl -- -s $MY_STAGE
...
Stack Outputs
ApiDistributionEndpoint: https://d2thj6o092tkou.cloudfront.net
...
```



Deploying a CloudFront distribution can often take upwards of 20 minutes.

9. Review the stack and resources in the AWS Console.
10. Invoke the endpoint shown in the stack output with the following `curl` command multiple times to see the difference in performance for the cached results:

```
$ curl -s -D - -w "Time: %{time_total}" -o /dev/null  https://<see
stack output>.cloudfront.net/things/123 | egrep -i 'X-Cache|Time'

X-Cache: Miss from cloudfront
Time: 0.712
$ curl ...
X-Cache: Hit from cloudfront
Time: 0.145

$ curl -v -X POST -H 'Content-Type: application/json' -d
'{"name":"thing 1"}' https://<see stack output>.cloudfront.net/things

...
< HTTP/1.1 200 OK
< Cache-Control: no-cache, no-store, must-revalidate
< X-Cache: Miss from cloudfront
...
```

11. Finally, remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

In this related recipe we focus on the service side of the equation. Cloud-native databases, such as DynamoDB, respond in the low 10s of milliseconds, and the overall latency across AWS API Gateway and AWS Lambda for a BFF service should typically execute in the low 100s of milliseconds. So long as the database capacity is set appropriately and throttling is minimized, it would be hard to make a noticeable improvement in this performance from the end user's perspective. The only way to really improve on this is to not have to make a request at all.

This is a case where cloud-native can really be counter-intuitive. Traditionally, to improve performance, we would need to increase the amount of infrastructure and add an expensive caching layer between the code and the database. In other words, we would need to spend a lot more money to improve performance. However, in this recipe, we are leveraging an extremely low-cost edge cache to both improve performance and lower cost. By adding `Cache-Control` headers, such as `max-age`, to our responses, we can tell a browser not to repeat a request and also tell the CDN to reuse a response for other users. As a result, we reduce load on the API Gateway and the function and reduce the necessary capacity for the database, which reduces the cost for all of these services. It is also good practice to explicitly control which actions should store `no-cache`, for example the `PUT`, `POST`, and `DELETE` methods.

Leveraging session consistency

The design of a cloud-native frontend application must account for the fact that the system should be eventually consistent. For example, in a traditional frontend application, it is not uncommon to save data and then immediately execute a query to retrieve that same data. However, in an eventually consistent system, it is very likely that the query would not find the data on the first try. Instead, cloud-native frontends leverage the fact that single page applications can—at minimum—cache data locally for the duration of the user's session. This approach is referred to as session consistency. The following recipe demonstrates how to use the popular Apollo Client (`https://www.apollographql.com/client`) with ReactJS to improve perceived performance and reduce load on the system by implementing **session consistency**.

How to do it...

1. Create the `service` and `spa` projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/se
ssion-consistency/spa --path cncb-session-consistency-spa

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch9/se
ssion-consistency/service --path cncb-session-consistency-service
```

2. Deploy the `service` with the following commands:

```
$ cd ./cncb-session-consistency-service
$ npm install
$ npm run dp:lcl -- -s $MY_STAGE
```

3. Navigate to the `cncb-session-consistency-spa` directory, `cd ../cncb-session-consistency-spa`.
4. Review the file named `src/index.js` with the following content and update the `uri` with the value output by the `service` stack, as follows:

```
...
const client = new ApolloClient({
    link: new HttpLink({
        // CHANGE ME
        uri: 'https://<API_ID>.execute-api.us-
east-1.amazonaws.com/<STAGE>/graphql',
    }),
    cache: new InMemoryCache(),
});
...
```

5. Review the file named `src/App.js` with the following content:

```
...
const AddThing = () => {
  ...
  return (
    <Mutation
      mutation={SAVE_THING}
      update={(cache, { data: { saveThing } }) => {
        const { things } = cache.readQuery({ query: GET_THINGS });
        cache.writeQuery({
          query: GET_THINGS,
          data: { things: { items: things.items.concat([saveThing]) } }
        });
      }}
    >
      ...
    </Mutation>
  );
};
...
```

6. Install the dependencies with `npm install`.
7. Run the app locally with `npm start`.
8. Browse to `http://localhost:3000`.
9. Add and update several things to notice how the query results update.
10. Finally, remove the service stack once you are finished, as follows:

```
$ cd ../cncb-session-consistency-service
$ npm run rm:lcl -- -s $MY_STAGE
```

How it works...

In this recipe, we use a GraphQL BFF that is similar to the one we created in the *Implementing a GraphQL CRUD BFF* recipe. The focus here is on the frontend application, which we create with ReactJS and the Apollo Client, and specifically on how to cache our interactions with the service. First, we create the `ApolloClient` in the `src/index.js` file and initialize it with the endpoint for the service and, most importantly, the `InMemoryCache` object.

Next, we implement the user interface in the `src/App.js` file. The screen displays a list of things that are returned from the `things` query. The Apollo Client will automatically cache the results of the query. The mutation that updates the individual objects will automatically update the cache and thus trigger the screen to re-render. Note that adding new data requires more effort. The `AddThing` function uses the mutation's `update` feature to keep the cache in sync and trigger a re-render. The `update` function receives a reference to the cache and the object that was returned from the mutation. We then call `readQuery` to read the query from the cache, append the new object to the query results, and finally update the cache by calling `writeQuery`.

The end result is a very low-latency user experience because we are optimizing the number of requests that are performed, the amount of data that is transferred, and the amount of memory that is used. Most importantly, for both new and updated data, we are not throwing away anything that was created on the client side and replacing it with the same, retrieved values—after all, it is just unnecessary work. We already have the data, so why should we throw it away and retrieve it again? We also cannot be certain that the data is consistent on the service side—unless we perform a consistent read that is slower, costs more and, as stated, is unnecessary. Session consistency becomes even more valuable for multi-regional deployments in the event of a regional failure. As we will discuss in `Chapter 10`, *Deploying to Multiple Regions*, eventually consistent, cloud-native systems are very tolerant of regional failure because they are already tolerant of eventual consistency, which can be more protracted during a failover. Therefore, session consistency helps make a regional failure transparent to the end user. For any data that must remain available during a regional failure, we can take session consistency a step further and persist the user session in local storage.

10
Deploying to Multiple Regions

In this chapter, the following recipes will be covered:

- Implementing latency-based routing
- Creating a regional health check
- Triggering regional failover
- Implementing regional replication with DynamoDB
- Implementing round-robin replication

Introduction

It is not a matter of if but when will a given cloud provider experience a news-worthy regional disruption. It is inevitable. In my experience, this happens approximately every two years or so. When such an event does occur, many systems have no recourse and become unavailable during the disruption because they are only designed to work across multiple availability zones within a single region. Meanwhile, other systems barely experience a blip in availability because they have been designed to run across multiple regions. The bottom line is that truly cloud-native systems capitalize on regional bulkheads and run in multiple regions. Fortunately, we leverage fully managed, value-added cloud services that already run across availability zones. This empowers teams to refocus that effort on creating an active-active, multi-regional system. The recipes in this chapter cover multi-regional topics from three interrelated perspectives—synchronous requests, database replication, and asynchronous event streams.

