Community Experience Distilled

RESTful Web Services with Scala

Learn the art of creating scalable RESTful web services with Scala

Jos Dirksen

[PACKT] open source*
PUBLISHING
community experience distilled

RESTful Web Services with Scala

Table of Contents

RESTful Web Services with Scala

RESTful Web Services with Scala

First published: November 2015

Production reference: 1241115

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-940-8

www.packtpub.com

Credits

Author

Jos Dirksen

Reviewer

William Scott

Commissioning Editor

Kunal Parikh

Acquisition Editor

Reshma Raman

Content Development Editor

Preeti Singh

Technical Editor

Ankita Thakur

Copy Editor

Swati Priya

Project Coordinator

Shweta H. Birwatkar

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Jason Monteiro

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Jos Dirksen has worked as a software developer and architect for more than a decade. He has a lot of experience in a large variety of technologies, ranging from backend technologies, such as Java and Scala, to frontend development using HTML5, CSS, and JavaScript. Besides working with these technologies, Jos also regularly speaks at conferences and likes to write about new and interesting technologies on his blog. He also likes to experiment with new technologies and see how they can best be used to create beautiful data visualizations, the results of which you can see on his blog at http://www.smartjava.org/.

Jos currently works as a Scala engineer for a large Dutch financial institution and has just given up a position as an enterprise architect for Malmberg, a large Dutch publisher of educational material. There, he helped develop a new digital platform for the creation and publishing of educational content for primary, secondary, and vocational education. Previously, Jos has worked in many different roles in private and public sectors, ranging from private companies, such as Philips and ASML, to organizations in the public sector, such as the Department of Defense of the Netherlands.

Jos has also written three books on Three.js. *Learning Three.js*, *Packt Publishing*, provides a complete overview of all the features of Three.js; *Three.js Essentials*, *Packt Publishing*, uses an example-based approach to explore the most important features of Three.js, and *Three.js Cookbook*, *Packt Publishing*, provides a recipe-based approach to cover important use cases of Three.js.

Besides his interest in Scala and REST, he is also interested in frontend development using JavaScript and HTML5.

Writing a book isn't something you do yourself. A lot of people have helped and supported me when I was writing this book. Special thanks to the following people:

All the guys from Packt Publishing who have helped me during the writing, reviewing, and laying out part of the process. Great work, guys!

The authors and contributors of all the great Scala frameworks that I've written about in this book.

Much thanks to the reviewers. Great feedback and comments really helped improve the book. Your positive remarks really helped shape the book!

And, of course, I'd like to thank my family. I'd like to thank my wife, Brigitte, for supporting me every time I start on a new book, and, of course, my two girls, Sophie and Amber, who continuously make me realize what is really important.

About the Reviewer

William Scott is a software engineer with over a decade of experience in building web applications. After graduating from Wake Forrest University with a master's degree in computer science, he focused on server-side Java for several projects before recently diving into JavaScript and AngularJS. In his spare time, he enjoys dabbling with iOS and Functional reactive Programming. He is an avid endurance runner, coffee drinker, and board gamer.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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

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

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Scala is a fantastic language that provides a great combination between functional and object-oriented programming. With Scala, you're able to write beautiful, concise, and easily maintainable code in a highly productive manner. Scala also provides very useful language constructs for creating REST services. In the Scala ecosystem, there is a large number of REST frameworks and each one of them allows you to create such services in a slightly different manner. This book gives you an overview of five of the most mature and flexible REST frameworks available, and shows you, through extensive examples, how you can use the various features of these frameworks.

What this book covers

Chapter 1, *Getting Started with RESTful Web Services*, shows you how to get the code, set up your IDE, and get a first minimal service up and running. It also gives some background information on REST and the model we'll use throughout the book.

Chapter 2, *A Functional-style REST Service with Finagle and Finch*, explains how you can use the Finch library together with Finagle to create a REST service that follows a functional programming approach.

Chapter 3, *A Pattern-matching Approach to REST Services with Unfiltered*, shows how you can use Unfiltered, a small REST library, to create a REST service. With Unfiltered, you can fully control how your HTTP request and responses are processed.

Chapter 4, *An Easy REST Service using Scalatra*, uses a part of the Scalatra framework to create a simple REST service. Scalatra is a great intuitive Scala framework, which makes creating REST services very easy.

Chapter 5, *Defining a REST Service Using Akka HTTP DSL*, focuses on the DSL part of Akka HTTP and explains how you can use the DSL, which is based on the well-known Spray DSL, to create easily composable REST services.

Chapter 6, *Creating REST Services with the Play 2 Framework*, explains how you can use part of the well-known Play 2 web framework to create a REST service.

Chapter 7, *JSON, HATEOAS, and Documentation*, goes deeper into the two important aspects of REST—JSON and HATEOAS. This chapter shows you, through examples, how to work with JSON and incorporate HATEOAS into the discussed frameworks.

What you need for this book

To work with the examples in this book, you don't need any special software besides Scala. The first chapter explains how to install the latest Scala version, and from there, you can get all the libraries for the frameworks explained in this book.

Who this book is for

If you are a Scala developer with some Scala experience and you want to get an overview of the frameworks that are available in the Scala world, then this book is perfect for you. You need to have a general knowledge of REST and Scala. This book is great for senior Scala (or other language) developers who are looking for a good REST framework to use together with Scala.

Conventions

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

Code words in text, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "In this case, we reject the request because the `HttpMethod` (the verb) didn't match anything we can process."

A block of code is set as follows:

```
class ScalatraBootstrapStep1 extends LifeCycle {
  override def init(context: ServletContext) {
    context mount (new ScalatraStep1, "/*")
  }
}
```

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

```
$ sbt runCH04-runCH04Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter4.steps.ScalatraRunnerStep1
10:51:40.313 [run-main-0] INFO  o.e.jetty.server.ServerConnector - Started
ServerConnector@538c2499{HTTP/1.1}{0.0.0.0:8080}
10:51:40.315 [run-main-0] INFO  org.eclipse.jetty.server.Server - Started
@23816ms
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In Postman you can use the **Step 03 – Get All Tasks** request for this."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

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

Piracy

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

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

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

Questions

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

Chapter 1. Getting Started with RESTful Web Services

Welcome to this book on how to create REST services with Scala. In this book, I'll introduce a couple of different Scala-based frameworks and show you how to create a RESTful service with them. Each of these frameworks has its own specific way of creating REST services; some are more functional, while others provide a rich **domain-specific language** (**DSL**). After reading this book and working through the examples, you'll be able to choose the approach that best suits you and your specific problem.

In this book, the following Scala frameworks will be discussed:

- **Akka HTTP / DSL**: Akka HTTP is a new REST framework built on top of Akka Streams. It provides a DSL-based approach based on Spray. Spray is one of the best-known REST frameworks in the Scala world, and the newest version will run on top of Akka HTTP. We'll explore the features of this DSL and show you how it can be used to create a REST service.
- **Unfiltered**: Unfiltered is a little REST framework which provides a very structured approach of creating REST services. This framework provides direct access to all parts of the HTTP request, and doesn't make assumptions on how you want to process REST services. This gives you complete control of how the request is processed and the response is produced.
- **Play 2**: Play 2 is one of the most popular Scala frameworks, which provides functionality to create complete web applications. Play 2 also provides great support for creating standard REST services. We'll focus on the REST-specific features of Play 2.
- **Finagle** and **Finch**: Finagle and Finch both come from the people at Twitter. With Finagle and Finch, it is possible to create REST services using a clean, functional programming approach.
- **Scalatra**: The last framework we'll discuss is the Scalatra framework. Scalatra is a lightweight framework, based on the better-known Sinatra framework, with which it is very easy to create REST services.

Besides these frameworks, in the last chapter of this book we'll also provide some guidelines on how to work with advanced topics such as HATEOAS, linking, and JSON processing.

In this first chapter, we are not going to explore a framework, but we'll use this chapter to introduce some concepts and set up some tools:

- We'll first have to make sure you can run all the examples provided with this book, so we'll show you how to get the code and setup SBT and an IDE
- We'll also do a short introduction into what RESTful services are
- And finally, we'll have a look at the API of the service that we'll implement using the different Scala frameworks

There are many different definitions of REST, so before we look into the technical details, let's first look at the definition of REST we'll use in this book.

Introduction to the REST framework

In this book, when we talk about REST, we talk about REST as described in the dissertation of Roy Fielding (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). Basically, REST is a software architecture style, which, when following the guidelines, can be used to create performant, reliable, and maintainable services. To better understand what REST is, it is a good idea to start with the constraints a service must follow to be RESTful. In his dissertation, Roy Fielding defines the following set of constraints:

- **Client-server**: This constraint means that clients and servers are separated from each other through a standardized interface. The advantage of this approach is that clients don't need to worry about persistency, databases, messaging, scalability, and other server-side concepts; instead, they can focus on user-oriented functionality. Another advantage is that clients and servers can be developed independently since the only dependency between them is the standardized contract. Note that if you require a very strict contract between the client and the server, a WSDL/SOAP-based service might be a better option than going for a RESTful approach.
- **Stateless**: Besides having a separate client and server, communication between these two components will have to be stateless. This means that each request the client sends should contain all the information necessary for the server. Note that for authentication, the server can temporarily store some session/user information in a persistent store, but all the real application state should be stored at the client. The big advantage of this approach is that this way it is very easy to scale out the servers horizontally by just adding more instances.
- **Cacheable**: In a RESTful architecture, clients are allowed to cache responses. It is up to the server side to indicate which responses might be cached and for how long. The goal of this constraint is to minimize interactions between the client and the server by avoiding sending requests, whose response will stay the same. This, of course, improves performance at the client side and reduces bandwidth.
- **Layered system**: This constraint describes that in a RESTful architecture, it is possible to create a layered system, where each layer has its own specific functionality. For instance, in between the client and the server, there might be a firewall, a load balancer, a reverse proxy, and so on. The client, however, doesn't notice these different layers.
- **Uniform interface**: From all the constraints, this is perhaps the most interesting one. This constraint defines what a uniform interface (the contract between the client and the server) should look similar to. This constraint itself consists of the following four sections:
    - **Identification of resources**: In requests, each resource should be uniquely identified. Most often, this is done through a form of URI. Note that the technical representation of a resource doesn't matter. A resource, identified through a URI, can be represented in JSON, CSV, XML, and PDF while still remaining the same resource.
    - **Manipulation of resources through these representations**: When a client has a representation of a resource (for example, a JSON message), the client can modify this resource by updating the representation and sending it to the server.
    - **Self-descriptive messages**: Each message sent between the client and the server should be

self-descriptive. The client need not know anything else to be able to parse and process the message. It should be able to learn from the message exactly what it can do with the resource.

- **Hypermedia as the engine of application state**: This constraint, also called **HATEOAS**, implies that a user of an API doesn't need to know beforehand what it can do with a specific resource. Through the use of links in the resource and the definition of media-types, a client can explore and learn the actions it can take on a resource.
- **Code on demand**: Code on demand is the only constraint that is optional. When comparing this constraint with the others, it is also one that is a bit different than the others. The idea behind this constraint is that servers could temporarily extend the functionality of clients by transferring executable code. In practice, this constraint is not seen that often though; most RESTful services deal with sending static responses, not executable code.

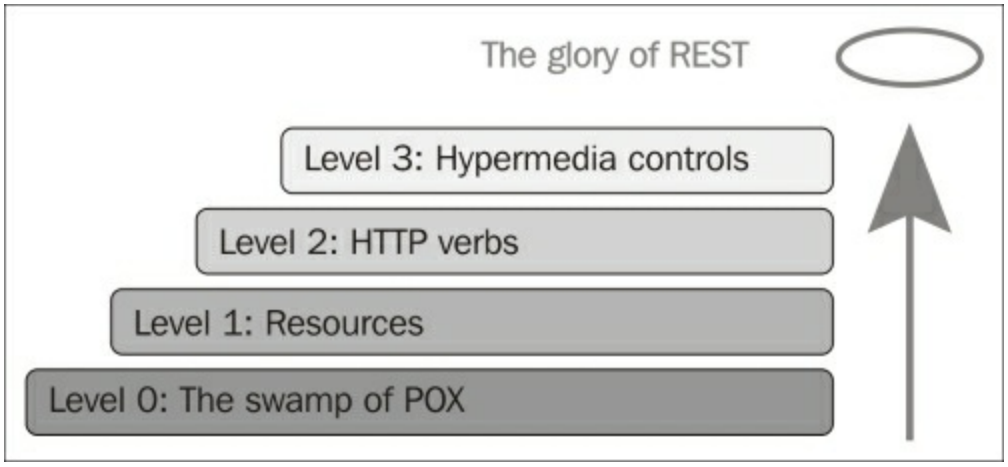
It's important to note that these constraints don't say anything about an implementation technology.

Tip

Often, when talking about REST, people immediately focus on HTTP and JSON. A RESTful architecture doesn't force you to adopt these technologies. On the other hand, most often, RESTful architectures are implemented on top of HTTP and currently use JSON as the message format. In this book, we will also focus on using HTTP and JSON to implement RESTful services.

The constraints mentioned here give an overview of how a service should act to be considered RESTful. However, when creating a service, it is often very hard to comply with all these constraints, and in some cases, not all the constraints might be that useful, or might be very hard to implement. Many people noticed this, and a couple of years ago, a more pragmatic view on REST was presented by Richardson's Maturity Model (http://martinfowler.com/articles/richardsonMaturityModel.html).

In Richardson's Maturity Model, you don't have to follow all the constraints to be considered RESTful; instead, a number of levels of maturity are defined that indicate how RESTful your service is. The higher the level, the more mature your service is, which will result in a more maintainable, more scalable, and easier-to-use service. This model defines the following levels:



The levels are described like this:

- Level 0 describes the situation where you just send XML or JSON objects to a single HTTP endpoint. Basically, you're not doing REST, but you're doing RPC over HTTP.
- Level 1 tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources
- Level 2 introduces a standard set of verbs so that we can handle similar situations in the same way, removing unnecessary variation
- Level 3 introduces discoverability, providing a way of making a protocol more self-documenting

In this book, we'll mostly focus on supporting REST at Level 2. So, we'll work with well-defined resources and use the appropriate HTTP verbs to indicate what we want to do with a resource.

In Chapter 7, *JSON, HATEOAS, and Documentation*, of this book, we'll address HATEOAS, which can help us reach maturity Level 3.

Now that we've got the theory out of the way, let's get the code, set up your favorite IDE, and define the API for the REST service we'll implement.

Getting the source code

There are a couple of different ways of getting the code for this book. We provide a download link at the book's website (https://www.packtpub.com/books/content/support/23321) from where you can download a ZIP file with the latest sources from GitHub (https://github.com/josdirksen/rest-with-scala/archive/master.zip), or even better, just use Git to clone the source repository.

Downloading the ZIP file

If you've downloaded the ZIP file, just unzip it to a directory of your choice:

```
Joss-MacBook-Pro:Downloads jos$ unzip rest-with-scala-master.zip
```

Using Git to clone the repository

Cloning the repository is also very easy if you've already got Git installed. If you haven't got Git installed, follow the instructions at https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

Once Git is installed, just run the following command:

```
Joss-MacBook-Pro:git jos$ git clone https://github.com/josdirksen/rest-with-scala
Cloning into 'rest-with-scala'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
Checking connectivity... done.
```

At this point, you've got the sources in a directory of your choice. Next, we need to make sure we can download all the frameworks' dependencies and run the samples. For this, we'll use SBT (more information can be found at http://www.scala-sbt.org/), which is the most common build tool for Scala-based projects.

Setting up Scala and SBT to run the examples

To run the examples provided in this book, we need to install Scala and SBT. Depending on your operating system, different steps need to be taken.

Installing Java

Before we can install SBT and Scala, we first need to install Java. Scala requires at least a Java Runtime version of 1.6 or higher. If you haven't installed Java on your system yet, follow the instructions at http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

Installing Scala and SBT

Once you have Java installed, installing Scala and SBT is just as easy. To install Scala, just go to http://www.scala-lang.org/download/ and download the binaries for your system. To install SBT, you can follow the instructions at http://www.scala-sbt.org/download.html.

To check whether everything is installed, run the following commands in a terminal:

```
Joss-MacBook-Pro:~ jos$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
Joss-MacBook-Pro:~ jos$ scala -version
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
Joss-MacBook-Pro:~ jos$ sbt -v
[process_args] java_version = '1.8.0_40'
# Executing command line:
java
-Xms1024m
-Xmx1024m
-XX:ReservedCodeCacheSize=128m
-XX:MaxMetaspaceSize=256m
-jar
/usr/local/Cellar/sbt/0.13.8/libexec/sbt-launch.jar

[info] Set current project to jos (in build file:/Users/jos/)
```

To exit SBT, hit *Ctrl* + *C*.

Running the examples

Now that you've got Java, Scala, and SBT installed, we can run the examples. You can, of course, run the examples from your IDE (see the next section on how to set up IntelliJ IDEA and Eclipse), but often, using SBT directly is just as easy. To run the examples, take the following steps:

1. Open a terminal and go to the directory where you've extracted the source ZIP file or cloned the repository.
2. To test the configuration, we've created a simple `HelloWorld` example. From the console, execute `sbt runCH01-HelloWorld`:

```
Joss-MacBook-Pro:rest-with-scala jos$ sbt runCH01-HelloWorld
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Compiling 2 Scala sources to /Users/jos/dev/git/rest-with-
scala/chapter-01/target/scala-2.11/classes...
[info] Running org.restwithscala.chapter1.HelloWorld
SBT successfully ran HelloWorld, configuration seems ok!
Press <enter> to exit.

[success] Total time: 18 s, completed Jun 13, 2015 2:34:41 PM
Joss-MacBook-Pro:rest-with-scala jos$
```

3. You might see a lot of output when the various dependencies are loaded, but after a while, you should see the message, `SBT successfully ran HelloWorld, configuration seems ok!`.
4. All the examples in this book wait for user input to terminate. So, once you're done playing around with the example, just hit *Enter* to terminate the running program.

In each chapter, we'll see the `sbt` command we need to execute. If you want to know all the examples you can run, you can also run the `sbt alias` command, which generates the following output:

```
Joss-MacBook-Pro:rest-with-scala jos$ sbt alias
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
        runCH01-HelloWorld = ; chapter01/runCH01HelloWorld
        runCH01-EchoServer = ; chapter01/runCH01EchoServer
```

Besides running the examples directly from the command line, it is also possible to run them from an IDE. In the following section, we'll see how to import the examples in IntelliJ IDEA and Eclipse.

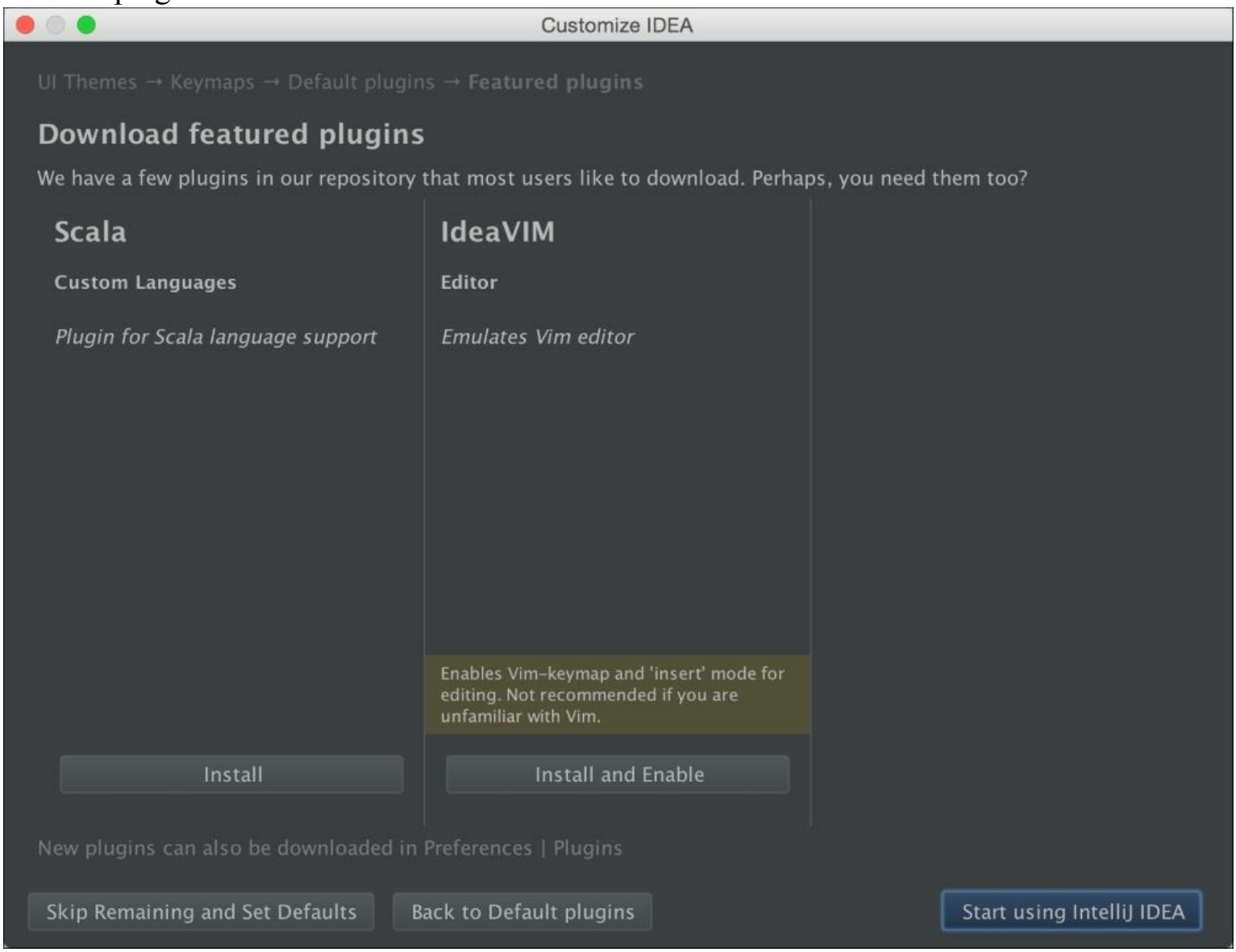
Setting up the IDE

With SBT and Scala installed, you have everything you need to run the examples. Sometimes, however, it is easier to play around and experiment with the examples directly from an IDE. The two most popular IDEs for working with Scala are IntelliJ IDEA and Eclipse. Both have great Scala plugins and excellent support for SBT.

Setting up IntelliJ IDEA

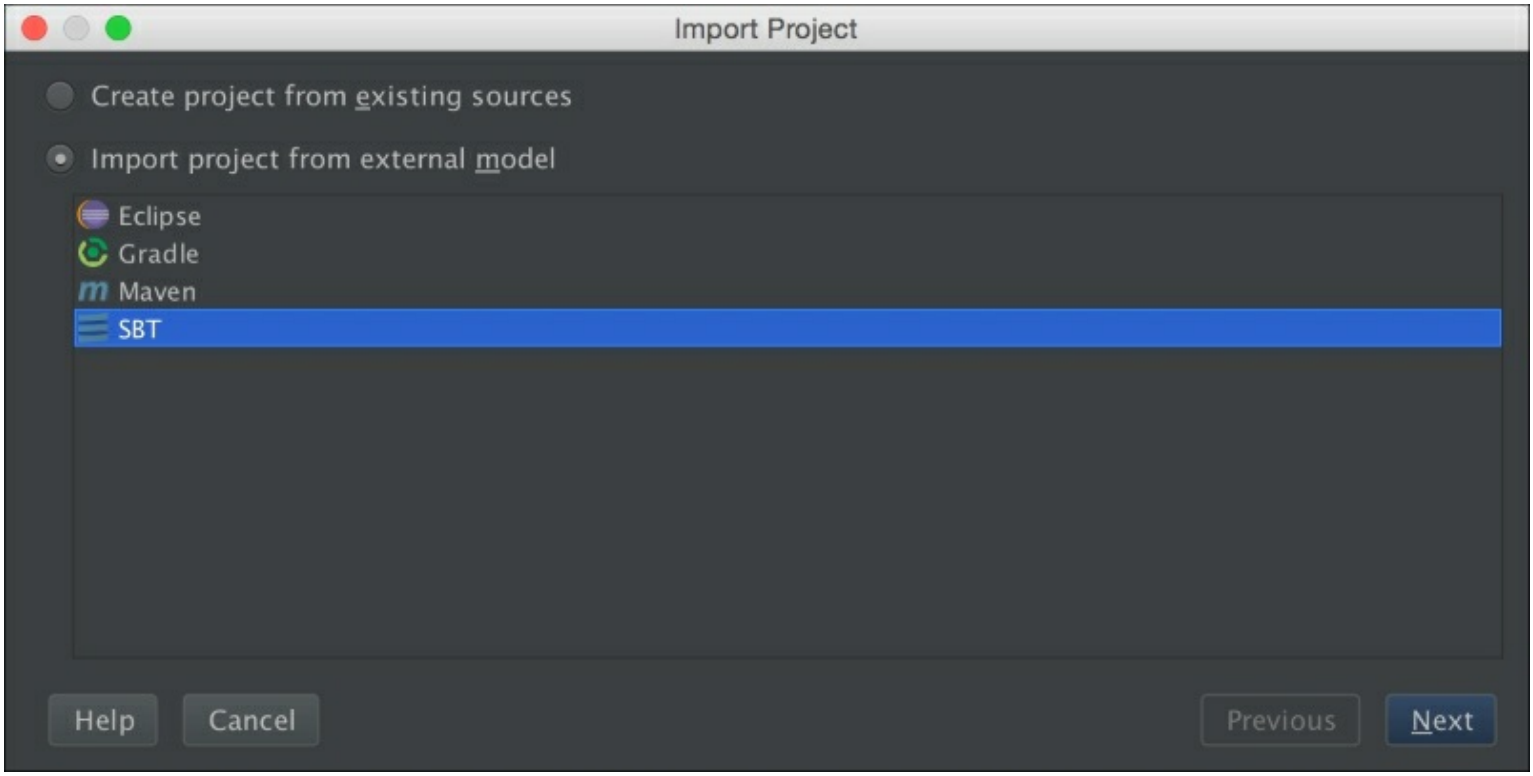
IntelliJ provides a community and a commercial version of its IDE, both of which can be used to run and play around with the examples in this book. The following steps are shown for the community edition, but can be applied in the same manner for the commercial variant:

1. The first thing to do is download the IDE. You can download a version for your OS from https://www.jetbrains.com/idea/download/. Once downloaded, run the installer and start the IDE. When you run IntelliJ for the first time, you're asked whether you want to install the featured plugins. Scala is one of them:



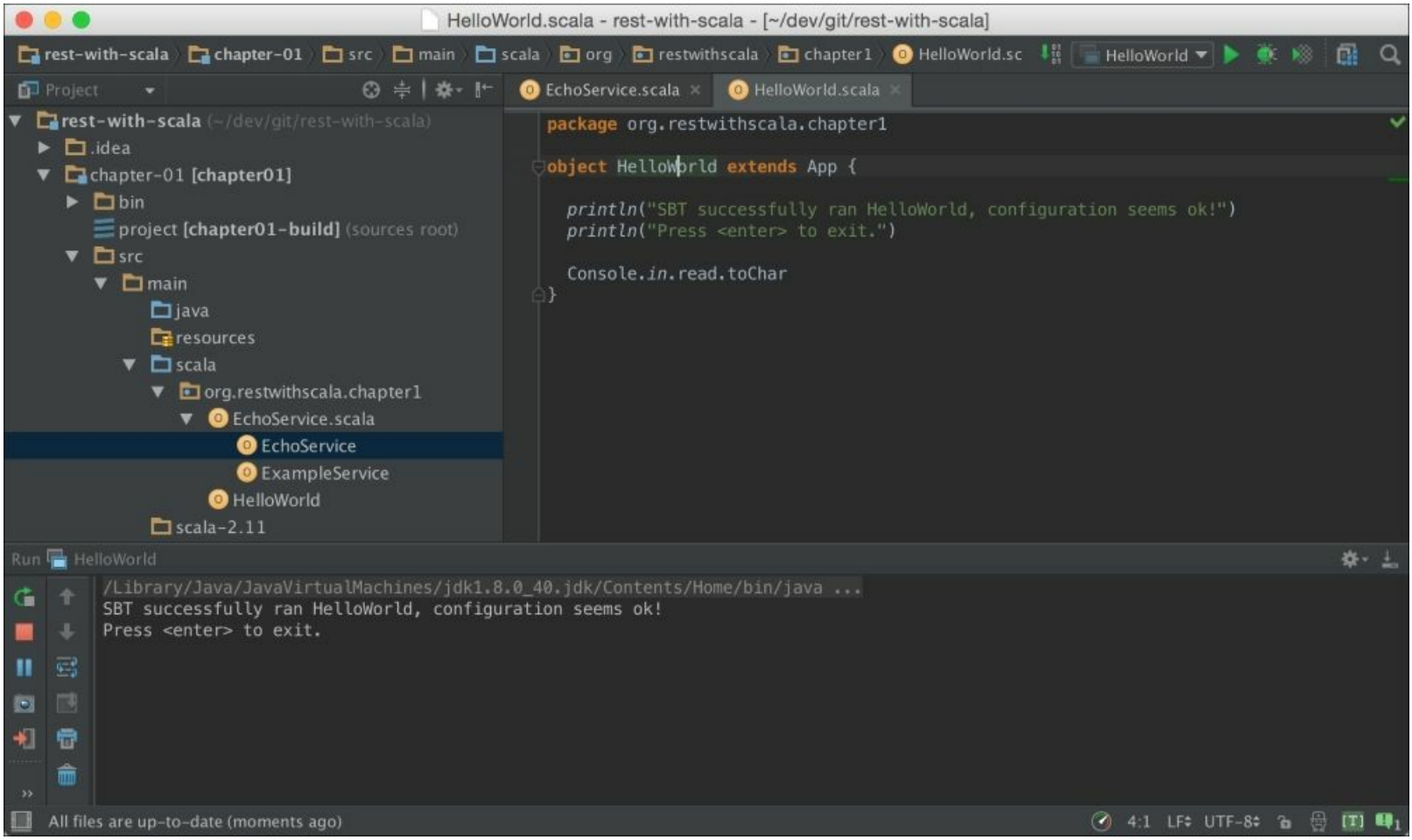
2. From here, click on **Install** below the **Scala** column to install Scala support in IntelliJ. After installing, click on **Start using IntelliJ IDEA**. After IntelliJ is started, you're shown a screen where you can import an existing project:

3. From here, select **Import Project**, and on the screen that opens, navigate to the directory where we extracted the downloaded sources, select that directory, and click on **OK**. On the screen that opens, select the **Import Project from external model** radio button and next, select **SBT**.



4. Now click on **Next** and fill in the screen that opens:
    1. Check the **Download sources and docs** checkbox.
    2. For **Project SDK**, click on **New**, select **JDK**, and navigate to the directory where you installed the JDK 1.8.
    3. And finally, click on **Finish**.

IntelliJ will now import all the projects and download all the required dependencies. Once done, you're shown a screen like this, where you see all the projects and can run the examples directly from the IDE:

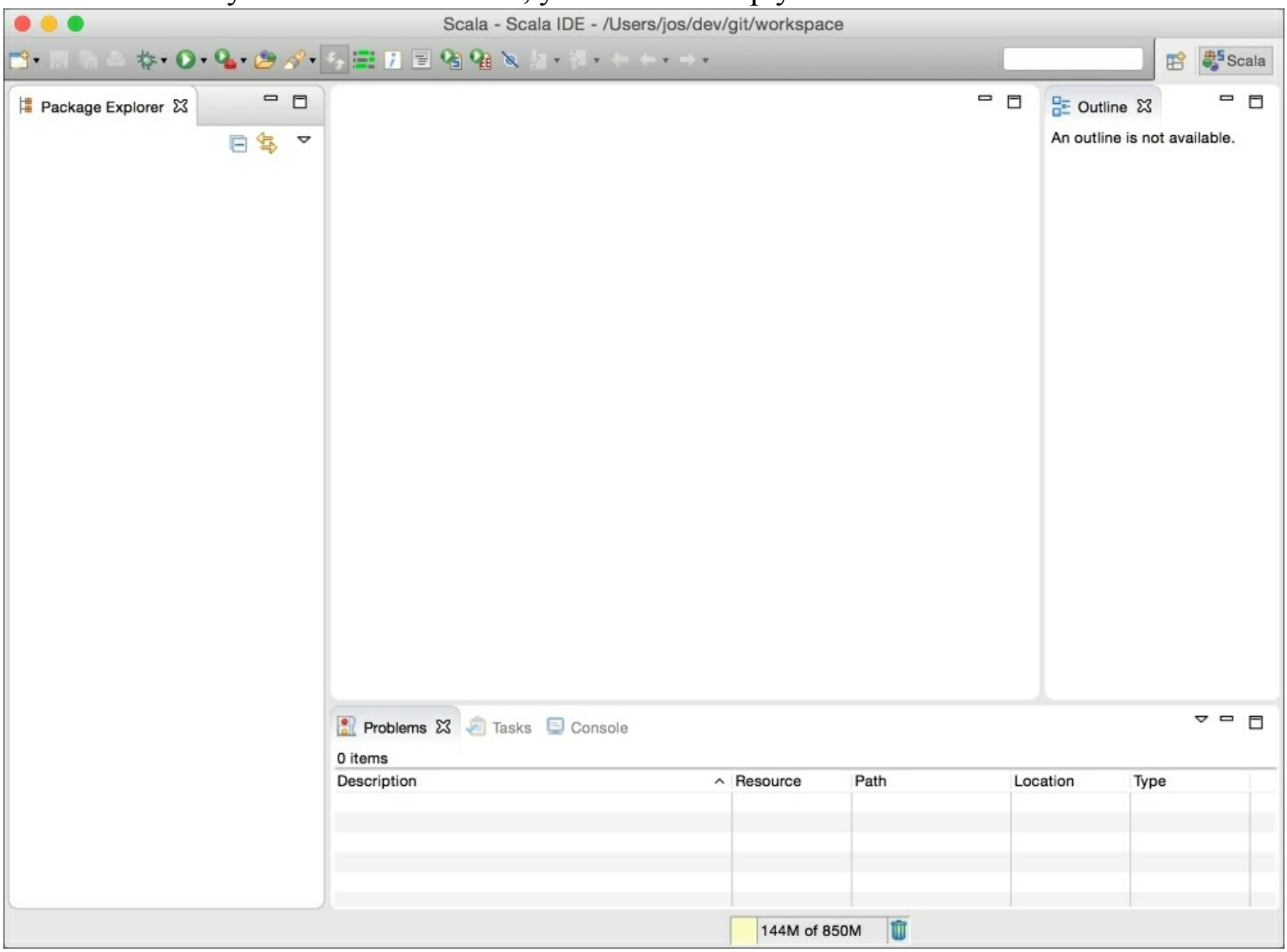


An alternative to using IntelliJ IDEA with great Scala support is Eclipse.

Setting up Eclipse

The Scala community provides a packaged version of Eclipse that contains everything you need for developing Scala.

1. To install this version of Eclipse, first download the version for your OS from their download site at http://scala-ide.org/download/sdk.html.
2. Once downloaded, extract the archive to a directory of your choice, start Eclipse, and select a location to store your file. Once started, you'll see an empty editor:

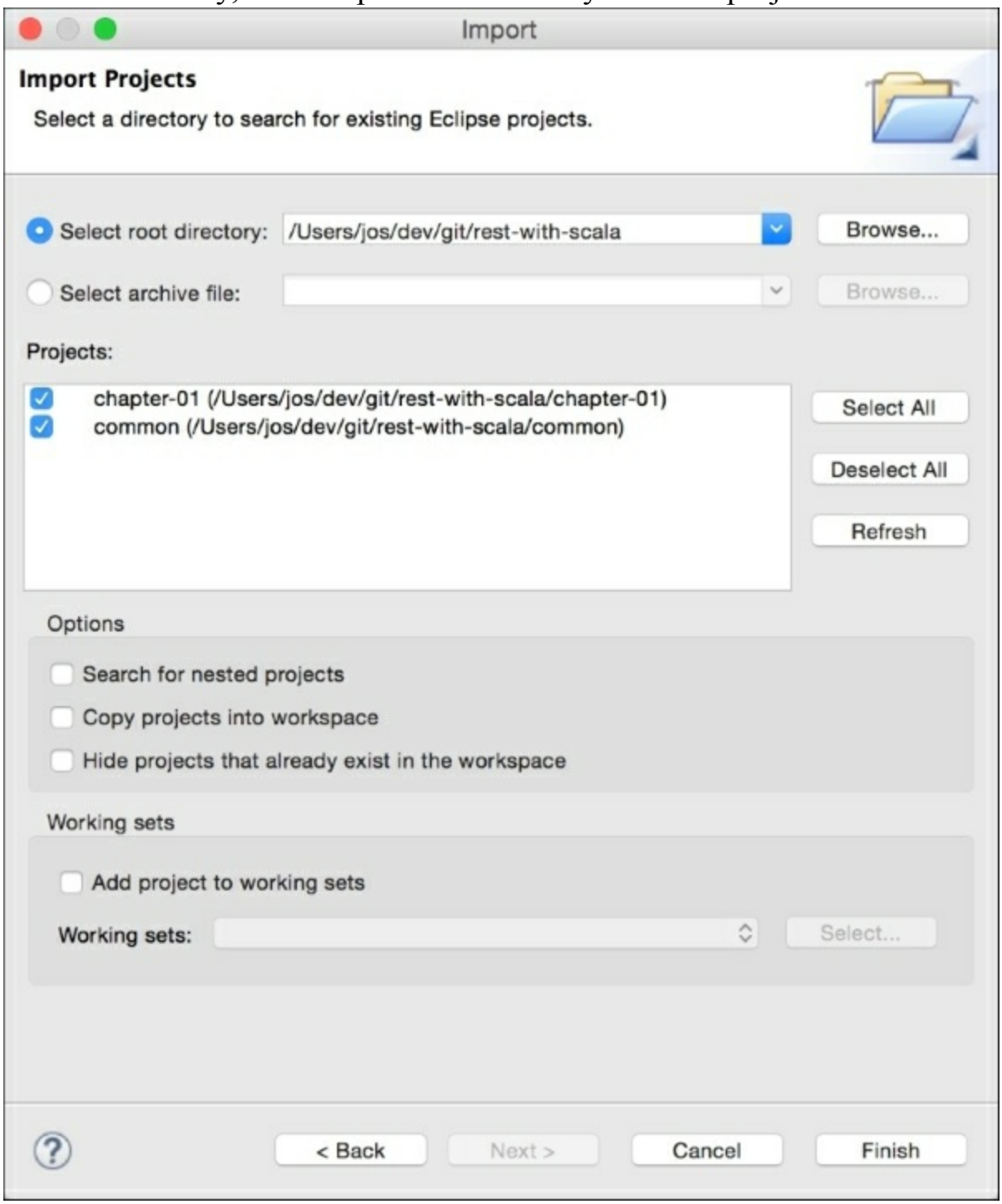


3. Before we can import the project, we must first create the required Eclipse project configuration. To do this, open a terminal and navigate to the directory where you extracted or cloned the sources. From that directory, run `sbt eclipse`:

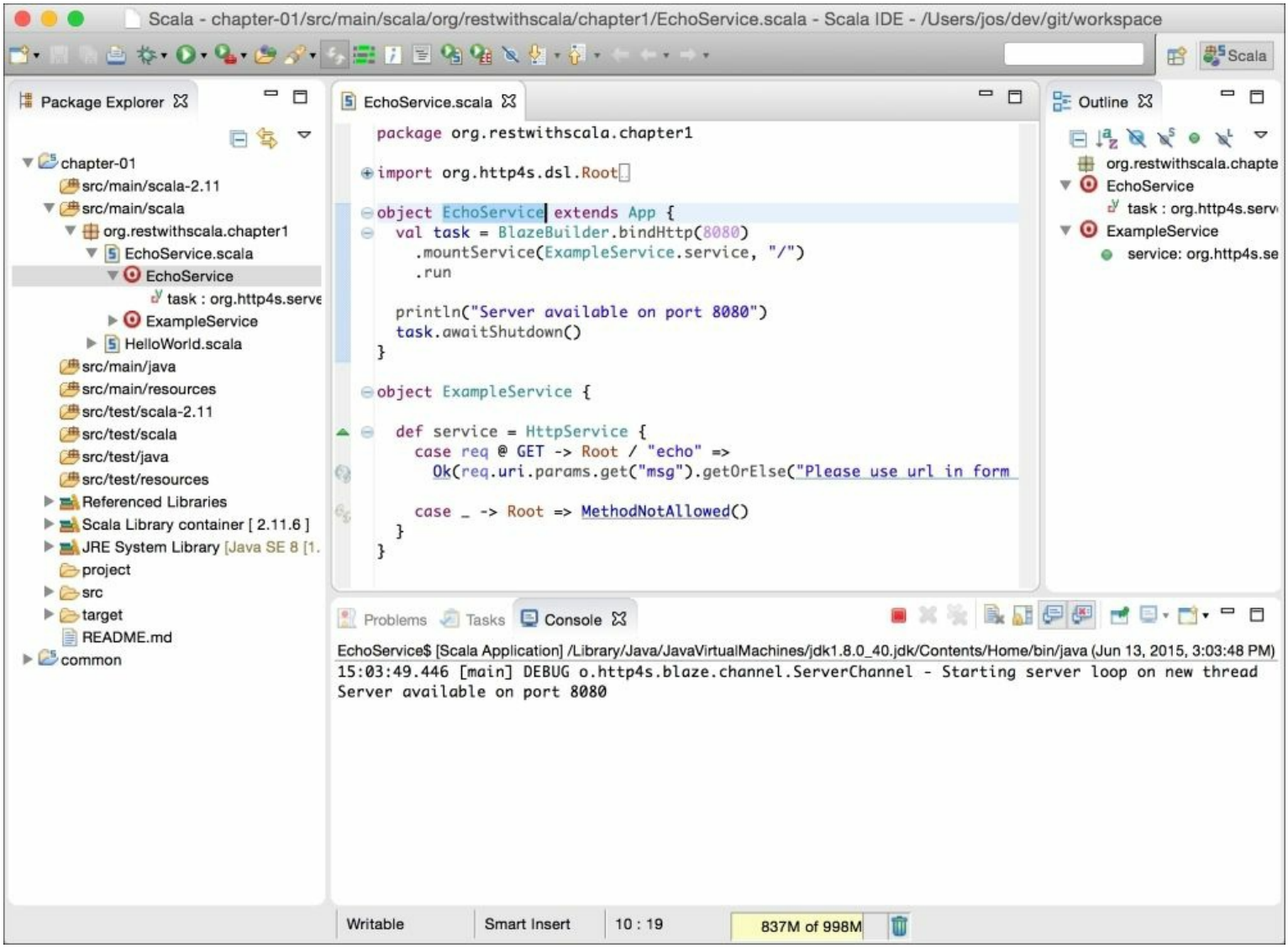
```
Joss-MacBook-Pro:rest-with-scala jos$ sbt eclipse
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Updating {file:/Users/jos/dev/git/rest-with-scala/project/}rest-with-
scala-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
```

```
[info] Done updating.
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] chapter-01
```

4. Now we can import the project. Go to **File** | **Import** from the menu and choose to import **Existing projects into Workspace**. On the next screen, select the directory with the sources as the root directory, and Eclipse should show you all the projects:



5. Now click on **Finish**, and the projects will be imported. Now you can edit and run the samples directly from Eclipse.


Scala - chapter-01/src/main/scala/org/restwithscala/chapter1/EchoService.scala - Scala IDE - /Users/jos/dev/git/workspace
Package Explorer
chapter-01
src/main/scala-2.11
src/main/scala
org.restwithscala.chapter1
EchoService.scala
EchoService
task : org.http4s.serve
ExampleService
HelloWorld.scala
src/main/java
src/main/resources
src/test/scala-2.11
src/test/scala
src/test/java
src/test/resources
Referenced Libraries
Scala Library container [2.11.6]
JRE System Library [Java SE 8 [1.
project
src
target
README.md
common
EchoService.scala
package org.restwithscala.chapter1
import org.http4s.dsl.Root
object EchoService extends App {
val task = BlazeBuilder.bindHttp(8080)
.mountService(ExampleService.service, "/")
.run
println("Server available on port 8080")
task.awaitShutdown()
}
object ExampleService {
def service = HttpService {
case req @ GET -> Root / "echo" =>
Ok(req.uri.params.get("msg").getOrElse("Please use url in form
case _ -> Root => MethodNotAllowed()
}
}
Outline
org.restwithscala.chapte
EchoService
task : org.http4s.serv
ExampleService
service: org.http4s.se
Problems
Tasks
Console
EchoService$ [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java (Jun 13, 2015, 3:03:48 PM)
15:03:49.446 [main] DEBUG o.http4s.blaze.channel.ServerChannel - Starting server loop on new thread
Server available on port 8080
Writable
Smart Insert
10 : 19
837M of 998M

Testing the REST API

Before we look at the API of the service we're going to create, we'll have a quick look at how to test your REST API. We can, of course, create a REST client in Scala and use that, but since a big advantage of REST services is that they can be read and understood by humans, we'll use a simple browser-based (Chrome in this case) REST client called **Postman**. Note that you can, of course, also use different REST clients. The reason we chose Postman is that with Postman, it is easy to create different kinds of request; it has HATEOAS support and also allows us to share requests, so you don't have to make them by hand.

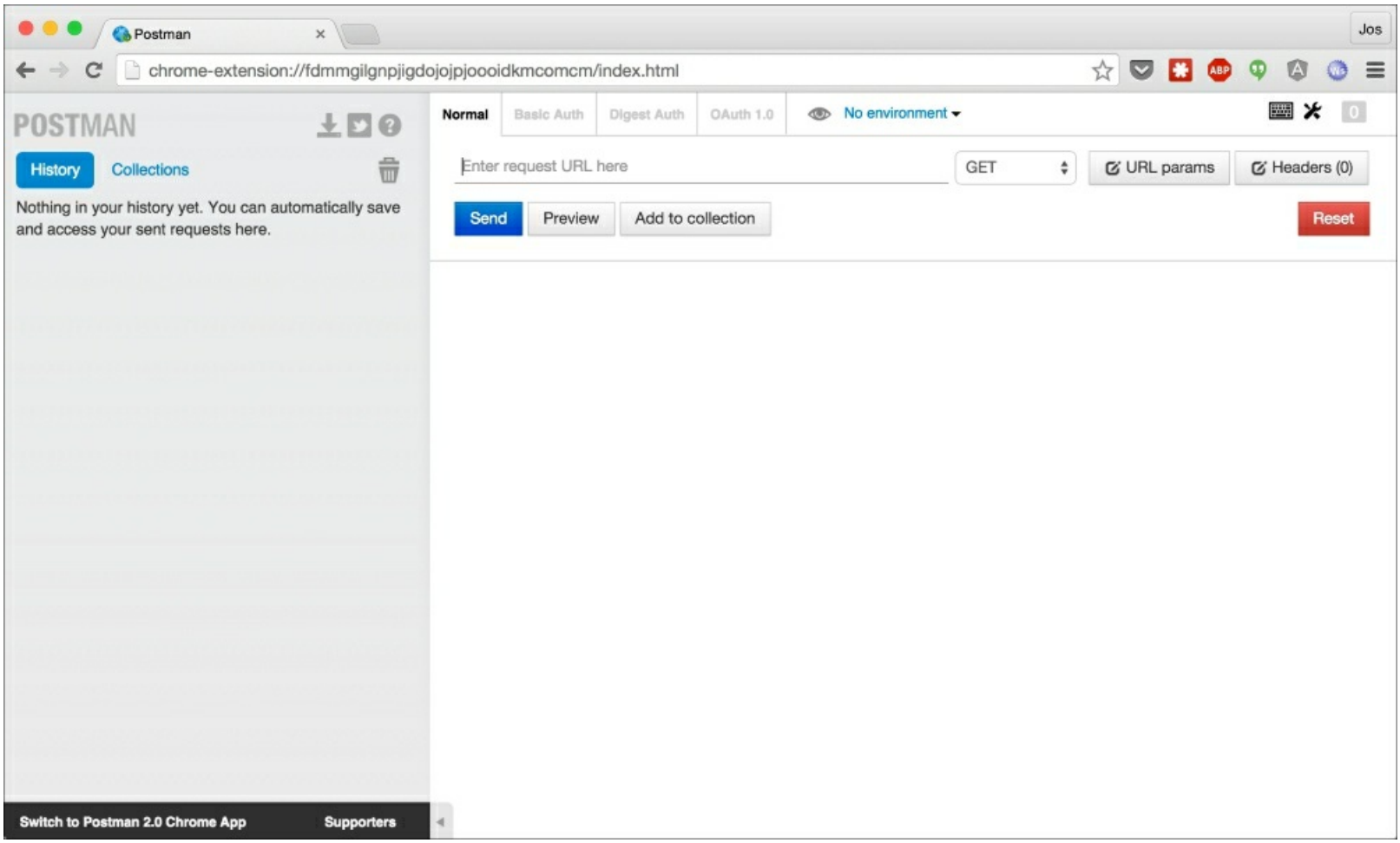
Installing Postman

Postman runs as a Chrome plugin, so to use this REST client, you need to use Chrome. Once you've started Chrome, open the URL, https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm?hl=en in your browser (or just search on Google for Chrome Postman).

Tip

Note that there are two versions of Postman: a simple Chrome plugin, which, at the time of writing, is version 0.8.4.14, and a Chrome app, which currently is at version 3.0.0.6. For this book, we'll use the simpler Chrome plugin, so make sure you install 0.8.4.14 in your browser.

Once installed, open up this app by going to `chrome://apps` and selecting the application or clicking on the newly added button at the top-right of your screen. When you open this plugin, you should see the following window:



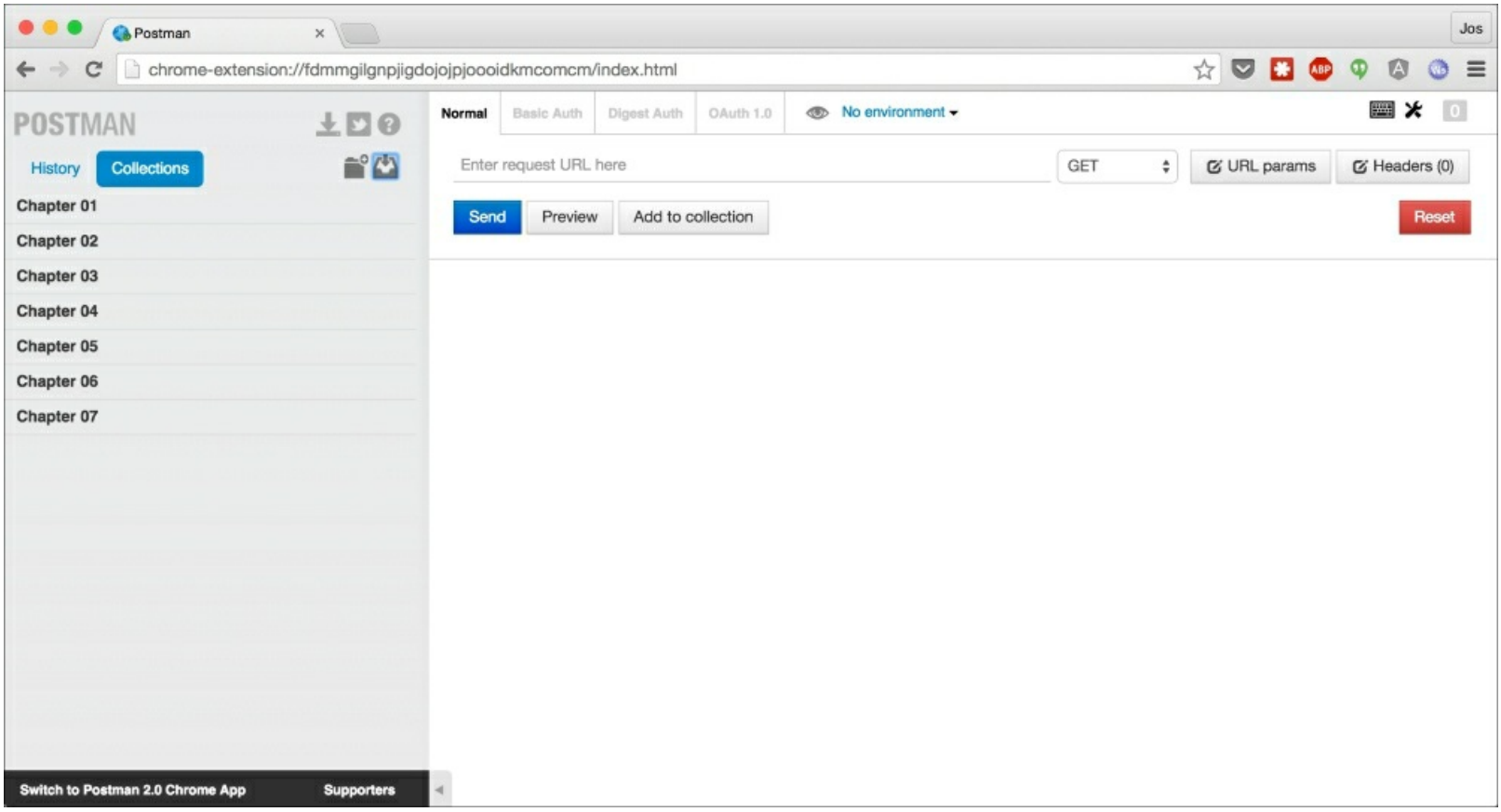
One interesting aspect of Postman is that you can very easily share REST queries. In the sources for this chapter, you can find a directory called `common`.

Importing request collection

In this directory, there are a number of files which contain the requests for each individual chapter. For instance, for this chapter, these are contained in the file, `ch01_requests.json`. Each file contains a number of requests that you can use to test the REST frameworks in this book. To import all these requests, take the following steps:

1. On the left-hand side of the screen, click on the **Collections** tab.
2. To the right of this tab, two icons pop up. Click on the right icon, which is called **import collection**.
3. On the screen that pops up, click on choose files, navigate to the `common` directory, and select all the `ch0#_requests.json` files and open them.

Now, you'll have a number of collections, one for each chapter, where you can find sample requests for the different chapters.



To run a request, just click on a collection. This will show all the requests for that chapter. Click on a request. Now clicking on the **Send** button will send the request to the server. In the upcoming chapters, we will see which request you can use to test the functionality of a specific Scala REST framework.

Testing the REST service

At this point, we've got Scala and SBT installed, and can use Postman as the REST client. The final step is to see whether everything is working correctly. For this, we'll start a very simple HTTP service, which echoes back a specific request parameter.

Tip

For this example, we've used HTTP4S. This is a basic HTTP server that allows you to quickly create HTTP services. If you're interested, you can find more information about this library at http://http4s.org/. The source code for our simple echo service can be found in the `chapter-01/src/main/scala` directory.

To run this example, we need to take a couple of steps:

1. First, open a console window and go to the directory where you downloaded and extracted the sources or cloned the Git repository.
2. From that directory, run the `sbt runCH01-EchoServer` command. This will start up the echo service:

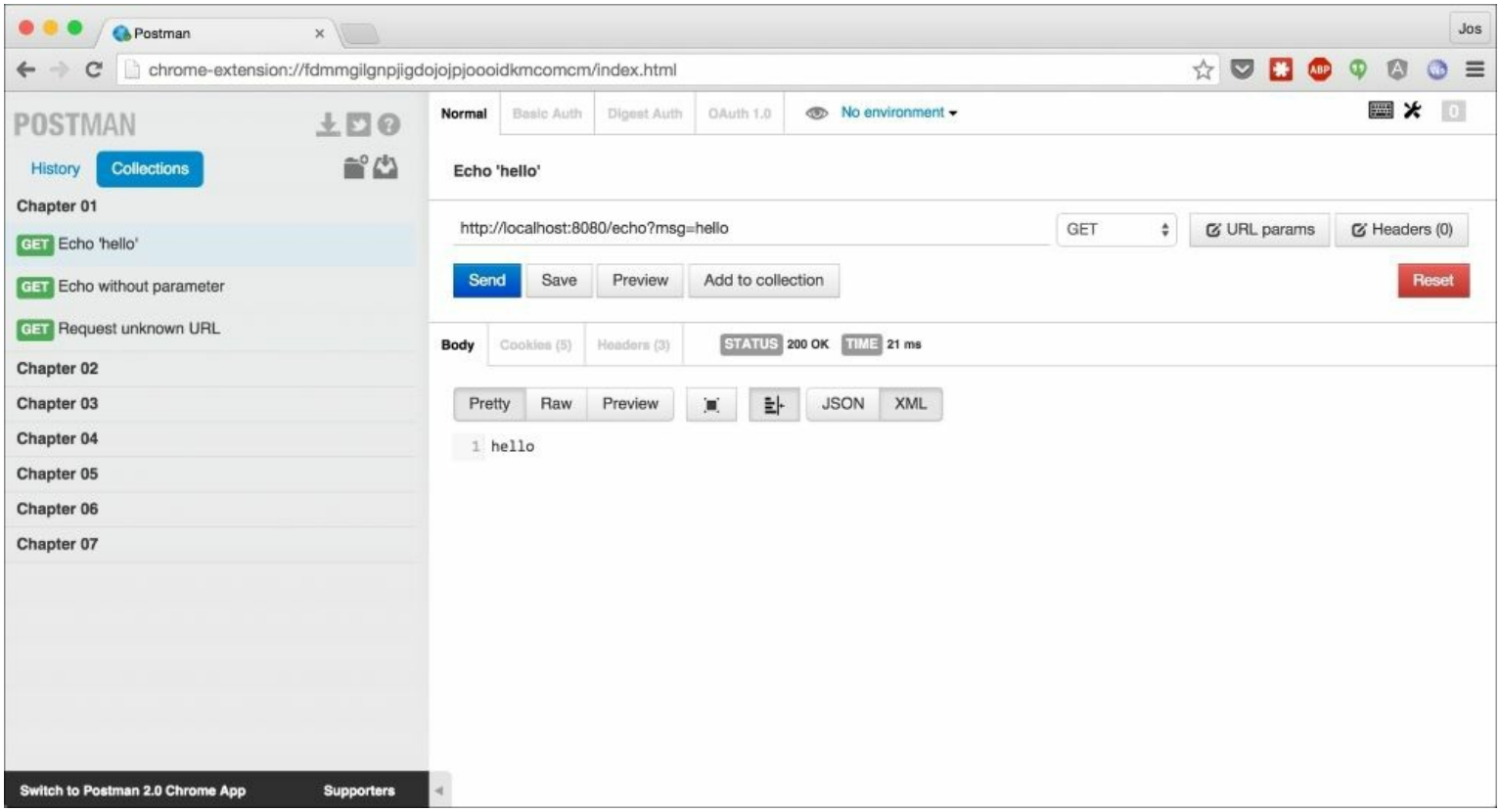
```
Joss-MacBook-Pro:rest-with-scala jos$ sbt runCH01-EchoServer
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter1.EchoService
11:31:24.624 [run-main-0] DEBUG o.http4s.blaze.channel.ServerChannel -
Starting server loop on new thread
Server available on port 8080
```

Tip

Downloading the example code

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

3. Now that we've got a server running, open up your Chrome browser, and from there, open Postman (remember, either use the added button or the `chrome://apps` URL). In the list of collections, click on the request labeled **Echo 'hello'**. This will open the request. You can now run this request by clicking on the **Send** button. The result, if everything is configured correctly, will be something like this:



4. By changing the value of the `msg request` parameter, you can test that the server is really echoing the user's input.
5. There are a couple of other requests in this collection that show some features of our current server. You can check what happens when the request parameter is omitted and when a call is made to an unknown URL.

At this point, the only thing left to do in this chapter is to look at the API we'll create using the different frameworks outlined in the upcoming chapters.

The REST service and model

To show the features of the various frameworks in this book and how they solve problems in a different manner, we'll define a simple API, which we'll implement with the REST frameworks. For this book, we'll implement a simple to-do list API.

API description

We want to create a RESTful API, so the most important part is to start with the description of the resources that can be managed through this API. For this API, we define the following resources:

Entity	Description

Task	A task is something that needs to be done. The JSON for a task looks similar to this:
	
	`{`
	`  "id": long,`
	`  "title": string,`
	`  "content": string,`
	`  "notes": [noteList],`
	`  "status": Status,`
	`  "assignedTo": Person,`
	`}`

Project	A project will allow us to group tasks together, and by assigning persons to a project, we can determine who can work on a specific task:
	
	`{`
	`  "id": string,`
	`  "title": string,`
	`  "tasks": [task],`
	`  "members": [person],`
	`  "updated": datetime`
	`}`

Person	A person is someone who can work on a task and when done, close the task. A person can only work on those tasks to which he is assigned, or when he is part of the project to which a task belongs:
	
	`{`
	`  "id": string,`
	`  "name": string`
	`}`

Note	Notes can be added to tasks to provide additional information on how the task should be performed:
	
	`{`
	`  "id": string,`
	`  "text": string,`
	`}`

Without going into too much detail here, we want to support approximately the following functionality in our API:

- **CRUD functionality**: We want to support some basic CRUD operations. It should be possible to perform the following actions:
  - Create, update, and delete a new task, project, person, and note.
  - Get a list of tasks, projects, persons, and notes.
  - Search through a list of tasks.
  - Add a note to a task. It should also be possible to update and delete the existing notes.
- **Advanced functions**: Besides the standard CRUD-like functionality, we also want to provide some more advanced features:

- Assign a task to a specific project
- Assign a person to a task
- Assign a person to a project
- Move a task from one project to another

Note that we won't implement all the functionality for each framework. We'll mainly use this API to explain how we can use the various REST frameworks.

Summary

That's it for the first chapter. In this chapter, we introduced what REST is, and which level of the Robertsons Maturity Model we'll be aiming at (Level 2). We'll explain HATEOAS in the final chapter. At this point, you should have Scala and SBT installed, and should be able to run all the examples in this book using SBT and test them using the supplied requests in Postman. We also saw how to use IDEs to play around with the code. And finally, we introduced the high-level API that we'll implement in the upcoming chapters.

In the next chapter, we'll dive into the first of the Scala frameworks we'll explore. The first one is Finch, which is a REST library on top of the networking library, Finagle. Both of these were initially created by Twitter.

Chapter 2. A Functional-style REST Service with Finagle and Finch

In this chapter, we're going to show you how you can create a REST service using the Finagle and Finch library. We'll do this using the following set of examples:

- **Your first Finagle and Finch service**: In this section, we'll create a minimal REST service, which will simply return a string.
- **HTTP verb and URL matching**: An important part of any REST service is how to handle various URL paths and the different HTTP verbs. In this part, we'll show you how Finch supports this through the use of matchers and extractors.
- **Use RequestReaders to process incoming requests**: When creating a REST service, you usually need to get information from the incoming HTTP request. Finch uses `RequestReader` instances to access information from the request, which we'll explain in this part.
- **JSON support**: REST services most often use JSON to represent resources. Finch supports a number of different JSON libraries. In this part, we'll explore one of these JSON libraries and how to use it from a Finch service.
- **Request validation and custom responses**: The final part of this chapter deals with validating incoming requests and creating custom responses. Finch has a very elegant way, using `RequestReader` instances and validation rules, to check whether incoming requests are valid and can be processed further.

Before we start looking at the code, let's quickly look at what Finagle and Finch are for libraries.

An introduction to Finagle and Finch

Finagle and Finch are actually two different frameworks. Finagle is an RPC framework, created by Twitter, which you can use to easily create different types of service. On its website (https://github.com/twitter/finagle), the team behind Finagle explains it like this:

> *"Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers. Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency. Most of Finagle's code is protocol agnostic, simplifying the implementation of new protocols."*

So, while Finagle provides the plumbing required to create highly scalable services, it doesn't provide direct support for specific protocols. This is where Finch comes in.

Finch (https://github.com/finagle/finch) provides an HTTP REST layer on top of Finagle. On their website, you can find a nice quote which summarizes what Finch aims to do:

> *"Finch is a thin layer of purely functional basic blocks atop of http://twitter.github.io/finagle for building composable REST APIs. Its mission is to provide the developers simple and robust REST API primitives being as close as possible to the bare metal Finagle API."*

In this chapter, we'll only be talking about Finch. Note, though, that most of the concepts provided by Finch are based on underlying Finagle ideas. Finch provides a very nice REST-based set of functions to make working with Finagle very easy and intuitive.

Building your first Finagle and Finch REST service

Let's start by building a minimal Finch REST service. The first thing we need to do is make sure we have the correct dependencies. Like we mentioned in the previous chapter, we use SBT to manage our dependencies. All the dependencies for the various chapters can be found in the `Dependencies.scala` file, which is located in the `project` directory in the location where you extracted your sources. For the Finch examples, which we will see in this chapter, we use the following dependencies:

```
  lazy val finchVersion = "0.7.0"

  val backendDeps = Seq(
    "com.github.finagle" %% "finch-core" % finchVersion
  )
```

This book uses a single SBT file (`build.sbt` located in the root) for all the chapters and uses a multimodule approach. Diving into the multimodule setup is a bit beyond the scope of this book. If you want to learn more about how we use SBT to manage and define the various modules, look at the `build.sbt` file.

Now that we've got our library dependencies loaded, we can start coding our very first Finch service. The next code fragment (the source can be found at `chapter-02/src/main/scala/org/restwithscala/chapter2/gettingstarted/HelloFinch.scala`) shows a minimal Finch service, which just responds with a `Hello, Finch!` message:

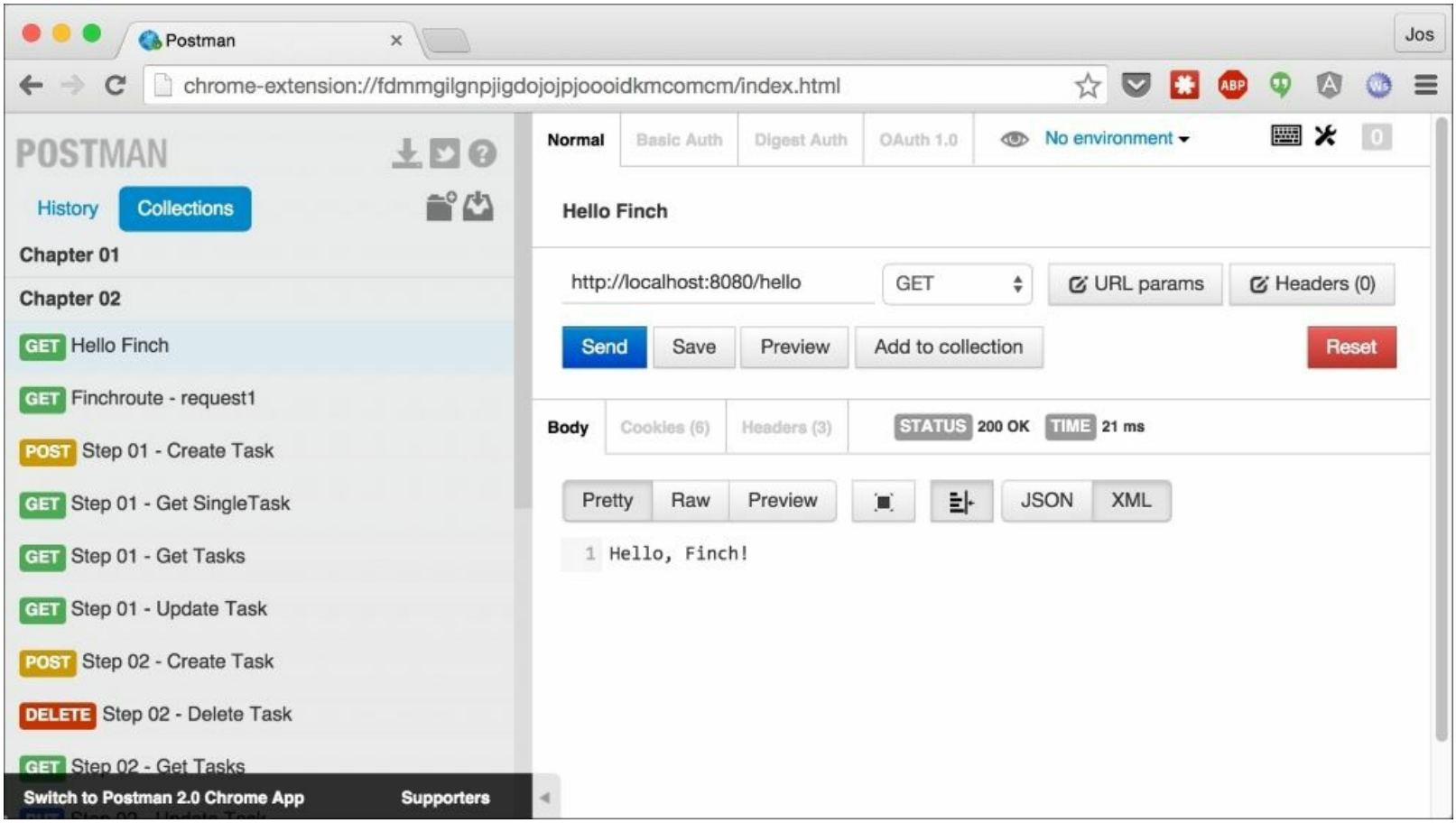
```
package org.restwithscala.chapter2.gettingstarted

import io.finch.route._
import com.twitter.finagle.Httpx

object HelloFinch extends App {

  Httpx.serve(":8080", (Get / "hello" /> "Hello, Finch!").toService)

  println("Press <enter> to exit.")
  Console.in.read.toChar
}
```

When this service receives a `GET` request on the URL path, `hello`, it will respond with a `Hello, Finch!` message. Finch does this by creating a service (using the `toService` function) from a route (more on routes is explained in the next section) and using the `Httpx.serve` function to host the created service. To run this example, open a terminal window in the directory where you've extracted the sources. In that directory, run the `sbt runCH02-HelloFinch` command:

```
$ sbt runCH02-HelloFinch
[info] Loading project definition from /Users/jos/dev/git/rest-with-
```

```
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter2.gettingstarted.HelloFinch
Jun 26, 2015 9:38:00 AM com.twitter.finagle.Init$$anonfun$1 apply$mcV$sp
INFO: Finagle version 6.25.0 (rev=78909170b7cc97044481274e297805d770465110)
built at 20150423-135046
Press <enter> to exit.
```

At this point, we have an HTTP server running on port `8080`. When we make a call to `http://localhost:8080/hello`, this server will respond with the `Hello, Finch!` message. To test this service, we've provided an HTTP request in Postman (see the previous chapter on how to install Postman and load requests). You can use the `GET Hello Finch` request to test the Finch service we just created:

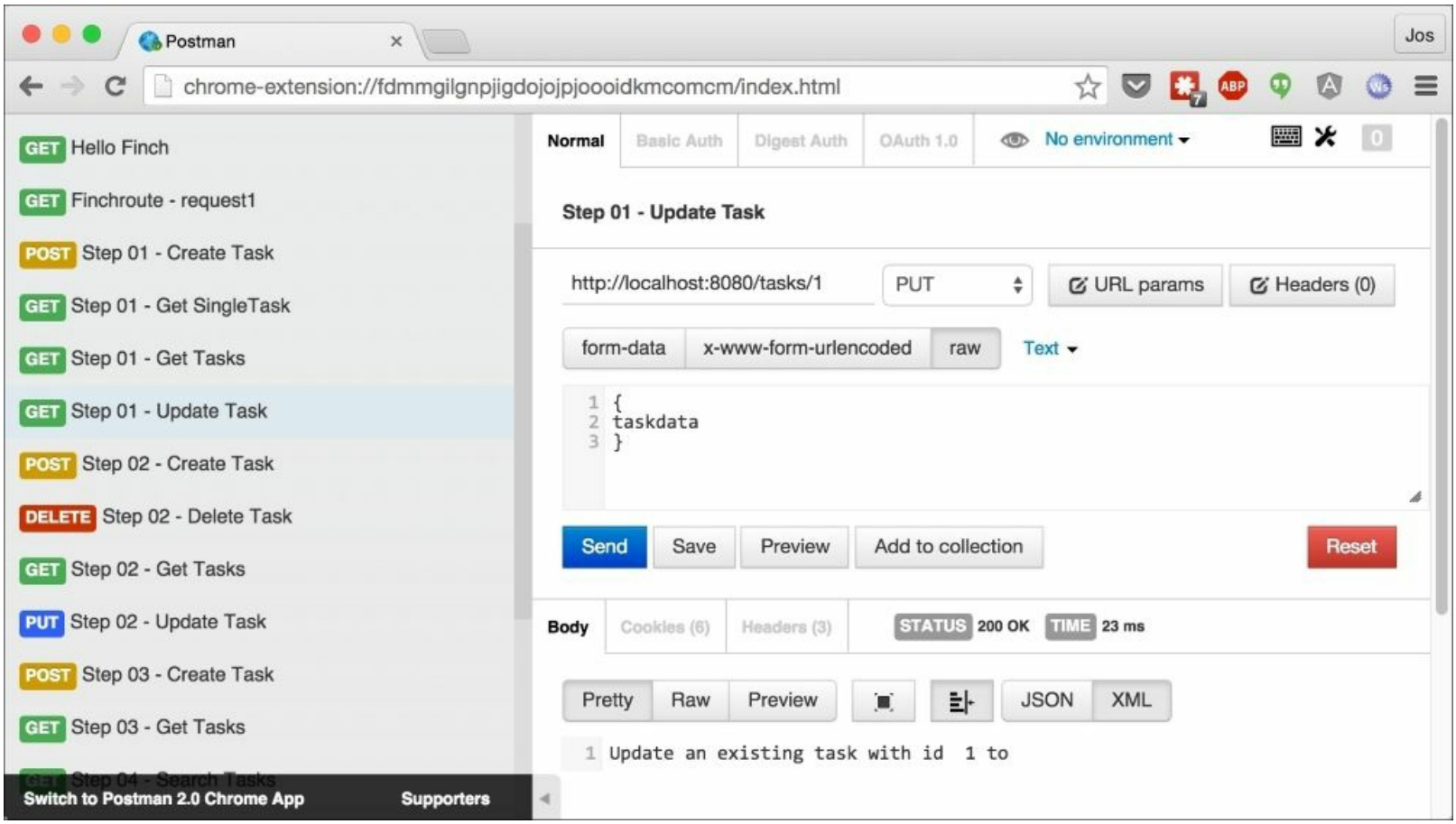


HTTP verb and URL matching

An important part of every REST framework is the ability to easily match HTTP verbs and the various path segments of the URL. In this section, we'll look at the tools Finch provide us with. Let's start with getting the service up and running though. To run this service, you can use the `sbt runCH02-runCH02Step1` command from the source directory:

```
$ sbt runCH02-runCH02Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter2.steps.FinchStep1
Jun 26, 2015 10:19:11 AM com.twitter.finagle.Init$$anonfun$1 apply$mcV$sp
INFO: Finagle version 6.25.0 (rev=78909170b7cc97044481274e297805d770465110)
built at 20150423-135046
Press <enter> to exit.
```

Once the server is started, you can once again use Postman to make requests to this service, using the requests from the **Chapter 02** collection. This service just returns a simple text message on each request:



Now let's look at some code and see how to do this with Finch (`chapter-`

02/src/main/scala/org/restwithscala/chapter2/steps/FinchStep1.scala):

```
package org.restwithscala.chapter2.steps

import com.twitter.finagle.Httpx
import io.finch.request._
import io.finch.route._
import io.finch.{Endpoint => _}

object FinchStep1 extends App {

  // handle a single post using a RequestReader
  val taskCreateAPI = Post / "tasks" /> (
    for {
      bodyContent <- body
    } yield s"created task with: $bodyContent")

  // Use matchers and extractors to determine which route to call
  // For more examples, see the source file.
  val taskAPI = Get / "tasks" />
            "Get a list of all the tasks" | Get / "tasks" / long />
            ( id => s"Get a single task with id: $id" ) | Put / "tasks" / long
/>
            ( id => s"Update an existing task with id  $id to " ) | Delete /
"tasks" / long />
            ( id => s"Delete an existing task with $id" )


  // a simple server that combines the two routes
  val server = Httpx.serve(":8080",
            (taskAPI :+: taskCreateAPI).toService )

  println("Press <enter> to exit.")
  Console.in.read.toChar

  server.close()
}
```

In this code fragment, we create a number of `Router` instances that process the requests which we sent from Postman. Let's start by looking at one of the routes of the `taskAPI` router, `Get / "tasks" / long /> (id => s"Get a single task with id: $id")`. The following table explains the various parts of the route:

Part	Description
`Get`	When writing routers, usually the first thing you do is determine which HTTP verb you want to match. In this case, this route will only match the `GET` verb. Besides the `Get` matcher, Finch also provides other matchers such as `Post`, `Patch`, `Delete`, `Head`, `Options`, `Put`, `Connect`, and `Trace`.
`"tasks"`	The next part of the route is a matcher that matches a URL path segment. In this case, we match the URL, `http://localhost:8080/tasks`. Finch will use an implicit conversion to convert this string object to a finch `Matcher` object. Finch also has two wildcard matchers: `*` and `**`. The `*` matcher allows any value for a single path segment, and the `**`

	matcher allows any value for multiple path segments.
`long`	The next part in the route is called an **extractor**. With an extractor, you turn part of the URL into a value which you can use to create the response (for example, retrieve an object from the database using the extracted ID). The `long` extractor, as the name implies, converts the matching path segment to a long value. Finch also provides an int, string, and boolean extractor.
`long => B`	The last part of the route is used to create the response message. Finch provides different ways of creating the response, which we'll show in the other parts of this chapter. In this case, we need to provide Finch with a function that transforms the long value we extracted, and returns a value Finch can convert to a response (you will learn more on this later). In this example, we just return a string.

If you've looked closely at the source code, you probably have noticed that Finch uses custom operators to combine the various parts of a route. Let's look a bit closer at these. With Finch, we get the following operators (also called **combinators** in Finch terms):

- `/` or `andThen`: With this combinator, you sequentially combine various matchers and extractors. Whenever the first part matches, the next one is called, for instance, `Get / "path" / long`.
- `|` or `orElse`: This combinator allows you to combine two routers (or parts thereof) as long as they are of the same type. So, we could do `(Get | Post)` to create a matcher, which matches the `GET` and `POST` HTTP verbs. In the code sample, we've also used this to combine all the routes that returned a simple string to the `taskAPI` router.
- `/>` or `map`: With this combinator, we pass the request and any extracted values from the path to a function for further processing. The result of the function that is called is returned as the HTTP response. As you'll see in the rest of the chapter, there are different ways of processing the HTTP request and creating a response.
- `:+:`: The final combinator allows you to combine two routers together of different types. In the example, we have two routers: `taskAPI`, which returns a simple string, and `taskCreateAPI`, which uses a `RequestReader` object (through the `body` function), to create the response. We can't combine these with `|` since the result is created using two different approaches, so we use the `:+:` combinator.

So far, we just return simple strings whenever we get a request. In the next section, we'll look at how you can use a `RequestReader` instance to convert the incoming HTTP requests to case classes and use those to create an HTTP response.

Processing incoming requests using RequestReaders

So far, we haven't done anything yet with the incoming request. In the previous example, we just returned a string without using any information from the request. Finch provides a very nice model using a **Reader monad**, which you can use to easily combine information from an incoming request to instantiate new objects.

Note

A Reader monad is a standard functional design pattern, which allows you to define functions that all access the same values. A great explanation of how Reader monads work can be found at http://eed3si9n.com/learning-scalaz/Monad+transformers.html.

Let's look at some code that uses a `RequestReader` to process incoming requests (the complete source code can be found in the `FinchStep2.scala` file):

```
object FinchStep2 extends App {

  val matchTask: Matcher = "tasks"
  val matchTaskId = matchTask / long

  // handle a single post using a RequestReader
  val taskCreateAPI =
        Get / matchTask /> GetAllTasks() :+:
       Post / matchTask /> CreateNewTask() :+:
     Delete / matchTaskId /> DeleteTask :+:
        Get / matchTaskId /> GetTask :+:
        Put / matchTaskId /> UpdateTask

  val taskAPI = ...


  val server = Httpx.serve(":8080",
                           (taskAPI :+: taskCreateAPI).toService )

  println("Press <enter> to exit.")
  Console.in.read.toChar

  server.close()

  sealed trait BaseTask {

    def getRequestToTaskReader(id: Long): RequestReader[Task] = {
      ( RequestReader.value(id) ::
        param("title") ::
        body ::
        RequestReader.value(None:Option[Person]) ::
        RequestReader.value(List.empty[Note]) ::
```

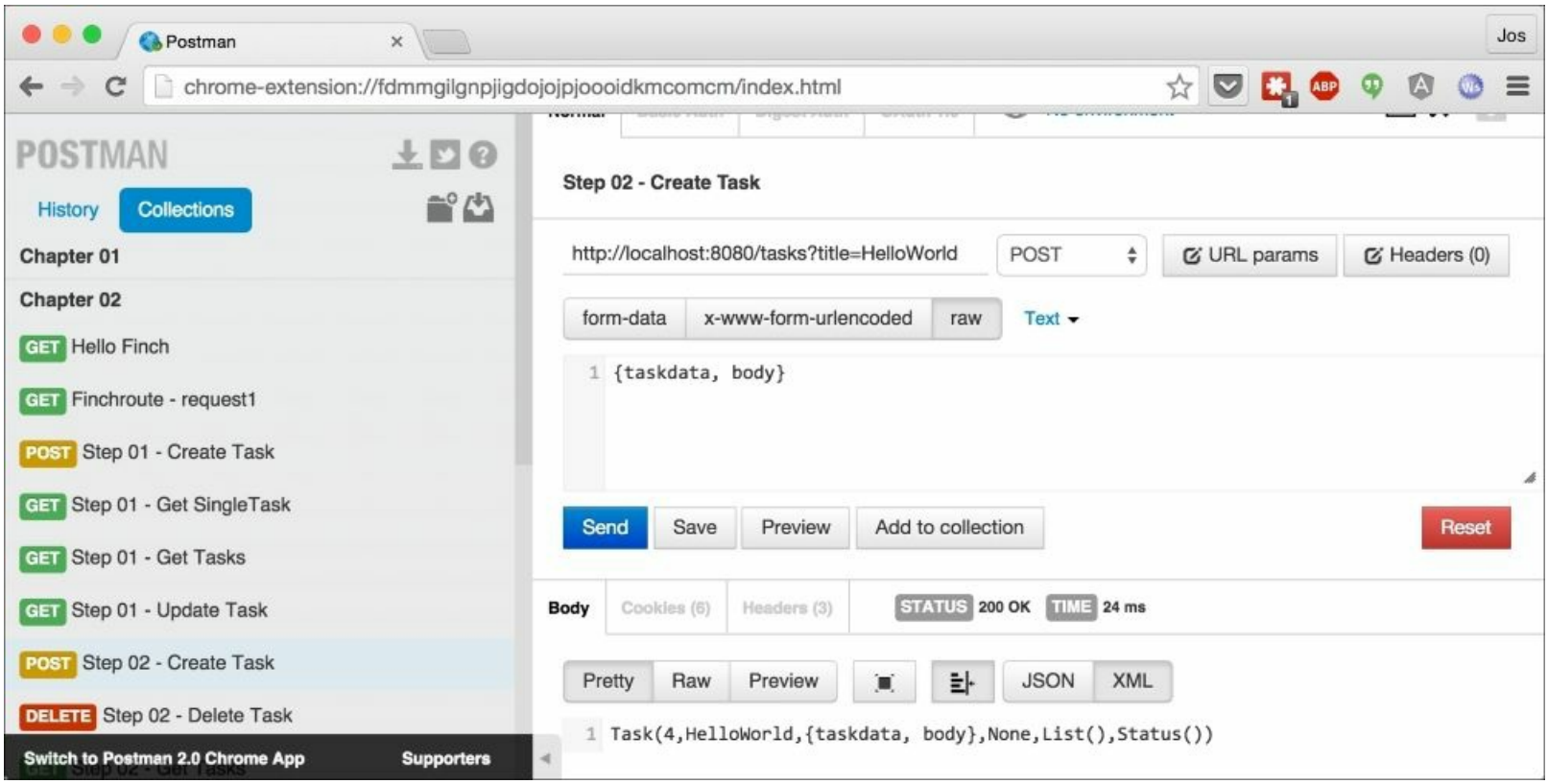
```
      RequestReader.value(Status(""))
      ).as[Task]
  }
}

case class CreateNewTask() extends Service[Request, String]
                                                  with BaseTask {

  def apply(req: Request): Future[String] = {
    val p = for {
      task <- getRequestToTaskReader(-1)(req)
      stored <- TaskService.insert(task)
    } yield stored

    p.map(_.toString)
  }
}

case class DeleteTask(id: Long)
                       extends Service[Request, HttpResponse] {

  def apply(req: Request): Future[HttpResponse] =
    TaskService.delete(id).map {
      case Some(task) => Ok()
      case None => NotFound()
    }
}

case class GetAllTasks() extends Service[Request, HttpResponse] {

  def apply(req: Request): Future[HttpResponse] = {
    for {
      tasks <- TaskService.all
    } yield Ok(tasks.mkString(":"))
  }
}

case class GetTask(taskId: Long)
                        extends Service[Request, HttpResponse] {
  def apply(req: Request): Future[HttpResponse] = {
    TaskService.select(taskId).map {
      case Some(task) => Ok(task.toString)
      case None => NotFound()
    }
  }
}

case class UpdateTask(taskId: Long)
           extends Service[Request, HttpResponse] with BaseTask {
  def apply(req: Request): Future[HttpResponse] =
    for {
      task <- getRequestToTaskReader(taskId)(req)
      stored <- TaskService.update(task)
    } yield stored match {
      case Some(task) => Ok(task.toString)
```