Implementing latency-based routing

Many systems make the conscious decision not to run across multiple regions because it is simply not worth the additional effort and cost. This is completely understandable when running in an active-passive mode because the additional effort does not produce an easily visible benefit until there is a regional disruption. It is also understandable when running in active-active mode doubles the monthly runtime cost. Conversely, serverless cloud-native systems are easily deployed to multiple regions and the increase in cost is nominal since the cost of a given transaction volume is spread across the regions. This recipe demonstrates how to run an AWS API Gateway and Lambda-based service in multiple regions and leverage Route53 to route traffic across these active-active regions to minimize latency.

Getting ready

You will need a registered domain name and a **Route53 Hosted Zone** that you can use in this recipe to create a subdomain for the service that will be deployed, such as we discussed in the *Associating a custom domain name with a CDN* recipe. You will also need a wildcard certificate for your domain name in the `us-east-1` and `us-west-2` regions, such as we discussed in the *Creating an SSL certificate for encryption in transit* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/l
atency-based-routing --path cncb-latency-based-routing
```

2. Navigate to the `cncb-latency-based-routing` directory with `cd cncb-latency-based-routing`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-latency-based-routing

plugins:
  - serverless-multi-regional-plugin

provider:
  ...
  endpointType: REGIONAL
```

```
custom:
  dns:
    hostedZoneId: ZXXXXXXXXXXXX
    domainName: ${self:service}.example.com
    regionalDomainName: ${opt:stage}-${self:custom.dns.domainName}
    us-east-1:
      acmCertificateArn: arn:aws:acm:us-
east-1:xxxxxxxxxxx:certificate/...
    us-west-2:
      acmCertificateArn: arn:aws:acm:us-
west-2:xxxxxxxxxxx:certificate/...
  cdn:
    region: us-east-1
    aliases:
      - ${self:custom.dns.domainName}
    acmCertificateArn: ${self:custom.dns.us-east-1.acmCertificateArn}

functions:
  hello:
    ...
```

4. Update the following fields in the `serverless.yml` file:
    - `custom.dns.hostedZoneId`
    - `custom.dns.domainName`
    - `custom.dns.us-east-1.acmCertificateArn`
    - `custom.dns.us-west-2.acmCertificateArn`
5. Review the file named `handler.js`.
6. Install the dependencies with `npm install`.
7. Run the tests with `npm test`.
8. Review the contents generated in the `.serverless` directory.
9. Deploy the stack in the `us-west-2` region with `npm run dp:lcl:w -- -s $MY_STAGE`.
10. Review the stack and resources in the AWS Console for the `us-west-2` region.
11. Test the `regional` endpoint of the service and note the region returned in the payload:

```
$ curl -v
https://$MY_STAGE-cncb-latency-based-routing.example.com/$MY_STAGE/hell
o

{"message":"Your function executed successfully in us-west-2!"}
```

12. Deploy the stack in the `us-east-1` region with `npm run dp:lcl:e -- -s $MY_STAGE`.



Deploying a CloudFront distribution can take 20 minutes or longer.

13. Review the stack and resources in the AWS console for the `us-east-1` region.
14. Test the `global` endpoint of the service and note the region returned in the payload, which should be different from above if you are not closest to the `us-west-2` region:

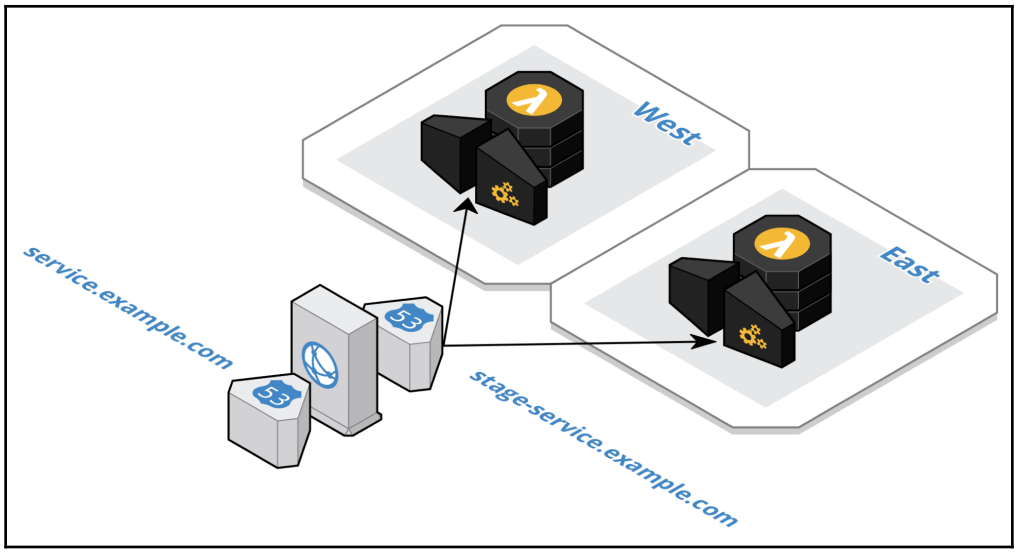
```
$ curl -v https://cncb-regional-failover-service.example.com/hello

{"message":"Your function executed successfully in us-east-1!"}
```

15. Remove the stacks once you have finished:

```
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAG
```

How it works...

In this recipe, we are taking a single service and deploying it to two regions—`us-east-1` and `us-west-2`. From the perspective of the API Gateway and the Lambda function, we are simply just creating two different CloudFormation stacks, one in each region. We have two scripts—`dp:lcl:e` and `dp:lcl:w`, and the only difference between the two is that they specify different regions. As a result, the effort to deploy to two regions is marginal, and there is no additional cost because we only pay per transaction. One thing of note in the `serverless.yml` file is that we are defining the `endpointType` for the API Gateway as `REGIONAL`, which will allow us to leverage the Route53 regional routing capabilities:



As shown in the preceding diagram, we need to configure Route53 to perform *latency*-based routing between the two regions. This means that Route53 will route requests to the region that is closest to the requester. The `serverless-multi-regional-plugin` encapsulates the majority of these configuration details, so we only need to specify the variables under `custom.dns`. First, we provide the `hostedZoneId` for the zone that hosts the top-level domain name, such as `example.com`. Next, we define the `domainName` that will be used as the `alias` to access the service globally via **CloudFront**. For this, we use the service name (that is, `${self:service}`) as a subdomain of the top-level domain to uniquely identify a service.

We also need to define a `regionalDomainName` to provide a common name across all the regions so that CloudFront can rely on Route53 to pick the best region to access. For this, we are using the stage (that is, `${opt:stage}-${self:custom.dns.domainName}`) as a prefix, and note that we are concatenating this with a dash so that it works with a simple wildcard certificate, such as `*.example.com`. The regional `acmCertificateArn` variables point to copies of your wildcard certificate in each region, as mentioned in the *Getting ready* section. API Gateway requires that the certificates live in the same region as the service. CloudFront requires that the certificate lives in the `us-east-1` region. CloudFront is a global service, so we only need to deploy the CloudFront distribution from the `us-east-1` region.

All requests to the global endpoint (`service.example.com`) will be routed to the closest CloudFront edge location. CloudFront then forwards the requests to the regional endpoint (`stage-service.example.com`), and Route53 will route the requests to the closest region. Once a request is in a region, all requests to services, such as Lambda, DynamoDB and Kinesis, will stay within the region to minimize latency. All changes to state will be replicated to the other regions, as we discuss in the *Implementing regional replication with DynamoDB* and *Implementing round-robin replication* recipes.



I highly recommend looking at the generated *CloudFormation* template to see the details of all the resources that are created.

Creating a regional health check

Health checks in a cloud-native system have a different focus from traditional health checks. Traditional health checks operate at the instance level to identify when a specific instance in a cluster needs to be replaced. Cloud-native systems, however, use fully managed, value-added cloud services, so there are no instances to manage. These serverless capabilities provide high availability across the availability zones within a specific region. As a result, cloud-native systems can focus on providing high availability across regions. This recipe demonstrates how to assert the health of the value-added cloud services that are used within a given region.

Getting ready

To complete this recipe in full, you will need a Pingdom (`https://www.pingdom.com`) account.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/r
egional-health-check --path cncb-regional-health-check
```

2. Navigate to the `cncb-regional-health-check` directory with `cd cncb-regional-health-check`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-regional-health-check

provider:
  name: aws
  runtime: nodejs8.10
  endpointType: REGIONAL
  iamRoleStatements:
    ...

functions:
  check:
    handler: handler.check
    events:
      - http:
          path: check
          method: get
    environment:
      UNHEALTHY: false
      TABLE_NAME:
        Ref: Table

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      ...
```

4. Review the file named `handler.js` with the following content:

```
module.exports.check = (request, context, callback) => {
  Promise.all([readCheck, writeCheck])
    .catch(handleError)
    .then(response(callback));
};

const db = new aws.DynamoDB.DocumentClient({
  httpOptions: { timeout: 1000 },
  logger: console,
});

const readCheck = () => db.get({
  TableName: process.env.TABLE_NAME,
  Key: {
```

```
    id: '1',
  },
}).promise();

const writeCheck = () => db.put({
  TableName: process.env.TABLE_NAME,
  Item: {
    id: '1',
  },
}).promise();

const handleError = (err) => {
  console.error(err);
  return true; // unhealthy
};

const response = callback => (unhealthy) => {
  callback(null, {
    statusCode: unhealthy || process.env.UNHEALTHY === 'true' ? 503 :
200,
    body: JSON.stringify({
      timestamp: Date.now(),
      region: process.env.AWS_REGION,
    }),
    headers: {
      'Cache-Control': 'no-cache, no-store, must-revalidate',
    },
  });
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack in the `us-east-1` and `us-west-2` regions:

```
$ npm run dp:lcl:e -- -s $MY_STAGE
...
  GET -
https://0987654321.execute-api.us-east-1.amazonaws.com/john/check

$ npm run dp:lcl:w -- -s $MY_STAGE
...
  GET -
https://1234567890.execute-api.us-west-2.amazonaws.com/john/check
```

9. Review the stack and resources in the AWS console for both regions.

10. For each region, invoke the endpoint shown in the stack output as shown here:

```
$ curl -v
https://0987654321.execute-api.us-east-1.amazonaws.com/$MY_STAGE/check
$ curl -v
https://1234567890.execute-api.us-west-2.amazonaws.com/$MY_STAGE/check
```

11. Create an **Uptime** check in your **Pingdom** account for each regional endpoint with an interval of `1 minute`.
12. Remove the stacks once you have finished:

```
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAGE
```

How it works...

A traditional health check typically asserts that an instance is able to access all the resources that it needs to operate properly. A regional health check does the same thing but from the perspective of the region as a whole. It asserts that all the value-added cloud services (that is, resources) used by the system are operating properly within the given region. If any one resource is unavailable, we will failover the entire region, as discussed in the *Triggering regional failover* recipe.

The health check service is implemented as a `REGIONAL` API Gateway based service and deployed to each region. We then need to periodically invoke the health check in each region to check that the region is healthy. We could have Route53 ping these regional endpoints, but it will ping them so frequently that the health check service could easily become the most expensive service in your entire system. Alternatively, we can use an external service, such as **Pingdom**, to invoke the health check in each region once per minute. Once a minute is sufficient for many systems, but extremely high traffic systems may benefit from the higher frequency provided by Route53.

The health check needs to assert that the required resources are available. The health check itself implicitly asserts that the API Gateway and Lambda services are available because it is built on those services. For all other resources, it will need to perform some sort of ping operation. In this recipe, we assume that DynamoDB is the required resource. The health check service defines its own DynamoDB table and performs a `readCheck` and a `writeCheck` on each invocation to assert that the service is still available. If either request fails, then the health check service will fail and return a `503` status code. For testing, the service provides an `UNHEALTHY` environment variable that can be used to simulate a failure, which we will use in the next recipe.

Triggering regional failover

As we discussed, in the *Creating a regional health check* recipe, our regional health checks assert that the fully managed, value-added cloud services that are used by the system are all up and running properly. When any of these services are down or experiencing a sufficiently high error rate, it is best to fail the entire region over to the next-best active region. This recipe demonstrates how to connect a regional health check to Route53, using **CloudWatch Alarms**, so that Route53 can direct traffic to healthy regions.

Getting ready

You will need a registered domain name and a **Route53 Hosted Zone** that you can use in this recipe to create a subdomain for the service that will be deployed, such as we discussed in the *Associating a custom domain name with a CDN* recipe. You will also need a wildcard certificate for your domain name in the `us-east-1` and `us-west-2` regions, such as we discussed in the *Creating an SSL certificate for encryption in transit* recipe.

How to do it...

1. Create the service and check projects from the following templates:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/r
egional-failover/check --path cncb-regional-failover-check

$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/r
egional-failover/service --path cncb-regional-failover-service
```

2. Navigate to the `cncb-regional-failover-check` directory with `cd cncb-regional-failover-check`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-regional-failover-check
...
functions:
  check:
    ...

resources:
  Resources:
```

```
    Api5xxAlarm:
      Type: AWS::CloudWatch::Alarm
      Properties:
        Namespace: AWS/ApiGateway
        MetricName: 5XXError
        Dimensions:
          - Name: ApiName
            Value: ${opt:stage}-${self:service}
        Statistic: Minimum
        ComparisonOperator: GreaterThanThreshold
        Threshold: 0
        Period: 60
        EvaluationPeriods: 1

    ApiHealthCheck:
      DependsOn: Api5xxAlarm
      Type: AWS::Route53::HealthCheck
      Properties:
        HealthCheckConfig:
          Type: CLOUDWATCH_METRIC
          AlarmIdentifier:
            Name:
              Ref: Api5xxAlarm
            Region: ${opt:region}
          InsufficientDataHealthStatus: LastKnownStatus
  Outputs:
    ApiHealthCheckId:
      Value:
        Ref: ApiHealthCheck
```

4. Review the file named `handler.js`.
5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack in the `us-east-1` and `us-west-2` regions:

```
$ npm run dp:lcl:e -- -s $MY_STAGE
...
  GET -
https://0987654321.execute-api.us-east-1.amazonaws.com/john/check

$ npm run dp:lcl:w -- -s $MY_STAGE
...
  GET -
https://1234567890.execute-api.us-west-2.amazonaws.com/john/check
```

9. Review the stack and resources in the AWS console for both regions.
10. Navigate to the `cncb-regional-failover-service` directory with `cd cncb-regional-failover-service`.
11. Review the file named `serverless.yml` with the following content:

```
service: cncb-regional-failover-service

plugins:
  - serverless-multi-regional-plugin
...
custom:
  dns:
    ...
    us-east-1:
      ...
      healthCheckId: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    us-west-2:
      ...
      healthCheckId: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
...
```

12. Update the following fields in the `serverless.yml` file:
    - `custom.dns.hostedZoneId`
    - `custom.dns.domainName`
    - `custom.dns.us-east-1.acmCertificateArn`
    - `custom.dns.us-east-1.healthCheckId` from the output of the `east` health check stack
    - `custom.dns.us-west-2.acmCertificateArn`
    - `custom.dns.us-west-2.healthCheckId` from the output of the `west` health check stack
13. Install the dependencies with `npm install`.
14. Run the tests with `npm test`.
15. Review the contents generated in the `.serverless` directory.
16. Deploy the stack in the `us-west-2` and `us-east-1` regions:

```
$ npm run dp:lcl:w -- -s $MY_STAGE
$ npm run dp:lcl:e -- -s $MY_STAGE
```



Deploying a CloudFront distribution can take 20 minutes or longer.

17. Review the stack and resources in the AWS console for both regions.
18. Test the `global` endpoint of the service and note the region returned in the payload:

```
$ curl -v https://cncb-regional-failover-service.example.com/hello

{"message":"Your function executed successfully in us-east-1!"}
```

19. Go to the `cncb-regional-failover-check-<stage>-check` Lambda function in the AWS console for the region that your request was routed to and change the `UNHEALTHY` environment variable to `true` and s*ave* the function. For example, the previous output shows `us-east-1`, so update the function in `us-east-1`.
20. Invoke the health check endpoint for that region multiple times over the course of several minutes to trigger a failover:

```
$ curl -v
https://0987654321.execute-api.us-east-1.amazonaws.com/$MY_STAGE/check
```

21. Test the `global` endpoint of the service and note that the region has changed:

```
$ curl -v https://cncb-regional-failover-service.example.com/hello

{"message":"Your function executed successfully in us-west-2!"
```

22. Review the status of the Route53 health checks in the AWS Console.
23. Remove the stacks once you are finished:

```
$ cd cncb-regional-failover-service
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAGE
$ cd ../cncb-regional-failover-check
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAGE
```

How it works...

Our regional health check service is designed to return a `5xx` status code when one or more of the required services returns an error. We add a CloudWatch alarm, named `Api5xxAlarm`, to the health check service that monitors the API Gateway `5xxError` metric in the given region and raises an alarm when there is at least one `5xx` in a minute. You will want to adjust the sensitivity of the alarm to your specific requirements. Next, we add a Route53 health check, named `ApiHealthCheck`, to the service that depends on the `Api5xxAlarm` and outputs the `ApiHealthCheckId` for use by other services. Finally, we associate the `healthCheckId` with the Route53 RecordSet for each service in each region, such as the `cncb-regional-failover-service`. When the alarm status is `Unhealthy`, Route53 will stop routing traffic to the region until the status is `Healthy` again.

In this recipe, we used the `UNHEALTHY` environment variable to simulate a regional failure and manually invoked the service to trigger the alarm. As we discussed in the *Creating a regional health check* recipe, the health check will typically be invoked on a regular basis by another service, such as Pingdom, to ensure that there is a constant flow of traffic asserting the health of the region. To increase coverage, we could also expand the alarm to check the `5xx` metric of all services in a region by removing the `ApiName` dimension from the alarm but still rely on pinging the health check service to assert the status when there is no other traffic.

Implementing regional replication with DynamoDB

Timely replication of data across regions is important to facilitate a seamless user experience when a regional failover occurs. During normal execution, regional replication will occur in near real time. During a regional failure, it should be expected that data would replicate more slowly. We can think of this as protracted eventual consistency. Fortunately, our cloud-native systems are designed to be eventually consistent. This means they are tolerant of stale data, regardless of how long it takes to become consistent. This recipe shows how to create global tables to replicate DynamoDB tables across regions and discusses why we do not replicate event streams.

Getting ready

Before starting this recipe, you will need an AWS Kinesis Stream in the `us-east-1` and `us-west-2` regions, such as the one created in the *Creating an event stream* recipe.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/d
ynamodb-global-table --path cncb-dynamodb-global-table
```

2. Navigate to the `cncb-dynamodb-global-table` directory with `cd cncb-dynamodb-global-table`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-dynamodb-global-table
...

plugins:
  - serverless-dynamodb-autoscaling-plugin
  - serverless-dynamodb-global-table-plugin

custom:
  autoscaling:
    - table: Table
      global: true
      read:
        ...
      write:
        ...

resources:
  Resources:
    Table:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${opt:stage}-${self:service}-things
        ...
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
```

4. Review the file named `handler.js` with the following content:

```
...
module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .filter(forOrigin)
    .map(toEvent)
    .flatMap(publish)
    .collect()
    .toCallback(cb);
};

const forOrigin = e => e.dynamodb.NewImage['aws:rep:updateregion'] &&
  e.dynamodb.NewImage['aws:rep:updateregion'].S ===
process.env.AWS_REGION;
...
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack in the `us-east-1` and `us-west-2` regions:

```
$ npm run dp:lcl:e -- -s $MY_STAGE
...
Serverless: Created global table: john-cncb-dynamodb-global-table-
things with region: us-east-1
...

$ npm run dp:lcl:w -- -s $MY_STAGE
...
Serverless: Updated global table: john-cncb-dynamodb-global-table-
things with region: us-west-2
...
```

9. Review the stack and resources in the AWS console in both regions.
10. Invoke the `command` function to save data to the `us-east-1` region:

```
$ sls invoke -f command -r us-east-1 -s $MY_STAGE -d
'{"id":"77777777-4444-1111-1111-111111111111","name":"thing one"}'
```

11. Invoke the `query` function to retrieve the data from the `us-west-2` region:

```
$ sls invoke -f query -r us-west-2 -s $MY_STAGE -d
77777777-4444-1111-1111-111111111111
{
    "Item": {
        "aws:rep:deleting": false,
        "aws:rep:updateregion": "us-east-1",
        "aws:rep:updatetime": 1534819304.087001,
        "id": "77777777-4444-1111-1111-111111111111",
        "name": "thing one",
        "latch": "open"
    }
}
```

12. Take a look at the logs for the `trigger` and `listener` functions in both regions:

```
$ sls logs -f trigger -r us-east-1 -s $MY_STAGE
$ sls logs -f listener -r us-east-1 -s $MY_STAGE
$ sls logs -f trigger -r us-west-2 -s $MY_STAGE
$ sls logs -f listener -r us-west-2 -s $MY_STAGE
```

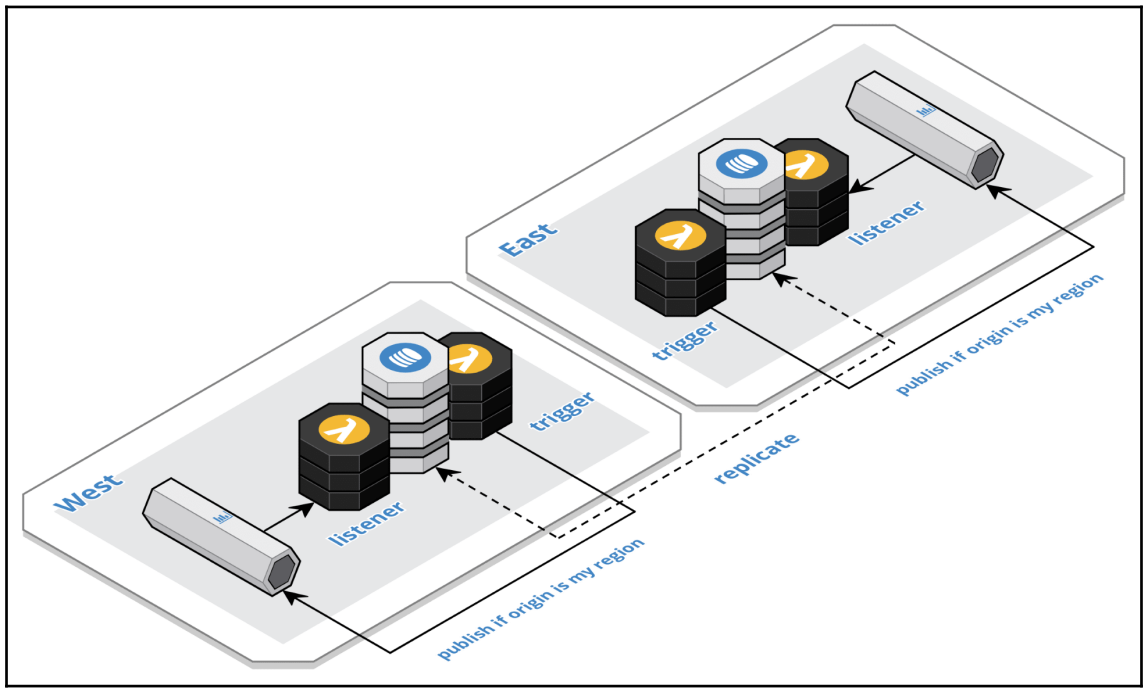
13. Remove both stacks once you are finished:

```
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAGE
```