```
        case None => NotFound()
      }
  }
}
```

In this code, we see a couple of new things. Instead of directly returning a string value, we use a case class that extends from `Service` to process the HTTP request and create a response. You can also run this service directly from SBT. Running the `sbt runCH02-runCH02Step2` command will start up the service:

```
$ sbt runCH02-runCH02Step2
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Compiling 1 Scala source to /Users/jos/dev/git/rest-with-scala/chapter-
02/target/scala-2.11/classes...
[info] Running org.restwithscala.chapter2.steps.FinchStep2
Jun 27, 2015 10:26:49 AM com.twitter.finagle.Init$$anonfun$1 apply$mcV$sp
INFO: Finagle version 6.25.0 (rev=78909170b7cc97044481274e297805d770465110)
built at 20150423-135046
Press <enter> to exit.
```

You can test this service once again using Postman. Let's start by testing whether we can create a new task. To do this, open up Postman and fire off the request **Step 02 – Create Task**:



The response we get back starts to look similar to real data. The text we entered in `body` as well as

the `title` request parameter is used.

Let's look at the router we used to create a new task in detail to see how this works:

```
val matchTask: Matcher = "tasks"
val taskCreateAPI = Post / matchTask /> CreateNewTask()

...

sealed trait BaseTask {

  def getRequestToTaskReader(id: Long): RequestReader[Task] = {
    ( RequestReader.value(id) ::
      param("title") ::
      body ::
      RequestReader.value(None:Option[Person]) ::
      RequestReader.value(List.empty[Note]) ::
      RequestReader.value(Status(""))
      ).as[Task]
  }
}

case class CreateNewTask() extends Service[Request, String]
                                                  with BaseTask {

  override def apply(req: Request): Future[String] = {
    val p = for {
      task <- getRequestToTaskReader(-1)(req)
      stored <- TaskService.insert(task)
    } yield stored

    p.map(_.toString)
  }
}
```

At the top of this code fragment, you can see how we define the router that handles the create task request we just made through Postman. Whenever a `POST` request is made to the `tasks` URL, this router matches and maps the request to the function to the right of the `/>` combinator. This time, however, we don't map to a function that returns a string, but we map to a case class that extends from `Service`. In our own class, we have to implement the `def apply(request: Req): Future[Rep]` function from the abstract `Service` class. In this specific example, we specified the type parameters for this service as `Request` and `String`, so the `apply` function should transform the incoming `Request` instance to a `Future[String]` object.

Tip

As the first type parameter, you normally specify `Request` (unless you apply filters before processing the request, as we'll explain in the last part of this chapter), and the second type parameter should be a type, which Finch can automatically convert to an HTTP response. To automatically convert, Finch looks for an implicit `EncodeResponse[A]` type-class in scope. Out of the box, Finch will transform strings to HTTP responses. It also supports a number of JSON libraries, where case classes are

automatically converted to HTTP responses with JSON bodies.

In the service for this route, we take a couple of steps:

1. First, we call the `getRequestToTaskReader` function defined in the base class, with the ID of the task we want to create. Since we're creating a new one, we just specify `-1` as the ID and let the backend generate a real ID. The result from this call is a `RequestReader[Task]` instance, which can convert a request into a `Task` class.
2. We then directly call the `apply` function on the returned `RequestReader[Task]` instance with the passed in request. This call returns a `Future[Task]` object, which we process further in the `for` comprehension.
3. When the future from step 2 resolves, we have access to a task. We store this task using the `TaskService.insert` method. This call also returns a `Future`.
4. Finally, we yield the stored `Task` object, as a `Future[Task]` instance.
5. The last step in the service is converting the `Future[Task]` object to a `Future[String]` object, which we just do by using a simple `map` function. The reason we need to do this is because Finch doesn't know how to automatically convert `Task` objects to an HTTP response.

Before we move on to the next section, let's look a bit closer at the `RequestReader[Task]` instance we used to convert a `Request` object to a Task object:

```
    def getRequestToTaskReader(id: Long): RequestReader[Task] = {
      ( RequestReader.value(id) ::
        param("title") ::
        body ::
        RequestReader.value(Option.empty[Person]) ::
        RequestReader.value(List.empty[Note]) ::
        RequestReader.value(Status(""))
        ).as[Task]
    }
```

In this function, we combine various RequestReader (`body`, `param`, and `RequestReader.value`) using the `::` combinator (we'll explain more about `body`, `param`, and `RequestReader.value` in the next section). When we pass in a `Request` to the result of this function, each `RequestReader` is executed against the request. The result of all these individual steps is combined using the `as[A]` function (you can also use `asTuple` to collect the results). Finch standard supports conversion to int, long, float, double, and boolean, and also allows you to convert to a case class. In this last case, you have to make sure the result from the individual `RequestReader` matches the constructor of your case class. In this example, `Task` is defined like this:

```
case class Task(id: Long, title: String, content: String,
                assignedTo: Option[Person], notes: List[Note],
                status: Status)
```

And this matches the results of the individual RequestReaders. Should you want to convert to a type that isn't supported, you can very simply write your own, and just make sure it is in scope:

```
implicit val moneyDecoder: DecodeRequest[Money] =
         DecodeRequest(s => Try(new Money(s.toDouble)))
```

In the example code so far, we've only used a couple of RequestReaders: `param` and `body`. Finch provides a number of other readers you can use to access information from the HTTP request:

Reader	**Description**
`param(name)`	This returns the request parameter as a string and throws a `NotPresent` exception when the parameter can't be found.
`paramOption(name)`	This returns the request parameter as an `Option[String]` object. This call will always succeed.
`paramsNonEmpty(name)`	This returns a multivalue parameter as a `Seq[String]` object. If the parameter can't be found, a `NotPresent` exception is thrown.
`params(name)`	This returns a multivalue parameter (for example, `?id=1,2,3&b=1&b=2`) as a `Seq[String]` object. If the parameter can't be found, an empty list is returned.
`header(name)`	This returns a request header with the specified name as a string and throws a `NotPresent` exception when the header can't be found.
`headerOption(name)`	This returns a request header with the specified name as an `Option[String]` object. This call will always succeed.
`cookie(name)`	This gets a `Cookie` object from the request. If the specified cookie isn't present, a `NotPresent` exception is thrown.
`cookieOption(name)`	This gets a `Cookie` object from the request. This call will always succeed.
`body`	This returns the request body name as a string and throws a `NotPresent` exception when there is no body present.
`bodyOption`	This returns the body as an `Option[String]` object. This call will always succeed.
`binaryBody`	This returns the request body name as an `Array[Byte]` object and throws a `NotPresent` exception when there is no body present.
`binaryBodyOption`	This returns the body as an `Option[Array[Byte]]` object. This call will always succeed.
`fileUpload`	This `RequestReader` reads an upload (a multipart/form) parameter from the request and throws a `NotPresent` exception when the upload can't be found.
`fileUploadOption`	This `RequestReader` reads an upload (a multipart/form) parameter from the request. This call will always succeed.

As you can see from this table, a large number of RequestReaders types are already available, and in most cases, this should be enough to handle your requirements. If the `RequestReader` object does not provide the required functionality, a number of helper functions are available that you can use to create your own custom `RequestReader`:

Function	**Description**
`value[A](value: A): RequestReader[A]`	This function creates a `RequestReader` instance that always succeeds and returns the specified value.
`exception[A](exc: Throwable): RequestReader[A]`	This function creates a `RequestReader` instance that always fails with the specified exception.
`const[A](value: Future[A]): RequestReader[A]`	This `RequestReader` will just return the specified value.
`apply[A](f: HttpRequest => A): RequestReader[A]`	This function returns a `RequestReader` instance that applies the provided function.

There is one part of the `RequestReader` that we haven't discussed yet. What happens when a `RequestReader` fails? Finch has a very elegant mechanism to handle these validation errors. We'll come back to that in the last part of this chapter.

JSON support

So far, we've just worked with plain strings as responses. In this section, we'll expand the previous sample and add JSON support and show you how you can control which HTTP response code should be used when handling a request. Using JSON with Finch is very straightforward since Finch already supports a number of JSON libraries out of the box:

- Argonaut (http://argonaut.io/).
- Jackson (https://github.com/FasterXML/Jackson)
- Json4s (http://json4s.org/)

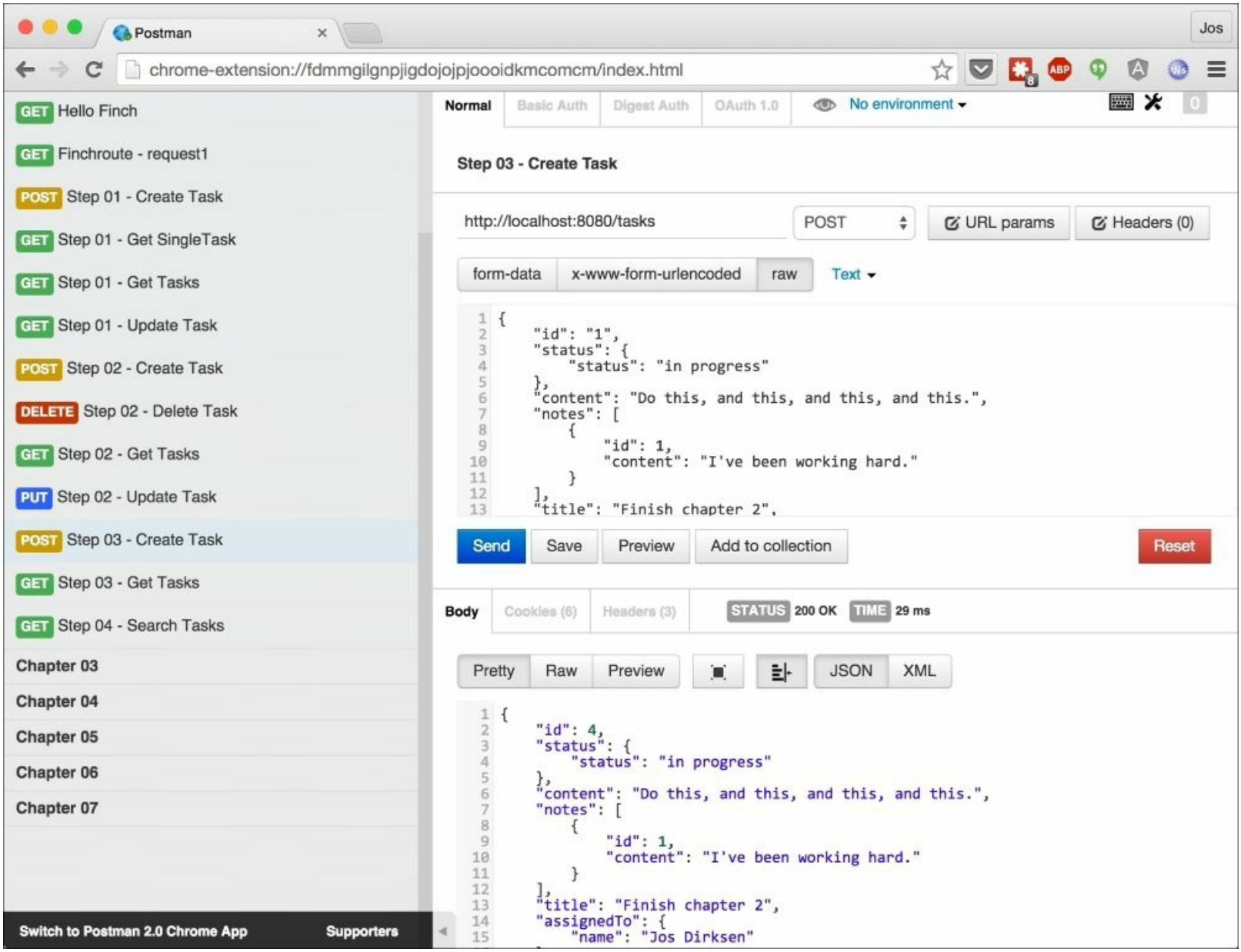
Argonaut

In this section, we'll look at how to use the Argonaut library to automatically convert our model (our case classes) to JSON. Should you want to use one of the other libraries, they work in pretty much the same manner.

We'll start by looking at the request and response message that our service should work with for this scenario. First, start up the server using the `sbt runCH02-runCH02Step3` command:

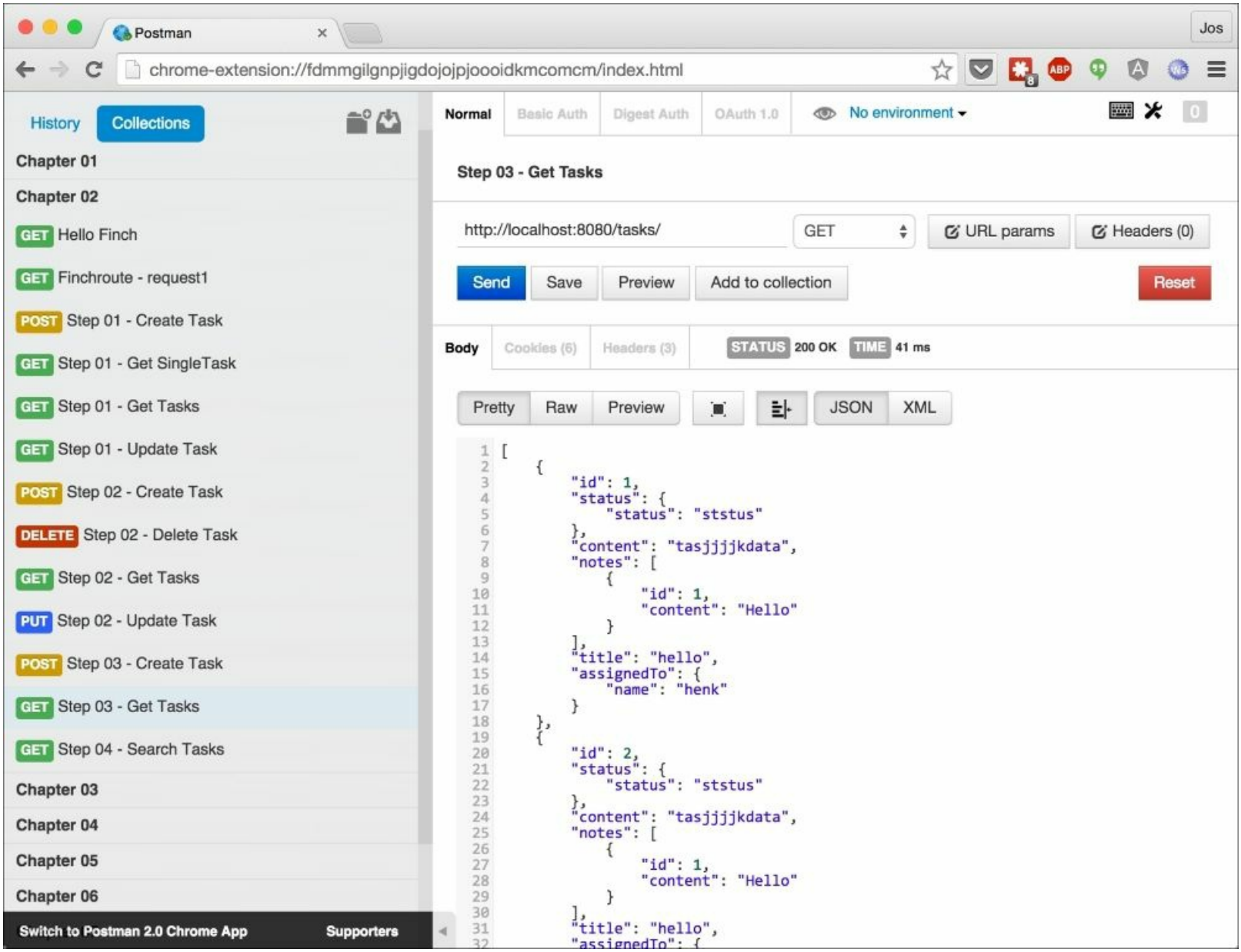
```
$ sbt runCH02-runCH02Step3
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter2.steps.FinchStep3
Jun 27, 2015 1:58:20 PM com.twitter.finagle.Init$$anonfun$1 apply$mcV$sp
INFO: Finagle version 6.25.0 (rev=78909170b7cc97044481274e297805d770465110)
built at 20150423-135046
Press <enter> to exit.
```

When the server is started, open Postman and select the request **Step 03 – Create Task** from the **Chapter 02** collection. When you send this request, the server will parse this to a case class, store it, and return the stored task once again as JSON.

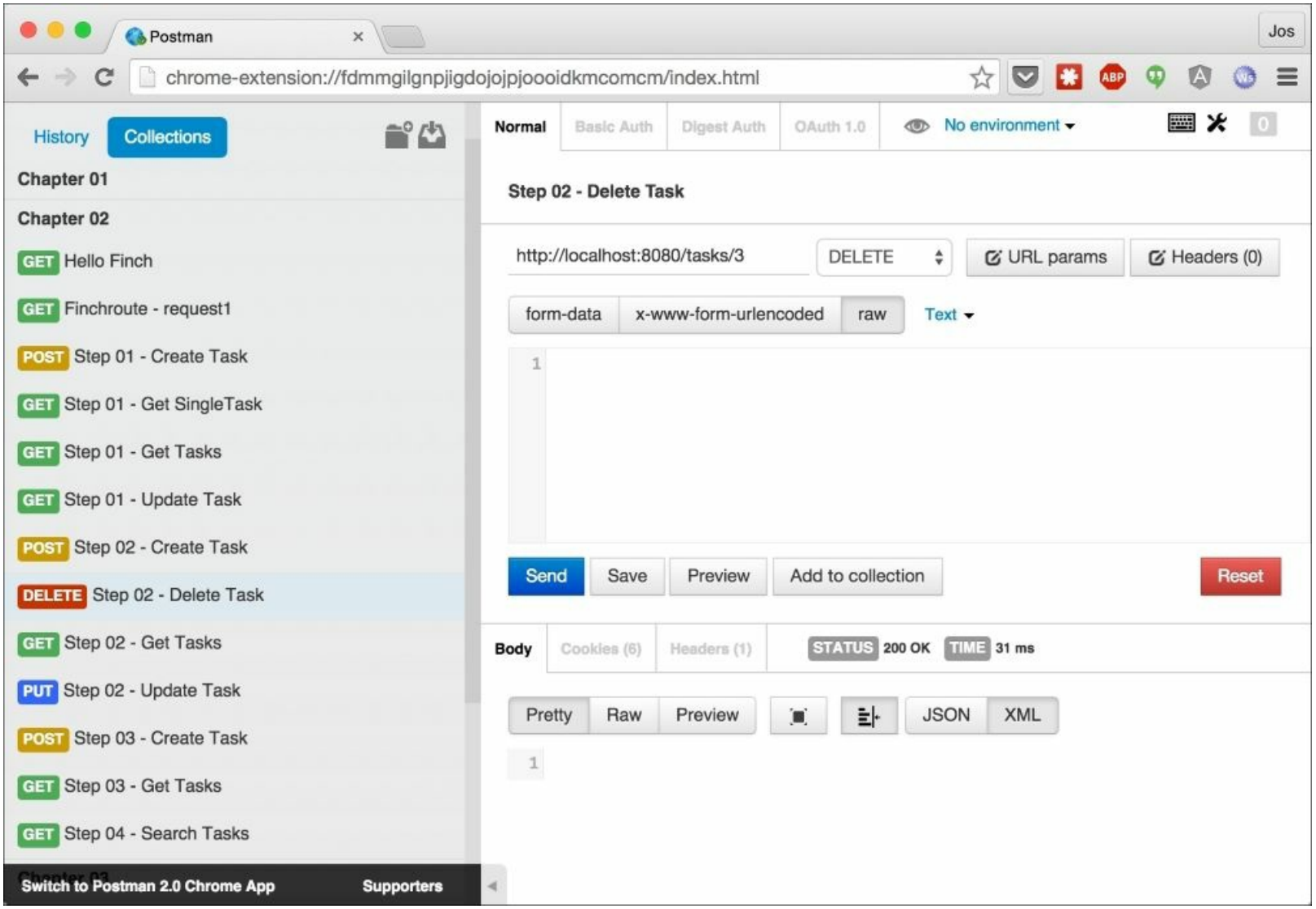


If you send the message a couple of times, you'll notice that the ID of the response increases. The reason is that we generate a new ID for newly created tasks, so ignore the ID from the incoming JSON message.

Once you've created a number of new tasks, you can also get all the stored tasks by using the **Step 03 – Get Tasks** request:



When you've stored a number of messages, you can also use the API to delete tasks. Click on **Step 02 – Delete Task**, change the URL to the ID you want to delete (for example, `http://localhost:8080/tasks/3`):



If the task with the ID you want to delete exists, it will return **200 Ok**, if the ID doesn't exist, you'll see **404 Not Found**.

To get this working, the first thing we need to do is get the required Argonaut dependencies. For this, we change the dependencies in our SBT build to this:

```
lazy val finchVersion = "0.7.0"

val backendDeps = Seq(
  "com.github.finagle" %% "finch-core" % finchVersion,
  "com.github.finagle" %% "finch-argonaut" % finchVersion
)
```

Jackson and Json4s

For Jackson and Json4s, you use the `finch-jackson` and `finch-json4s` modules instead.

To automatically convert our case classes to and from JSON, we need to tell Argonaut how we can convert to our case classes and vice versa. For our example, we've done this in the `chapter2` package object (located in the `package.scala` file):

```
implicit def personDecoding: DecodeJson[Person] = jdecode1L(Person.apply)
("name")

implicit def personEncoding: EncodeJson[Person] = jencode1L((u: Person) =>
(u.name))("name")

implicit def statusDecoding: DecodeJson[Status] = jdecode1L(Status.apply)
("status")

implicit def statusEncoding: EncodeJson[Status] = jencode1L((u: Status) =>
(u.status))("status")

implicit def noteDecoding: DecodeJson[Note] = jdecode2L(Note.apply)("id",
"content")

implicit def noteEncoding: EncodeJson[Note] = jencode2L((u: Note) => (u.id,
u.content))("id", "content")

implicit def taskDecoding: DecodeJson[Task] = jdecode6L(Task.apply)
    ("id", "title", "content", "assignedTo", "notes", "status")
implicit def taskEncoding: EncodeJson[Task] = jencode6L( (u: Task) => (u.id,
u.title, u.content,
                u.assignedTo, u.notes, u.status))
  ("id", "title", "content", "assignedTo", "notes", "status" )
```

For each of the case classes we want to support, we need a set of implicit values. To convert from JSON, we need a `DecodeJson[A]` instance and to convert to JSON, a `EncodeJson[A]` instance. Argonaut already provides some helper methods you can use to easily create these instances, which we've used in the previous example. For instance, with `jdecode2L` (the `2` stands for two arguments), we convert two JSON values to a case class, and with `jencode2L`, we convert two parameters of a case class to JSON. To learn more about Argonaut, you can look at its website at http://argonaut.io/; the part that deals with automatic conversion (as explained here) can be found at http://argonaut.io/doc/codec/.

Now that we've defined the mapping between JSON and the case classes we're using, we can look at how this changes our implementation. In the following code fragment, we see the code that handles the **Create Task**, **Delete Task**, and **Get Tasks** requests:

```
val matchTask: Matcher = "tasks"
val matchTaskId = matchTask / long

val taskCreateAPI =
```

```
      Get / matchTask /> GetAllTasks() :+:
     Post / matchTask /> CreateNewTask() :+:
   Delete / matchTaskId /> DeleteTask

...


case class CreateNewTask() extends Service[Request, HttpResponse] {

  def apply(req: Request): Future[HttpResponse] = {
    for {
      task <- body.as[Task].apply(req)
      stored <- TaskService.insert(task)
    } yield Ok(stored)
  }
}

case class DeleteTask(id: Long)
         extends Service[Request, HttpResponse] {
  def apply(req: Request): Future[HttpResponse] =
                                      TaskService.delete(id).map {
    case Some(task) => Ok()
    case None => NotFound()
  }
}

case class GetAllTasks() extends Service[Request, HttpResponse] {
  def apply(req: Request): Future[HttpResponse] = {
    for {
      tasks <- TaskService.all
    } yield Ok(tasks)
  }
}
```

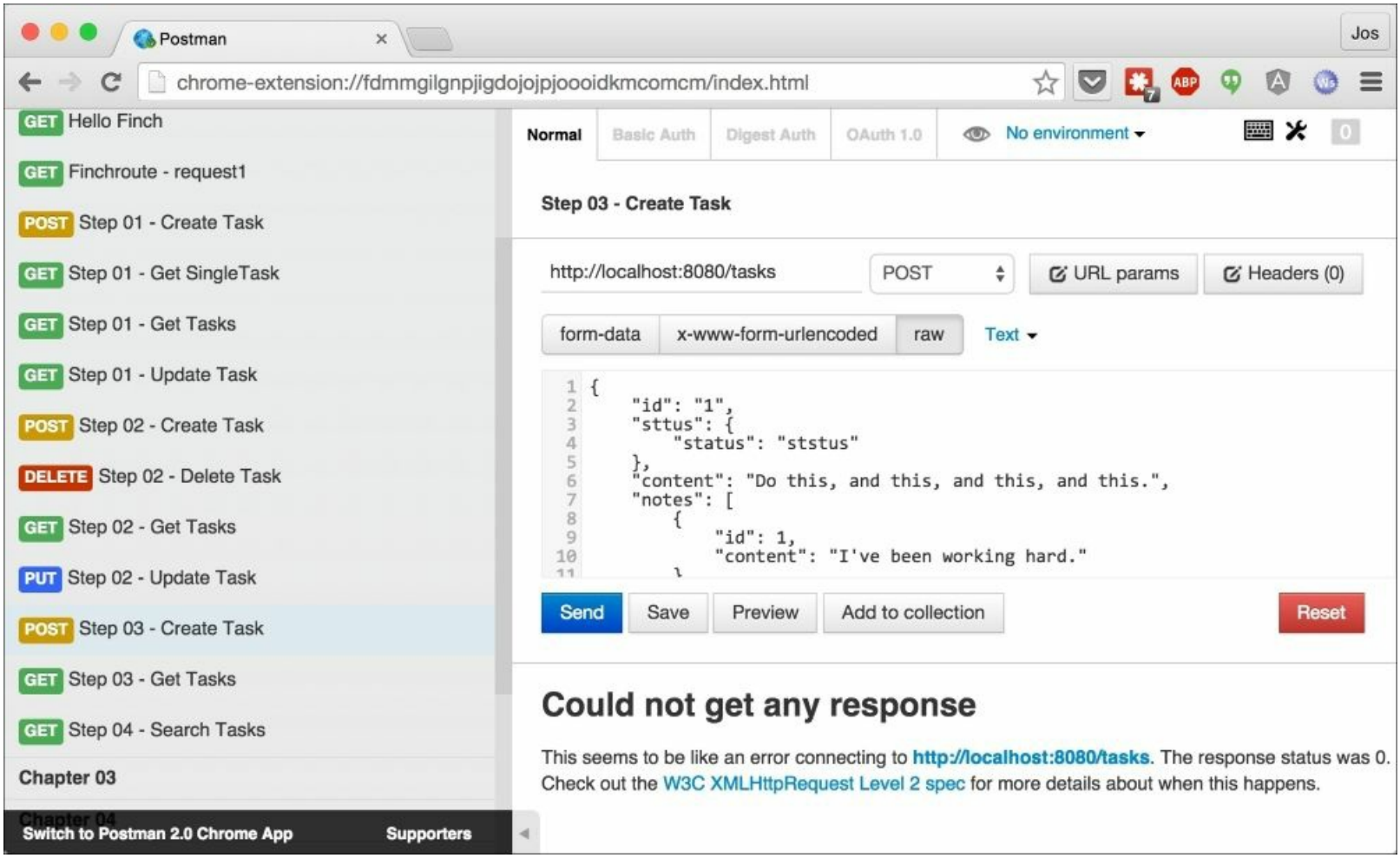
First, we'll look at the `CreateNewTask` class. As you can see, the code has become much simpler since we don't have to explicitly define how an incoming request is transformed to a `Task` anymore. This time, all we need to do in the `apply` function of the `CreateNewTask` service is use the body, `RequestReader`, and use `as[Task]` to automatically convert the provided request to a `Task`. This works since we implicitly defined a `DecodeJson[Task]` instance. Once the `Task` is created from the `Request`, we pass it into the `TaskService` to store it. The `TaskService` returns a `Future[Task]`, with the `Task` that is stored (this will have the correct ID filled in). Finally, we return `Ok` with the stored `Task` as a parameter. Finch will convert this `Ok` object to an `HttpResponse` with code 200, and it will convert the provided `Task` to JSON using the implicit `EncodeJson[Task]` instance. We'll look a bit closer at how to build and customize the HTTP response in the following section. The `GetAllTasks()` class works in pretty much the same manner. It retrieves a `Future[Seq[Task]]` object from the `TaskService` instance and Finch, together with the implicitly defined objects, and knows how to convert this sequence of tasks to the correct JSON message.

Before we move on to the next section, let's quickly look at the `DeleteTask` class. As you can see in the code, this case class takes an additional parameter. This parameter will contain the long value,

which was extracted by the `long` extractor in the router that mapped to this `Service`. If you have multiple extractors in the router, your case class should have the same amount of parameters.

Request validation and custom responses

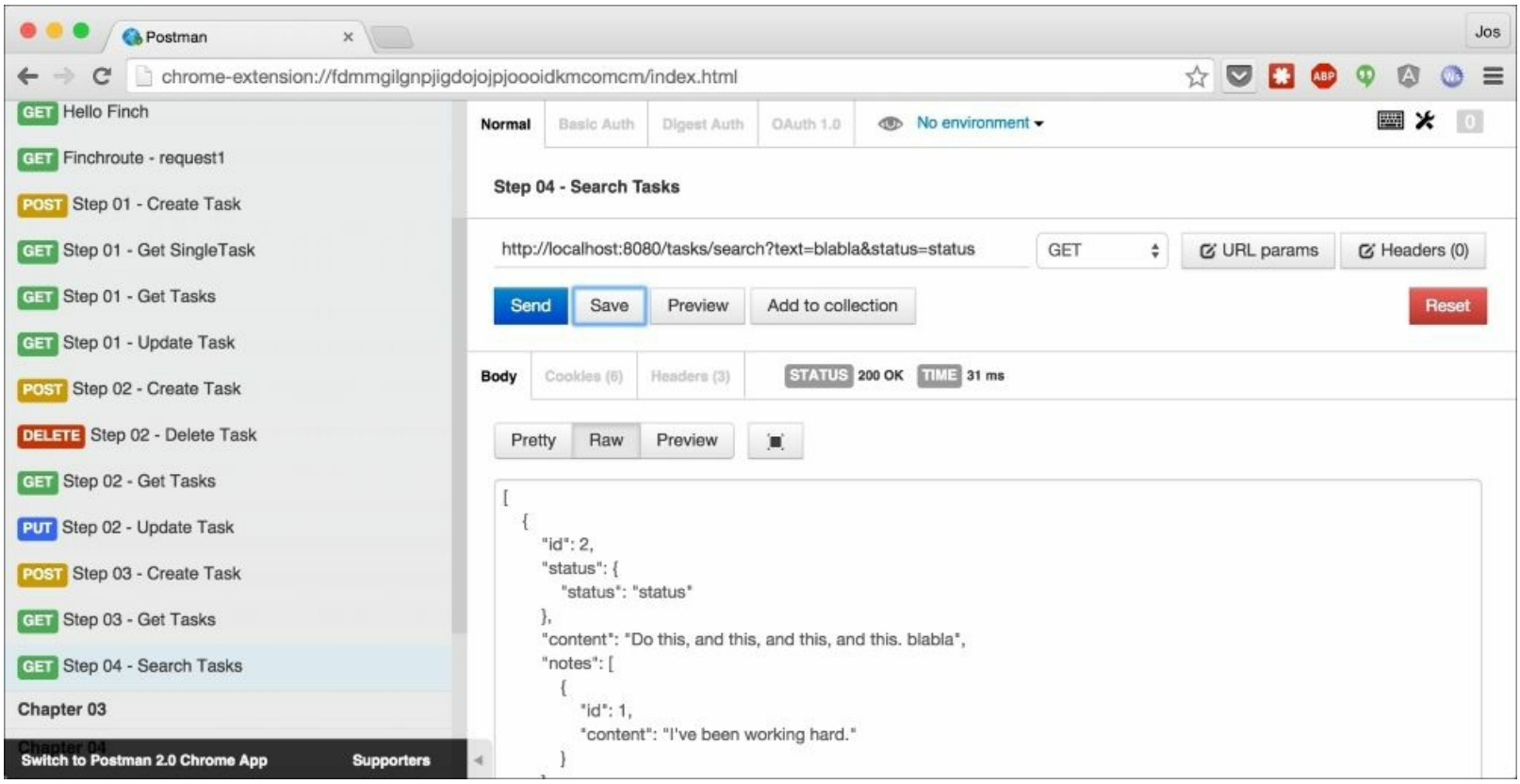
So far, we haven't looked at what happens when one of our RequestReaders can't read the required information. A header might be missing, a parameter might be in the incorrect format, or a cookie isn't present. If, for instance, you rename some fields in the JSON for the **Step 03 – Create Task** request, and make the request, it will fail silently:



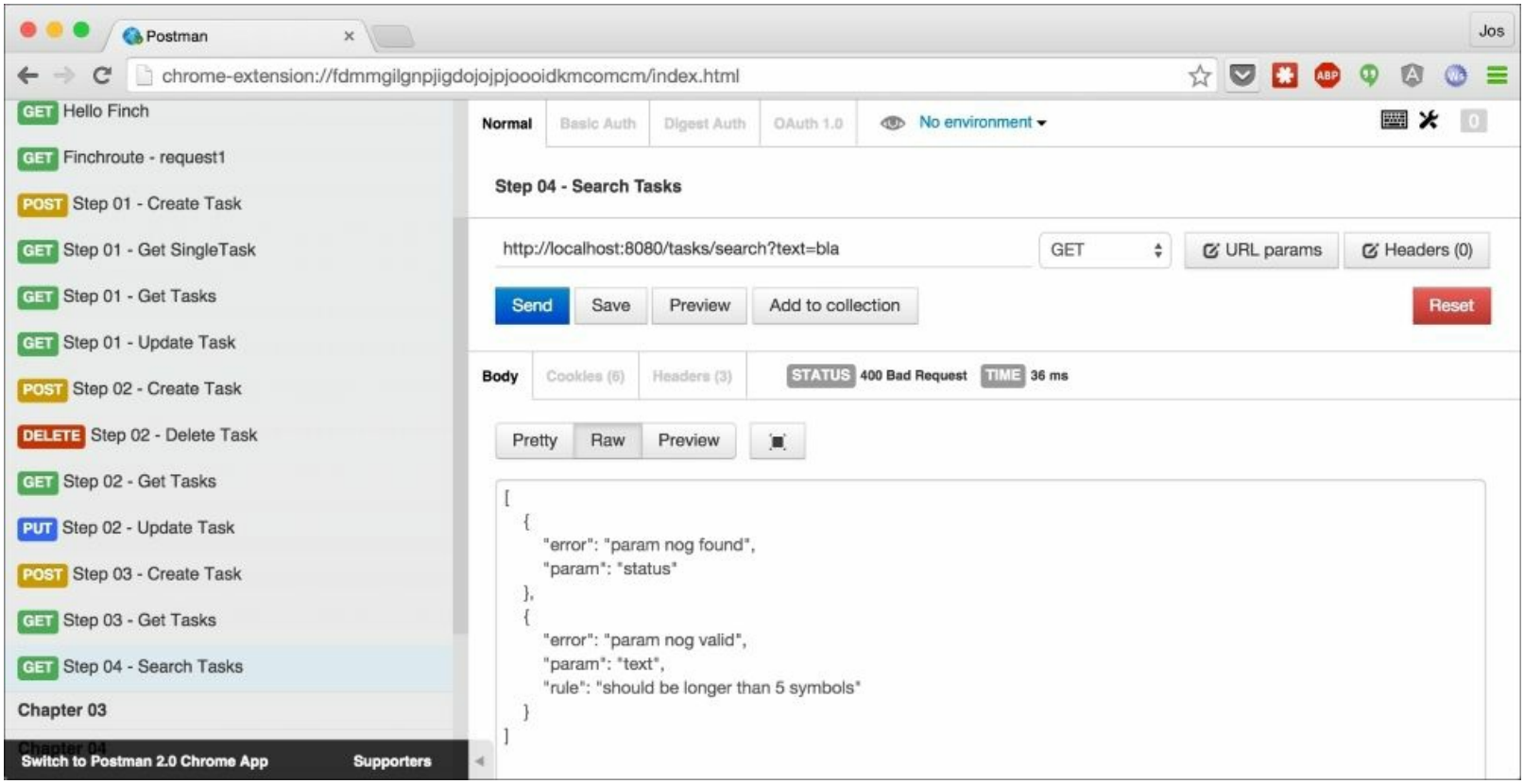
Finch, however, provides an elegant way to handle all the exceptions from the RequestReaders. First, we'll look at the result we'll be aiming for. First, start another `sbt` project like this:

```
$ sbt runCH02-runCH02Step4
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Compiling 1 Scala source to /Users/jos/dev/git/rest-with-scala/chapter-
02/target/scala-2.11/classes...
[info] Running org.restwithscala.chapter2.steps.FinchStep4
Jun 28, 2015 2:10:12 PM com.twitter.finagle.Init$$anonfun$1 apply$mcV$sp
INFO: Finagle version 6.25.0 (rev=78909170b7cc97044481274e297805d770465110)
built at 20150423-135046
Press <enter> to exit.
```

Open Postman and use **Step 03 – Create Task** to create some tasks in our database. For this example, we've added a search functionality that you can access through the **Step 04 – Search Tasks** request.



To show how validation works, we've added a couple of rules to the request parameters. The `status` request parameter is required, and when the `text` parameter is used, its value should be at least five characters. To test how this works, either remove the `status` parameter or change the value of the `text` parameter to something smaller than five characters. The following screenshot shows the resulting error messages:



The following code fragment shows the case class we use to search the database and show changes we have to make to our application to get these validation results:

```
// uses the following route
// Get / matchTask / "search" /> SearchTasks()

case class SearchParams(status: String, text: Option[String])

case class SearchTasks() extends Service[Request, HttpResponse] {

    def getSearchParams: RequestReader[SearchParams] = (
          param("status") ::
          paramOption("text").should(beLongerThan(5))
        ).as[SearchParams]

    def apply(req: Request): Future[HttpResponse] = {
      (for {
        searchParams <- getSearchParams(req)
        tasks <- TaskService.search
                    (searchParams.status, searchParams.text)
      } yield Ok(tasks)).handle({case t: Throwable =>
                                    BadRequest(errorToJson(t))})
    }

    def errorToJson(t: Throwable):Json = t match {

      case NotPresent(ParamItem(param)) =>
        Json("error" -> Json.jString("param not found"),
```

```
                                    "param" -> Json.jString(param))
      case NotValid(ParamItem(param), rule) =>
        Json("error" -> Json.jString("param not valid"),
                                    "param" -> Json.jString(param),
                                    "rule" -> Json.jString(rule))
      case RequestErrors(errors) =>
                          Json.array(errors.map(errorToJson(_)):_*)
      case error:Throwable => Json("error" ->
                          Json.jString(error.toString))
    }
  }
```

When a request is passed into this `Service`, the `apply` function is invoked. In this function, we pass the request to a `RequestReader[SearchParams]` object that looks similar to this:

```
    def getSearchParams: RequestReader[SearchParams] = (
          param("status") ::
          paramOption("text").should(beLongerThan(5))
        ).as[SearchParams]
```

When this `RequestReader` is called with a request, it will first try and get the `status` parameter. If this parameter can't be found, a `NotPresent` exception will be thrown. This, however, doesn't stop the processing of the request, and the `RequestReader` gets the value of the `text` parameter. If the `text` parameter is available, it should be longer than five characters (note that we also have a `shouldNot` function for when you want to check whether a rule doesn't apply). If it isn't, a `NotValid` exception will be thrown. In the previous examples, if this happened, the processing of the request would have stopped, and the service would not have returned any response. To process these exceptions, we need to call the `handle` function on the `Future[HttpResponse]` instance (or the `Future` returned from the `RequestReader()` function).

Note

When you start working with Finch yourself, you might notice that it doesn't use the standard `scala.concurrent.Future` class from Scala, but uses `the Future` defined in `com.twitter.util.Future`. The reasons are that Finch (and Finagle, which is used internally by Finch) is a Twitter project, and that the Future from Twitter has a lot of additional functionality. For instance, the `handle` function, which is discussed in the next section, is a standard function on the Twitter `Future` object. The `TaskService` that we use in this book, however, uses standard Scala `Future` objects. To make sure we can easily interoperate between the Scala `Future` and Twitter `Future` objects, we've created some implicit conversions. If you're interested in how they look, you can find these implicit conversions in the `src/main/scala/org/restwithscala/chapter2/package.scala` file.

The `handle` function takes a `partial` function, and in this scenario, it should return an `HttpResponse` instance. As you can see in the code, we just convert the validation related exceptions as a JSON object and wrap it in a `BadRequest` class.

In the example, we showed we used the `beLongerThan` rule. Finch provides a number of standard

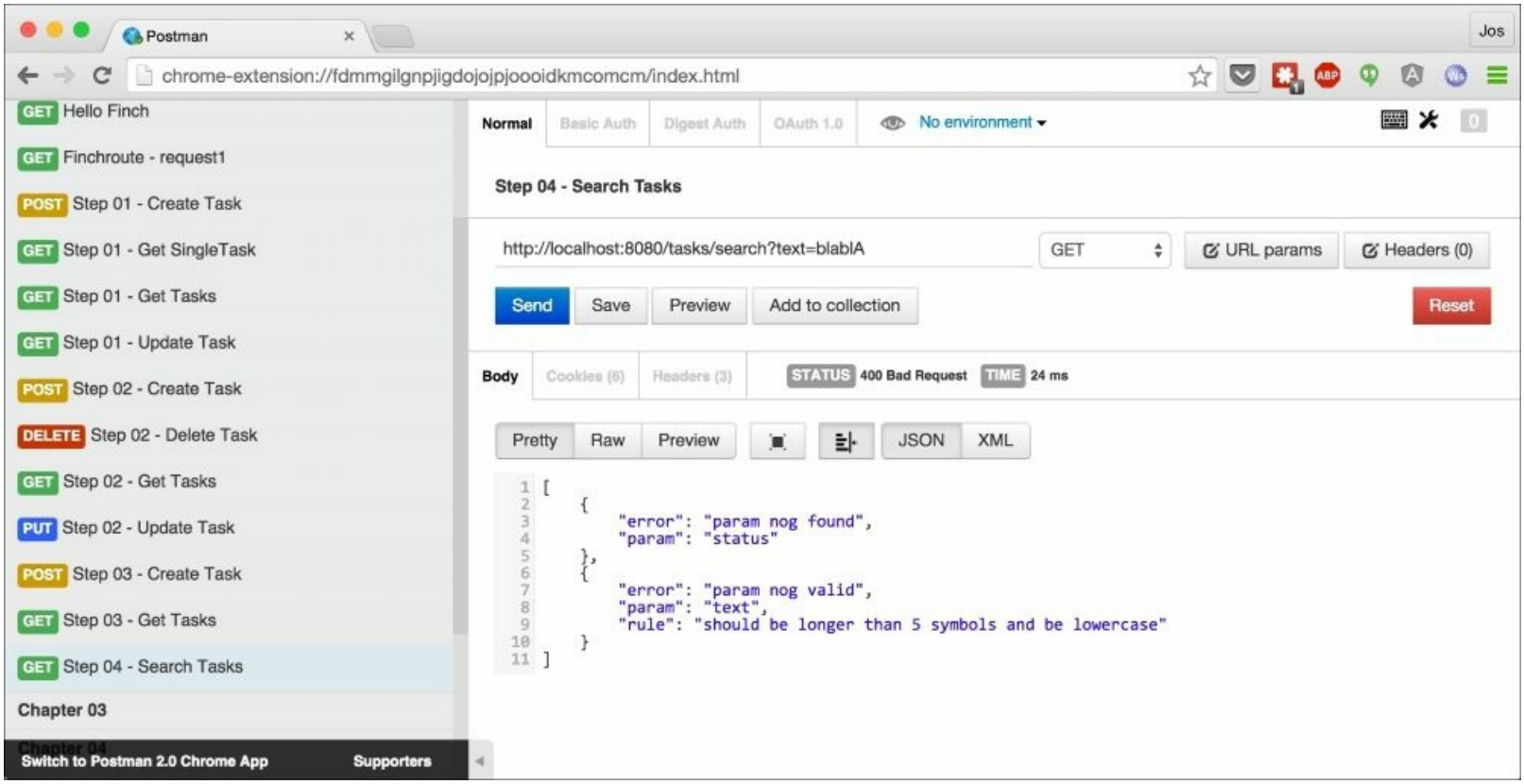
rules out of the box that you can use to check whether the result from a specific `RequestReader` is valid:

Rule	Description
`beLessThan(n: Int)`	This checks whether a numeric value is less than the specified integer.
`beGreaterThan(n: Int`	This checks whether a numeric value is greater than the specified integer.
`beShorterThan(n: Int)`	This checks whether the length of a string is shorter than the specified integer.
`beLongerThan(n: Int)`	This checks whether the length of a string is longer than the specified integer.
`and`	This combines two rules together. Both rules must be valid.
`or`	This combine two rules together. One of the rules must be valid.

Creating a custom validation rule is very simple. For instance, the following code creates a new rule that checks whether a string contains any uppercase characters:

```
    val shouldBeLowerCase = ValidationRule[String]("be lowercase")
{!_.exists((c: Char) => c.isUpper) }

    def getSearchParams: RequestReader[SearchParams] = (
          param("status") ::
          paramOption("text").should(beLongerThan(5) and shouldBeLowerCase)
        ).as[SearchParams]
```

When we now run a query, we also get a message if we use uppercase characters in our `text` parameter:



The last part that we'll look at a bit closely before we move on to the next chapter is how to create HTTP responses. We've already seen a little bit of this with the `Ok`, `BadRequest`, and `NotFound` case classes. Finch also provides a number of additional functions to further customize the HTTP response message. You can use the following functions to create the response:

Function	**Description**
`withHeaders(headers: (String, String)*)`	This adds the provided headers to the response.
`withCookies(cookies: Cookie*)`	This adds the provided cookies to the response.
`withContentType(contentType: Option[String])`	This sets the content-type of the response to the specified `Option[String]` value.
`withCharset(charset: Option[String])`	This sets the character-set of the response to the provided `Option[String]` object.

For example, if we wanted to create an `Ok` response with a custom character set, a custom content-type, some custom headers, and a string body, we'd do something like this:

```
Ok.withCharset(Some("UTF-8"))
  .withContentType(Some("text/plain"))
  .withHeaders(("header1" -> "header1Value"),
               ("header2" -> "header2Value"))("body")
```

Summary

In this chapter, we walked through the Finch framework. With the Finch framework, you can create REST services using a functional approach. Handling requests is done by mapping a request to a `Service`; validating and parsing requests is done using a Reader monad, the `RequestReader`; and all the parts are composable to create complex routes, RequestReaders, rules, and services from simple parts.

In the next chapter, we'll dive into a Scala REST framework that uses a different approach. We'll look at Unfiltered, which uses a pattern matching-based approach of defining REST services.

Chapter 3. A Pattern-matching Approach to REST Services with Unfiltered

In this chapter, we'll introduce a light-weight REST framework called **Unfiltered**. With Unfiltered, you can use standard Scala pattern matching to take complete control over how to process an HTTP request and create an HTTP response. In this chapter, we'll look at the following topics:

- Setting up the basic skeleton for an Unfiltered-based REST service
- Using matchers and extractors to process incoming HTTP requests
- Processing requests in synchronous and asynchronous ways
- Converting and validating incoming requests and parameters using extractors and directives
- Customizing response codes and response formats

What is Unfiltered

Unfiltered is an easy to use light-weight REST framework, which provides a set of constructs you can use to create your own REST services. For this, Unfiltered uses Scala pattern matching, together with a set of matchers and extractors. One of the interesting parts of Unfiltered is that it gives you complete control over how you handle your request and define your response. The framework itself won't add any headers, or makes assumptions regarding content-types or response codes unless you tell it to.

Unfiltered has been around for a couple of years and is used by a large number of companies. Some of the best-known ones are the following two:

- **Remember the Milk**: Remember the Milk is one of the best-known to-do apps. It uses Unfiltered to handle all its public APIs.
- **Meetup**: With Meetup, groups of people who share interests come together to share knowledge and schedule meetups. Meetup uses Unfiltered to serve its real-time APIs.

For more information and documentation about Unfiltered, you can check out the website at http://unfiltered.databinder.net/.

Your first Unfiltered service

Just like we did in the previous chapter, we will start by creating the most basic Unfiltered REST service. The dependencies for the examples and frameworks used in this chapter can be found in the `Dependencies.scala` file, which is located in the `project` directory. For the Unfiltered examples explained in this chapter, we use the following SBT dependencies:

```
  lazy val unfilteredVersion = "0.8.4"
  val backendDeps = Seq(
    "net.databinder" %% "unfiltered-filter" % unfilteredVersion,
    "net.databinder" %% "unfiltered-jetty" % unfilteredVersion
  )
```

To work with Unfiltered, we at least need the `unfiltered-filter` module, which contains the core classes we need to create routes and handle requests and responses. We also need to define which type of server we want to use to run Unfiltered on. In this case, we run Unfiltered on an embedded Jetty (http://www.eclipse.org/jetty/) instance.

With these dependencies in place, we can create a minimal Unfiltered service. You can find the code for this service in the `HelloUnfiltered.scala` source file, which is located in the sources for this chapter:

```
package org.restwithscala.chapter3.gettingstarted

import unfiltered.request._
import unfiltered.response._

object HelloUnfiltered extends App {

  // explicitly set the thread name. If not, the server can't be
  // stopped easily when started from an IDE
  Thread.currentThread().setName("swj");

  // Start a minimalistic server
  val echo = unfiltered.filter.Planify {
    case GET(Path("/hello")) => ResponseString("Hello Unfiltered")
  }
  unfiltered.jetty.Server.http(8080).plan(echo).run()

  println("Press <enter> to exit.")
  Console.in.read.toChar

}
```

What we do here is create a simple route that responds to a `GET` request on the `/hello` path. When Unfiltered receives this request, it will respond with the `Hello Unfiltered` response using the `ResponseString` object to create a response (we will discuss this more later). This route is bound to a Jetty server, which runs on port `8080`.

Tip

You probably have noticed the strange `Thread.currentThread().setName` call at the beginning of this example. The reason we do this is to avoid Unfiltered starting in daemon mode. Unfiltered tries to detect whether we have started from SBT or run normally; if run from SBT, it allows us to stop the server by just pressing a key (the behavior we want). If not, it runs in the background and requires a shutdown hook to stop the server. It does this by checking the name of the current thread. If the name is `main`, it will run Unfiltered in daemon mode, if it is something else, it will run normally, which allows us to easily stop the server. So by setting the name of the main thread to something else, we can also have this nice shutdown behavior when run from the IDE.

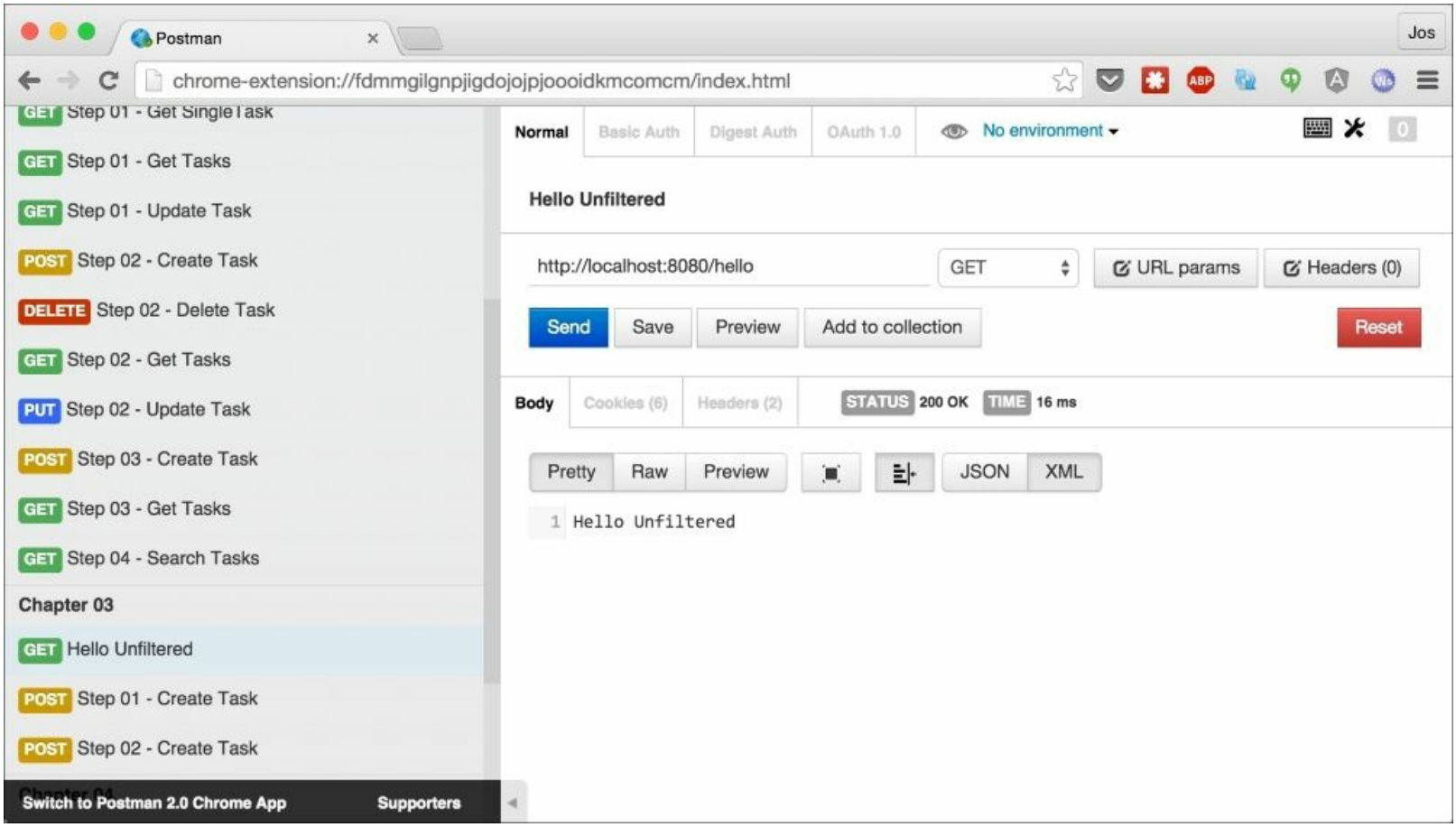
Additionally, we also need to set up logging for Jetty, which is the engine used by Unfiltered. Jetty has some prolific logging in its default configuration. To minimize the logging of Jetty to only log useful information, we need to add a `logback.xml` configuration file:

```
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern> %d{HH:mm:ss.SSS}[%thread]%-5level%logger{36}-%msg%n
            </pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

Now let's start the server and use Postman to make a call to the Unfiltered service. To start this example, run `sbt runCH03-HelloUnfiltered` from the root directory of the sources:

```
$ sbt runCH03-HelloUnfiltered
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter3.gettingstarted.HelloUnfiltered
10:04:32.108 [swj] INFO  org.eclipse.jetty.server.Server - jetty-
8.1.13.v20130916
10:04:32.178 [swj] INFO  o.e.jetty.server.AbstractConnector - Started
SocketConnector@0.0.0.0:8080
Embedded server listening at
  http://0.0.0.0:8080
Press any key to stop.
```

Open up Postman, click on the collection name **Chapter 03**, and open the first request. Once you click on **Send**, you should see the response, **Hello Unfiltered**, being returned by the server:



In the next section, we'll look a bit more closely at how Unfiltered uses pattern matching to map incoming requests to functions.

HTTP verb and URL matching

Unfiltered uses standard Scala pattern matching to determine what to do with a specific request. The following code shows how Unfiltered provides matchers for a number of simple REST calls:

```
package org.restwithscala.chapter3.steps

import unfiltered.request._
import unfiltered.response._

object Step1 extends App {

Thread.currentThread().setName("swj");


  object api extends unfiltered.filter.Plan {
    def intent = taskApi.onPass(fallback)

    def taskApi = unfiltered.filter.Intent {
      case GET(Path("/tasks"))
            => ResponseString(s"Get all tasks" )
      case GET(Path(Seg("tasks" :: id :: Nil)))
            => ResponseString(s"Get a single task with id: $id" )
      case DELETE(Path(Seg("tasks" :: id :: Nil)))
           => ResponseString
                     (s"Delete an existing task with id $id")

      case req @ Path("/tasks") => req match {
        case POST(_)
          => ResponseString(s"Create a new" +
                          s" task with body ${Body.string(req)}")
        case PUT(_)
          => ResponseString(s"Update a new task with" +
                           s" body ${Body.string(req)}")
        case _ => Pass
      }
      case _ => Pass
    }

    def fallback = unfiltered.filter.Intent {
      case _ => NotImplemented ~>
                ResponseString("Function is not implemented")
    }
  }


  unfiltered.jetty.Server.http(8080).plan(api).run()

  println("Press <enter> to exit.")
  Console.in.read.toChar

}
```

There is a lot to see in this code fragment, so let's start at the beginning. To create a REST API in Unfiltered, we need to create a `Plan` object. A `Plan` describes how to respond to specific requests. There are two different ways to create such a `Plan`. You can directly pass in a partial function to `unfiltered.filter.Planify`, like we did in the getting started example earlier in this chapter or explicitly extend from `unfiltered.filter.Plan` and set the `intent val` to your route configuration. In the rest of this chapter, we'll use the latter approach since it allows us to combine API parts in an easy manner. Let's start with the first set of matchers from the `taskApi` instance:

```
case GET(Path("/tasks"))
      => ResponseString(s"Get all tasks" )
case GET(Path(Seg("tasks" :: id :: Nil)))
      => ResponseString(s"Get a single task with id: $id" )
case DELETE(Path(Seg("tasks" :: id :: Nil)))
      => ResponseString(s"Delete an existing task with id $id")
```

As you can see, we use standard pattern matching from Scala. These patterns will be checked against the current `HttpRequest` instance and will make use of a number of matchers provided by Unfiltered:

Matcher	Description
`GET` and the other HTTP verbs.	With this matcher, we match the HTTP verb of the request. Unfiltered provides the following number of standard matchers for this: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`, `LINK`, and `UNLINK`. Should you have edge cases where you need to match other verbs, you can easily create your own matcher like this: `object WOW extends Method("WOW")`
`Path`	The next matcher we see is the `Path` matcher. With this matcher, you check whether you match the complete URL path. So in the preceding example, the first pattern only matches when the exact `/tasks` path is called.
`Seq`	If you want to extract path segments or match more flexibly on multiple path segments, you can use the `Seq` matcher. This matcher will check the URL on which the request was made, splits it into path segments, and check whether individual path segments match or extract path segments for further processing.

So, in our API:

- The first case matches a `GET` request on the `/tasks` path.
- The second case matches a `GET` request on the `/tasks/:id` path. The ID is passed into the function handling this case.
- The third case does the same as the second, but this time for a `DELETE` request.

Besides these matchers shown in this example, Unfiltered also provides the following:

Matcher	Description
`HTTP` and `HTTPS`	These two matchers allow you to check whether the request was received over an HTTP or HTTPS connection.

`Json`/`JavaScript`/`XML` and more	Unfiltered also allows you to check the Accepts header of a request. For instance, the JSON matcher will match when the `Accepts` header is `application/json` or a request is made on a URL with the extension `.js`, without an `Accepts` header. Unfiltered provides the following set of standard matchers of this type: `Json`, `JavaScript`, `AppJavaScript`, `Jsonp`, `Xml`, `Html`, and `Csv`. If you want to specify a new content-type matcher, you can do it like this: **`object Custom extends Accepting {`** **`  val contentType = "application/vnd+company.category"`** **`  val ext = "json"`** **`}`**
`HTTP_1_0`/`HTTP_1_1`	Check whether the protocol was made using HTTP version 1.0 or HTTP version 1.1.
`Mime`	This matcher allows you to check whether the request conforms to a specific mime-type.
Any header	Unfiltered also provides matchers and extractors for a large set of other HTTP headers. There are too many to list here; for a complete overview look at the objects in the `headers.scala` file from the Unfiltered sources.
`Params`	With this matcher-extractor, you can match a specific request parameter. We will show an example of this extractor in the next section.
`RemoteAddr`	Specific matcher that checks whether the `XForwaredFor` header contains a trusted address.

Let's look back at the example, especially the `PUT` and `POST` calls on the `/tasks` URL. For these two routes, we used an alternative approach:

```
  case req @ Path("/tasks") => req match {
      case POST(_)
        => ResponseString(s"Create a new" +
                          s" task with body ${Body.string(req)}")
      case PUT(_)
        => ResponseString(s"Update a new task with" +
                          s" body ${Body.string(req)}")
      case _ => Pass
    }
```

Here, we first match on the `Path("/tasks")` route and then use the matching request to determine what to do for the different verbs. This is a convenient way to handle multiple verbs that can be used to make a call to the same URL.

In this case, we can handle a `POST` and a `PUT` call, and just ignore any other call by returning `Pass`. When we return `Pass`, we just tell Unfiltered that this intent can't handle the request. When Unfiltered can't match a request with the current intent, it will just try the next. We've used this approach in our example:

```
object api extends unfiltered.filter.Plan {
  def intent = taskApi.onPass(fallback)

  def taskApi = unfiltered.filter.Intent { ... }
  def fallback = unfiltered.filter.Intent {
```

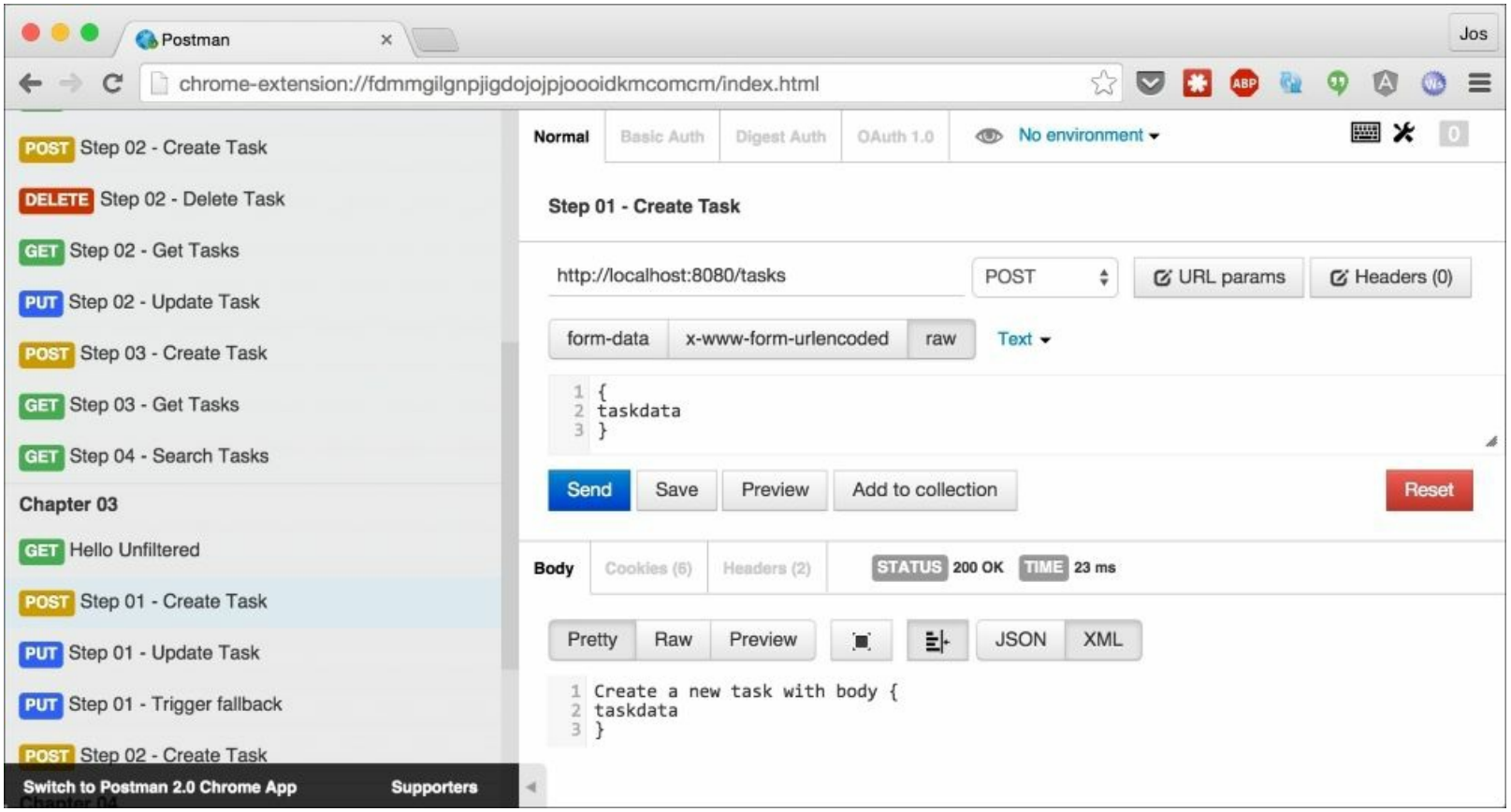
```
    case _ => NotImplemented ~>
              ResponseString("Function is not implemented")
  }
```

In our `Plan`, we defined two intents: one that handled our API, the `taskAPI` intent, and one that could serve as fallback when the `taskAPI` intent didn't match, aptly named `fallback` (we'll explain more about this later in this chapter). The `fallback` intent returns the HTTP code, `NotImplemented`, with a message. By calling `taskApi.onPass(fallback)`, we tell Unfiltered that when the `taskAPI` intent returns a `Pass` result, it should try the `fallback` intent. The `fallback` intent can also configure an `onPass` result, and this way, you can easily chain and combine APIs.

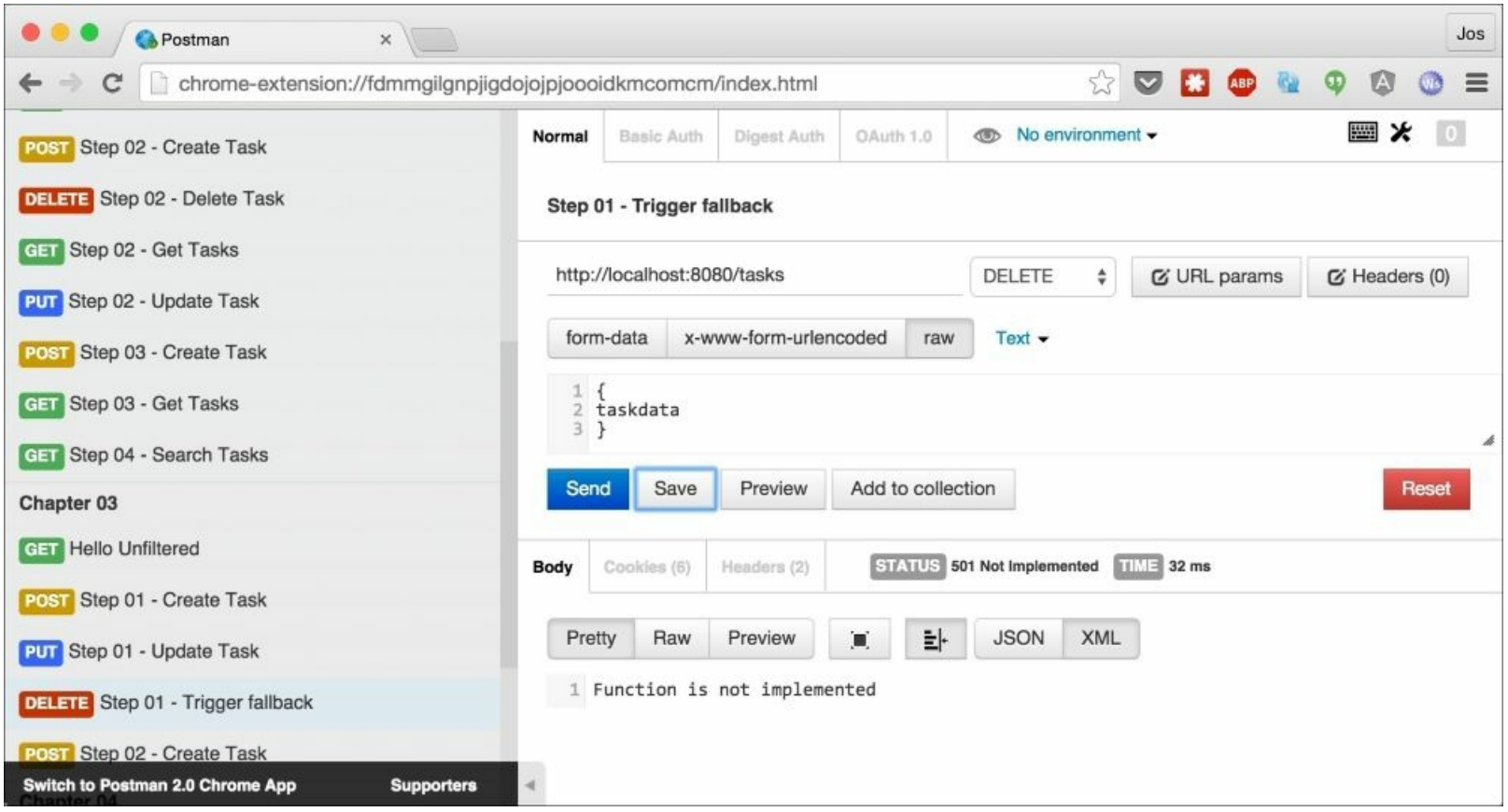
To test this service, start it with the `sbt runCH03-runCH03Step1` command:

```
$ sbt runCH03-runCH03Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter3.steps.Step1
15:06:36.085 [swj] INFO  org.eclipse.jetty.server.Server - jetty-
8.1.13.v20130916
15:06:36.160 [swj] INFO  o.e.jetty.server.AbstractConnector - Started
SocketConnector@0.0.0.0:8080
Embedded server listening at
  http://0.0.0.0:8080
Press any key to stop.
```

The collection for **Chapter 03** in Postman provides you with a number of requests you can use to test this server. For instance, the call to create a new task looks similar to this:



And when we hit the `fallback` route, we see the following screenshot:



So far, we've only focused on routing and matching requests. In the next section, we'll look at how we can access the request parameters and use the Scala `Future` object to create asynchronous responses.

Extracting request parameters and using futures for asynchronous responses

Now that we've got the basics covered, let's see what we need to do to convert the incoming request parameters and body into our domain model (our case classes). In this section, we'll take the following steps:

1. Convert the incoming request to a `Task` case class.
2. Store the converted `Task` in our dummy `TaskService`.
3. The `TaskService` object returns a `Future[Task]`; we will change the Unfiltered configuration to start handling requests asynchronously.

Let's start with the first part and look at the route configuration and how to convert the incoming request to a `Task` case class. The complete source for this example can be found in the `Step2.scala` file in the `rest-with-scala/chapter-03/src/main/scala/org/restwithscala/chapter3/steps/` directory.

Let's first show you the complete code and then we'll look at the individual parts in more detail. Note that we've only implemented a part of the complete task API here:

```
object Step2 extends App {

  implicit def ec = ExecutionContext.Implicits.global

  @io.netty.channel.ChannelHandler.Sharable
  object api extends future.Plan with ServerErrorResponse {

    implicit def executionContext = ec

    def intent =  {
      case GET(Path("/tasks")) => Future
       {ResponseString(s"Get all tasks" )}
      case GET(Path(Seg("tasks" :: id :: Nil))) => Future
       {ResponseString(s"Get a single task with id: $id" )}
      case DELETE(Path(Seg("tasks" :: id :: Nil))) => Future
       {ResponseString(s"Delete an existing task with id $id")}

      case req @ Path("/tasks")) => req match {
        case POST(_) =>
         handleCreateTask(req).map(ResponseString(_))
        case PUT(_) =>
           handleCreateTask(req).map(ResponseString(_))
        case _ => Future {Pass}
      }

      case _ => Future{Pass}
    }
  }
```

```
  @io.netty.channel.ChannelHandler.Sharable
  object fallback extends future.Plan with ServerErrorResponse {
    implicit def executionContext = ec
    def intent =  {
      case _ => Future {NotImplemented ~>
                ResponseString("Function is not implemented")}
    }
  }

  def handleCreateTask(req: HttpRequest[Any]): Future[String] = {
    val task = requestToTask(TaskService.nextTaskId(), req)
    val inserted = task map(TaskService.insert(_).map(_.toString))

    inserted.getOrElse(Future{"Error inserting"})
  }

  def paramExtractor(param: String): Extract[String] = {
     new Extract[String]( param,
                Params.first ~> Params.nonempty ~> Params.trimmed)
  }

  def requestToTask(id: Long, req: HttpRequest[Any])
                                          : Option[Task] = {
    val title = paramExtractor("title")

    req match {
      case Params(title(param)) => Some(Task(
        id, param, Body.string(req),
        None, List.empty[Note], Status("")))

      case _ => None
    }
  }

  unfiltered.netty.Server.http(8080)
         .handler(api)
         .handler(fallback).run
  dispatch.Http.shutdown()
}
```

The first part we'll look at more closely is how to convert an incoming request to a `Task` case class.

Converting a request to a Task class

The following code fragment shows how we can convert a request to a `Task` case class with Unfiltered:

```
def paramExtractor(param: String): Extract[String] = {
   new Extract[String]( param, Params.first ~> Params.nonempty)
}

def requestToTask(id: Long, req: HttpRequest[Any]): Option[Task] = {
  val title = paramExtractor("title")

  req match {
    case Params(title(param)) => Some(Task(
      id,
      param,
      Body.string(req),
      None,
      List.empty[Note],
    Status("")))

    case _ => None
  }
}
```

We defined a function named `requestToTask` which converts an incoming `HttpRequest` and an ID to a `Task`. The first step we take in this function is to create a custom extractor, based on a number of standard Unfiltered constructs:

```
def paramExtractor(param: String): Extract[String] = {
   new Extract[String]( param, Params.first ~> Params.nonempty)
}
```

In this function, we create a custom extractor by extending the `Extract` class provided by Unfiltered. This class has a constructor with two arguments. The first one is the name of the parameter we want to extract and the second is a function with the `Seq[String] => Option[T]` signature. The second parameter can be used to provide a set of predicates, which Unfiltered uses to check whether the value is available, and has the correct format. In this scenario, we used two predicates provided by Unfiltered. `Params.first` returns the first parameter value with the provided name or `None`, and the `Params.nonempty` returns `None` if the result from `Params.first` is empty, and returns `Some[String]` if it isn't. As you can see, we can use the `~>` operator to chain predicates together (this operator is just syntactic sugar for the `andThen` function).

We now can use this extractor in our pattern matching logic:

```
  req match {
    case Params(title(param)) => ...
```

This means that this pattern will only match if one of the provided parameters has the name `title` and

isn't empty. Out of the box Unfiltered provides the following constructs you can use when creating new matchers and extractors:

Predicate	Description
`even`	This returns the parameter value if it can be converted to an integer and the result is even.
`odd`	This returns the parameter value if it can be converted to an integer and the result is odd.
`nonempty`	This returns the parameter value if it isn't empty.
`first`	This returns the first value of a parameter. For example, in the case of `?id=10&id=20`, it will return `10`.
`int`	If the parameter value can be converted to an integer, it will return the integer value.
`long`	If the parameter value can be converted to a long, it will return the long value.
`float`	If the parameter value can be converted to a float, it will return the float value.
`double`	If the parameter value can be converted to a double, it will return the double value.
`trimmed`	This will use `String.trim` to trim the parameter value.
`~>`	This creates a sequence of predicates, for example, `first ~> nonempty ~> trimmed`.

At this point, we check whether we've got a non-empty title parameter, and if we do, we convert the incoming request to a `Task`.

Tip

Unfiltered also provides a way you can use multiple extractors at the same time. With the `&` matcher, you can combine two extractors. The pattern will only match if both sides of the `&` matched succeed. For instance, we could check whether the parameters contain a non-empty title and an integer amount like this:

```
val title = new Extract[String]( "title", Params.first ~> Params.nonempty)
val amount  = new Extract[String]( "amount", Params.first ~> Params.nonempty ~>
int)
```

And use it in a pattern, like this:

```
case Params(title(titleValue) & amount(amountValue))
```

We'll see more of this in the section on validation later in this chapter.

To finally convert the request into a `Task`, we just call the constructor directly, like this:

```
case Params(title(param)) => Some(Task(
  id,
  param,
  Body.string(req),
  None,
  List.empty[Note],
Status("")))
```

Now that we can convert a request to a `Task`, let's look at how we call this from our route and store it.

Storing a request in the TaskService

To store the request in the `TaskService`, we have to call the `TaskService.insert` function. We use the following code to do that:

```
case req @ Path("/tasks") => req match {
  case POST(_) => handleCreateTask(req).map(ResponseString(_))
  case PUT(_) => handleCreateTask(req).map(ResponseString(_))
  case _ => Future {Pass}
}

...

def handleCreateTask(req: HttpRequest[Any]): Future[String] = {
  val task = requestToTask(TaskService.nextTaskId(), req)
  val insertedTask = task
         map(TaskService.insert(_).map(_.toString))

  insertedTask.getOrElse(Future{"Error inserting"})
}
```

When we receive a `POST` or `PUT` request on `/tasks`, we pass the request on to the `handleCreateTask` function. In this function, we convert the request to a `Task` using the code we discussed previously, and store it with the `TaskService.insert` function. For now, we'll just return the `toString` of the created `Task` if we converted and stored the `Task` successfully. If something went wrong, we return a simple error message, also a string. We then return the created `Task` or the error message using the `ResponseString` function.

The `handleCreateTask` function returns a `Future[String]` object, so what we have to do is make sure that our Unfiltered configuration can handle futures.

Configuring Unfiltered to work with futures

Changing the configuration from a synchronous to an asynchronous model is very easy. The first thing we need to do is change the underlying server from Jetty to Netty. We need to do this because the asynchronous functionality is built on top of Netty, and thus, won't work with Jetty. To use Netty, we need to add the following two dependencies to our SBT configuration:

```
    "net.databinder" %% "unfiltered-netty" % "0.8.4"
    "net.databinder" %% "unfiltered-netty-server" % "0.8.4"
```

Next, we need to change the way we create our API:

```
  implicit def ec = ExecutionContext.Implicits.global

  @io.netty.channel.ChannelHandler.Sharable
  object api extends future.Plan with ServerErrorResponse {

    def executionContext = ec

    def intent =  ..

  }
```

Instead of extending from `unfiltered.filter.Plan`, we extend from `future.Plan` (with `ServerErrorResponse`, which you can use to handle exception in a generic way). If we work with futures, we also need to define the execution context we want to use. This is used by Unfiltered to run futures asynchronously. In this case, we just use the default global execution context. Note that we need to add the `io.netty.channel.ChannelHandler.Sharable` annotation to our API. If we don't do this, Netty will block any incoming requests, and only one thread can access the API at the same time. Since we move our service to Netty, we also need to start the server in a slightly different way:

```
  unfiltered.netty.Server.http(8080)
      .handler(api)
      .handler(fallback).run
  dispatch.Http.shutdown()
```

The last step we need to take to work with futures is to make sure that all our pattern match cases also return a `Future` object. We do this simply by wrapping the responses in a `Future`:

```
      case GET(Path("/tasks")) => Future
       {ResponseString(s"Get all tasks" )}
      case GET(Path(Seg("tasks" :: id :: Nil))) => Future
       {ResponseString(s"Get a single task with id: $id" )}
      case DELETE(Path(Seg("tasks" :: id :: Nil))) => Future
       {ResponseString(s"Delete an existing task with id $id")}

      case req @ Path("/tasks")) => req match {
        case POST(_) =>
         handleCreateTask(req).map(ResponseString(_))
        case PUT(_) => handleCreateTask(req).map(ResponseString(_))
```

```
      case _ => Future {Pass}
    }

    case _ => Future{Pass}
```

Most of the responses are explicitly wrapped in a `Future` object. For the `handleCreateTask` function, we already receive a `Future[String]` object, so just use `map` to convert it to the correct type using a `ResponseString` instance.

As for all the examples, we've also provided a set of sample requests you can use to test this REST API. You can find the requests in the **Chapter 03** collection in Postman. The most interesting requests to play around with are **Step 02 – CreateTask**, **Step 02 – Create Task – Invalid**, and **Step 02 – Trigger Fallback**.