How it works...

A **DynamoDB Global Table** is responsible for replicating data across all the regional tables that have been associated with the global table and keep the data synchronized, all in near real time. The `serverless-dynamodb-global-table-plugin` will create the global tables and is designed to work with the `serverless-dynamodb-autoscaling-plugin`. For each table that has the `global` flag set to true, the plugin will create the global table when the service is deployed to the first region. For each successive regional deployment the plugin will add the regional table to the global table. Each regional table must have the same name, have streams enabled, and have the same autoscaling policies, which is handled by the plugins. One thing that is not handled by the plugins is that the tables must all be empty when the global table is initially deployed.

We will start with the happy-path scenario, where there is no regional disruption and everything is working smoothly, and walk through the following diagram. When data is written to the table in a region, such as `us-east-1`, then the data is replicated to the `us-west-2` region. The `trigger` in the `us-east-1` region is also executed. The trigger has a `forOrigin` filter that will ignore all events where the `aws:rep:updateregion` field is not equal to the current `AWS_REGION`. Otherwise, the trigger will publish an event to the Kinesis Stream in the current region and all subscribers to the event will execute in the current region and replicate their own data to the other regions. The `listener` for the current service will ignore any events that it produced itself. In the `us-west-2` region the `trigger` will also be invoked after the replication, but the `forOrigin` filter will short-circuit the logic so that a duplicate event is not published to Kinesis and reprocessed by all the subscribers in that region. The inefficiency of duplicate event processing and the potential for infinite replication loops are two reasons why it is best not to replicate event streams and instead reply on replication at the leaf data stores:





Note that this recipe builds on the code from the *Implementing bi-directional synchronization* recipe, so you can review that recipe for additional details about the code.

During a regional failover, in the best-case scenario, a user's data will have already been replicated and the failover process will be completely seamless. The user's next commands will execute in the new region, the chain of events will process in the new region, and the results will eventually replicate back to the failed region. When there is some replication latency, **session consistency** helps make the failover process appear seamless, as we discussed in the *Leveraging session consistency* recipe. However, during a regional failover, it is likely that some subscribers in the failing region will fall behind on processing the remaining events in the regional stream. Fortunately, a regional disruption typically means that there is lower throughput in the failing region, as opposed to no throughput. This means that there will be a higher latency for replicating the results of event processing to the other regions but they will eventually become consistent. A user experience that is designed for eventual consistency, such as an email app on a mobile device, will handle this protracted eventual consistentcy in its stride.

The complexity of trying to keep track of which events have processed and which events are stuck in a failing region is another reason why it is best not to replicate event streams. In cases where this protracted eventual consistency cannot be tolerated, then the latest events in the new region can rely on session consistency for more up-to-date information and use the techniques discussed in the *Implementing idempotency with an inverse oplock* and *Implementing idempotency with event sourcing* recipes to handle the older events that are received out of order from the slowly recovering region.

Implementing round-robin replication

Not all of the databases in our polyglot persistence architecture will support turnkey regional replication as we have with AWS DynamoDB, yet we still need to replicate their data to multiple regions to improve latency and support regional failover. This recipe demonstrates how to use AWS S3 as a surrogate to add regional replication to any database.

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch10/r
ound-robin-replication --path cncb-round-robin-replication
```

2. Navigate to the `cncb-round-robin-replication` directory with `cd cncb-round-robin-replication`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-round-robin-replication
...
functions:
  listener:
    ...
  trigger:
    ...
  search:
    ...
  replicator:
    handler: replicator.trigger
    events:
      - sns:
          arn:
            Ref: BucketTopic
          topicName: ${self:service}-${opt:stage}-trigger
    environment:
      REPLICATION_BUCKET_NAME:
${self:custom.regions.${opt:region}.replicationBucketName}

custom:
  regions:
    us-east-1:
      replicationBucketName: cncb-round-robin-replication-${opt:stage}-
bucket-WWWWWWWWWWWW
    us-west-2:
      replicationBucketName: cncb-round-robin-replication-${opt:stage}-
bucket-EEEEEEEEEEEE
...
```

4. Review the file named `replicator.js` with the following content:

```
module.exports.trigger = (event, context, cb) => {
  _(event.Records)
    .flatMap(messagesToTriggers)
    .flatMap(get)
    .filter(forOrigin)
    .flatMap(replicate)
    .collect()
    .toCallback(cb);
};

...
```

```
const forOrigin = uow => uow.object.Metadata.origin !==
process.env.REPLICATION_BUCKET_NAME;
...
const replicate = uow => {
  const { ContentType, CacheControl, Metadata, Body } = uow.object;

  const params = {
    Bucket: process.env.REPLICATION_BUCKET_NAME,
    Key: uow.trigger.s3.object.key,
    Metadata: {
      'origin': uow.trigger.s3.bucket.name,
      ...Metadata,
    },
    ACL: 'public-read',
    ContentType,
    CacheControl,
    Body,
  };

  const s3 = new aws.S3(...);
  return _(
    s3.putObject(params).promise()
    ...
  );
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test -- -s $MY_STAGE`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack in the `us-east-1` and `us-west-2` regions:

```
$ npm run dp:lcl:e -- -s $MY_STAGE
$ npm run dp:lcl:w -- -s $MY_STAGE
```

9. Update the `replicationBucketName` variables in the `serverless.yml` so that `us-east-1` replicates to `us-west-2` and visa versa, and then redeploy the stacks.
10. Review the stacks and resources in the AWS Console.
11. Publish an event from a separate Terminal with the following commands:

```
$ cd <path-to-your-workspace>/cncb-event-stream
$ sls invoke -f publish -r us-east-1 -s $MY_STAGE -d '{"type":"thing-
created","thing":{"new":{"name":"thing
two","id":"77777777-5555-1111-1111-111111111111"}}}'
```

12. Invoke the following curl command to search the data in the `us-west-2` region:

```
$ curl
https://<API-ID>.execute-api.us-west-2.amazonaws.com/$MY_STAGE/search?q
=two | json_pp