Adding validation to parameter processing

So far, we haven't really validated the incoming request. We just checked whether a parameter was provided, and if not, completely failed the request. This works, but is a rather cumbersome way of validating input parameters. Luckily, Unfiltered provides an alternative way of validation by using something called directives. With a directive, you tell Unfiltered what you expect, for example, a parameter that can be converted to an int, and Unfiltered will either get the value or respond with an appropriate response message. In other words, with a directive, you define a set of criteria the request must fulfill.

Introducing directives

Before we look at how we can use a directive in our scenario, let's look at how you can use a directive in your code:

```
import unfiltered.directives._, Directives._

val intent = { Directive.Intent {
  case Path("/") =>
    for {
      _ <- Accepts.Json
      _ <- GET
      amount <- data.as.Option[Int].named("amount") } yield JsonContent ~>
ResponseString(
         """{ "response": "Ok" }""")
}
```

In this intent, we use `Directive.Intent` to indicate that we want to create an intent that uses directives to process the request. In this sample, `Accepts.Json`, `GET`, and `data.as.Option[Int].named("amount")` are all directives. When one of the directives fails, an appropriate error response is returned automatically. With directives, you can pretty much move your matcher and extractor logic to a set of directives.

Out of the box, Unfiltered comes with a number of directives, which automatically return a response when they don't match:

Directive	Description
`GET`, `POST`, and the other HTTP verbs	You can match on all the methods. If a method doesn't match, a `MethodNotAllowed` response is returned.
`Accepts.Json`	All the `Accepts.Accepting` definitions are supported. If one of these should fail, you get a `NotAcceptable` response.
`QueryParams`	Get all the query parameters from the request.
`Params`	You can check whether a single parameter is available. If not, a `BadRequest` response is returned.
`data.as`	The `data.as` directive allows you to get a parameter and convert it to a specific value. It provides standard directives to convert a parameter to: BigDecimal, BigInt, Double, Float, Int, Long, and String. Besides that, it also allows you to specify whether a value is an option or required.

Let's look a bit closer at the `data.as` directive since that is the most interesting one when trying to validate input. To use this directive, we first define an implicit function like this:

```
implicit val intValue =
  data.as.String ~> data.as.Int.fail { (k,v) =>
```

```
    BadRequest ~> ResponseString(
      s"'$v' is not a valid int for $k"
    )
}
```

And use it from the `for` comprehension like this:

```
 ...
 for {
  value <- data.as.Int.named("amount")
 } yield {...}
```

If the request parameter with the name `amount` is present and can be converted to an int, we'll retrieve that value. If it isn't present, nothing will happen, and if it can't be converted to an int, the specified `BadRequest` message will be returned. We can also make it `Optional` by just requesting `data.as.Option[Int]` instead. Having a value as an option is nice, but sometimes you want to make sure that a specific query parameter is always present. For that, we can use `Required`. To use `Required`, we first have to add another `implicit` function to define what happens when a required field isn't present:

```
implicit def required[T] = data.Requiring[T].fail(name =>
  BadRequest ~> ResponseString(name + " is missing\n")
)
```

This means that we return a `BadRequest` message with the specified response when a `Required` field is missing. To use the `Required` field, we simply change `data.as.Int` to `data.as.Required[Int]`:

```
...
 for {
  value <- data.as.Required[Int].named("amount")
 } yield {...}
```

Now, Unfiltered will first check if the field is present, and if it is, will check whether it can be converted to an int. When one of the checks fails, the correct response message will be returned.

Adding search functionality to our API

Now let's move on to our example. For this scenario, we'll add a `search` function to our API. This `search` function will allow you to search on the status and the text of a task, and return a list of tasks that match. Let's first look at the complete code before we dive into the individual parts:

```
package org.restwithscala.chapter3.steps

import org.restwithscala.common.model._
import org.restwithscala.common.service.TaskService
import unfiltered.directives.{Directive => UDirective, ResponseJoiner, data}
import unfiltered.request._
import unfiltered.response._
import unfiltered.netty._
import scala.concurrent.{ExecutionContext}
import scala.concurrent.Future
import scalaz._
import scalaz.std.scalaFuture._

object Step3 extends App {


  /**
   * Object holds all the implicit conversions used by Unfiltered to
   * process the incoming requests.
   */
  object Conversions {

  case class BadParam(msg: String) extends ResponseJoiner(msg)(
    msgs =>
      BadRequest ~> ResponseString(msgs.mkString("","\n","\n"))
  )


    implicit def requiredJoin[T] = data.Requiring[T].fail(name =>
      BadParam(name + " is missing")
    )

    implicit val toStringInterpreter = data.as.String

    val allowedStatus = Seq("new", "done", "progress")
    val inStatus = data.Conditional[String](
                        allowedStatus contains (_)).fail(
      (k, v) => BadParam(s" value not allowed: $v, should be one of
${allowedStatus.mkString(",")} ")
    )
  }


  Thread.currentThread().setName("swj");
  implicit def ec = ExecutionContext.Implicits.global

  // This plan contains the complete API. Works asynchronously
```

```
  // directives by default don't work with futures. Using the d2
  // directives, we can wrap the existing directives and use the
  // async plan.
  @io.netty.channel.ChannelHandler.Sharable
  object api extends async.Plan with ServerErrorResponse {

    // Import the required d2 directives so we can work
    // with futures and directives together. We also bring
    // the implicit directive conversions into scope.
    val D = d2.Directives[Future]
    import D._
    import D.ops._
    import Conversions._

    // maps the requests so that we can use directives with the
    // async intent. In this case we pass on the complete request
    // to the partial function
    val MappedAsyncIntent = d2.Async.Mapping[Any, HttpRequest[Any]] {
      case req: HttpRequest[Any] => req
    }

    // d2 provides a function to convert standard Unfiltered
    // directives to d2 directives. This implicit conversion
    // makes using this easier by adding a toD2 function to
    // the standard directives.
    implicit class toD2[T, L, R](s: UDirective[T, L, R]) {
      def toD2 = fromUnfilteredDirective(s)
    }

    // our plan requires an execution context,
    def executionContext = ec
    def intent = MappedAsyncIntent {
      case Path("/search") => handleSearchSingleError
    }

    def handleSearchSingleError = for {
      status <- inStatus.named("status").toD2
      text1 <- data.as.Required[String].named("text").toD2
      tasks <- TaskService
               .search(status.get,Some(text1)).successValue
    } yield {Ok ~> ResponseString(tasks.toString()) }
  }
  unfiltered.netty.Server.http(8080).handler(api).run
  dispatch.Http.shutdown()
}
```

Lots of code and some of it might look a bit strange. In the following sections, we'll see why we did it this way.

Directives and working with futures

Before we dive into the directives, you might notice some additional code regarding d2. For this example, we needed to use the `Directives2` library from https://github.com/shiplog/directives2/ so that we can correctly work combining futures and directives. The standard directives, as provided by Unfiltered, don't support asynchronous plans and only allow you to use the synchronous Jetty approach. With the `Directives2` directives, we can work with futures and use one of the available asynchronous plans provided by Unfiltered.

There is, however, some glue code required for this to get it to work. The following changes needed to be made to the previous step to make directives work nicely together with futures:

Move from `future.Plan` to `async.Plan`:

```
@io.netty.channel.ChannelHandler.Sharable
  object api extends async.Plan with ServerErrorResponse {
```

The d2 directives support the `async.Plan` class, but don't support the `future.Plan`. Luckily, this doesn't change the rest of the code for us. The next step is to import the d2 classes and objects:

```
    val D = d2.Directives[Future]
    import D._
    import D.ops._
    import Conversions._
```

With these imports, we get the ability to work with futures as directives, and it allows us to convert standard Unfiltered directives to d2 directives. The next step to do is to use the `d2.Async.Mapping` object to glue our new asynchronous directives to our `async.Plan`:

```
    val MappedAsyncIntent = d2.Async.Mapping[Any, HttpRequest[Any]] {
      case req: HttpRequest[Any] => req
    }
```

In this setup, we'll just pass any request we receive to a partial function, which we define like this:

```
    def intent = MappedAsyncIntent {
      case Path("/search") => handleSearchSingleError
    }
```

Now whenever we receive a request on the `/search` path, we pass it on to the `handleSearchSingleError` function. The final step we do is that we create a simple helper method to make converting our standard directives to d2 directives:

```
    implicit class toD2[T, L, R](s: UDirective[T, L, R]) {
      def toD2 = fromUnfilteredDirective(s)
    }
```

When this `implicit` is in scope, we can just call `toD2` on our normal directives so that they can work correctly with the ones from the d2 library.

Adding validation to the request parameters

Now that we've got the d2 stuff out of the way, let's look at the definition of our validations. We've defined all our directives implicitly in a `Conversions` object:

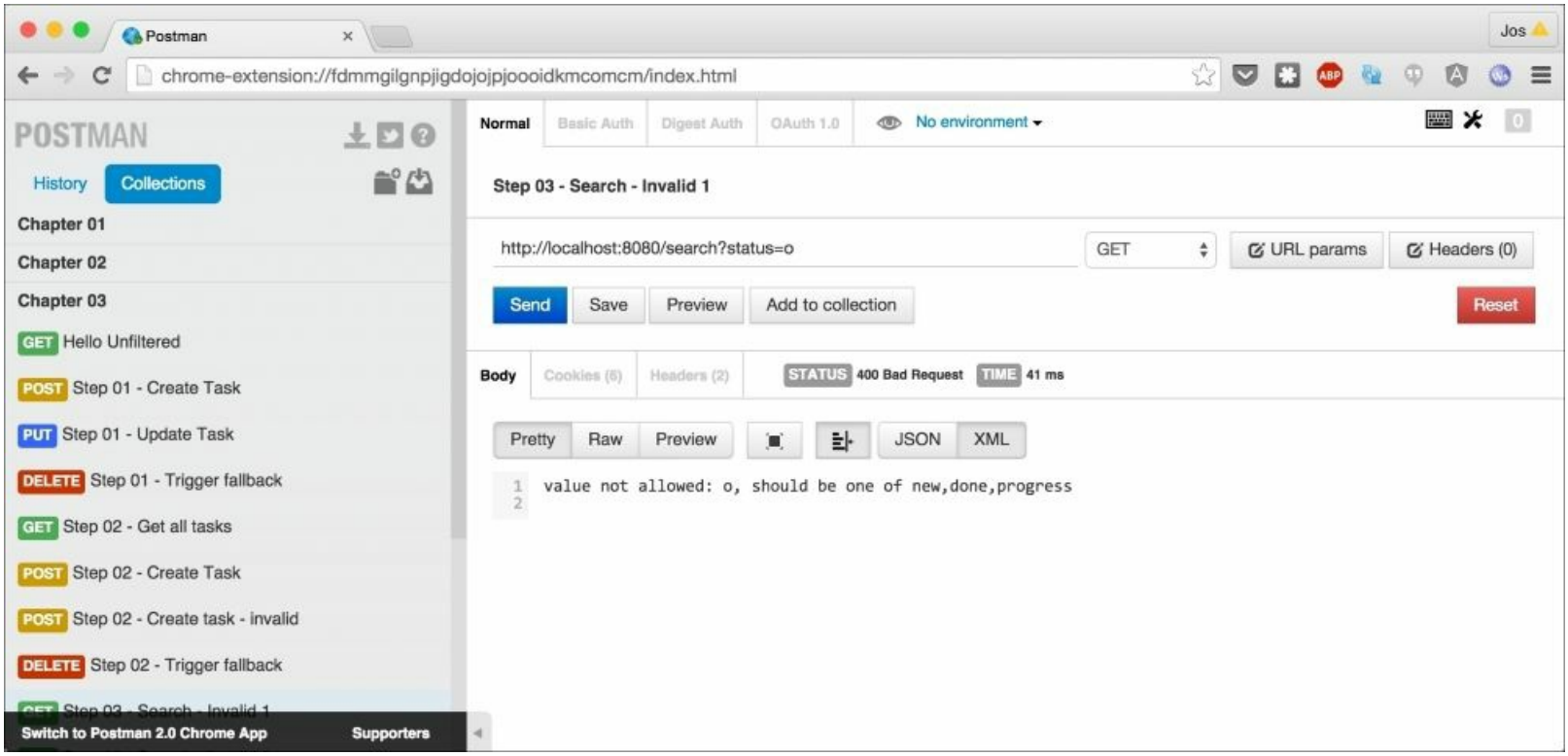
```
  object Conversions {

    case class BadParam(msg: String) extends ResponseJoiner(msg)(
      msgs =>
        BadRequest ~> ResponseString(msgs.mkString("","\n","\n"))
    )

    implicit def required[T] = data.Requiring[T].fail(name =>
      BadParam(name + " is missing")
    )

    implicit val toStringInterpreter = data.as.String

    val allowedStatus = Seq("new", "done", "progress")
    val inStatus = data.Conditional[String](
                        allowedStatus contains (_)).fail(
      (k, v) => BadParam(s" value not allowed: $v, should
                be one of ${allowedStatus.mkString(",")} ")
    )
  }
```

Here, we define three implicit values—`required` checks whether a value is present, `toStringInterpreter` tries to convert a parameter to a string, and `inStatus` checks whether a string is one of the specific set of values. If one of them fails, the `fail` function is called and an error is returned. Here, however, we don't directly return the error as an `HttpResponse`, but return it as a `BadParam` class. This `BadParam` case class serves as a collector of errors and allows a standard way to report one or more errors. In the next section, we'll come back to this. For now, we'll just report the first error we see. We do this by setting up a `for` comprehension like this:

```
    def handleSearchSingleError = for {
      status <- inStatus.named("status").toD2
      text1 <- data.as.Required[String].named("text").toD2
      tasks <- TaskService
               .search(status.get,Some(text1)).successValue
    } yield {Ok ~> ResponseString(tasks.toString()) }
```

This `for` comprehension works just like any normal one. First, we check whether the status query parameter is valid. If it is, we get the text value and then we use both these values to search through the `TaskService`, and finally return the result from the `TaskService` as a string. One remark here is the `successValue` function we call on the `Future` returned from the `TaskService`. This is a d2 specific call, which transforms the `Future` into a directive.

Let's open Postman and first make a request with an invalid status:

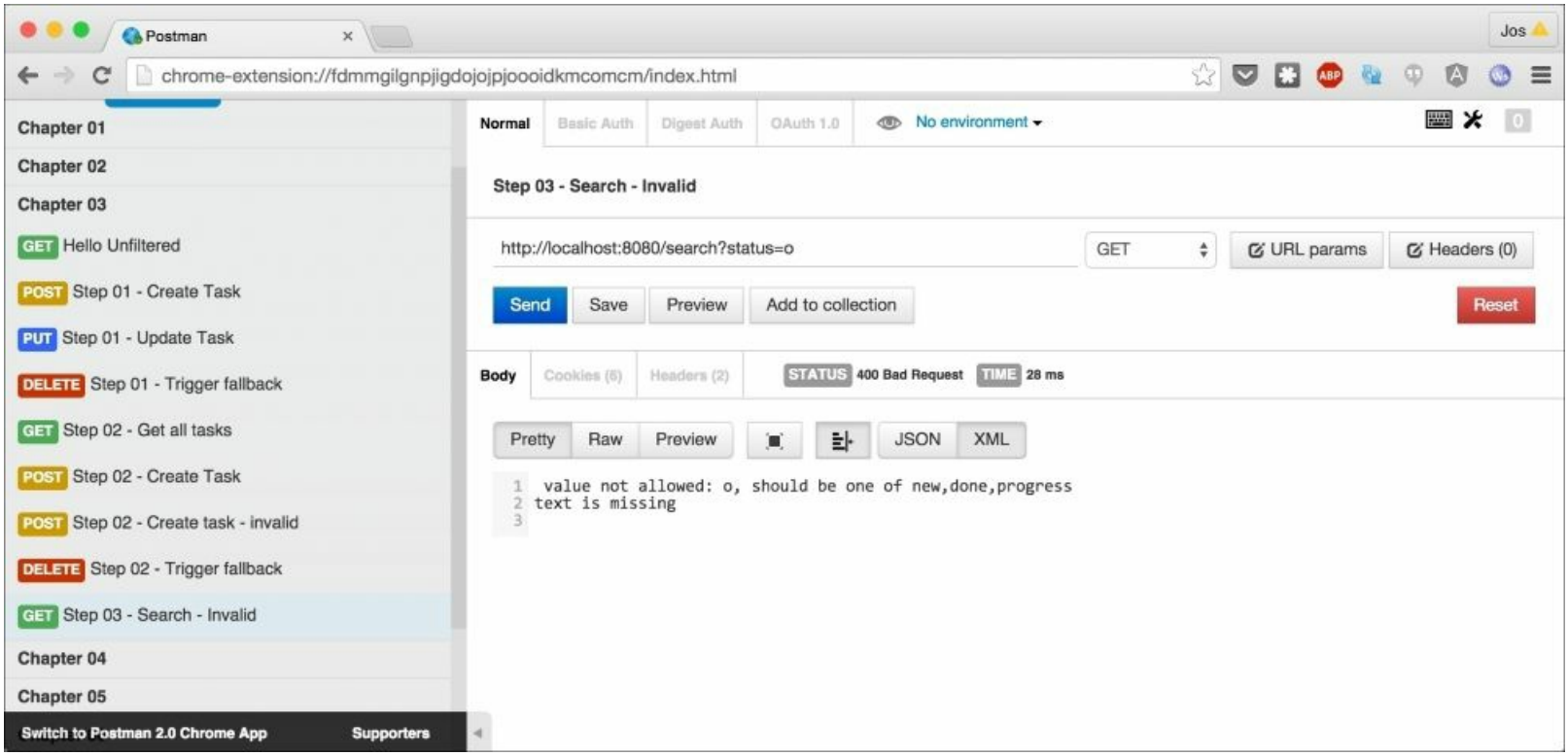


As you can see, the error message shows what we expect. However, as you might have noticed, we also didn't enter a text value, but the error message didn't show that. The reason is that our `for` comprehension stops at the first error.

Luckily, Unfiltered provides a way to combine errors. All we have to do is change our `for` loop to the following:

```
def handleSearchCombinedError = for {
  (status, text) <- (
         (data.as.Required[String].named("status")) &
         (data.as.Required[String].named("text"))
       ).toD2
  tasks <- TaskService.search(status,Some(text)).successValue
} yield {
    Ok ~> ResponseString(tasks.toString())
```

By using the `&` operator we can combine directives together. Now each part of the combined directives will log its errors using the `BadParam` case class, which will respond with all the errors it has collected. You can see how this works in the following screenshot:


Postman
Jos
chrome-extension://fdmmgilgnpjigdojojpjoooidkmcomcm/index.html
Chapter 01
Chapter 02
Chapter 03
GET Hello Unfiltered
POST Step 01 - Create Task
PUT Step 01 - Update Task
DELETE Step 01 - Trigger fallback
GET Step 02 - Get all tasks
POST Step 02 - Create Task
POST Step 02 - Create task - invalid
DELETE Step 02 - Trigger fallback
GET Step 03 - Search - Invalid
Chapter 04
Chapter 05
Switch to Postman 2.0 Chrome App
Supporters
Normal
Basic Auth
Digest Auth
OAuth 1.0
No environment
Step 03 - Search - Invalid
http://localhost:8080/search?status=o
GET
URL params
Headers (0)
Send
Save
Preview
Add to collection
Reset
Body
Cookies (6)
Headers (2)
STATUS 400 Bad Request
TIME 28 ms
Pretty
Raw
Preview
JSON
XML
1 value not allowed: o, should be one of new,done,progress
2 text is missing
3

Summary

In this chapter, we saw some of the most important aspects of Unfiltered. You learned how to handle requests, use matchers and extractors to route requests, and access parameters and path segments. You also learned that Unfiltered provides different processing models, synchronous and asynchronous, and how to run your service either on top of Jetty or on Netty. In the last section, we explored how directives in Unfiltered can be used to extract parameters in a more powerful way and the additional steps you need to take to use those in an asynchronous manner.

All in all, as you saw, Unfiltered is a very flexible framework, which is easily extensible, and gives you full control over the response-request cycle.

Chapter 4. An Easy REST Service Using Scalatra

In the previous chapters, you learned how to create REST services using frameworks that use a functional, Scala language-like approach. Finch used a very functional programming-based approach, and Unfiltered used pattern matching. In this chapter, we explore a Scala framework, Scalatra, whose main goal is simplicity.

In this chapter, we'll explain Scalatra's functionality using the following examples:

- **First Scalatra service**: We'll create a simple Scalatra service that shows how you can get up and running.
- **Verb and path handling**: Scalatra provides a number of constructs you can use to define a route. A route can match a specific HTTP verb and path and on a match, it will return a specific response.
- **Add support for futures and simple validation**: In its standard configuration, Scalatra works synchronously. In this part, we'll show you how you can add support for futures and also add some basic validation.
- **Convert to and from JSON and support advanced validations**: In the last part of this chapter, we'll look at JSON support and explain how Scalatra supports a more advanced way of validating incoming requests.

First, though, let's have a quick look at what Scalatra is, and what it aims to do.

Introduction to Scalatra

Scalatra is a small Scala web framework which tries to keep things simple. It provides a set of guidelines and helper classes to create complete web applications. In this chapter, we'll focus on the support it provides to create REST services.

Scalatra is built with a number of principles in mind (from the Scalatra home page):

> *Start small, build upwards: Start with a small core, and have lots of easy integrations for common tasks.*
>
> *Freedom: Allow the user freedom to choose whatever structure and libraries make the most sense for the application being built.*
>
> *Solid, but not stolid: Use solid base components. For instance, servlets aren't cool, but they are extremely stable and have a huge community behind them. At the same time, work to advance the state of web application development by using new techniques and approaches.*
>
> *Love HTTP: Embrace HTTP and its stateless nature. People get into trouble when they fool themselves into thinking things which aren't true - fancy server-side tricks to give an illusion of statefulness aren't for us.*

As you'll see in this chapter, the main goal of Scalatra is to keep things simple.

Your first Scalatra service

To get our first Scalatra service up and running, we need to take a couple of extra steps. The reason is that Scalatra is designed to run in a servlet container (for example, Tomcat or Jetty). While this works great for test and production environments, it doesn't allow us to do some quick prototyping or easily run from SBT or an IDE. Luckily, you can also start the Jetty servlet container programmatically and run your Scalatra service from there. So, in this section, we'll:

- Start by showing you the dependencies required to run Scalatra
- Setting up Jetty in such a way that it can run our Scalatra REST service
- Creating a simple Scalatra service that responds to a specific `GET` request

First, let's look at the dependencies for Scalatra. You can find these in the `Dependencies.scala` file in the `project` directory of the `sources` directory. For Scalatra (and Jetty), we define the following:

```
lazy val scalatraVersion = "2.3.0"
val backendDeps = Seq(
 "org.scalatra"      %% "scalatra"          % scalatraVersion,
 "ch.qos.logback"    %  "logback-classic"   % "1.1.3",
 "org.eclipse.jetty" %  "jetty-webapp"      % "9.2.10.v20150310"
)
```

The first dependency in this `Seq` pulls in all the required Scalatra libraries, the second one allows us to define how and what Jetty will log, and the final dependency is needed so that we can start Jetty programmatically from our project.

With these dependencies defined, we can create an embedded Jetty server, which we can use to serve our REST services. The code for this launcher can be found in the `chapter4/package.scala` file:

```
object JettyLauncher {

  def launch(bootstrapClass: String): Server = {

    // define the servlet context, point to our Scalatra servlet
    val context = new WebAppContext()
    context.setContextPath "/"
    context.setResourceBase("src/main/webapp")
    context.setInitParameter(ScalatraListener
                            .LifeCycleKey, bootstrapClass)
    context.addEventListener(new ScalatraListener)
    context.addServlet(classOf[DefaultServlet], "/")

    // create a server and attach the context
    val server = new Server(8080)
    server.setHandler(context)

    // add a lifecycle listener so to stop the server from console
    server.addLifeCycleListener(new AbstractLifeCycleListener() {
      override def lifeCycleStarted(event: LifeCycle): Unit = {
        println("Press <enter> to exit.")
```

```
          Console.in.read.toChar
          server.stop()
        }
      })

      // start and return the server
      server.start
      server.join
      server
    }
  }
```

We'll not dive too much into this code since it isn't really related to Scalatra. The main thing to understand here is that we've defined a function called `launch`, which takes the name of a bootstrap class as a parameter (more on this later) and that we've added a `ScalatraListener` instance using the `addEventListener` function. Once the Jetty server has finished starting up, the `ScalatraListener` will be called and start the Scalatra service using the provided `bootstrapClass`.

Now that we've created a way to launch our Scalatra service, let's look at the most basic example (source can be found in the `HelloScalatra.scala` file):

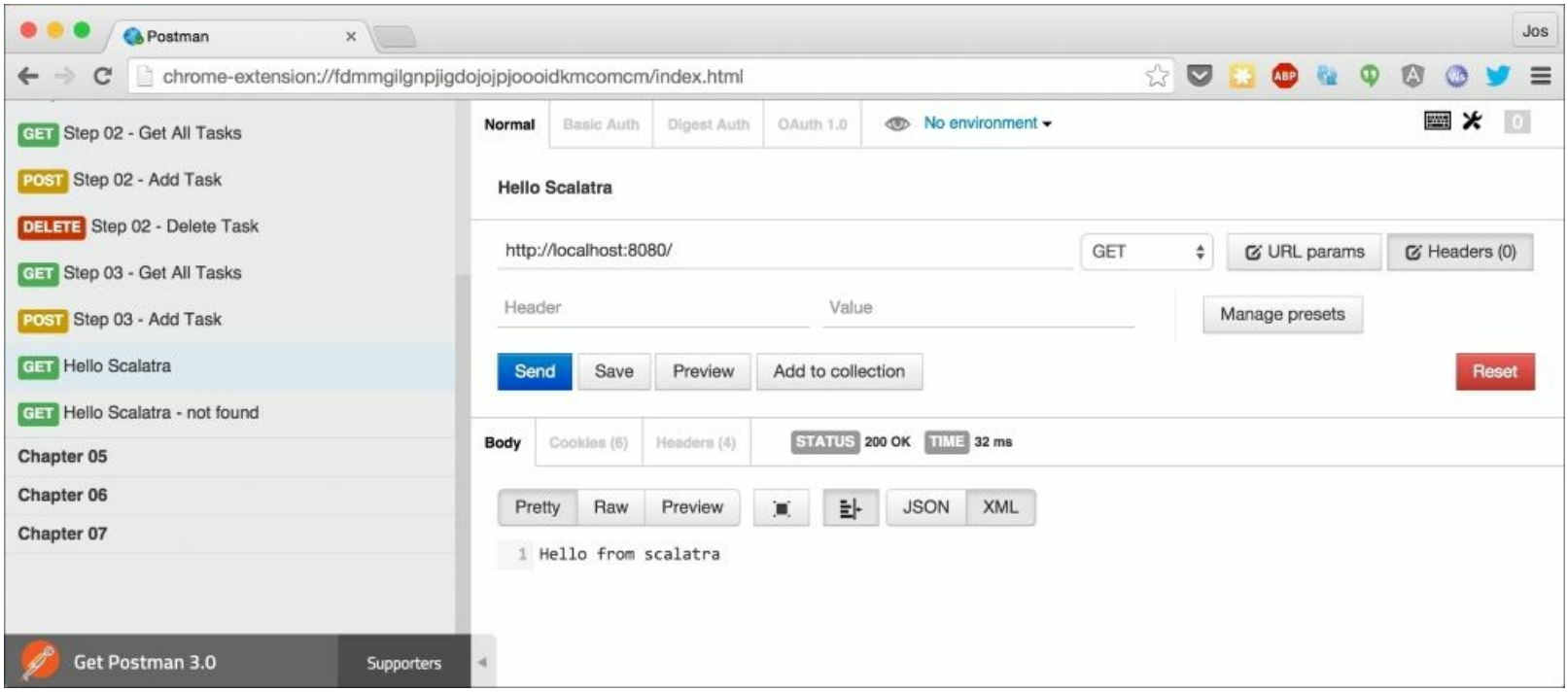
```
package org.restwithscala.chapter4.gettingstarted

import org.restwithscala.chapter4.JettyLauncher
import org.scalatra.{ScalatraServlet, LifeCycle}
import javax.servlet.ServletContext

// run this example by specifying the name of the bootstrap to use
object ScalatraRunner extends App {
  JettyLauncher.launch(
      "org.restwithscala.chapter4.gettingstarted.ScalatraBootstrap")
}

// used by jetty to mount the specified servlet
class ScalatraBootstrap extends LifeCycle {
  override def init(context: ServletContext) {
    context mount (new HelloScalatra, "/*")
  }
}

// the real servlet code
class HelloScalatra extends ScalatraServlet  {

  notFound {
    "Route not found"
  }

  get("/") {
    "Hello from scalatra"
  }
}
```

Let's walk through this file from top to bottom. At the top, we define an object called `ScalatraRunner`. With this object, we start our REST service by calling launch on the `JettyLauncher` we saw previously. We also pass the name of the `ScalatraBootstrap` class to the launcher so that the `ScalatraListener` we saw earlier can call the ScalatraBootstrap's `init` method when Jetty has finished starting up. In the `ScalatraBootstrap` class, we implement the `init` method and use that to instantiate our REST service (in this example, it is called `HelloScalatra`) and make it available to the outside world by calling mount. For each of the examples in this chapter, we'll use this same approach. In the `HelloScalatra` class, we finally see our REST service definition. In this case, we define a route which returns `Hello from scalatra` when it receives a `GET` request on the `/` path. If no route matches, the `notFound` function is triggered which returns a `404` message stating `route not found`.

All that is left to do is test these two scenarios. From the sources directory, run `sbt runCH04-HelloScalatra`. This should show an output similar to this:

```
$ sbt runCH04-HelloScalatra
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter4.gettingstarted.ScalatraRunner
20:42:09.020 [run-main-0] INFO  org.eclipse.jetty.util.log - Logging initialized
@31722ms
20:42:09.536 [run-main-0] INFO  org.eclipse.jetty.server.Server - jetty-
9.2.10.v20150310
20:42:09.940 [run-main-0] INFO  o.e.j.w.StandardDescriptorProcessor - NO JSP
Support for /, did not find org.eclipse.jetty.jsp.JettyJspServlet
20:42:10.015 [run-main-0] INFO  o.scalatra.servlet.ScalatraListener - The cycle
class name from the config:
org.restwithscala.chapter4.gettingstarted.ScalatraBootstrap
20:42:10.304 [run-main-0] INFO  o.scalatra.servlet.ScalatraListener -
Initializing life cycle class: ScalatraBootstrap
20:42:10.643 [run-main-0] INFO  o.e.j.server.handler.ContextHandler - Started
o.e.j.w.WebAppContext@78dac2c7{/,file:/Users/jos/dev/git/rest-with-
scala/src/main/webapp,AVAILABLE}
20:42:10.924 [run-main-0] INFO  o.e.jetty.server.ServerConnector - Started
ServerConnector@15f336ae{HTTP/1.1}{0.0.0.0:8080}
20:42:10.925 [run-main-0] INFO  org.eclipse.jetty.server.Server - Started
@33637ms
Press <enter> to exit.
```

At this point, we can press *Enter* to stop the server or fire up Postman and test our service. In Postman, you'll find a collection of requests for this chapter; let's just test the request (`hello scalatra`), which returns our `Hello from scalatra` message so that we know everything is working as it should be:



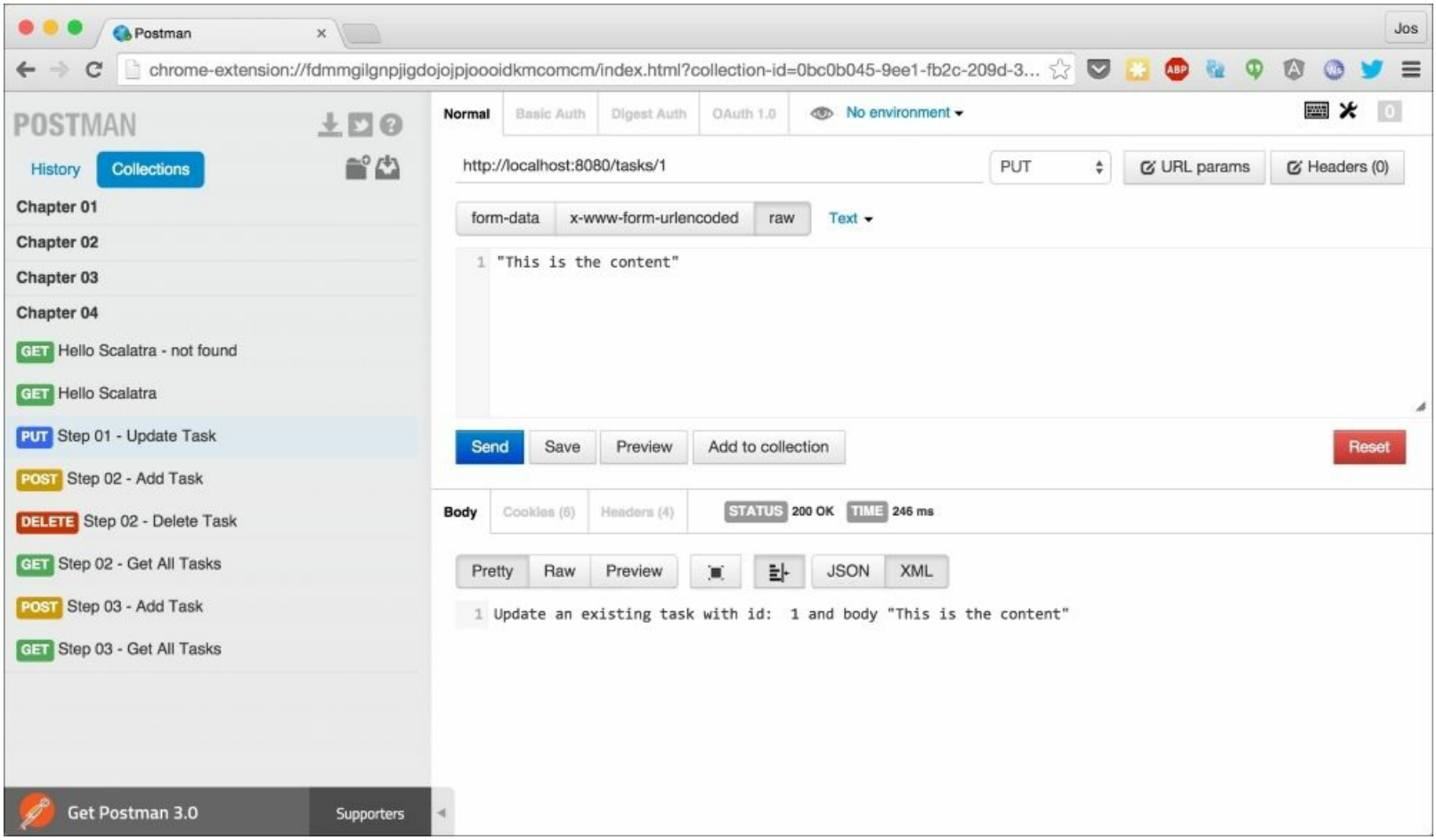
As we can see in the preceding screenshot, the response from Scalatra is as we expected, so our basic Scalatra setup is working correctly.

Verb and path handling

Now that we've got our basic Scalatra REST service running, let's look at a more elaborate example, which we'll use to explore some more features of Scalatra. Before we look at the code, let's make a request from Postman. First, start the server by calling `sbt runCH04-runCH04Step1` from the console, which shows something similar to this:

```
$ sbt runCH04-runCH04Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter4.steps.ScalatraRunnerStep1
10:51:40.313 [run-main-0] INFO  o.e.jetty.server.ServerConnector - Started
ServerConnector@538c2499{HTTP/1.1}{0.0.0.0:8080}
10:51:40.315 [run-main-0] INFO  org.eclipse.jetty.server.Server - Started
@23816ms
Press <enter> to exit.
```

Next, open up Postman and from the folder, `chapter-04`, select `request Step 01 - Update Task` and send it to the server. This request will simulate creating a new task and will respond by echoing some of the information it has received:



As you can see from the preceding screenshot, our server is up and running, and responds with a message containing the updated task. Next, we'll walk through the code of this Scalatra service:

```
package org.restwithscala.chapter4.steps

import javax.servlet.ServletContext

import org.restwithscala.chapter4.JettyLauncher
import org.scalatra.{NotFound, BadRequest, ScalatraServlet, LifeCycle}

import scala.util.{Failure, Success, Try}

// run this example by specifying the name of the bootstrap to use
object ScalatraRunnerStep1 extends App {
  JettyLauncher.launch(
    "org.restwithscala.chapter4.steps.ScalatraBootstrapStep1")
}

class ScalatraBootstrapStep1 extends LifeCycle {
  override def init(context: ServletContext) {
    context mount (new ScalatraStep1, "/*")
  }
}

class ScalatraStep1 extends ScalatraServlet  {

  notFound { "Route not implemented" }

  post("/tasks") { s"create a new task with body ${request.body}"}
  get("/tasks") { "Get all the tasks" }
  get("/tasks/:id") {
    Try { params("id").toInt }  match {
      case Success(id) => s"Get task with id: ${params("id")} "
      case Failure(e) => BadRequest(reason = "Can't parse id")
    }
  }
  delete("/tasks/:id") { s"Delete task with id: ${params("id")} "}
  put("/tasks/:id") { s"Update an existing task " +
    s"with id:  ${params("id")} " +
    s"and body ${request.body}"}
}
```

As we mentioned in the previous section, we need to call `JettyLauncher` and define a `ScalatraBootstrapStep1` class to run our service. The interesting code in this example is located in the `ScalatraStep1` class, which defines a number of routes that we can call from our REST client.

Tip

When you look at the routes in this chapter, you'll notice that the more generic routes are at the top of the `ScalatraServlet` class. The reason is that Scalatra tries to match incoming requests starting from the bottom and from there, it moves up. So when defining routes, take care that you place the most generic routes at the top and the most specific ones at the bottom.

In the `ScalatraStep1` class, we define a number of routes. Let's look at a couple of these:

```
  get("/tasks/:id") {
    Try { params("id").toInt }  match {
      case Success(id) => s"Get task with id: ${params("id")} "
      case Failure(e) => BadRequest(reason = "Can't parse id")
    }
  }
  ...
  put("tasks/:id") { s"Update an existing task " +
    s"with id:  ${params("id")} " +
    s"and body ${request.body}"}
```

Here we see two routes. The `get("/tasks/:id")` matches `GET` HTTP requests made on a URL, which looks similar to `/tasks/12`. When the request is made, we use the `params` function to get the value of the path segment and try and convert it to an integer. If this is successful, we just return a string, and if not, we return the HTTP error, `BadRequest`. The `put("tasks/:id")` route matches `PUT` HTTP requests and always returns a string, which contains the provided ID using the `params` function, and also shows the sent message body that can be accessed through the `request.body` value. Besides the `params` function, Scalatra also provides a `multiParams` function. With the `multiParams` function, you don't get a single string, but a `Seq[String]` instance. This is especially useful if you want to access request parameters that have multiple values. For instance, if we match on `/task/search?status=new,old`, we can get a `Seq[String]` containing new and old by calling `multiParams("status")`.

Besides matching directly on a path element, Scalatra also supports a number of other ways to match an HTTP request. The following table shows how you can match specific HTTP verbs and routes in Scalatra:

Construct	**Description**
`get("/")`, `post("/")`, `put("/")`, `delete("/")` and the other HTTP verbs.	Scalatra allows you to match a specific HTTP verb. Scalatra supports the following HTTP verbs, which you can directly use in your route definition: `options`, `get`, `head`, `post`, `put`, `delete`, `trace`, `connect`, and `patch`.
`get("/full/path")`	The most basic way to match a request on a specific path is by specifying the full path to match on. This will only match if the provided path matches exactly. In this case, `"/full/path"` will match, while `"/full/path/something"` won't.
`get("/path/:param")`	As we've seen in the examples, you can also extract variables from the path by prefixing a path segment with a `:`. This will match paths such as `"/path/12"` and `"path/hello"`.
`get("""^\/tasks\/(.*)""".r)`	You can also use a regular expression to match a specific path in Scalatra. To access the groups that match, you can either use the `params("splat")` call or through the `multiParams("splat")` function call.
`get("/*/*")`	Scalatra also support the use of wildcards. You can access the parameters that match either through the `params("splat")` call or through the `multiParams("splat")` function call.

`get("/tasks", condition1, condition2)`	You can further fine-tune the matching by supplying conditions. A condition is a function that returns either `True` or `False`. If all conditions return `True`, the route matches, and is executed. For example, you can use something similar to `post("/tasks", request.getHeader("headername") == "headerValue")`.

Before we move on to the next section, let's have a quick look at how you can access all the attributes of a request. So far, we've seen `params`, `multiParams`, and `request.body` to access certain parts. Scalatra also exposes the other parts of a request. The following table shows a complete overview of the helper functions and request properties exposed by Scalatra (note that you can easily use these request properties as conditions in your routes):

Function	**Description**
`requestPath`	This returns the path against which the route is matched.
`multiParams(key)`	This returns the value of a request parameter (or match path segment) as a `Seq[String]`.
`Params(key)`	This returns the value of a request parameter (or match path segment) as a string.
`request.serverProtocol`	This returns an `HttpVersion` object, which is either `Http11` or `Http10`.
`request.uri`	This is the URI of the request as a `java.net.URI`.
`request.urlScheme`	This returns a `Scheme` object, either `Http` or `Https`.
`request.requestMethod`	This returns an `HttpMethod`, for example, `Get`, `Post`, or `Put`.
`request.pathInfo`	This returns the path info from the request or an empty string if no pathinfo is available.
`Request.scriptName`	This returns the servlet path part of the request as a string.
`Request.queryString`	This returns the query string of the request or an empty string if no query string is present.
`Request.multiParameters`	This returns a map of all the parameters of this request as a `MultiMap`. This contains the parameters from the query string and any posted form data.
`request.headers`	This returns all the headers as a `Map[String, String]` object.
`request.header(key)`	This gets a specific header from the request and returns an `Option[String]`.
`request.characterEncoding`	This returns, if present, the character encoding of the request and an `Option[String]`.