[
   {
      "id" : "77777777-5555-1111-1111-111111111111",
      "url" :
"https://s3.amazonaws.com/cncb-round-robin-replication-john-
bucket-1cqxst40pvog4/things/77777777-5555-1111-1111-111111111111",
 "name" : "thing two"
 }
]
```

13. Take a look at the logs for the `trigger` and `listener` functions in both regions:

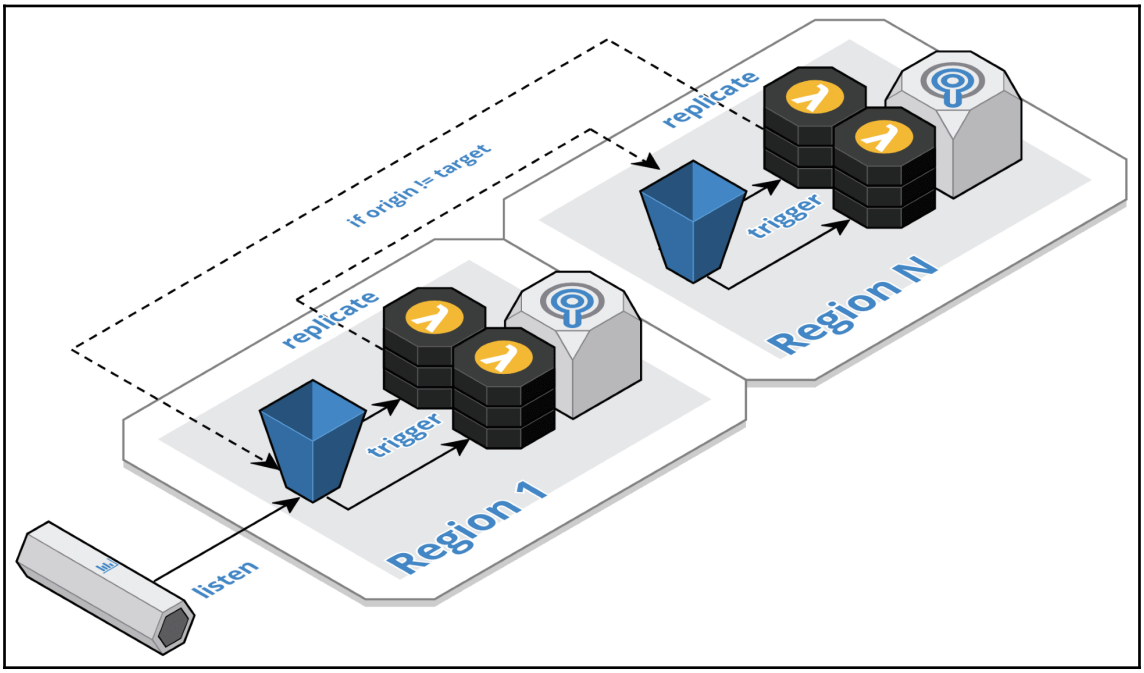
```
$ sls logs -f replicator -r us-east-1 -s $MY_STAGE
...
2018-08-19 17:00:05 ... [AWS s3 200 0.04s 0 retries] getObject({
Bucket: 'cncb-round-robin-replication-john-bucket-1a3rh4v9tfedw',
  Key: 'things/77777777-5555-1111-1111-111111111111' })
2018-08-19 17:00:06 ... [AWS s3 200 0.33s 0 retries] putObject({
Bucket: 'cncb-round-robin-replication-john-bucket-1cqxst40pvog4',
  Key: 'things/77777777-5555-1111-1111-111111111111',
  Metadata:
   { origin: 'cncb-round-robin-replication-john-bucket-1a3rh4v9tfedw'
},
  ACL: 'public-read',
  ContentType: 'application/json',
  CacheControl: 'max-age=300',
  Body: <Buffer ... > })
...

$ sls logs -f replicator -r us-west-2 -s $MY_STAGE
...
2018-08-19 17:00:06 ... [AWS s3 200 0.055s 0 retries] getObject({
Bucket: 'cncb-round-robin-replication-john-bucket-1cqxst40pvog4',
  Key: 'things/77777777-5555-1111-1111-111111111111' })
2018-08-19 17:00:06 ... {... "object":{..."Metadata":{"origin":"cncb-
round-robin-replication-john-bucket-1a3rh4v9tfedw"}, ...}}
...
```

14. Empty the bucket in each region before removing the stacks
15. Remove both stacks once you have finished:

```
$ npm run rm:lcl:e -- -s $MY_STAGE
$ npm run rm:lcl:w -- -s $MY_STAGE
```

How it works...

In this recipe, we are creating a **materialized view** in Elasticsearch, and we want to allow users to search against the data in the region that they are closest to. However, Elasticsearch does not support regional replication. As we discussed in the *Implementing regional replication with DynamoDB* recipe, we do not want to replicate the event stream because that solution is complex and too difficult to reason about. Instead, as shown in the following diagram, we will place an S3 bucket in front of Elasticsearch in each region and leverage S3 triggers to update Elasticsearch and to implement a round-robin replication scheme:





Note that this recipe builds on the code from the *Implementing a search BFF* recipe, so you can review that recipe for additional details about the code.

The service listens to the Kinesis Stream in the current region and writes the data to an S3 bucket in the current region, which generates an S3 trigger that is routed to an SNS topic. A function reacts to the topic and creates the materialized view in Elasticsearch in the current region. Meanwhile, a `replicator` function also reacts to the same topic. The replicator copies the contents of the object from the S3 bucket to the matching bucket in the next region, as specified by the `REPLICATION_BUCKET_NAME` environment variable. This in turn generates a trigger in that region. Once again, a function responds to the topic and creates the materialized view in Elasticsearch in that region as well. The `replicator` in that region also responds and looks to copy the object to the next region. This process of `trigger` and `replicate` will round robin for as many regions as necessary, until the `forOrigin` filter sees that the origin bucket (that is, `uow.object.Metadata.origin`) is equal to the target of the current replicator (that is, `process.env.REPLICATION_BUCKET_NAME`). In this recipe, we have two regions—`us-east-1` and `us-west-2`. The data originates in the east region, so the east replicator copies the data to the west bucket (`1cqxst40pvog4`). The west replicator does not copy the data to the east bucket (`1a3rh4v9tfedw`) because the origin is the east bucket.

This round robin replication technique is a simple and cost-effective approach that builds on the event architecture that is already in place. Note that we cannot leverage the built-in S3 replication feature for this purpose because it only replicates to a single region and does not create a chain reaction. However, we could add S3 replication to these buckets for backup and disaster recovery, as we discussed in the *Replicating the data lake for disaster recovery* recipe.

11
Welcoming Polycloud

In this chapter, the following recipes will be covered:

- Creating a service in Google Cloud Functions
- Creating a service in Azure Functions

Introduction

Vendor lock-in is a common concern with serverless, cloud-native development. However, this concern is a relic of the monolithic thinking that stems from monolithic systems that must be changed in whole from one vendor to another. Autonomous services, on the other hand, can be changed one by one. Nevertheless, the elusive promise of *write once; run anywhere* is still the battle cry of the *multi-cloud* approach. Yet, this approach ignores the fact that the part that is written once is only the tip of a very big iceberg, and what lies below the waterline embodies the most risk and does not translate directly between cloud providers. This inevitably leads to the use of a least-common denominator set of tools and techniques that can more easily be lifted and shifted from one provider to another.

We chose instead to embrace the *disposable architecture* of fully managed, value-added cloud services. This *serverless-first* approach empowers self-sufficient, full-stack teams to be lean and experiment with new ideas, fail-fast, learn, and adjust course quickly. This leads naturally to a *polyglot-cloud* or *polycloud* approach, where teams select the best cloud provider service by service. Ultimately, companies do have a preferred cloud provider. But there is an argument to be made for diversification, where some percentage of services are implemented on different cloud providers to gain experience and leverage. The goal then is to have a consistent pipeline experience across services with similar, if not the same, tools, techniques, and patterns for development, testing, deployment, and monitoring. The recipes in many of the previous chapters focused on AWS fully managed, value-added cloud services. The recipes in this chapter demonstrate how a consistent cloud-native development pipeline experience is possible with additional cloud providers, such as Google and Azure.

Creating a service with Google Cloud Functions

The **Serverless Framework** provides an abstraction layer above many different cloud providers that facilitates a consistent development and deployment experience. This recipe demonstrates how to create a service with Google Cloud Functions.

Getting ready

Before starting this recipe, you will need a Google Cloud Billing Account, project, and credentials that are configured for the Serverless Framework (`https://serverless.com/framework/docs/providers/google/guide/credentials`).

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch11/g
cp --path cncb-gcp
```