`request.contentType`	This gets the content-type of the request if present and returns an `Option[String]`.
`request.contentLength`	This gets the length of the content and returns an `Option[Long]`.
`request.serverName`	This returns the server name part of the complete path and a string.
`request.serverPort`	This returns the port of the server as an integer.
`request.referrer`	This tries to get the referrer from the request and returns an `Option[String]`.
`request.body`	This returns the body of the request as a string.
`request.isAjax`	This checks whether the request is an AJAX request. It does this by checking the presence of an `X-Requested-With` header.
`request.isWrite`	Checks whether the request is not safe (see RFC 2616).
`request.multiCookie`	This returns a map of all the cookies of this request as a `MultiMap`.
`request.cookies`	This returns all the cookies as a `Map[String, String]`.
`request.inputStream`	This gets the `inputStream` of the request, which can be used to read the body. Note that this `inputStream` is already consumed when you call `request.body`.
`request.remoteAddress`	This tries to get the clients IP address and returns it as a string.
`request.locale`	This returns the `Locale` value from the request.

As you can see, Scalatra wraps all the normal request properties and attributes you'd expect and makes them easily available either through some helper functions or as properties on the available request value.

Now that we've explored the basic functionality of Scalatra and seen how we can match HTTP verbs and paths, we'll look at some more advanced features in the next section.

Add support for futures and simple validation

In this section, we'll add support for futures to Scalatra and show a couple of first steps to validate incoming requests. To work asynchronously, Scalatra needs some additional dependencies. The complete list of dependencies required for this examples are the following:

```
  lazy val scalatraVersion = "2.3.0"
  val backendDeps = Seq(
      "org.scalatra" %% "scalatra" % scalatraVersion,
      "ch.qos.logback"% "logback-classic" % "1.1.3",
      "org.eclipse.jetty" %  "jetty-webapp" % "9.2.10.v20150310",
      "com.typesafe.akka" %% "akka-actor" % "2.3.4"
```

As you can see from the dependencies, Scalatra uses Akka (http://akka.io) to handle requests asynchronously. However, you don't need to know much about Akka to get everything up and running. In the following code fragment, we show you the basic glue that is required to connect all the moving parts:

```
package org.restwithscala.chapter4.steps

import javax.servlet.ServletContext

import akka.actor.ActorSystem
import org.restwithscala.chapter4.JettyLauncher
import org.restwithscala.common.model.{Status, Task}
import org.restwithscala.common.service.TaskService
import org.scalatra._
import org.slf4j.LoggerFactory

import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Failure, Success, Try}


object ScalatraRunnerStep2 extends App {
  JettyLauncher.launch(
       "org.restwithscala.chapter4.steps.ScalatraBootstrapStep2")
}

class ScalatraBootstrapStep2 extends LifeCycle {

  val system = ActorSystem()

  override def init(context: ServletContext) {
    context.mount(new ScalatraStep2(system), "/*")
  }

  override def destroy(context: ServletContext) {
    system.shutdown()
  }
}
```

```
class ScalatraStep2(system: ActorSystem) extends ScalatraServlet
                                                  with FutureSupport {

  protected implicit def executor: ExecutionContext
                                       = system.dispatcher
  val Log = LoggerFactory.getLogger(this.getClass)

  ...
}
```

In this code fragment, we use the JettyLauncher we've already seen to start the Jetty server and specify the Scalatra bootstrap class that we want to start when Jetty is started. In the bootstrap for this example, we take a couple of additional steps:

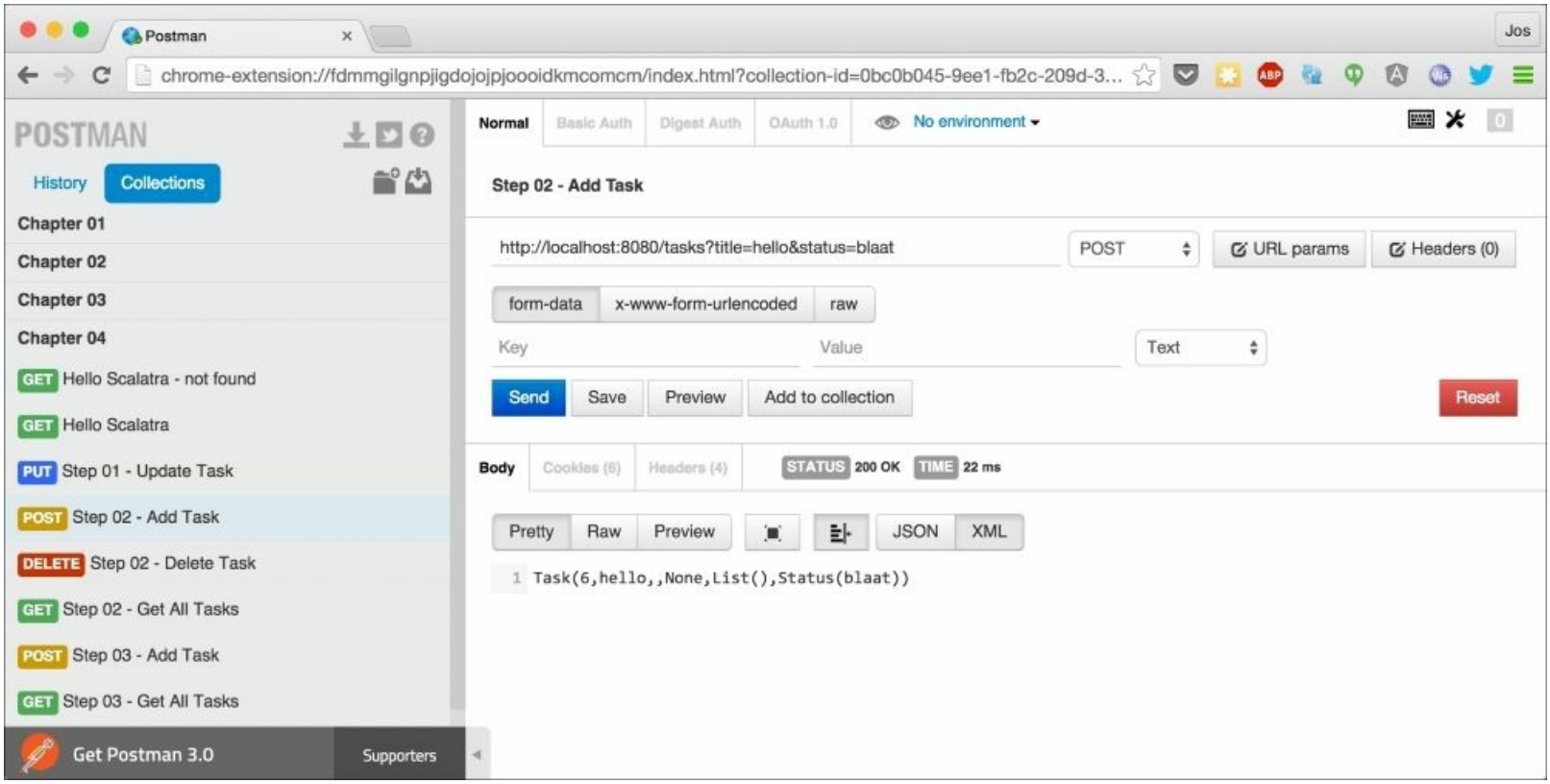
```
class ScalatraBootstrapStep2 extends LifeCycle {

  val system = ActorSystem()

  override def init(context: ServletContext) {
    context.mount(new ScalatraStep2(system), "/*")
  }

  override def destroy(context: ServletContext) {
    system.shutdown()
  }
}
```

When this class is instantiated, we create a new Akka, `ActorSystem`, which is required by Akka. We pass this system to the constructor of our Scalatra route (`ScalatraStep2`) so that we can use it from there. In this bootstrap class, we also override the `destroy` function. When the Jetty server is shut down, this will neatly close the `ActorSystem` and clean up any open resources.

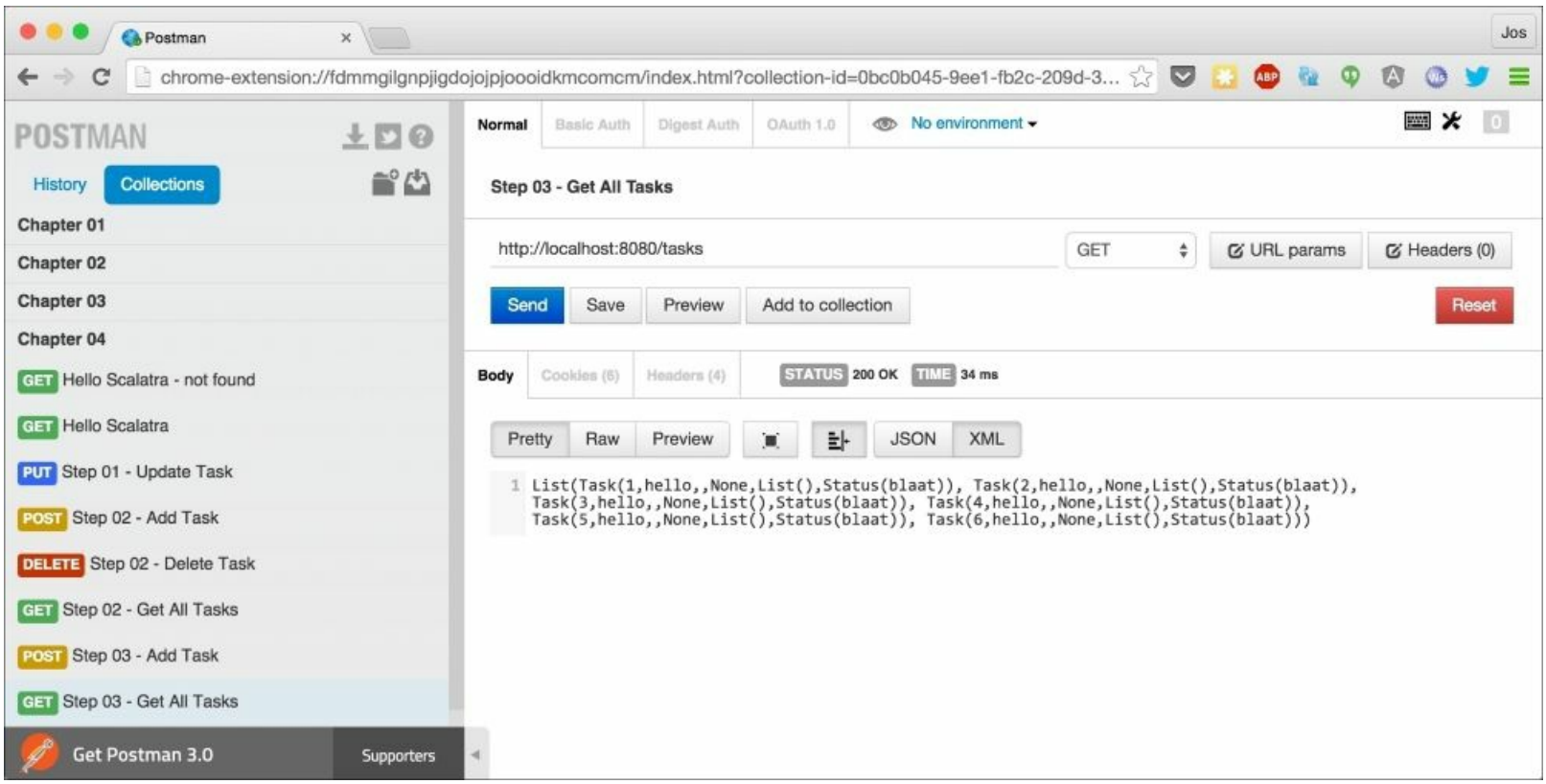
Before we look at the code that handles our routes, we'll first make some calls from Postman to better understand what our routes need to do. So, start the server for this part with `sbt runCH04-runCH04Step2`:

```
$ sbt runCH04-runCH04Step2
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter4.steps.ScalatraRunnerStep2
17:46:00.339 [run-main-0] INFO  org.eclipse.jetty.util.log - Logging initialized
@19201ms
17:46:00.516 [run-main-0] INFO  org.eclipse.jetty.server.Server - jetty-
9.2.10.v20150310
17:46:01.572 [run-main-0] INFO  o.e.jetty.server.ServerConnector - Started
ServerConnector@5c3c276c{HTTP/1.1}{0.0.0.0:8080}
17:46:01.572 [run-main-0] INFO  org.eclipse.jetty.server.Server - Started
@20436ms
Press <enter> to exit.
```

Now create a number of tasks by using the **Step 02 – Create Task** request from this chapter:



To get an overview of all the tasks that have been stored, use the **Step 02 – Get All Tasks** request:



As you can see from the two screenshots, this time we store the tasks, and you can retrieve them through other REST calls. The code for these two routes, and a route to delete a task based on an ID, is shown next:

```
class ScalatraStep2(system: ActorSystem) extends ScalatraServlet
                                                with FutureSupport {

  protected implicit def executor: ExecutionContext
                                        = system.dispatcher
  val Log = LoggerFactory.getLogger(this.getClass)

  before("/*") {
    Log.info(s"Processing request for: ${params("splat")}")
  }

  after("""^\/tasks\/(.*)""".r) {
    Log.info(s"Processed request for tasks: ${params("captures")}")
  }

  notFound {
    "Route not implemented"
  }

  post("/tasks") {
    new AsyncResult() {
      // we use a AsyncResult since we access the parameters
      override val is  = {
        // convert provided request parameters to a task and store it
        val createdTask = TaskService.insert(Task( -1,
          params.getOrElse("title",
                          halt(status = 400,
                               reason="Title is required")), request.body, None,
List.empty, Status(params.getOrElse("status", "new"))))

        // the result is a Future[Task]; map this to a string
        createdTask.map(_.toString)
      }
    }
  }

  get("/tasks") {
      // directly return Future since we don't access request
      TaskService.all.map(_.toString) }

  delete("/tasks/:id") {
    new AsyncResult() {
      override val is = Try { params("id").toLong } match {
        case Success(id) => TaskService.delete(id).map(_.toString)
        case Failure(e) => Future{BadRequest(
                        reason = s"Can't parse id: ${e.getMessage}")}
      }
    }
  }
}
```

Let's walk through this code fragment and see what is happening. Let's start with the definition of the class that holds the routes and the first statement inside the class:

```
class ScalatraStep2(system: ActorSystem) extends ScalatraServlet
                                              with FutureSupport {

  protected implicit def executor: ExecutionContext
                                     = system.dispatcher
```

This time, besides extending from `ScalatraServlet`, we also mix in the `FutureSupport` trait. When we mix in this trait, we add support for futures to Scalatra. This trait also requires us to define an `ExecutionContext` instance. In this example, we use the default, `ExecutionContext`, provided by Akka. You can, of course, also define and configure one yourself.

Tip

An `ExecutionContext` is used by programs to execute logic asynchronously and for the developer to gain a finer control of threading. You could, for instance, execute a piece of code by passing a `Runnable` instance to the `execute` method. Scalatra and Akka hide all the details of how the `ExecutionContext` is used, but it is up to the developer to specify which `ExecutionContext` to use and how it is configured.

Now that we've configured the last part, we can look at running calls asynchronously. The first route we'll look at is the `get("/tasks")` route:

```
  get("/tasks") {
      // directly return future since we don't access request
      TaskService.all.map(_.toString) }
```

This route is very simple. It calls the `TaskService.all` function, which returns a `Future[List[Task]]` and we transform this to a `Future[String]` through the `map` function. Internally, Scalatra will run this request on top of Akka, and wait, non-blocking, for the `Future` to finish. Once it is finished, Scalatra will return the string to the client. The great thing is that you don't have to do anything yourself. Just return a `Future` object, and Scalatra knows how to handle everything since we added `FutureSupport`. In this sample, we just return a string inside the `Future`. Scalatra also supports a number of other return types you can use:

Type	Description
`ActionResult`	An `ActionResult` type is a class where you can set the status, body, and headers which are returned. Scalatra comes with a large number of standard `ActionResult` you can use—`OK`, `Created`, `Accepted`, and so on. For a complete overview, look at the source for the `ActionResult.scala` file.
`Array[Byte]`	The content-type of the response (if not set) is set to application/octet-stream, and the byte array is returned.
`NodeSeq`	The content-type of the response (if not set) is set to `text/HTML` and the `NodeSeq` instance is converted to a string and is returned.

`Unit`	If you don't specify anything, Scalatra assumes you've set the correct values in the response object yourself.
`Any`	The content-type of the response (if not set) is set to `text/plain` and the `toString` method is called on the object and the result is returned.

Note that you can override this functionality, or add new functionality, by overriding the `renderResponse` function.

Now let's look at the `delete("/tasks/:id")` route:

```
  delete("/tasks/:id") {
    new AsyncResult() {
      override val is = Try { params("id").toLong } match {
        case Success(id) => TaskService.delete(id).map(_.toString)
        case Failure(e) => Future{BadRequest(
                        reason = s"Can't parse id: ${e.getMessage}")}
      }
    }
  }
```

The `TaskService.delete` service returns a `Future[Option[Task]]`, which we transform to a `Future[String]`, just like we did in the previous code fragment. The main difference here is that we don't directly return a `Future`, but wrap the block inside an `AsyncResult` object. The reason why we need to do this is that we access values from the request. We use `params("id")` to get the value from the URL path. If you access any information from the request, you need to wrap it in an `AsyncResult` object to avoid timing issues and strange exceptions.

Tip

When working with futures or when adding new complex routes, it is often practical to add some logging to see what is happening before and after the request has been processed. Scalatra supports this through the `before("path")` and `after("path")` functions you can define in your class. In this example, we, for instance, log every request by specifying a `before(/*)` function and log some additional information afterward for the request made to a specific path defined by a regular expression: `after("""^\/tasks\/(.*)""".r)`.

Now let's move on to some simple validations. Look at the following code from the `POST` method:

```
       val createdTask = TaskService.insert(Task( -1,
          params.getOrElse("title",
                          halt(status = 400,
                               reason="Title is required")), request.body, None,
List.empty, Status(params.getOrElse("status", "new"))))

          // the result is a Future[Task]; map this to a string
          createdTask.map(_.toString)
      }
```

What you see here is that we can use `getOrElse` on the parameters to check whether it is provided, and if not, we can either throw an error or add a default value.

Tip

Note that we've used a special construct here from Scalatra—`halt`. When this function is called, Scalatra will immediately stop processing the request and return the specified HTTP response. Besides halt, Scalatra also provides a `pass` function which can be used to stop processing inside the current route and instead try to see if there are other routes that might match.

This is just some basic simple validation. In the next section, we'll look at the more advanced ways of adding validation.

Advanced validation and JSON support

For the final example, we're going to add JSON support to the service and some more advanced validations. To test the examples in this section, run `sbt runCH04-runCH04Step3`.

Add JSON support

Let's start by adding JSON support. First off, add the following dependencies to the SBT build file:

```
"org.json4s"   %% "json4s-jackson" % "3.2.9",
"org.scalatra" %% "scalatra-json" % scalatraVersion,
```

Adding JSON support only takes a couple of simple steps. First, change the class definition of our route to the following:

```
class ScalatraStep3(system: ActorSystem) extends ScalatraServlet
                  with FutureSupport
                  with JacksonJsonSupport {
```

With the `JacksonJsonSupport` trait added, we next need to add the following line to enable automatic JSON parsing:

```
protected implicit val jsonFormats: Formats = DefaultFormats
```

Now we only need to inform the routes that we want to work with JSON. For this, we use the `before()` function, where we set the following:

```
  before("/*") {
    contentType = formats("json")
}
```

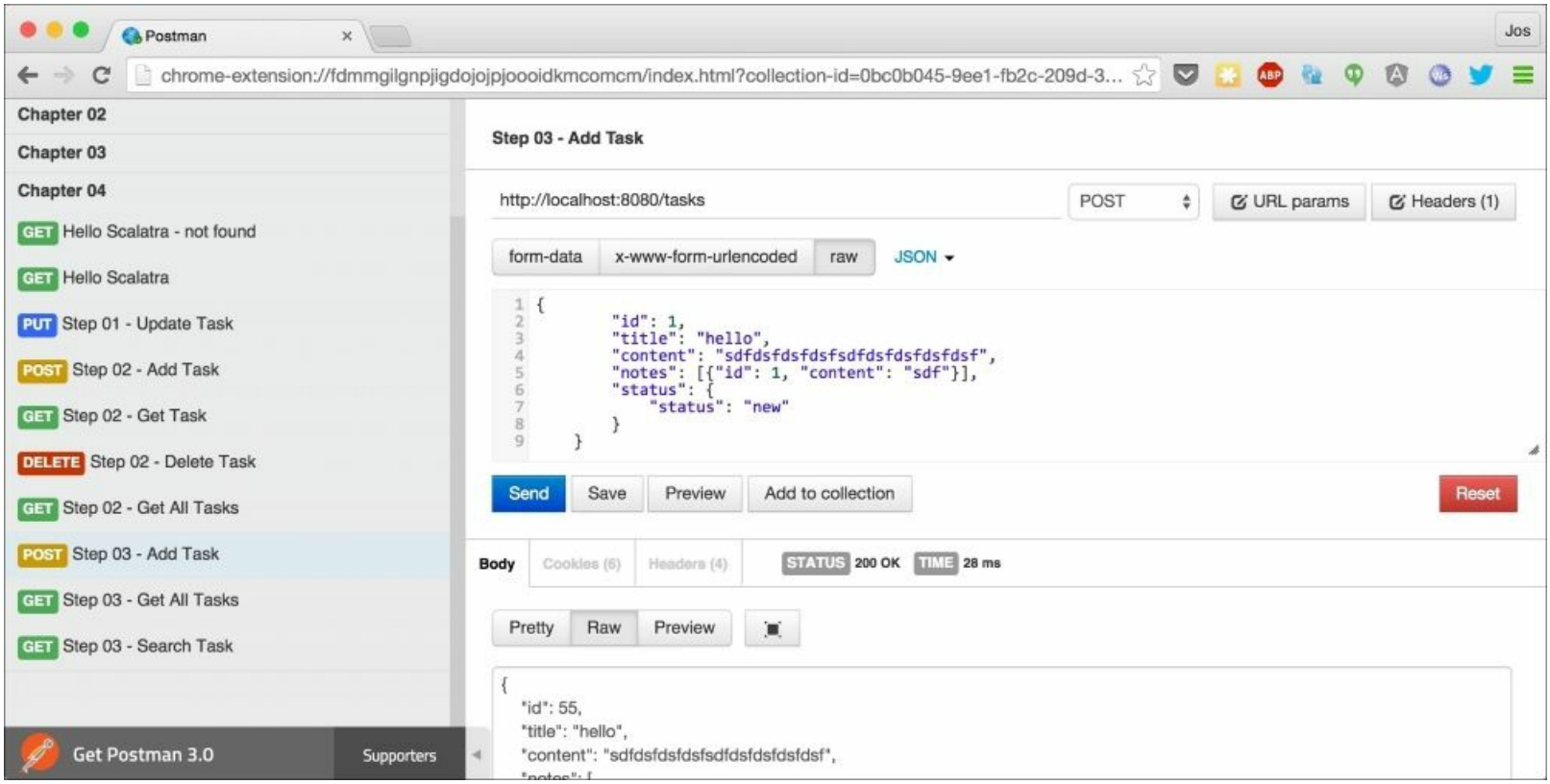
At this point, we can just return our case classes, and Scalatra will automatically convert them to JSON. For instance, for the get all tasks service, it looks similar to this:

```
  get("/tasks") {
      TaskService.all // we use json4s serialization to json.
  }
```

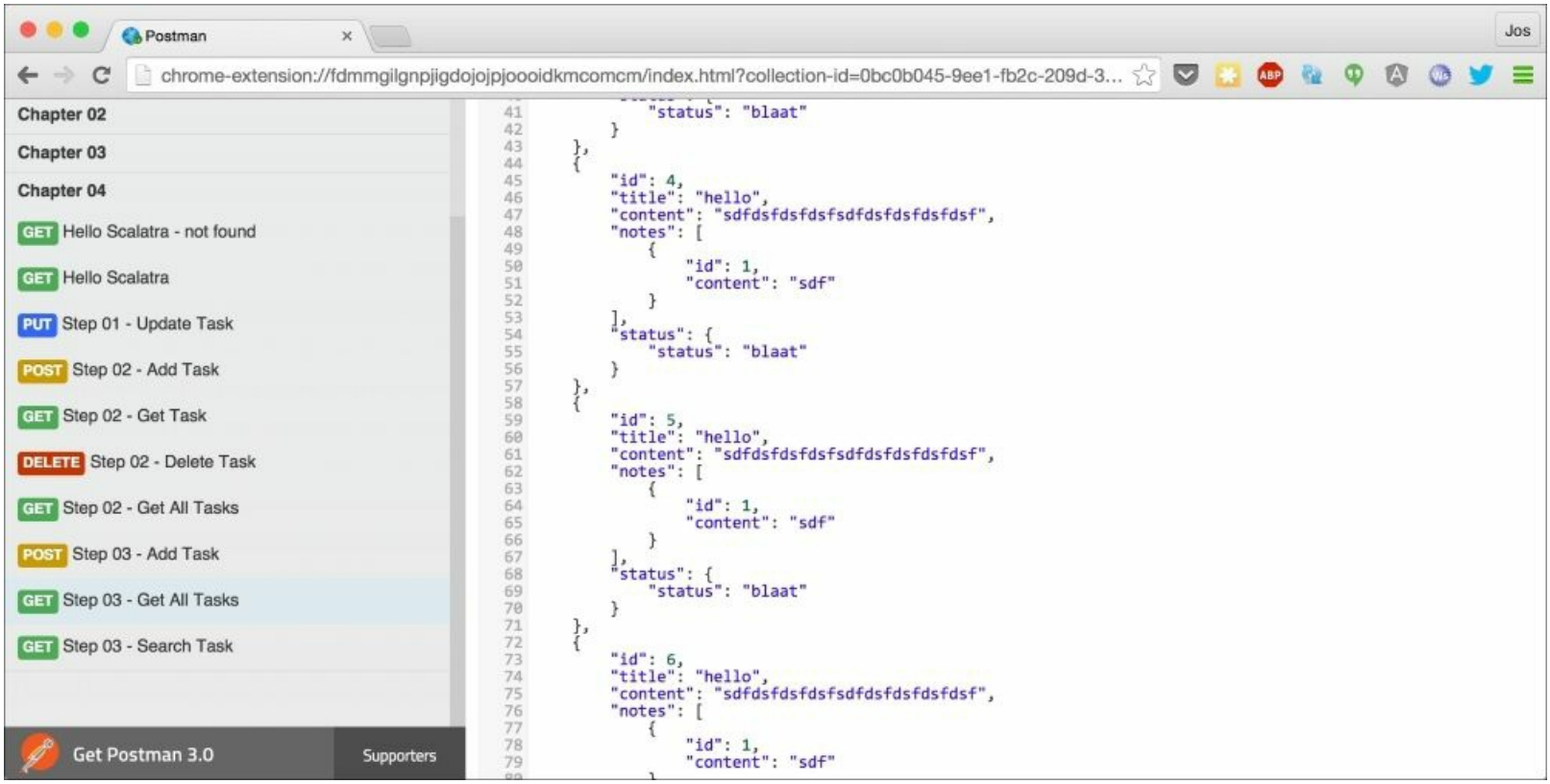
Before we look at Postman and see the requests, we need to take one final step so that we can also store the incoming JSON messages. Let's look at the `post("/tasks")` function:

```
  post("/tasks") {
    new AsyncResult() {
      override val is  = {
        // convert provided request parameters to a task and store it
        TaskService.insert(parsedBody.extract[Task])
      }
    }
  }
```

Let's see this in action. Open Postman and, using the **Step 03 – Add Task** request, add some tasks:



As you can see, the body we sent is a JSON message describing the task. Retrieving the messages works in pretty much the same way. In Postman, you can use the **Step 03 – Get All Tasks** request for this:



And here, you can see that the tasks you just added are returned.

Advanced validations

Now that we've got JSON support added, let's look at the final part of this chapter and explore how you can add more advanced validations to your Scalatra routes. The first thing we need to do is add one more dependency to our `sbt` build file (`scalatra-commands`). At this point, our dependencies should look similar to this:

```
lazy val scalatraVersion = "2.3.0"
val backendDeps = Seq(
    "org.scalatra" %% "scalatra" % scalatraVersion,
    "org.scalatra" %% "scalatra-json" % scalatraVersion,
    "org.scalatra" %% "scalatra-commands" % scalatraVersion,
    "org.json4s"   %% "json4s-jackson" % "3.2.9",
    "ch.qos.logback"    %  "logback-classic"   % "1.1.3",
    "org.eclipse.jetty" %  "jetty-webapp"      % "9.2.10.v20150310",
    "com.typesafe.akka" %% "akka-actor" % "2.3.4"
)
```

Through the use of commands, we can add more complex validations to our input parameters. For this, we need to change a couple of things in our route. The first thing we need to do is add `CommandSupport` to our route:

```
class ScalatraStep3(system: ActorSystem) extends ScalatraServlet
                  with FutureSupport
                  with JacksonJsonSupport with CommandSupport {
```

This allows us to process incoming requests using commands. Next, we need to specify which types of command our service should process. Since we only use a single command in this case, we set the `CommandType` to `SearchTaskCommand` (more on this file later):

```
override type CommandType = SearchTasksCommand
```

Let's look a bit closer at the commands and validations we'll use in this example:

```
object SearchCommands {

  object SearchTasksCommand {
   implicit def createSearchParams(cmd: SearchTasksCommand):
                SearchParams =
                SearchParams(cmd.status.value.get, cmd.text.value)
  }

  class ValidStatusValidations(b: FieldDescriptor[String]) {

    def validStatus(message: String =
        "%s be either 'new' or 'in progress'.") =
        b.validateWith(_ =>
               _ flatMap { new PredicateValidator[String](
                    b.name,
                    List[String]("new",
                                 "in progress").contains(_),
```

```
                                    message).validate(_) }
    )
  }

  /**
   * Params only command parses incoming parameters
   */
  class SearchTasksCommand extends ParamsOnlyCommand {
    implicit def statusValidator(b: FieldDescriptor[String])
                      = new ValidStatusValidations(b)

    val text: Field[String] = asType[String]("text")
    val status: Field[String] =
                asType[String]("status").
                    notBlank.
                    minLength(3).
                    validStatus()
  }

}
```

At the bottom of this object, we define a `SearchTasksCommand` class. This command will process the incoming parameters (since we extend from `ParamsOnlyCommand`) and check whether the incoming parameters are valid. In this case, we don't validate the text parameter, but expect the `status` parameter to validate against the `notBlank`, `minLength`, and the custom `validStatus` validators. In this object, we also define an implicit conversion between the `SearchTaskCommand` class and the `SearchParams` case class, which we can use in our service. This makes the code in our route more clean, as we'll see later.

We also define a custom validator in this object—the `ValidStatusValidations` class. This class takes a `FieldDescriptor` as its input and defines a `validStatus` function. In the `validStatus` function, we use the `validateWith` function of the `FieldDescriptor` to validate the value of the parameter. We can write this ourselves or use the `PredicateValidator`, as we did in this example. Once the validation is defined, we make it available in our command by defining the `implicit def statusValidator` conversion.

Scalatra comes with a large set of validators out of the box, which you can use in the manner explained here:

Name	**Description**
`notEmpty`	This checks whether the provided field contains a value
`greaterThan`	This tries to convert value to a number and checks whether it is greater than the provided value
`lessThan`	This tries to convert value to a number and checks whether it is less than the provided value

`greaterThanOrEqualTo`	This tries to convert value to a number and checks whether it is greater than or equal to the provided value
`lessThanOrEqualTo`	This tries to convert value to a number and checks whether it is less than or equal to the provided value
`notBlank`	This removes any spaces and checks whether the provided field still contains a value
`validEmail`	This checks whether the value is a valid e-mail value
`validAbsoluteUrl`	This checks whether the value is a valid absolute URL
`validUrl`	This checks whether the value is a valid URL
`validForFormat`	This checks the value against the provided regular expression
`minLength`	This validates whether the value's length is at least a specific amount of characters
`maxLength`	This validates whether the value's length is less than a specific amount of characters
`enumValue`	This checks whether the value is one of the provided values

Now, we can finally use the command in our route:

```
  get("/tasks/search") {
    new AsyncResult() {
      override val is = (command[SearchTasksCommand] >>
                  (TaskServiceWrapper.WrapSearch(_))).fold (
        errors => halt(400, errors),
        tasks => tasks
      )
    }
  }
```

The syntax might be a bit rough, but the following steps happen here:

- `Command[SearchTasksCommand] >>` means that we execute the command specified in the `SearchTasksCommand` class and call the function to the right of `>>`
- The provided function in this case is `TaskServiceWrapper.wrapSearch()`
- The result of this service is a `ModelValidation[T]` on which we can call `fold`
- If our validation returns errors, we halt the call and return the errors
- If our validation returns tasks, we just return those

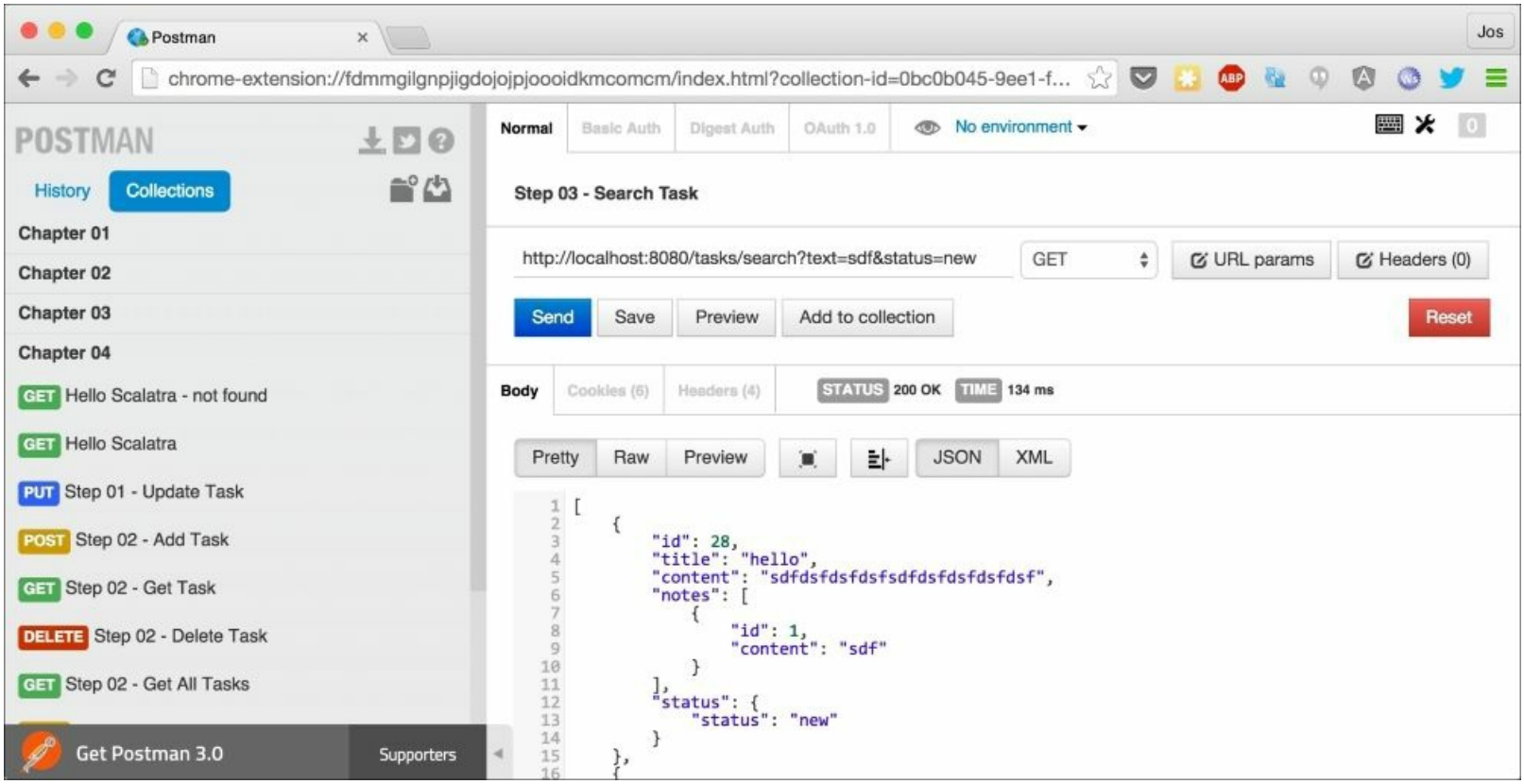
Since we need to return a `ModelValidation[T]` instance, we have created a simple wrapper service around the `TaskService`:

```
object TaskServiceWrapper {

  def WrapSearch(search: SearchParams): ModelValidation[Future[List[Task]]] = {
```

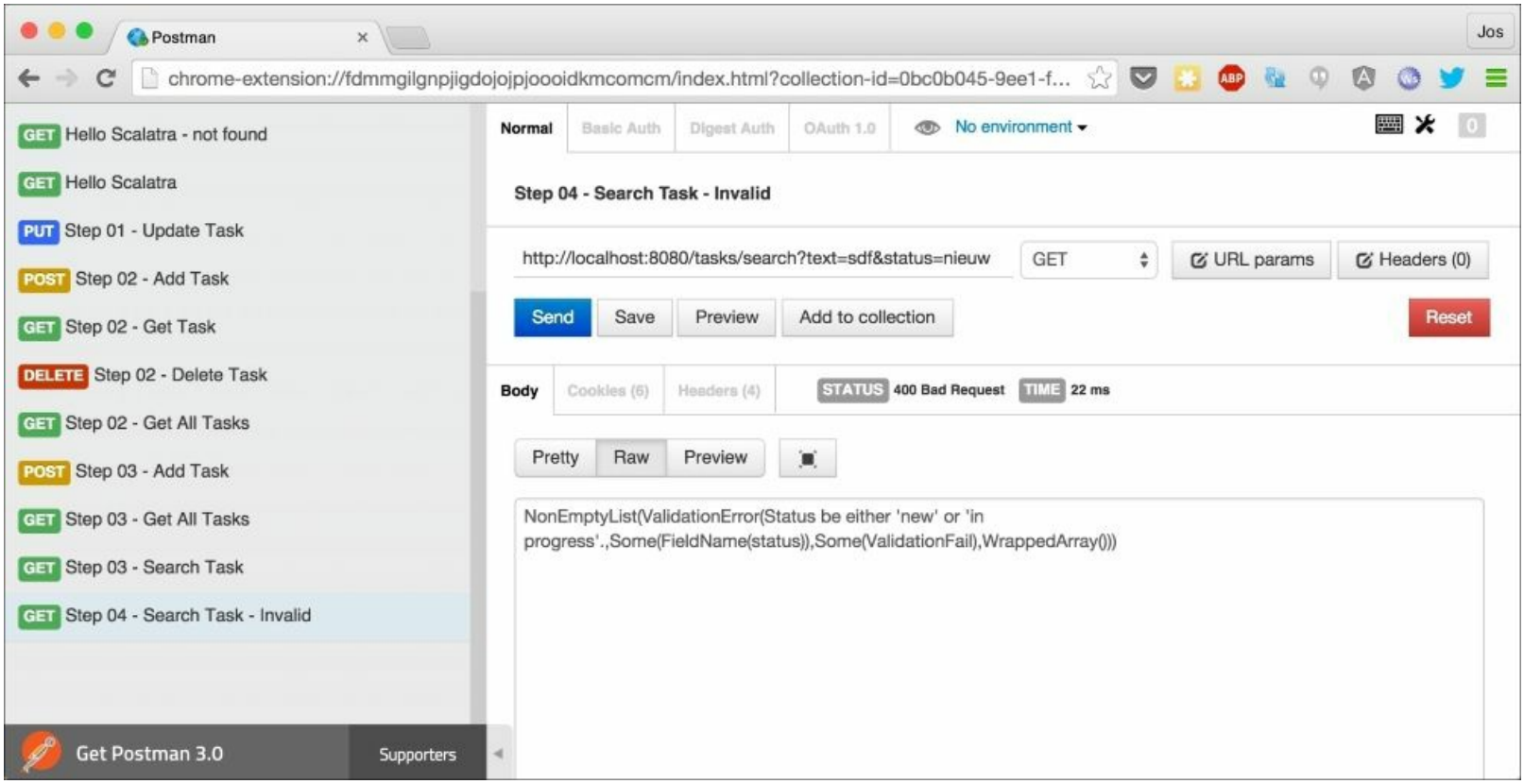
```
    allCatch.withApply(errorFail) {
      println(search)
      TaskService.search(search).successNel
    }
  }

def errorFail(ex: Throwable) = ValidationError(ex.getMessage,
UnknownError).failNel
}
```

At this point, when we call the `/tasks/search` URL, Scalatra will create the command, execute the validations, and if successful, call the `WrapSearch` function and return a set of tasks. If errors occur during validations, those will be returned instead.

You can easily test this with Postman. First, add some tasks through the **Step 03 – Add Task** request. Now when you call **Step 03 – Search Tasks**, you'll just get a set of valid results:



If, on the other hand, you call **Step 03 – Search Tasks – Invalid**, you'll see an error message:



In this section, we saw some of the possibilities Scalatra commands provide for validation. You can do much more with commands than shown in this section; for more information, refer to the Scalatra commands documentation at http://www.scalatra.org/2.3/guides/formats/commands.html.