2. Navigate to the `cncb-gcp` directory with `cd cncb-gcp`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-gcp

provider:
  name: google
  runtime: nodejs8
  project: cncb-project
  region: ${opt:region}
  credentials: ~/.gcloud/keyfile.json

plugins:
  - serverless-google-cloudfunctions

functions:
  hello:
    handler: hello
    events:
      - http: path
```

```
...
#resources:
#  resources:
#    - type: storage.v1.bucket
#      name: my-serverless-service-bucket
```

4. Review the file named `index.js` with the following content:

```
exports.hello = (request, response) => {
  console.log('env: %j', process.env);
  response.status(200).send('... Your function executed
successfully!');
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-gcp@1.0.0 dp:lcl <path-to-your-workspace>/cncb-gcp
> sls deploy -v --r us-east1 "-s" "john"
...
Serverless: Done...
Service Information
service: cncb-gcp
project: cncb-project
stage: john
region: us-east1

Deployed functions
hello
  https://us-east1-cncb-project.cloudfunctions.net/hello
```

9. Review the deployment and resources in the Google Cloud console.
10. Invoke the endpoint shown in the stack output in the following commands:

```
$ curl -v https://us-east1-cncb-project.cloudfunctions.net/hello
...
JavaScript Cloud Native Development Cookbook! Your function executed
successfully!
```

11. Take a look at the logs:

```
$ sls logs -f hello -r us-east1 -s $MY_STAGE
...
2018-08-24T05:10:20...: Function execution took 12 ms, finished with
status code: 200
...
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The first thing to note is that the steps of the *How to do it...* section are virtually the same as all the previous recipes. This is because the Serverless Framework abstracts away the deployment details, and we further wrap all the commands with *NPM* scripts to encapsulate dependency management. From here, we can use all the same tools and techniques for development and testing as outlined in `Chapter 6`, *Building a Continuous Deployment Pipeline*. This enables team members to transition smoothly when working across services that are implemented on different cloud providers.

The `serverless-google-cloudfunctions` plugin handles the details of interacting with the Google Cloud APIs, such as Cloud Functions and Deployment Manager, to provision the service. The `serverless.yml` file should look very familiar. We specify the `provider.name` as `google` and set up the `plugins`, and then we focus on defining the `functions` and `resources`. The details are cloud provider-specific, but the details of cloud provider-specific, value-added services are usually why we choose a specific provider for a specific service. The Node.js code in the `index.js` file is familiar as well, though the function signature is different. Ultimately, there is a clear mapping of Google Cloud services for implementing the cloud-native patterns and techniques enumerated in the recipes throughout this cookbook.

Creating a service with Azure Functions

The Serverless Framework provides an abstraction layer above many different cloud providers that facilitates a consistent development and deployment experience. This recipe demonstrates how to create a service with Azure Functions.

Getting ready

Before starting this recipe, you will need an Azure account and credentials configured for the Serverless Framework (`https://serverless.com/framework/docs/providers/azure/guide/credentials`).

How to do it...

1. Create the project from the following template:

```
$ sls create --template-url
https://github.com/danteinc/js-cloud-native-cookbook/tree/master/ch11/a
zure --path cncb-azure
```

2. Navigate to the `cncb-azure` directory with `cd cncb-azure`.
3. Review the file named `serverless.yml` with the following content:

```
service: cncb-azure-${opt:stage}

provider:
  name: azure
  location: ${opt:region}

plugins:
  - serverless-azure-functions

functions:
  hello:
    handler: handler.hello
    events:
      - http: true
        x-azure-settings:
          authLevel : anonymous
      - http: true
        x-azure-settings:
          direction: out
          name: res
```

4. Review the file named `handler.js` with the following content:

```
module.exports.hello = function (context) {
  context.log('context: %j', context);
  context.log('env: %j', process.env);

  context.res = {
```

```
    status: 200,
    body: '... Your function executed successfully!',
  };

  context.done();
};
```

5. Install the dependencies with `npm install`.
6. Run the tests with `npm test`.
7. Review the contents generated in the `.serverless` directory.
8. Deploy the stack:

```
$ npm run dp:lcl -- -s $MY_STAGE

> cncb-azure@1.0.0 dp:lcl <path-to-your-workspace>/cncb-azure
> sls deploy -v -r 'East US' "-s" "john"
...
Serverless: Creating resource group: cncb-azure-john-rg
Serverless: Creating function app: cncb-azure-john
...
Serverless: Successfully created Function App
```

9. Review the deployment and resources in the Azure console.
10. Tail the logs from another terminal:

```
$ sls logs -f hello -r 'East US' -s $MY_STAGE
Serverless: Logging in to Azure
Serverless: Pinging host status...
2018-08-25T04:02:34  Welcome, you are now connected to log-streaming
service.
2018-08-25T04:05:00.843 [Info] Function started (Id=...)
2018-08-25T04:05:00.856 [Info] context: {...}
2018-08-25T04:05:00.856 [Info] env: {...}
2018-08-25T04:05:00.856 [Info] Function completed (Success, Id=...,
Duration=19ms)
```

11. Invoke the endpoint shown in the stack output in the following commands:

```
$ curl -v https://cncb-azure-$MY_STAGE.azurewebsites.net/api/hello
...
JavaScript Cloud Native Development Cookbook! Your function executed
successfully!
```

12. Remove the stack once you are finished with `npm run rm:lcl -- -s $MY_STAGE`.

How it works...

The first thing to note is that the steps of the *How to do it…* section are virtually the same as in all the previous recipes. This is because the Serverless Framework abstracts away the deployment details, and we further wrap all the commands with *NPM scripts* to encapsulate dependency management. From here we can use all the same tools and techniques for development and testing as outlined in `Chapter 6`, *Building a Continuous Deployment Pipeline*. This enables team members to transition smoothly when working across services that are implemented on different cloud providers.

The `serverless-azure-cloudfunctions` plugin handles the details of interacting with the Azure APIs, such as Azure Functions and Resource Manager, to provision the service. The `serverless.yml` file should look very familiar. We specify the `provider.name` as `azure` and set up the `plugins`, and then we focus on defining the `functions`. The details are cloud provider-specific, but the details of the cloud provider-specific, value-added services are usually why we choose a specific provider for a specific service. The Node.js code in the `handler.js` file is familiar as well, though the function signature is different. Ultimately, there is a clear mapping of Azure services for implementing the cloud-native patterns and techniques enumerated in the recipes throughout this cookbook.

Other Books You May Enjoy

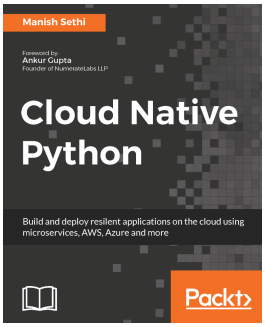
If you enjoyed this book, you may be interested in these other books by Packt:



Cloud Native Python Cloud Native Development Patterns and Best Practices
John Gilbert

ISBN: 978-1-78847-392-7

- Enable massive scaling by turning your database inside out
- Unleash flexibility via event streaming
- Leverage polyglot persistence and cloud-native databases
- Embrace modern continuous delivery and testing techniques
- Minimize risk by evolving your monoliths to cloud-native
- Apply cloud-native patterns and solve major architectural problems in cloud environment



Cloud Native Python
Manish Sethi

ISBN: 978-1-78712-931-3

- Get to know "the way of the cloud", including why developing good cloud software is fundamentally about mindset and discipline
- Know what microservices are and how to design them
- Create reactive applications in the cloud with third-party messaging providers
- Build massive-scale, user-friendly GUIs with React and Flux
- Secure cloud-based web applications: the do's, don'ts, and options
- Plan cloud apps that support continuous delivery and deployment

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

access control list (ACL) 204
alerts
  creating 241, 242, 244
Amazon Resource Names (ARN) 23
analytics BFF
  implementing 112, 115, 117
API Gateway
  creating 29, 30, 31, 32
  securing, with OpenID Connect 176, 179
Apollo Client
  reference link 285
asynchronous API
  contract tests, writing 225, 227
asynchronous non-blocking IO
  leveraging 275, 277
AWS Kinesis 39
Azure Functions
  used, for creating service 316, 318, 319

B

Backend For Frontend (BFF) 29, 66, 98, 282
backpressure
  implementing 250, 252, 254
bi-directional synchronization
  implementing 91, 94, 96

C

cache-control
  utilizing 282, 284
CI/CD pipeline
  creating 211, 213, 215
cloud account
  securing 168, 170
cloud-native system
  monitoring 234, 235, 236
CloudFormation stacks 9
code
  executing, in cloud 162, 164, 166
Cognito datasets
  materialized view, creating 76, 78, 80
  used, for applying database-first variant of Event Sourcing pattern 58, 60, 62
collaboration
  orchestrating, between services 125, 128, 129
Command Query Responsibility Segregation (CQRS) 39, 63
Command-Line Interface (CLI) 85
configuration by exception approach 12, 18
Content Delivery Network (CDN)
  about 136
  custom domain name, associating 140, 143
  invalidation of content, triggering 157, 159, 161
  service, deploying 149, 152, 153
  single-page application, serving 136, 139, 140
  static JSON, serving 153, 155, 157
  website, serving 143, 148
Content Management Systems (CMS) 143
contract tests
  writing, for asynchronous API 225, 227
  writing, for synchronous API 223, 224
cross-site scripting (XSS) 201
custom authorizer
  implementing 179, 182, 184
custom domain name
  associating, with CDN 140, 143
custom metrics
  implementing 236, 238

D

data lake
  about 39, 41, 43
  creating 39, 41, 43

indexing 86, 89, 90
replicating, for disaster recovery 205, 207, 209
disaster recovery
data lake, replicating 205, 207, 209
domain events
monitoring 239, 240
downstream services 24
DynamoDB Streams 27
DynamoDB
autoscaling 280, 282
materialized view, creating 63, 65, 66
used, for applying database-first variant of Event Sourcing pattern 52, 54, 57
used, for implementing regional replication 302, 304, 306, 307

E

Elasticsearch
materialized view, creating 71, 73, 75
envelope encryption
using 193, 194, 197, 198
Event Sourcing pattern
database-first variant, applying with Cognito datasets 58, 60, 62
database-first variant, applying with DynamoDB 52, 54, 57
event-first variant, applying 43, 45, 46
Event Sourcing
used, for implementing idempotence 264, 265, 266
event stream
creating 19, 20, 21, 23
event
publishing 19, 20, 21, 23
replaying 81, 84, 86
External Service Gateway (ESG)
about 98, 118, 121
implementing 118, 121

F

faults events
handling 255, 257, 258
resubmitting 258, 260
feature flags
leveraging 230, 231, 232
federated identity pool
creating 171, 173, 174
Function as a Service (Faas)
about 14, 270
tuning 268, 269, 271
Functional Reactive Programming (FRP) 254
functions
creating 14, 15, 16, 17, 18

G

Google Cloud Functions
used, for creating service 314, 316
GraphQL CRUD BFF
implementing 98, 101, 105, 106
GraphQL-based service
authorizing 184, 186, 188

H

Highland.js
reference link 28, 254

I

idempotence
implementing, with Event Sourcing 264, 265, 266
implementing, with inverse OpLock 261, 262, 264
integration tests
writing 219, 220, 222
inverse OpLock
about 264
used, for implementing idempotence 261, 262, 264

J

JSON Web Token (JWT) 176
JWT filter
implementing 188, 190, 192

K

Key Management Service (KMS) 193
Kibana 90
Kinesis 27

L

latency-based routing
  implementing 290, 291, 293
logging 18
logs
  working with 14, 15, 16, 17, 18

M

materialized view
  creating, in Cognito dataset 76, 78, 80
  creating, in DynamoDB 63, 65, 66
  creating, in Elasticsearch 71, 73, 75
  creating, in S3 67, 69, 70
metrics
  working with 14, 15, 16, 17, 18
micro event store
  creating 47, 50, 52
multi-factor authentication (MFA) 170

N

Node Package Manager (NPM)
  reference link 12

O

object relation mapping (ORM) 270
OpenID Connect
  used, for securing API Gateway 176, 179
outbound External Service Gateway
  implementing 122, 124, 125

P

Pingdom
  reference link 244

R

rate limiting
  implementing 250
real user monitoring (RUM) 247
RED method
  reference link 243
regional failover
  triggering 298, 300, 302
regional health check
  creating 294, 296, 297
regional replication
  implementing, with DynamoDB 302, 304, 306, 307
requests
  batching 271, 273, 274
retries
  employing 248, 250
round-robin replication
  implementing 307, 309, 311, 312
RxJS
  reference link 28, 254

S

S3
  materialized view, creating 67, 69, 70
Saga
  implementing 130, 132, 134
search BFF
  implementing 106, 109, 111
Serverless Framework
  reference link 12, 314
service
  collaboration, orchestrating between 125, 128, 129
  creating, with Azure Functions 316, 318, 319
  creating, with Google Cloud Functions 314, 316
session consistency
  leveraging 285, 287
sign in
  implementing 174, 176
sign out
  implementing 174, 176
sign up
  implementing 174, 176
Simple Notification Service (SNS) 111
single-page application (SPA)
  about 135
  deploying 33, 34, 35, 37
  serving, from CDN 136, 139, 140
SSL certificate
  creating, for encryption in transit 198, 200, 201
stack
  creating 9, 11, 12
stage 13

static JSON
  serving, from CDN 153, 155, 157
stream processor
  about 24
  creating 25, 26, 27, 29
  events, grouping 277, 279
synchronous API
  contract tests, writing 223, 224
synthetic transaction tests
  creating 244, 246, 247

T

timeouts
  employing 248, 250
transitive end-to-end tests
  assembling 227, 229, 230

U

unit of work (UOW) 258, 275
unit tests
  creating 217, 218
  writing 216
upstream services 24

W

web application firewall (WAF)
  about 140, 201, 203, 204
  configuring 201, 203
Web Application Firewall (WAF)
  configuring 204
website
  serving, from CDN 143, 148