Summary

In this chapter, you learned how to use Scalatra to create a REST service. We saw that you need a custom Jetty launcher to run Scalatra standalone since it's been created to run inside servlet containers. Once you've got Scalatra running, you can just add routes to a class, and they will be automatically picked up when you run Scalatra. Remember, though, that routes are matched starting from the bottom and going up. Scalatra also provides easy access to all the properties of a request and some basic validations through the use of Options. For more advanced validations, you can use the `scalatra-commands` module, where you can automatically parse incoming parameters and validate against a large set of validators. Finally, adding JSON support to Scalatra is very easy. All you have to is add the correct modules to your build and import the conversions classes.

In the next chapter, we'll show you how to use the Spray-based DSL from Akka HTTP to create REST services.

Chapter 5. Defining a REST Service Using Akka HTTP DSL

In this chapter, we'll look at the successor to one of the most popular REST frameworks available in the Scala space, called **Spray**. Spray has been around for a couple of years, and provides a very extensive **domain-specific language** (**DSL**), which you can use to define your REST services. Spray itself isn't being actively developed and has merged into the Akka HTTP initiative provided by Typesafe. The DSL structure and way you create REST services, however, hasn't changed that much. So in this chapter, we'll explore the following features provided by Akka HTTP's DSL:

- The first DSL-based service
- Verb and path handling through directives
- Exception handling
- Validations and JSON support

In the next section, we'll first look a bit deeper at what this DSL entails and the history of the Akka HTTP project.

What is Akka HTTP?

Akka HTTP is part of the Akka set of libraries and framework. Akka itself is a very well-known actor framework, which is used to create highly scalable, distributed, and resilient applications. Akka HTTP is built on top of the Akka framework, and the 1.0 version was released in the summer of 2015. You can work with Akka HTTP in two different manners. You can use the low-level API and directly work with reactive flows to process the raw HTTP information, or you can use the high-level API and use an advanced DSL to process your requests. In this chapter, we'll use the latter approach.

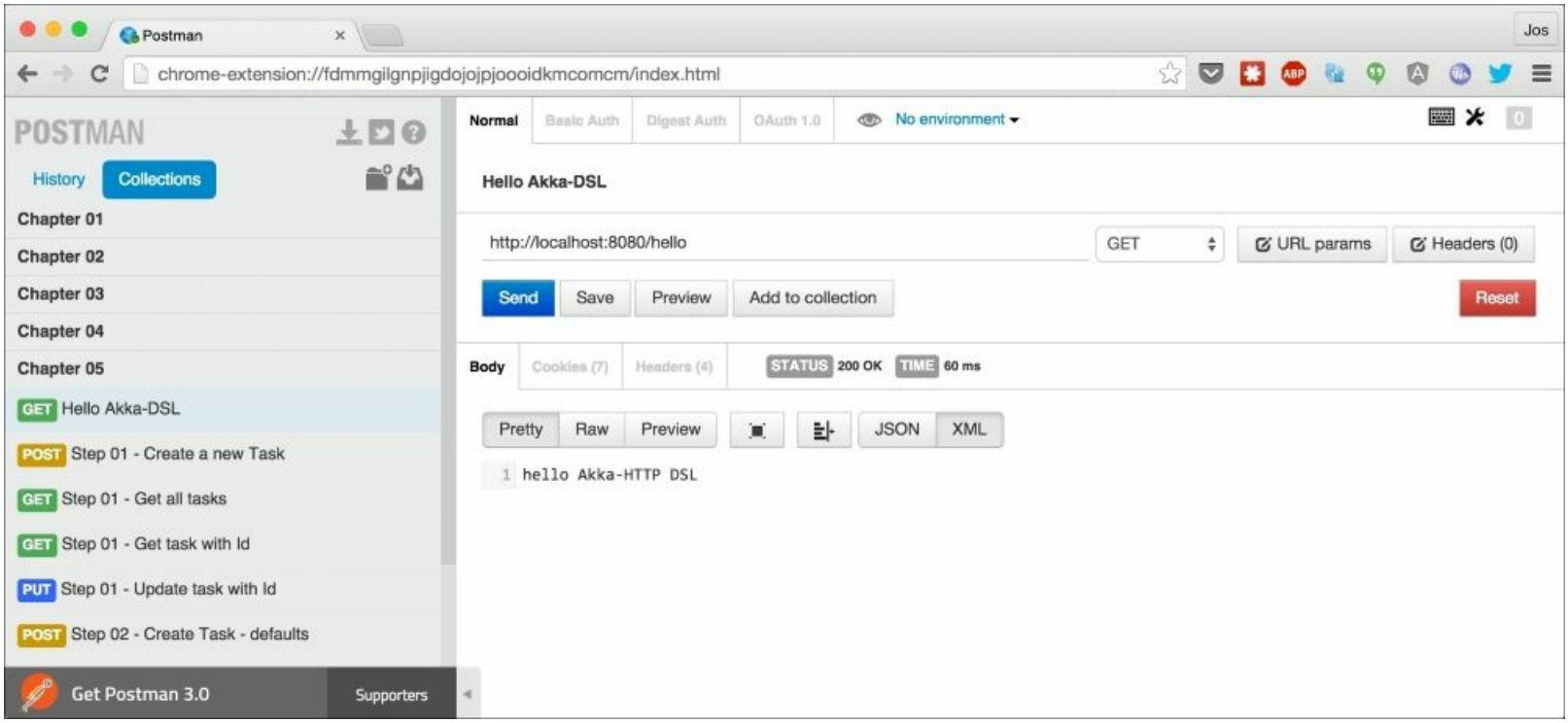
You might think that Akka HTTP isn't a very mature framework since the 1.0 version was only recently released. This isn't the case. The Akka HTTP DSL is based on the well-known Spray framework, which has been around for a couple of years. Development of Spray has stopped, and has continued in the Akka HTTP DSL project. So, for those of you who have experience with Spray, the DSL will look pretty much the same, and you'll recognize all the standard constructs from Spray.

Creating a simple DSL-based service

For each framework in this book, we create a simple getting started service. So for Akka HTTP, we did the same thing. Before we look at the code, let's begin by starting the service and firing a request using Postman. To start the service, from the command line, run the `sbt runCH05-HelloAkka-DSL` command:

```
$ sbt runCH05-HelloAkka-DSL
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Running org.restwithscala.chapter5.gettingstarted.HelloDSL
Press <enter> to exit.
```

Open up Postman, and from the **Chapter 05** collection, run the `Hello Akka-DSL` command. The server will respond with a simple message:



To create this example, we of course need to import the external dependencies. For this sample, the following `sbt` dependencies are used:

```
lazy val akkaHttpVersion = "1.0"

val backendDeps = Seq (
  "com.typesafe.akka" %% "akka-stream-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-core-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json-experimental" % akkaHttpVersion
```

```
)
```

Note that the dependencies still have the experimental tag in their name. This means that the implementation might change and, at this point, there isn't any official support from Typesafe yet. So there might be some changes in the future, which aren't binary-compatible. Typesafe itself defines it as:

> *"This module of Akka is marked as experimental, which means that it is in early access mode, which also means that it is not covered by commercial support. An experimental module doesn't have to obey the rule of staying binary compatible between minor releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. An experimental module may be dropped in major releases without prior deprecation."*

So at this point, it might be wise to not yet convert all your existing Spray code to this codebase, but wait until they move out of the experimental phase.

With these dependencies in place, we can create our simple service:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Directives._

object HelloDSL extends App {
  // used to run the actors
  implicit val system = ActorSystem("my-system")
  // materializes underlying flow definition into a set of actors
  implicit val materializer = ActorMaterializer()

  val route =
    path("hello") {
      get {
        complete {
          "hello Akka-HTTP DSL"
        }
      }
    }

  // start the server
  val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)

  // wait for the user to stop the server
  println("Press <enter> to exit.")
  Console.in.read.toChar

  // gracefully shut down the server
  import system.dispatcher
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.shutdown())
```

```
}
```

The first thing we do in this service is define two implicit values. The implicit `ActorSystem` is needed to define the actor system that will be used to run the various processing steps of a request asynchronously. Akka HTTP will convert the DSL we create into a flow definition (which is a construct of Akka Streams). This flow can be seen as a blueprint of the steps that a request takes from the beginning to the end. The implicit `ActorMaterializer` will convert this flow into a set of Akka actors so that multiple requests can be executed concurrently without interfering with each other, which runs on the implicitly defined `ActorSystem`.

With the implicits defined, we can define the route:

```
val route =
  path("hello") {
    get {
      complete {
        "hello Akka-HTTP DSL"
      }
    }
  }
```

Each request is passed through this route and, when one matches, its inner route is executed. So in this case, the following steps are executed:

1. First, the provided URL path is checked. In this case, if the path matches `hello`, the inner route of the path function (this is called a **directive**) is executed.
2. The next check Akka HTTP makes is to see whether the verb matches. In this example, we check for a `GET` verb.
3. The final inner route completes the request by calling complete. When complete is called, the result of the provided block is returned as response. In this example, we just return a string.

The last piece of code in this hello world example shuts down the server when a key is pressed. Shutting down the server is done through the following code:

```
import system.dispatcher
bindingFuture
  .flatMap(_.unbind())
  .onComplete(_ => system.shutdown())
```

This might seem a complex way to shut down the server, but when you look at the types, it is actually really simple. We call `flatMap` on the `bindingFuture` instance (of the type `Future[ServerBinding]`), so when the `Future` is ready (the server is started successfully), we call unbind on the `ServerBinding` instance. This, in itself, also returns a `Future`, which is flattened since we called `flatMap`. When this last `Future` resolves, we close the Akka system to cleanly shut down everything.

We will use the same way to start and stop the service in other examples.

Working with paths and directives

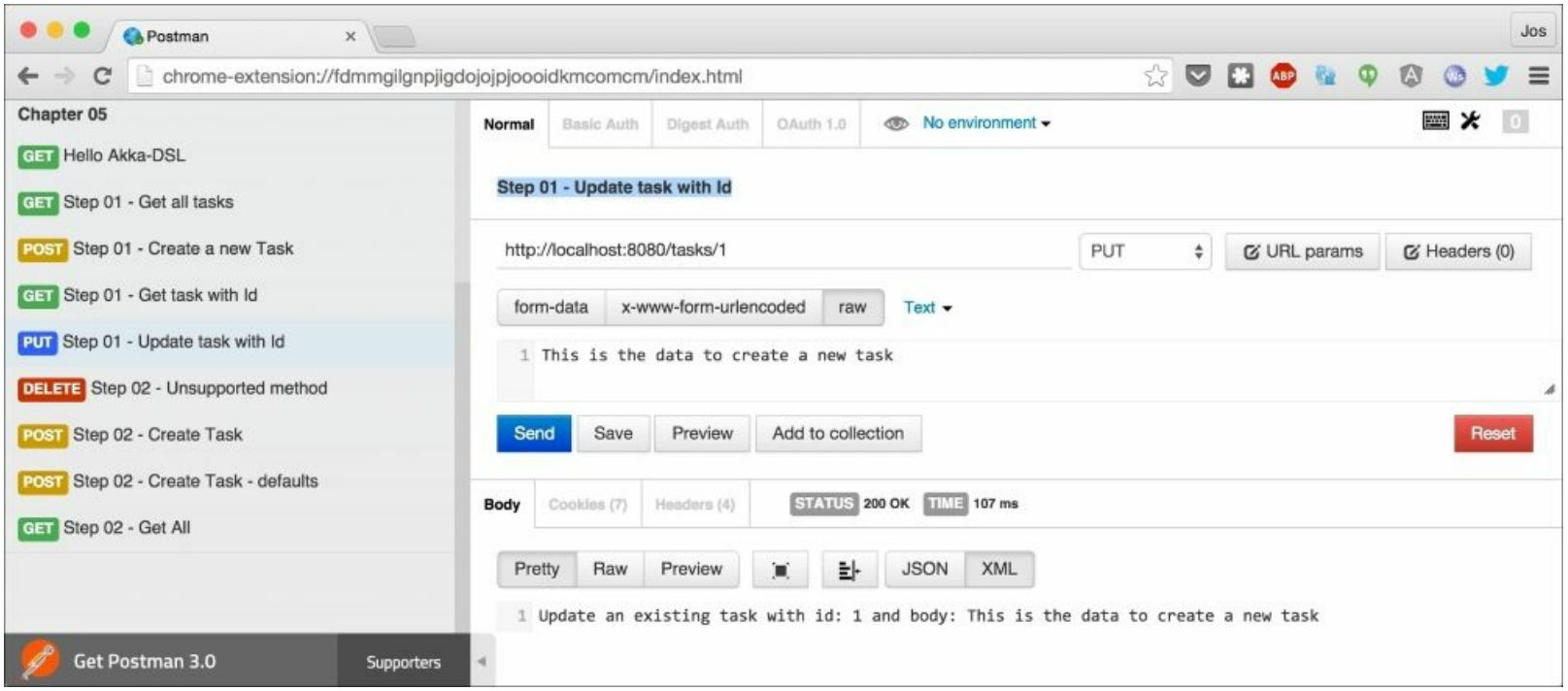
The first example we'll look at is the first simple implementation of our API. We won't be returning real objects or JSON yet, but a couple of strings. The code for this step looks like this:

```
  val route =
    // handle the /tasks part of the request
    path("tasks") {
      get {
        complete { "Return all the tasks" }
      } ~
      post {
        complete { s"Create a new task" }
      } // any other request is also rejected.
    } ~ { // we handle the "/tasks/id separately"
      path("tasks" / IntNumber) {
        task => {
          entity(as[String]) { body => {
            put { complete {
        s"Update an existing task with id: $task and body: $body" } }
          } ~
            get { complete {
        s"Get an existing task with id : $task and body: $body" } }
          } ~ {
            // We can manually add this rejection.
            reject(MethodRejection(HttpMethods.GET),
                   MethodRejection(HttpMethods.PUT))
          }
        }
      }
    }
```

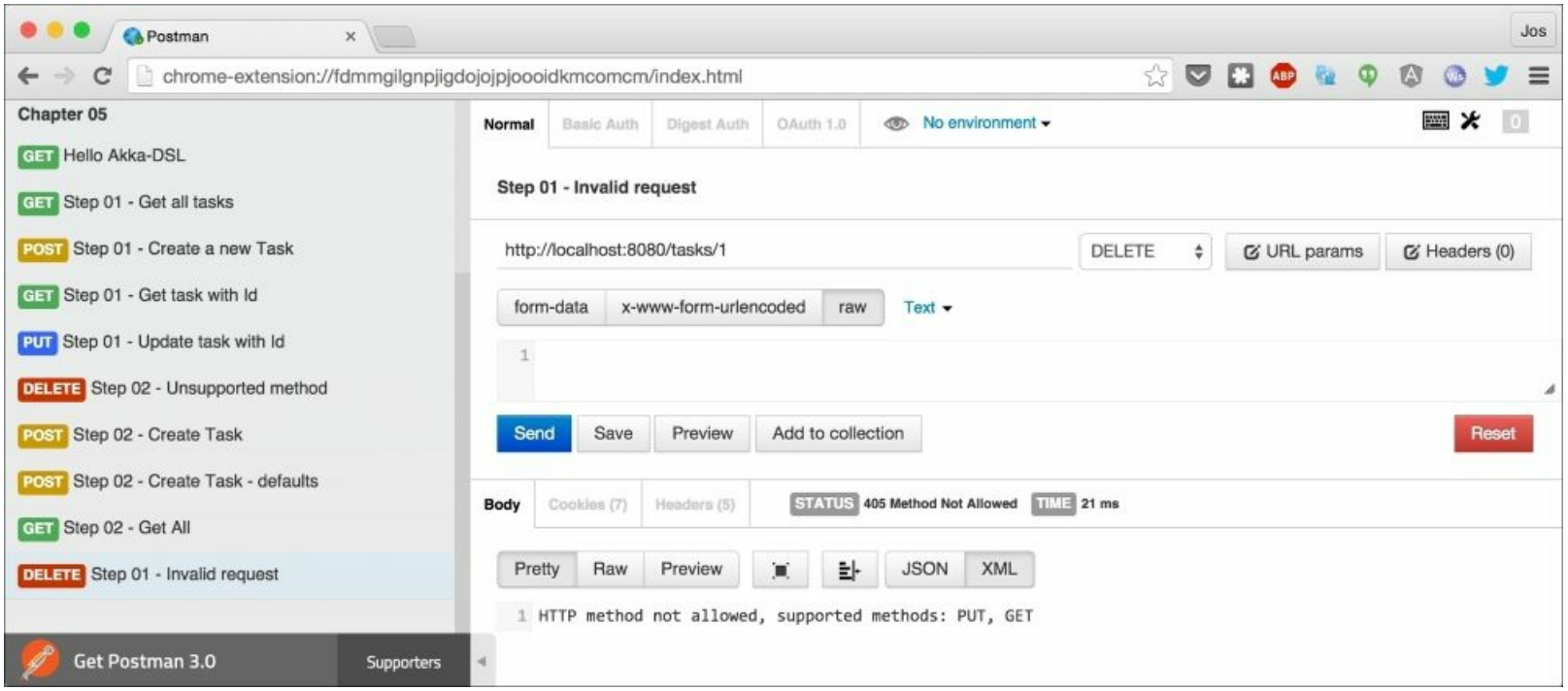
This code doesn't look that different from the previous example we saw. We define a route by using directives such as `path`, `get`, `post`, and `put` and return values by using the `complete` function. We do, however, use a couple of new concepts. Before we explain the code and the concepts provided by Akka HTTP, first let's fire some requests. For this, start up the example for this section:

```
$ sbt runCH05-runCH05Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Compiling 3 Scala sources to /Users/jos/dev/git/rest-with-scala/chapter-
05/target/scala-2.11/classes...
[info] Running org.restwithscala.chapter5.steps.AkkaHttpDSLStep1
Press <enter> to exit.
```

Open up Postman in your browser and firstly execute the **Step 01 - Update task with id** request:



As you can see, we return a simple response, which shows what data was sent to the server. Another interesting example is where we send an invalid request—**Step 01 - Invalid request**:



Here, you can see that we can easily provide the user with additional information on how to use our service.

Let's look a bit closer at the second part of the code we saw at the beginning of this section:

```
path("tasks" / IntNumber) {
        task => {
          entity(as[String]) { body => {
            put { complete {
        s"Update an existing task with id: $task and body: $body" } }
          } ~
            get { complete {
        s"Get an existing task with id : $task and body: $body" } }
          }
        }
      }
    }
```

Here, we once again see the familiar `path`, `get`, and `put` directives, and we also use additional directives to get extra information from the request. We use the `IntNumber` path matcher to convert part of the path to an integer and use the `entity(as[String])` extractor to extract the body of the request as a string (we'll see more of this directive at the end of this chapter when we use the same approach to handle JSON input). Let's start though by looking a bit closer at the `path` directive.

In this example, we already used three different path matchers. We used a string value to match a part of the URL, the `/` matcher to indicate that we expected a forward slash, and the `IntNumber` path matcher to match and extract a number. Besides these, you can also use the matchers explained in the following table:

Path matchers	**Description**
`"/hello"`	This matcher matches part of the URL and also consumes it. Nothing is extracted here.
`"[a-b]"r`	You can also specify a regular expression with a maximum of one capture group with this matcher. The capture group is extracted.
`Map[String, T]`	Using this matcher, you can extract a value, based on the path that matches `Map("path1" -> 1, "path2" -> 2, "path3" -> 3)`.
`Slash (or /)`	This matcher matches a single forward slash.
`Segment`	This matches if the path starts with a path segment (not a forward slash). The current path segment is extracted as a string.
`PathEnd`	This matches the end of the path and extracts nothing.
`Rest`	This matches the rest of the path and returns it as a string.
`RestPath`	This matches the rest of the path and returns it as a path.
`IntNumber`	This matches a number of digits that can be converted to an integer. The matched integer is extracted.

`LongNumber`	This matches a number of digits that can be converted to a long number and extracts the matched long number.
`HexIntNumber`	This matches a number of hex-digits that can be converted to an integer and extracts the matched integer.
`HexLongNumber`	This matches a number of hex-digits that can be converted to a long number and extracts the matched long number.
`DoubleNumber`	This matches a number of digits that can be converted to a double number and extracts the matched double number.
`JavaUUID`	This matches and extracts the string representation of a `java.util.JavaUUID` object. The result is a `.java.util.JavaUUID` instance.
`Neutral`	This matches everything and doesn't consume anything.
`Segments`	This is the same as the `Segment` matcher but, this time, matches all the remaining segments and returns these as a `List[String]` object.
`separateOnSlashes`	This creates a matcher that interprets slashes as path segment separators.
`provide`	This matcher always matches and extracts the provided tuple value.
`~`	This operator allows you to concatenate two matchers, for example `"hello" ~ "world"` is the same as `"helloworld"`.
`|`	This combines two matchers together. The right-hand side one will only be evaluated when the left-hand side one fails to match.
Postfix: `?`	The `?` postfix makes the matcher optional, and it will always match. The result of the extracted value will be an `Option[T]` object.
Prefix: `!`	This prefix inverses the matcher.
Postfix: `.repeat`	With `repeat`, you can make a matcher that repeats itself the specified amount of time.
`transform, flatMap, map`	This allows you to customize the matcher and create your own custom logic.

So even in the `path` directive, you can already extract a lot of information and apply multiple matchers. Besides the `path` directive, there are a lot of other directives. We've already seen the entity extracted in this example used like this:

```
entity(as[String]) { body => {
            put { complete {
        s"Update an existing task with id: $task and body: $body" } }
```

```
    }
```

When you use an extractor, the extracted value is passed as an argument to the inner route (`body` in this code fragment). Akka HTTP comes with a large number of extractors you can use to get values out of the request. The following table shows the most useful ones:

Directive	Description
`cookie("name")`	This extracts a cookie with the specified name and returns an `HttpCookiePair` instance. There is also an `Option` variant—`optionalCookie`.
`entity(as[T])`	This unmarshals the request entity to the specified type (for more information, see the section on JSON).
`extractClientIp`	This extracts the client's IP from either the X-Forwarded- or Remote-Addressor X-Real-IP header as a `RemoteAddress`.
`extractCredentials`	This gets an `Option[HttpCredentials]` from the authorization header.
`extractExecutionContext`	This provides access to the Akka `ExecutionContext` instance.
`extractMaterializer`	This provides access to the Akka `Materializer`.
`extractHost`	This gets the `hostname` part of the host request header value as a string.
`extractMethod`	This extracts the request method as an `HttpMethod`.
`extractRequest`	This provides access to the current `HttpRequest`.
`extractScheme`	This returns the URI scheme (`http`, `https`, and others) from the request as a string.
`extractUnmatchedPath`	This extracts the part of the path that is unmatched at this point as a `Uri.Path`.
`extractUri`	This accesses the full URI of the request as a `Uri`.
`formFields`	This extracts fields from a POST made in an HTML form. For more information, see the section on path matchers.
`headerValueByName`	This extracts the value of the first HTTP request header with a given name and returns it as a string. You can also get an `Option[String]` by using `OptionalHeaderValueByName`.
`headerValueByType [T]`	You can also extract a header and automatically convert it to a specific type with this directive. For this one, there is also an `Option` variant—`OptionalHeaderValueByType`.

`parameterMap`	This gets all the parameters from the request as a `Map[String, String]`. If multiple parameters with the same name exist, only the last one will be returned.
`parameterMultiMap`	This gets all the parameters from the request as a `Map[String, List[String]]`. If multiple parameters with the same name exist, all will be returned.
`parameterSeq`	This extracts all the parameters in order as a `Seq[(String, String)]` of tuples.
`provide("value")`	This injects the provided value into the inner route. There is also a `tprovide` function, which injects a tuple.

Most of these extractors are pretty much self-explanatory. For instance, when you want to extract a specific HTTP header, you can write a route like this:

```
val exampleHeaderRoute = path("example") {
  (headerValueByName("my-custom-header")) { header => {
      complete(s"Extracted header value: $header")
    }
  }
}
```

Now, let's get back to our example and look again at a very simple part of our route:

```
      get {
        complete { "Return all the tasks" }
      }
```

So far, we've only seen a small number of Akka directives. We looked at the possible extractors in the previous table, and the simple directives to match parts of the path and the specific-HTTP verbs. Besides these, Akka HTTP provides a very large number of directives, much more than we can explain in this single chapter. In the following table, we'll list the directives we feel are the most important and flexible for you to use in the routes:

Directive	**Description**
`conditional`	This provides support for conditional requests as specified in http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-26.
`decodeRequest / encodeRequest`	This compresses and decompresses a request that is sent using gzip or deflate compression.
`decodeRequestWith / encodeRequestWith`	This compresses and decompresses a request with the specified encoder and decoder.
`get / delete / post / put / head / options / patch`	The inner route of this directive will be executed when the specified HTTP verb matches.
`host`	This only runs the inner route if the host specified in this directive matches. You can use strings or a regex.

`method(httpMethod)`	This checks whether the request matches the specified `HttpMethod`.
`onComplete(future)`	This runs the inner route when the provided `Future` completes.
`onSuccess`	This runs the inner route when the provided `Future` completes successfully.
`overrideMethodWithParameter`	This changes the method (HTTP verb) of the incoming request to the provided `HttpMethod`.
`pass`	This always passes the request to the inner route.
`path`	If the provided path matches, this passes the request to the inner route (you will get to know more about it in the next example).
`pathEnd`	This only passes the request on if its complete path is matched.
`pathEndOrSingleSlash`	This only passes the request on if it's complete path is matched or only a single slash remains.
`pathPrefix`	This matches first part of a path. A leading slash is automatically added. If you just want to test and not consume the path, use `pathPrefixTest`. If you don't want a leading slash, you can use `rawPrefix` and `rawPrefixTest`.
`pathSingleSlash`	This only runs the inner route if the path contains a single slash.
`pathSuffix`	This checks the end of the current path and, if it matches, runs the inner route. If you just want to test and not consume the path, use the `pathSuffixTest` directive.
`requestEncodedWith`	This checks whether the request encoding matches the specified `HttpEncoding`.
`requestEntityEmpty`	This matches if the request doesn't contain a body.
`requestEntityPresent`	This matches if the request contains a body.
`scheme("http")`	This checks the scheme of the request. If the scheme matches, the request is passed on to the inner route.
`validate`	This allows you to test against an arbitrary condition.

Before we move on to the next example, we'll quickly look at one last inner route that we saw at the beginning of this section:

```
{
// We can manually add this rejection.
reject(MethodRejection(HttpMethods.GET),
   MethodRejection(HttpMethods.PUT))
}
```

With this inner route, we inform Akka HTTP that the inner route rejects the request. In this case, we reject the request because the `HttpMethod` (the verb) didn't match anything we can process. When you reject a request, Akka HTTP will check whether there are any routes that might match and, if not, will convert the rejection in to an HTTP error message. Further, in this chapter, we'll look a bit closer at how Akka HTTP works with rejections and handles exceptions.

Processing request parameters and customizing the response

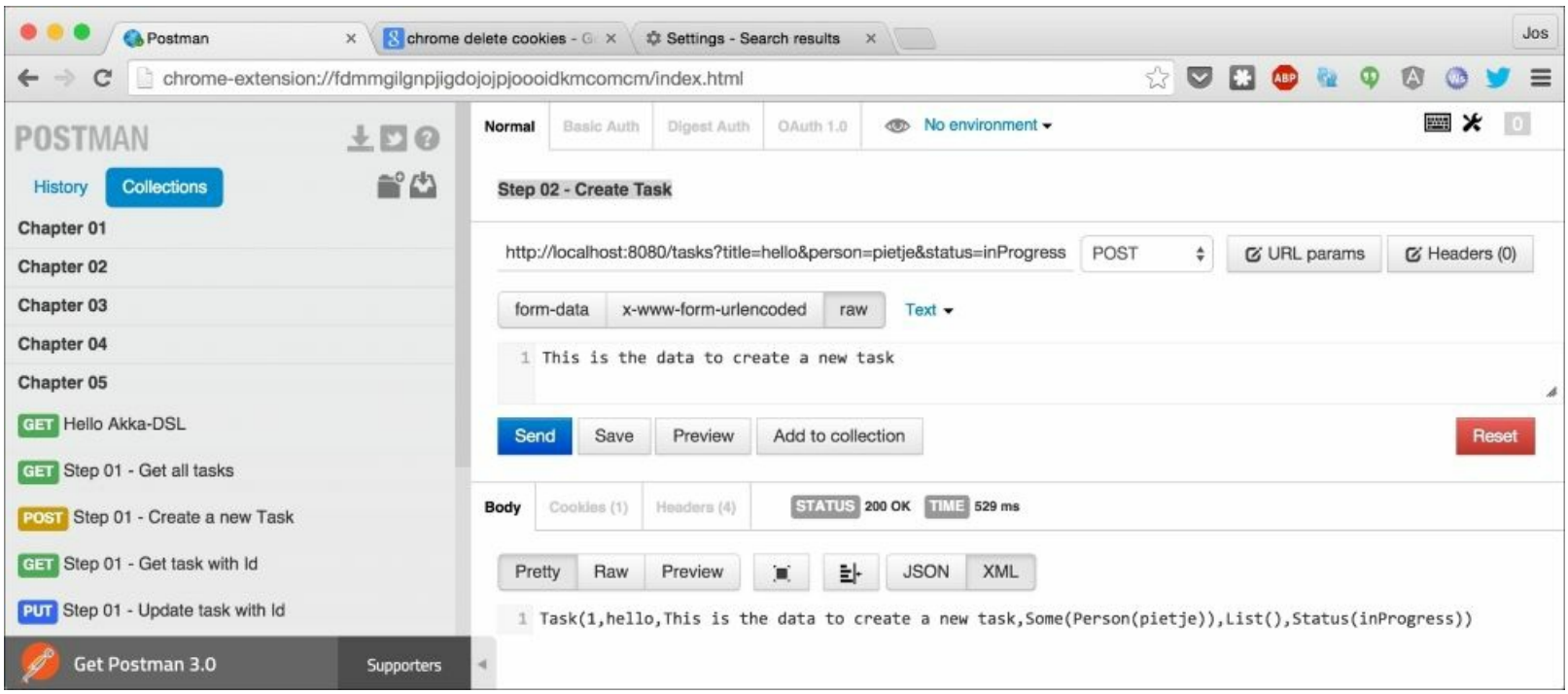
In this section, we'll dive a bit deeper into how we can extract query parameters from requests and how to customize the response you send back to the client. In this section, we'll look a bit closer at how to implement the following requests:

- **Create a task**: We'll use a set of query parameters to create a new task
- **Get all tasks**: We will return all the tasks that have been created
- **Get a task**: We will return a specific task based on the provided ID

For each of these requests, we'll first show the call from Postman and then how it is implemented with Akka HTTP. First, start up the correct server with the `sbt runCH05-runCH05Step2` command:

```
$ sbt runCH05-runCH05Step2
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)
[info] Compiling 1 Scala source to /Users/jos/dev/git/rest-with-scala/chapter-
05/target/scala-2.11/classes...
[info] Running org.restwithscala.chapter5.steps.AkkaHttpDSLStep2
Press <enter> to exit.
```

Now open Postman and execute the **Step 02 - Create Task** request:



In the response, you can see that we return a string representation of the task that was added, and that

the content of the task is based on the query parameters in the URL. To accomplish this, we implement the following route:

```
    ((post) &
    (parameters("title", "person".?, "status" ? "new"))) {
                              (title, assignedTo, status) => {
      (entity(as[String])) { body => {
        complete {
          val createdTask = TaskService.insert(
             Task(-1,
                  title,
                  body,
                  assignedTo.map(Person(_)),
                  List.empty, Status(status)))
          createdTask.map(_.toString)
        }
       }
      }
    }
  }
```

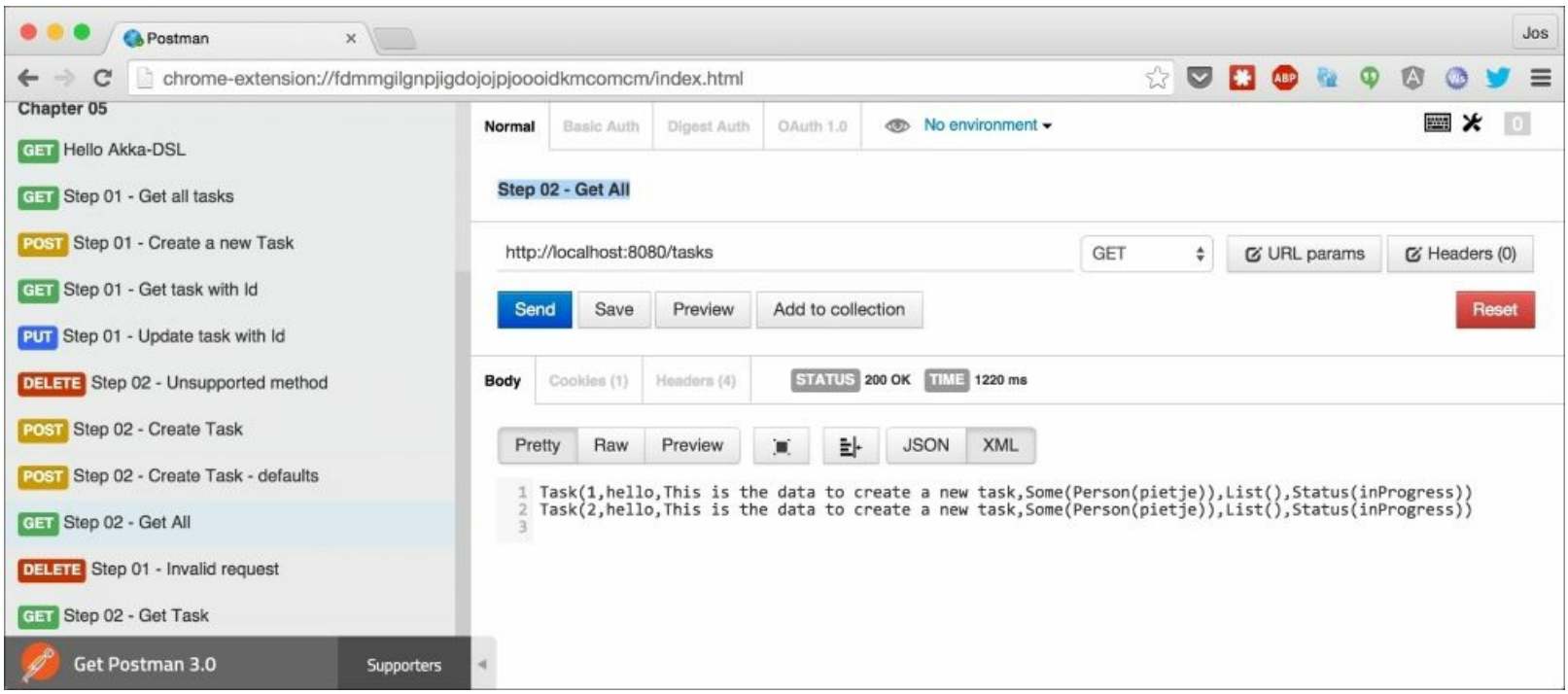
In this route, we first combine two directives, the `post` directive and the `parameters` directive by using the `&` symbol. This means that both directives should match before the inner route is executed. There is also a `|` symbol you can use, which means either the left or the right directive should match (`put | post`). The parameters, together with the body, are used to create a `Task` instance using the `TaskService` object. The resulting `Future[Task]` is then converted to a `Future[String]` and returned. We need to convert the `Task` to a string here manually since we haven't told Akka HTTP how to deal with the `Task` class (you will get to know more about it later). If you look at the parameters directive, you'll not only recognize the query parameters from the original request, but also see a number of modifiers. The following bullets explain how the parameters directive works:

- `"title"`: This extracts the value of the parameter as a string. This marks this query parameter as required.
- `"person".?`: By using the `.?` postfix, you make a value optional. This result is an `Option[String]`.
- `"status" ? "new"`: This retrieves the parameter and, if it can't be found, uses the default value (`"new"` in this case). The result is a string.
- `"status" ! "new"`: This requires the query parameter with the name `"status"` to be `"new"`.
- `"number".as[Int]`: This tries to convert the parameter to the specified type.
- `"title.*"`: This extracts multiple instances of the title parameter to an `Iterable[String]`.
- `"number".as[Int].*`: This is same as the previous function, but works on types.

So for our sample, we require a `title` parameter, an optional `person` parameter, and an optional `status` parameter, which has a default value of `"new"`. With these constructs, it is very easy to extract the correct values from a request's query parameters.

Now let's look a bit closer at the response we send back from this example. We use the complete directive which we've seen earlier, and just return a `Future[String]` instance. Since it is a string,

Akka HTTP knows how to marshal it to an HTTP response. We use the same approach for the get all tasks request. From Postman, after you've created a couple of tasks, make the **Step 02 - Get All** request:

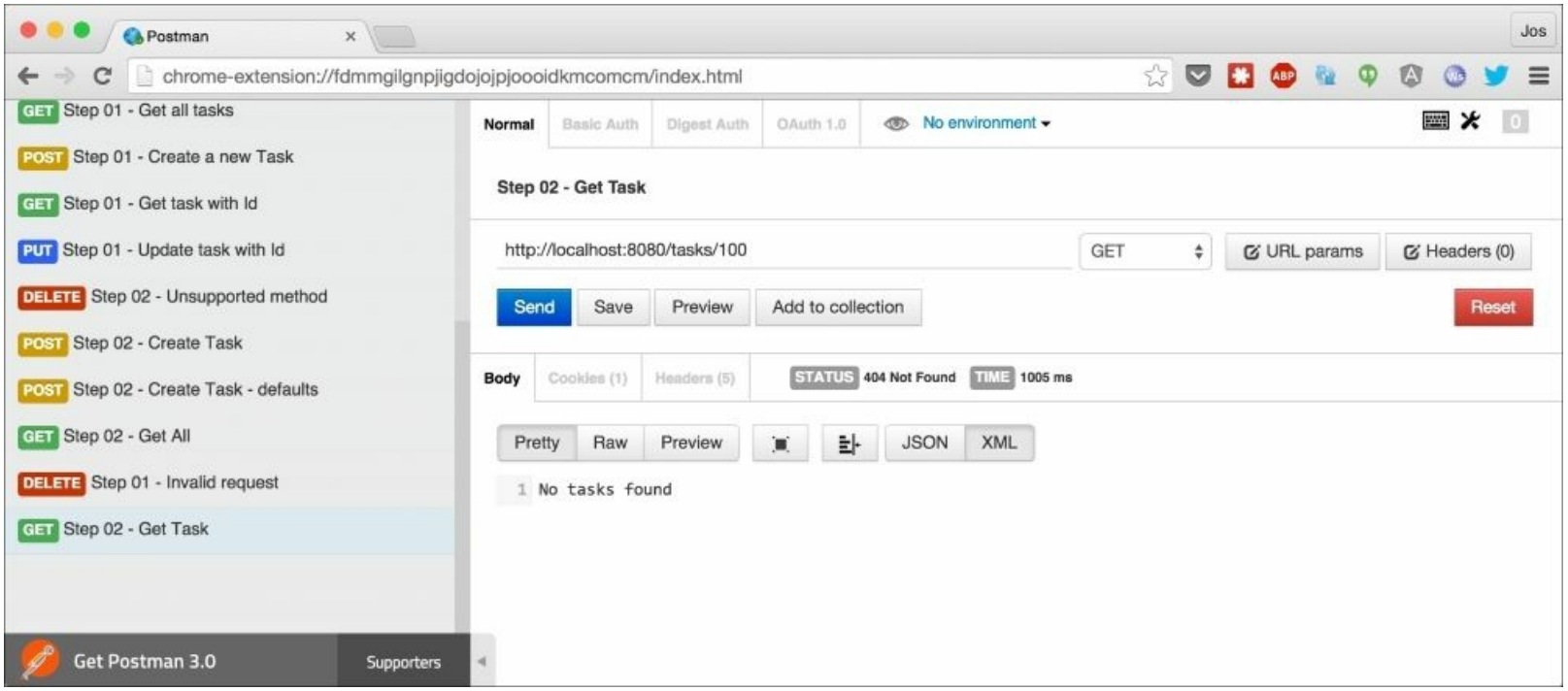


You'll see that you get back the list of tasks you've created. Once again, it's very simple:

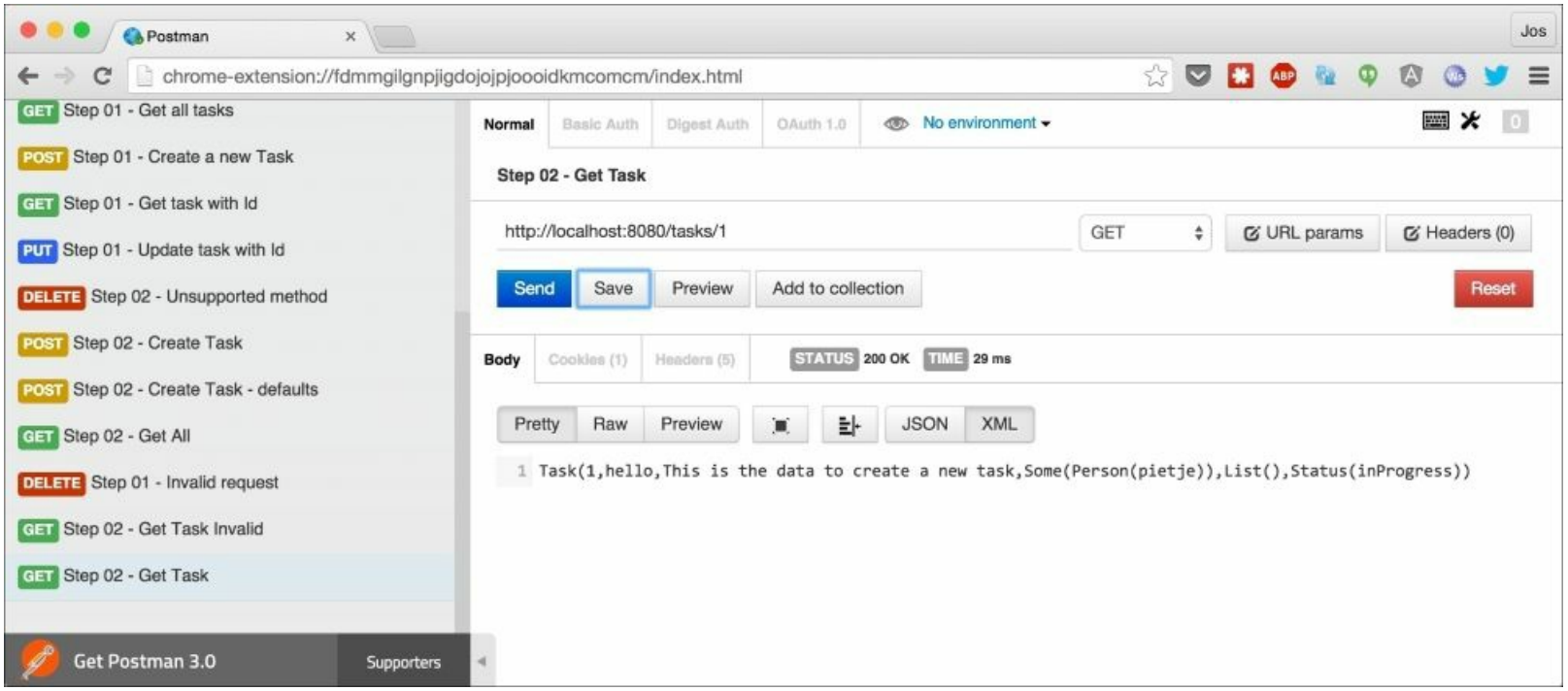
```
complete {
  // our TaskService returns a Future[List[String]]. We map // this to a single
Future[String] instance that can be returned // automatically by Akka HTTP
  TaskService.all.map(_.foldLeft("")((z, b) => z + b.toString + "\n"))
}
```

We make a call to the `TaskService` object, which returns a `Future[List[Task]]` instance. Since we haven't told Akka HTTP yet how to handle this type, we manually convert it to the correct type and use `complete` to send back the response. So far, we've only seen the use of complete. In most cases, the default behavior provided by Akka HTTP is good enough, however, there are also different ways you can customize the response that is sent by Akka HTTP.

Open Postman again, use the **Step 02 - Create Task** option to create a number of tasks and this time execute the **Step 02 – Get Task Invalid** request to get a single request. This request will try to get the request with ID 100, which is invalid (unless you added 100 tasks). The result looks similar to this:



As you can see, this time we get a **404** message, with a custom error message. If you make a request for a task which is available (use the Step **02 – Get Task** request), you'll see the following:



For the rest, there isn't too much special in these two requests. However, if you look at the **Cookies** and **Headers** tabs, you might notice that we get additional results there. If you open these tabs, you'll notice a custom hello cookie with `world` as the value, and in the **Headers** tab, we added a custom header called **helloheader**.

To accomplish this, we used a couple of standard Akka HTTP features. The following code fragment shows how this is done:

```
val sampleHeader: HttpHeader =
          (HttpHeader.parse("helloheader","hellovalue") match {
    case ParsingResult.Ok(header, _) => Some(header)
    case ParsingResult.Error(_) => None
}).get
...

implicit val StringMarshaller: ToEntityMarshaller[Task] =
    Marshaller.opaque { s =>
           HttpEntity(ContentType(`text/plain`), s.toString) }
...

get {
  ( setCookie(HttpCookie(name = "hello", value = "world")) &
    respondWithHeader(sampleHeader)
    ) {
    onComplete(TaskService.select(task)) {
      case Success(Some(value)) => complete(value)
      case Success(None) => complete(StatusCodes.NotFound,
                                     "No tasks found")
      case Failure(ex)  => complete(StatusCodes.InternalServerError,
                           s"An error occurred: ${ex.getMessage}")
    }
  }
}
```

In this code fragment, we use a couple of directives provided by Akka HTTP that can be used to customize the response to a request. With the `setCookie` and `respndWithHeader` directives, we add the custom cookie and header. In the inner route of this directive, we use the `onComplete` directive to determine what to do with the response from the `TaskService.select` function. If the `Future` succeeds with `Some[Task]`, we respond with this `Task`. If no `Task` is found, we return with a `404` (the `NotFound` status code). Finally, if the `Future` fails to complete successfully, we respond with an `InternalServerError`. You might notice that this time, we didn't convert the `Task` to a string, but just returned it. This works in this case because we've also defined an implicit, `ToEntityMarshaller[Task]`. This marshaller allows Akka HTTP to convert the `Task` case class to an `HTTPEntity` instance, which Akka HTTP knows how to marshal to an `HttpResponse`.

Besides the directives shown here, Akka HTTP provides a number of other directives and functions you can use to customize the responses. The following table shows the directives related to customizing the response:

Directive	**Description**
`complete`	This completes the request with the provided arguments.
	This returns the result of the provided `Future` object. If the `Future` fails the exception is extracted

`completeOrRecoverWith`	and the inner route is run.
`completeWith`	This extracts a function that can be called to complete the request.
`getFromBrowseableDirectories`	This serves the content of the given directories as a file-system browser.
`getFromBrowseableDirectory`	This serves the content of the given directory as a file-system browser.
`getFromDirectory`	This matches a `GET` request and returns with the content of a file in a specific directory.
`getFromFile`	This matches a `GET` request and returns with the content of a file.
`getFromResource`	This matches a `GET` request and returns with the content of a classpath resource.
`getFromResourceDirectory`	This matches a `GET` request and returns with the content of a classpath resource in a specified classpath directory.
`listDirectoryContents`	This matches a `GET` request and returns with the content of a specific directory.
`redirect`	This sends a redirection response. There are also more specific directives: • `redirectToNoTrailingSlashIfPresent` • `redirectToTrailingSlashIfMissing`
`respondWithHeader`	This sets a specific header on the response object. You can also add multiple headers at once using the `responseWithHeaders` directive. This directive overwrites the already set headers. If you don't want to override the existing headers, use the following directives: • `respondWithDefaultHeaders` • `respondWithDefaultHeader`
`setCookie, deleteCookie`	This adds or deletes a cookie.
`overrideStatusCode`	This sets the HTTP status code of the response. Akka HTTP provides a `StatusCode` object, which you can use to access all the available response codes.

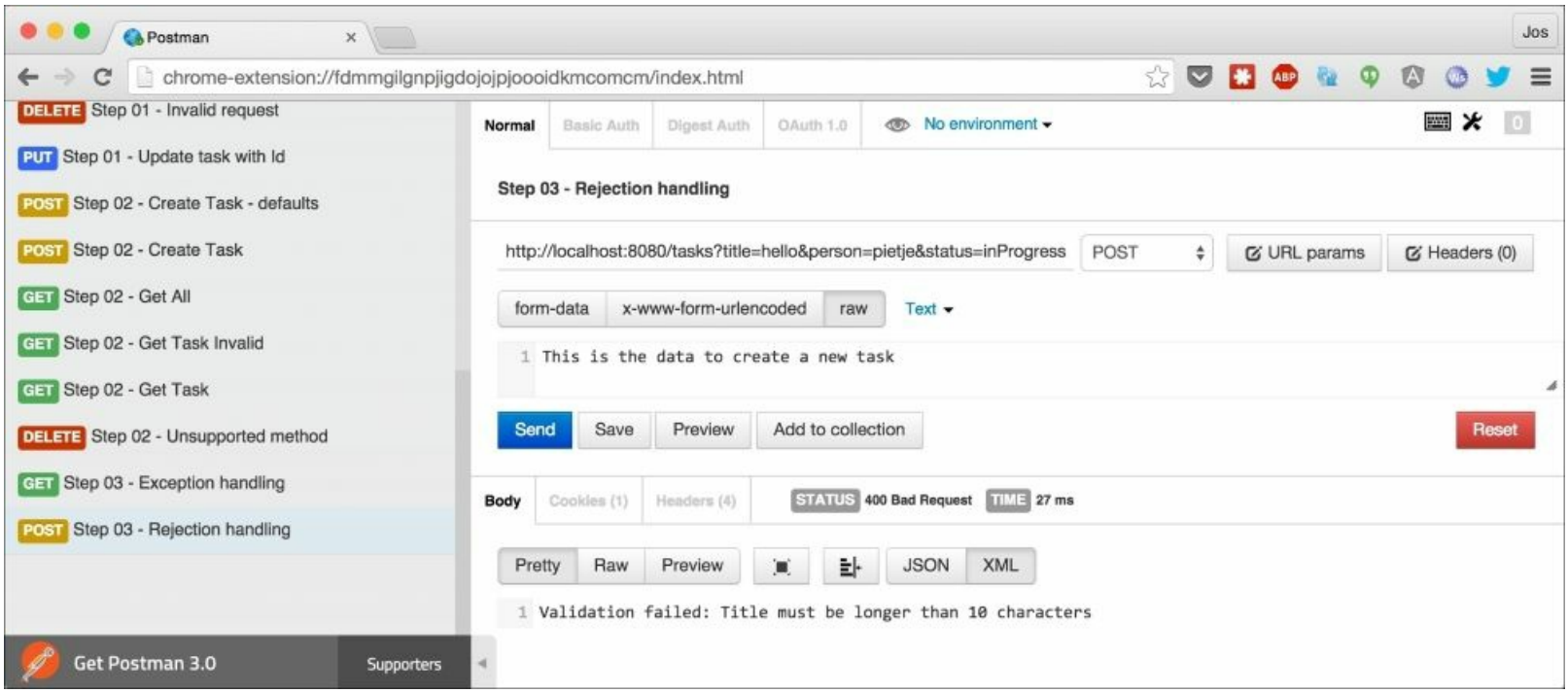
If these directives aren't enough, there is also the option to simply return an `HttpResponse` object. Akka HTTP provides an `HttpResponse` case class for that, which you can use like this:

```
final case class HttpResponse(status: StatusCode = StatusCodes.OK,
                              headers: immutable.Seq[HttpHeader] = Nil,
                              entity: ResponseEntity = HttpEntity.Empty,
                              protocol: HttpProtocol = HttpProtocols.`HTTP/1.1`)
```

So far in this chapter, we saw how to access and extract information from the request and ways to set and customize the response. In the last couple of pages from this chapter, we'll look at the different ways you can handle errors and exception situations from Akka HTTP.

Exception handling and rejections

In this section, we'll look at how Akka HTTP handles exceptions and rejections. Before we look at the code, we'll once again use Postman to show what we want to accomplish. Let's start with how rejections are handled. Rejections are functional errors that are either thrown by directives or which you can throw yourself. In this example, we add some validations to the create task request. So run `sbt runCH05-runCH05Step3`, open up Postman, and execute the request, **Step 03 - Rejection handling**:



As you can see from the response, the title of the task we want to create must be at least 10 characters. The code where we add this validation and also configure the rejection handler is shown here:

```
handleRejections(customRejectionHandler) {
  ((post) & (parameters("title", "person".?, "status" ? "new"))) {
         (title, assignedTo, status) => {
    (entity(as[String])) { body => {
      (validate(title.length > 10,
           "Title must be longer than 10 characters") &
        validate(List("new", "done", "in progress").contains(status),
           "Status must be either 'new', 'done' or 'in progress'") &
        validate(body.length > 0,
           "Title must be longer than 10 characters")
        ) {
        complete {
TaskService.insert(Task(-1, title, body,
                   assignedTo.map(Person(_)),
                   List.empty, Status(status)))
```

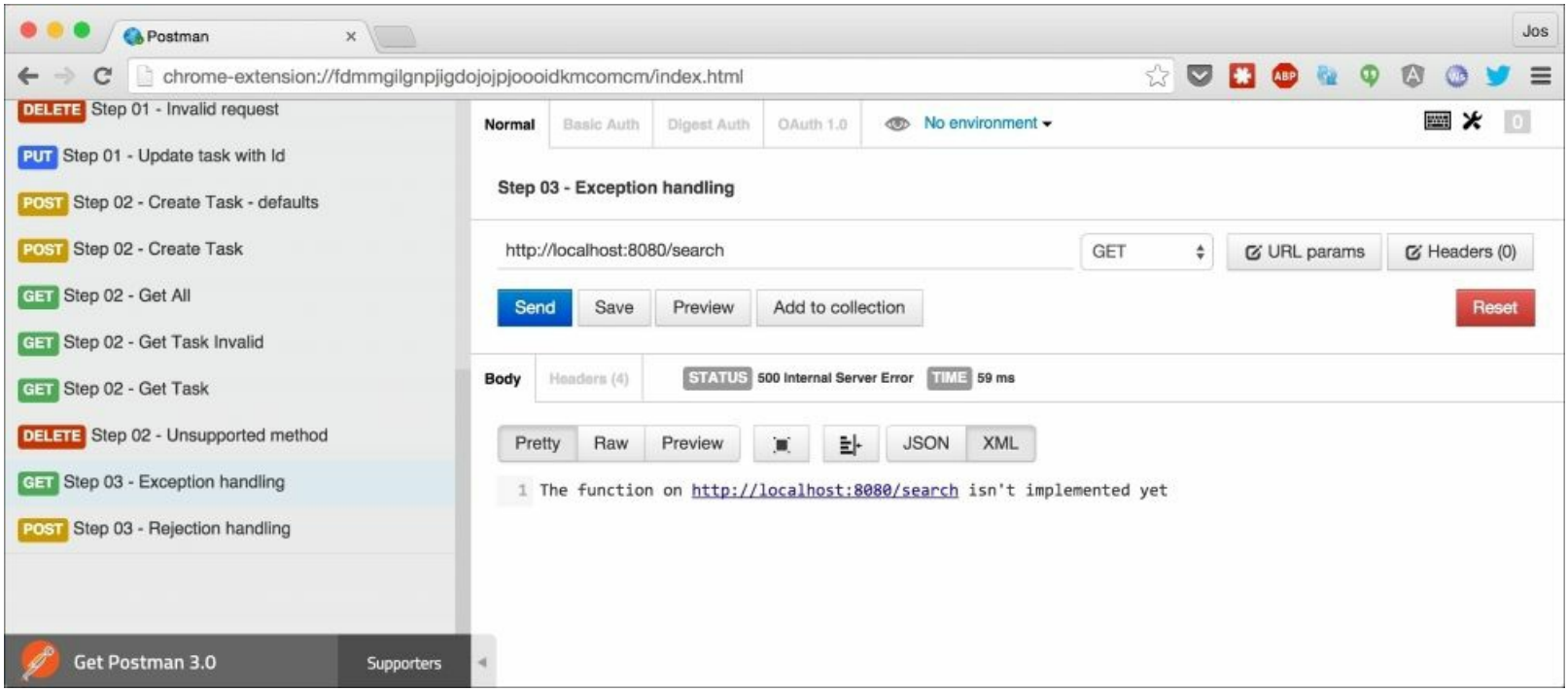
```
          }
        }
      }
      }
    }
    }
}
```

In this code we use the `validate` directive to check the incoming parameters. If one of the validations fails, the request will be rejected with a `ValidationRejection` result. We've wrapped this route with the `handleRejections` directive, so all the rejections from this route will be caught by this handler. The following code fragment shows what this handler looks like:

```
  val customRejectionHandler =
    RejectionHandler.newBuilder()
      .handle {
      case ValidationRejection(cause, exception) =>
        complete(HttpResponse(StatusCodes.BadRequest, entity =
                 s"Validation failed: $cause"))
    }.result()
```

When you want to create a rejection handler, it's easiest to use the `RejectionHandler.newBuilder` object. This object provides you with a couple of functions you can call to define this handler's behavior. The `handle` function we used here allows us to handle single rejections, and we used this to respond with a 400 bad request, when a `ValidationRejection` occurs. Any other rejections bubble up the route. Besides `handle`, you can also use `handleAll` to handle multiple instances of the same rejection at the same time. Finally, there is a specific `handleNotFound` function you can use to define behavior for when a resource isn't found.

Handling exceptions works in pretty much the same way. Open up Postman again and use the **Step 03 - Exception handling** request to trigger an exception:



In this case, we return an internal server error. The following code fragment shows how to do this with Akka HTTP:

```
val customExceptionHandler = ExceptionHandler {
  case _: IllegalArgumentException =>
    // you can easily access any request parameter here using extractors.
    extractUri { uri =>
      complete(HttpResponse(StatusCodes.InternalServerError, entity = s"The
function on $uri isn't implemented yet"))
    }
}
...
path("search") {
  handleExceptions(customExceptionHandler) {
      failWith(IllegalArgumentException("Search call not implemented"))
  }
}
```

As you can see, using an exception handler is not that different from the rejection handler we discussed previously. This time, we define an `ExceptionHandler` and use the `handleExceptions` directive to connect it to a specific part of the route. For the `ExceptionHandler`, we only provide a partial function where we specify the exceptions to catch.

Tip

In this section, we explicitly used the `handleExceptions` and `handleRejections` functions to define which handler to use for part of the route. If you want to create a custom handler that matches the complete request, you can also define the handlers as implicits. This way, they will be used for all the rejections and exceptions produced by the route. Also, good to note is that you don't necessarily

have to match all the rejections and exceptions. If you don't match a specific exception or rejection, it will bubble up to one higher in the route and, eventually, it will reach the Akka HTTP provided default handlers.

In this sample, we've so far seen the `handleRejections` and `handleExceptions` directives; there are a couple of other directives and functions available that are related to rejections and exceptions. The following table lists all of these:

Directive	**Description**
`handleRejection`	This handles the current set of rejections using the provided `RejectionHandler`.
`handleException`	This handles exceptions thrown from the inner route using the provided `ExceptionHandler`.
`cancelRejection`	This allows you to cancel a rejection coming from the inner route. You can also use the `cancelRejections` directive to cancel multiple ones.
`failWith`	This directive raises the specified exception; this should be used instead of throwing exceptions.
`recoverRejections`	This directive and `RecoverRejectionsWith` allow you to handle rejections and transform them in to a normal result.
`reject`	This rejects the request with the specified rejections.

With these directives, and the use of global rejection and exception handlers, you should be able to cleanly handle the fault situations in your REST service.

For the last subject in the discussion on Akka HTTP we'll quickly show you how to add JSON support to your service.

Adding JSON support

As a final step, we'll add JSON support to our Akka HTTP REST service. With this added, we'll be able to send JSON to our service, and Akka HTTP will automatically convert it to the case classes we use. The first thing we need to do is add an additional SBT dependency:

```
val backendDeps = Seq (
  "com.typesafe.akka" %% "akka-stream-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-core-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-experimental" % akkaHttpVersion,
  "com.typesafe.akka" %% "akka-http-spray-json-experimental" % akkaHttpVersion
)
```

Once added, we need to inform Akka HTTP which of our case classes should be marshaled to and from JSON. The common way to do this is by defining a specific trait, which contains the implicits needed by Akka HTTP. For our example, we define that trait like this:

```
trait AkkaJSONProtocol extends DefaultJsonProtocol {
  implicit val statusFormat = jsonFormat1(Status.apply)
  implicit val noteFormat = jsonFormat2(Note.apply)
  implicit val personFormat = jsonFormat1(Person.apply)
  implicit val taskFormat = jsonFormat6(Task.apply)
}
```

This code is pretty self-explanatory. What we do here is define which case classes should be marshaled to and from JSON. Next, we need to make sure we import Akka HTTP's JSON library:

```
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport
```

And finally, make sure we extend this trait in our application:

```
object AkkaHttpDSLStep4 extends App with AkkaJSONProtocol
```

At this point, we can remove the custom marshallers we defined earlier in this chapter, and make all our methods just return the `Task` instances instead of converting the `Task` instances into strings. For instance, to retrieve all the current stored tasks, we just do this:

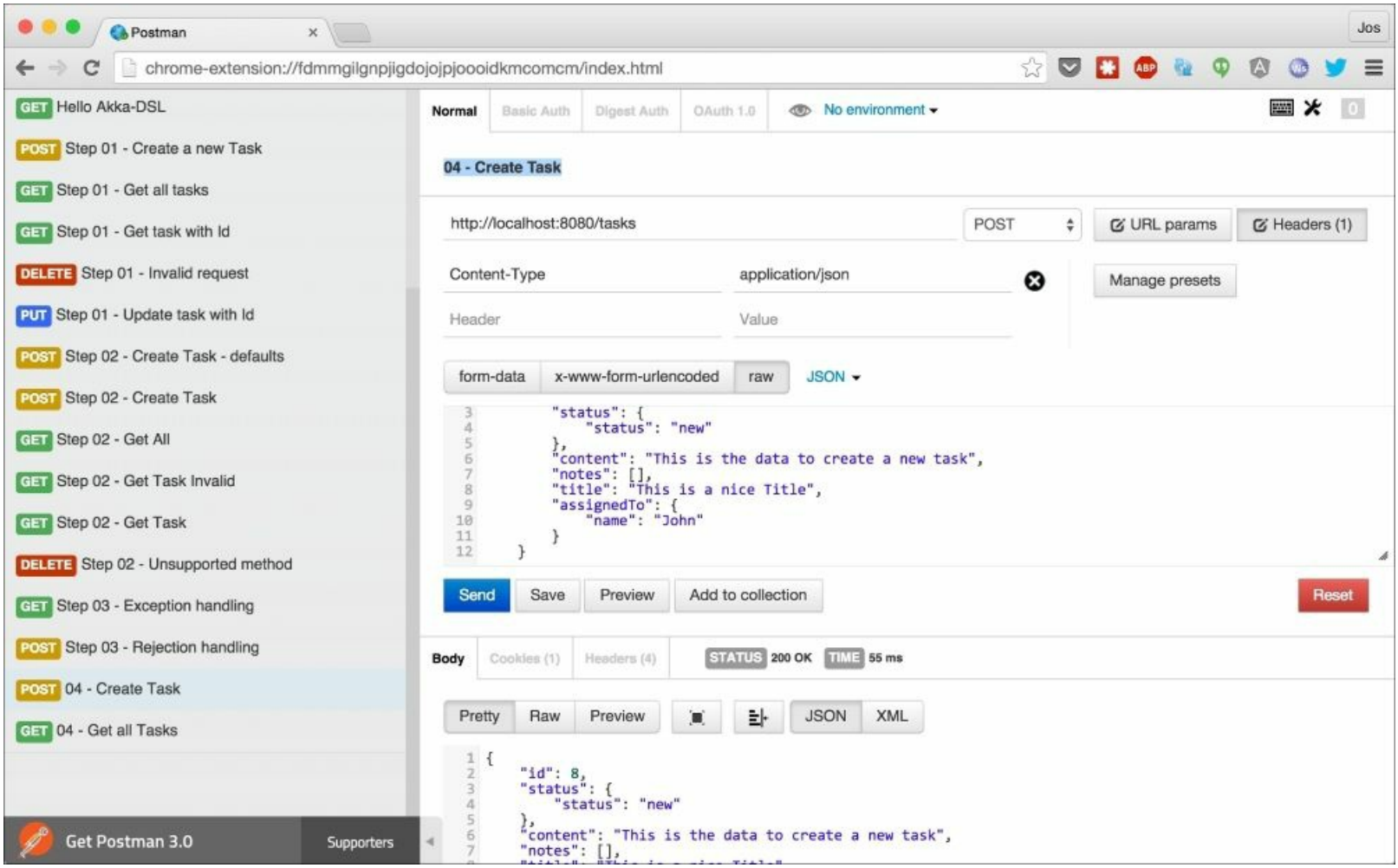
```
path("tasks") {
  get {
    complete {
      TaskService.all
    }
  }
}
```

To also convert incoming JSON to a `Task` object, we need to use the `entity` directive. The code to add a `Task` object now becomes this:

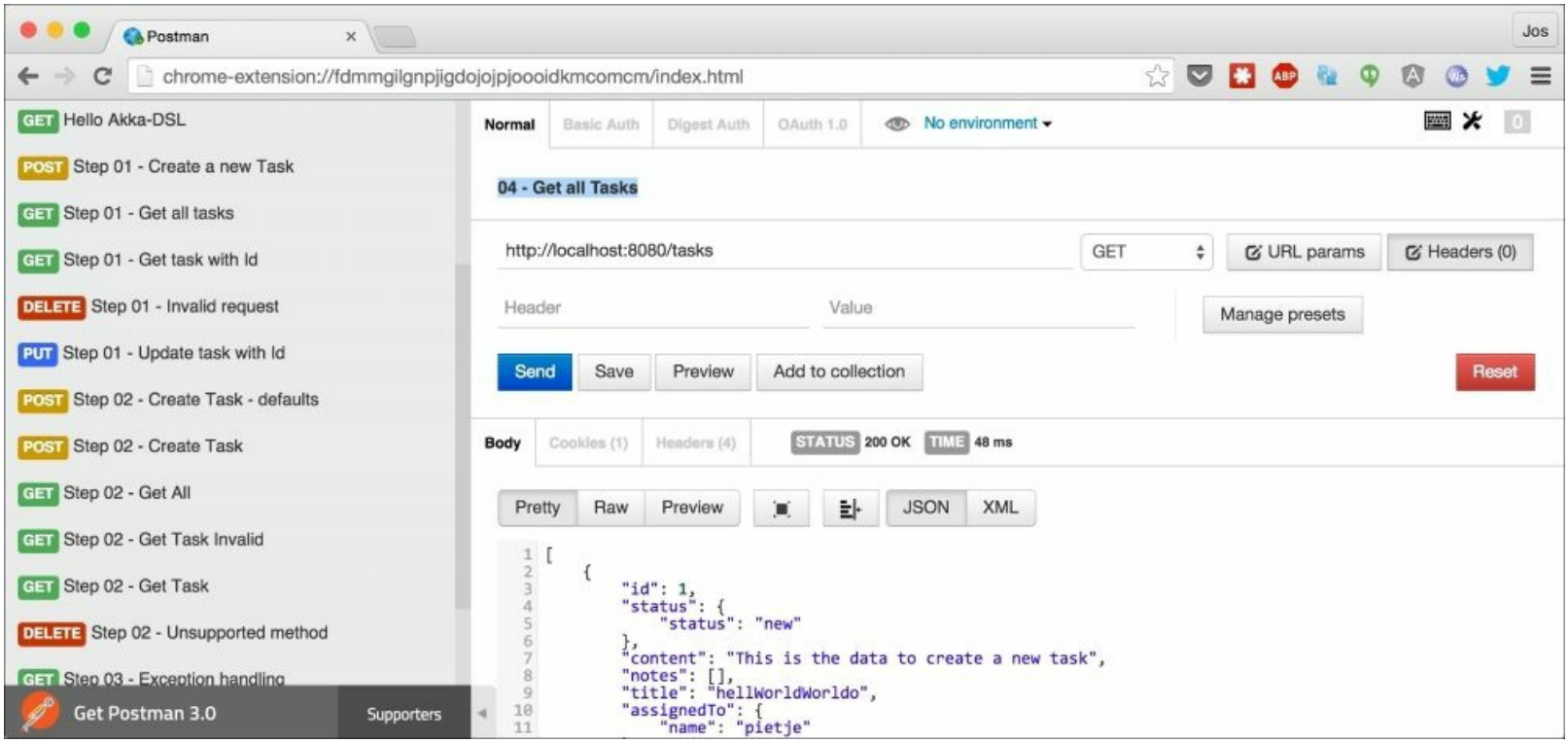
```
post {
```

```
    (entity(as[Task])) { task => {
      complete {
        TaskService.insert(task)
      }
    }
  }
}
```

With these steps taken, we can open up Postman and see whether it works. In Postman, open request, **04 - Create Task**, and run it:



Here, we can see that the JSON sent is being processed by the server and is successfully added. Now repeat this a couple of times and then use the **04 - Get all Tasks** request to see whether we can retrieve the list of all the tasks we've added:


Postman
Jos
chrome-extension://fdmmgilgnpjigdojojpjoooidkmcomcm/index.html
GET Hello Akka-DSL
POST Step 01 - Create a new Task
GET Step 01 - Get all tasks
GET Step 01 - Get task with Id
DELETE Step 01 - Invalid request
PUT Step 01 - Update task with Id
POST Step 02 - Create Task - defaults
POST Step 02 - Create Task
GET Step 02 - Get All
GET Step 02 - Get Task Invalid
GET Step 02 - Get Task
DELETE Step 02 - Unsupported method
GET Step 03 - Exception handling
Get Postman 3.0
Supporters
Normal
Basic Auth
Digest Auth
OAuth 1.0
No environment
04 - Get all Tasks
http://localhost:8080/tasks
GET
URL params
Headers (0)
Header
Value
Manage presets
Send
Save
Preview
Add to collection
Reset
Body
Cookies (1)
Headers (4)
STATUS 200 OK
TIME 48 ms
Pretty
Raw
Preview
JSON
XML
1 [
2 {
3 "id": 1,
4 "status": {
5 "status": "new"
6 },
7 "content": "This is the data to create a new task",
8 "notes": [],
9 "title": "hellWorldWorldo",
10 "assignedTo": {
11 "name": "pietje"

Summary

In this chapter, we explored most of the features provided by Akka HTTP. You learned to use the available directives to match specific request properties and extract information from the headers, parameters, and body of the request. Besides directives to process incoming requests, Akka HTTP also provides directives to create and customize the response that is sent back to the client. If errors occur while processing the request, Akka HTTP provides standard exception and rejection handlers. You also learned how you can override the default behavior by adding custom handlers. Finally, this chapter also showed you how easy it is to add JSON support and automatically marshal your classes to and from JSON.

In the next chapter, we'll look at the final REST framework that we'll discuss in this book, **Play 2**.

Chapter 6. Creating REST Services with the Play 2 Framework

For the last REST framework in this book, we will look at Play 2. Play 2 is a modern web framework that can be used to create complete applications. Part of the tools this framework provides allows you to quickly and easily create REST services. In this chapter, we'll focus on that part.

In this chapter, we'll discuss the following topics:

- Route matching with the route file
- Working with the incoming HTTP request and customizing the response
- Adding JSON support, custom validations, and error handling

First, let's get a bit more information about what the Play 2 framework is, and where you can find additional information about it.

An introduction to the Play 2 framework

The Play 2 framework is one of the best-known and most used web frameworks in the Scala ecosystem. It provides a very developer-friendly way of creating web applications and REST services. Some of the most interesting features of Play 2 are:

- **Automatically reload changes**: When creating Play 2 applications, you don't have to restart the application after a change in the code. Just resend a request, and Play 2 will automatically reload the changes.
- **Scalable**: Play 2 is based on Akka. Through the Akka framework, it provides an easy way to scale up and scale out.
- **Great tool support**: Play 2 is based on SBT, the standard build tool for Scala. Through an SBT plugin, you can easily start, reload, and distribute your Play 2 applications.

Even though Play 2 isn't specifically build as a REST framework, we can use parts of it to easily create REST applications, and still take advantage of the features of Play 2.

Hello World with Play 2

To create a simple Hello World REST service in Play 2, we first need to set up the SBT project correctly. Play 2 uses an SBT plugin to run an HTTP server that you can use to access your REST service. The first thing we need to do is add this plugin to the `plugins.sbt` file (which you can find in the `project` directory):

```
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.0")
```

Next, we define the dependencies so that we can create a Play 2-based REST service. For Play 2, we use the following dependencies:

```
lazy val playVersion = "2.4.0"

val backendDeps = Seq (
  "com.typesafe.play" %% "play" % playVersion,
  "com.typesafe.play" %% "play-docs" % playVersion
)
```

There is one more step we need to add to the SBT configuration (`build.sbt`) to be able to run the samples in this chapter:

```
addCommandAlias("runCH06-HelloPlay", "; chapter06/run -Dhttp.port=8080  -
Dplay.http.router=hello.Routes")

addCommandAlias("runCH06-runCH06Step1", "; chapter06/run -Dhttp.port=8080  -
Dplay.http.router=step01.Routes")

addCommandAlias("runCH06-runCH06Step2", "; chapter06/run -Dhttp.port=8080  -
Dplay.http.router=step02.Routes")

addCommandAlias("runCH06-runCH06Step3", "; chapter06/run -Dhttp.port=8080  -
Dplay.http.router=step03.Routes")

import PlayKeys._
lazy val chapter06 = (project in file ("chapter-06"))
  .enablePlugins(PlayScala)
  .dependsOn(common)
  .settings(commonSettings: _*)
  .settings(
    name := "chapter-06",
    libraryDependencies := DependenciesChapter6.backendDeps
  )
```

The important parts here are the `addCommandAlias` and `enablePlugins` functions. With the `addCommandAlias` function, we define on which port Play 2 should listen and which route configuration it should use (more on that in the *Working with the routes files* section). The first `addCommandAlias` function defines that we'll use the `hello.Routes` file and listen on port `8080`. To be able to run Play 2 during development, we also need to add the plugin we defined earlier to this project. We do this through the `enablePlugins(PlayScala)` call.

Now let's look a bit closer at the `hello.routes` file that we use to define the routes for this Hello World example. You can find this file in the `routes` directory. The content of the `hello.routes` file looks similar to this:

```
GET     /hello                      controllers.Hello.helloWorld
```

This means that a `GET` request that matches the `/hello` path will invoke the action defined by `controllers.Hello.helloWorld`. This action is defined in the `Hello.scala` file, which Play 2, by default, stores in the `app` directory:

```
package controllers

import play.api.mvc._

object Hello extends Controller {

  def helloWorld = Action {

    Ok("Hello Play")
  }

}
```

Now start the project by calling `sbt runCH06-HelloPlay`:

```
Joss-MacBook-Pro:rest-with-scala jos$ sbt runCH06-HelloPlay
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)

--- (Running the application, auto-reloading is enabled) ---

[info] p.a.l.c.ActorSystemProvider - Starting application default Akka system:
application
[info] p.c.s.NettyServer$ - Listening for HTTP on /0:0:0:0:0:0:0:0:8080

(Server started, use Ctrl+D to stop and go back to the console...)
```

At this point, Play 2 starts and we can access our REST service on port `8080`. Note that this time, we did not create our own custom launcher, but used the one provided by Play 2. To stop this server, you can hit *Ctrl + D*.

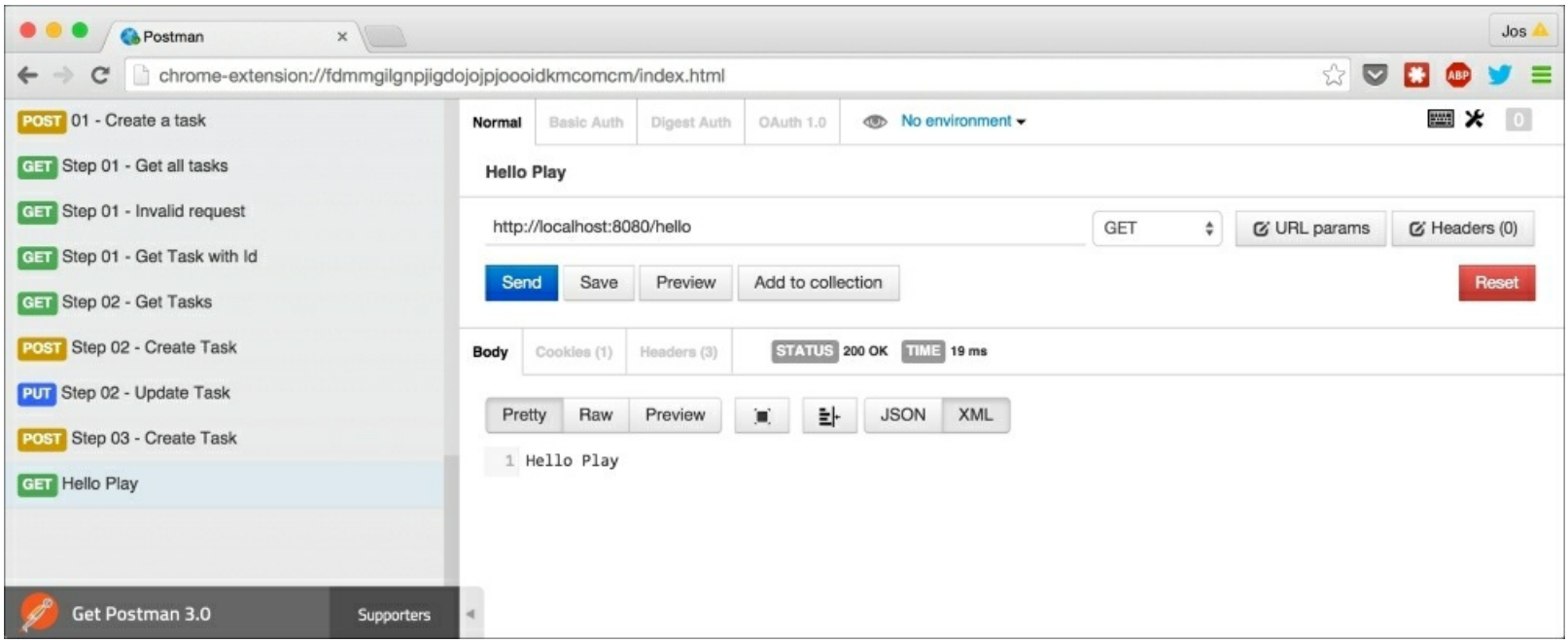
Tip

When you start Play 2 the way we do in this chapter, we start Play 2 in development mode. In development mode, Play 2 will automatically reload server changes and provide detailed error messages when problems occur. However, you shouldn't run a production Play 2 application in development mode. The performance in development mode is worse than production mode, and through the detailed error messages, much of the internal state is exposed to the clients, which isn't

something you want to do in production.

Therefore, when you are done with development, and start performance testing or deploying your application to production, you should start Play 2 in production mode. An explanation on how to do this can be found on the Play 2 website at https://www.playframework.com/documentation/2.4.x/Production.

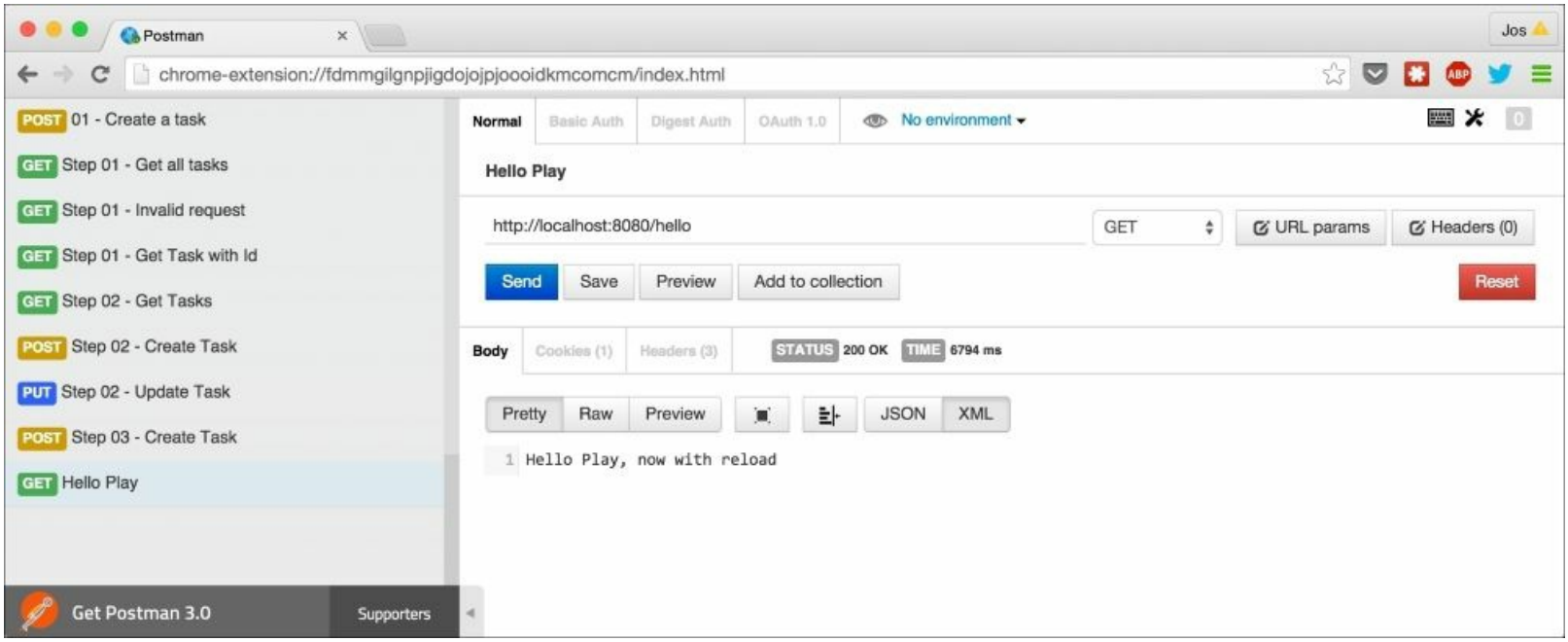
To test whether everything works correctly, open Postman and execute the **Hello Play** request from the request in the `Chapter 6` folder:



The response is as you probably expect. A `GET` request made to `/hello` just returns the `Hello Play` value. One of the great things about Play 2 is that you can change your code, without having to restart the server. As an example, change the response message in the `Hello` controller to the following:

```
def helloWorld = Action {

  Ok("Hello Play, now with reload!")
}
```

Save the file, don't stop or restart the Play 2 server; instead open up Postman and execute the same request again:



As you can see, the server now returns the updated string value, without requiring you to restart the server.

In the following section, we will explore the routes file a bit more.

Working with the routes file

In the previous section, we saw a very small routes file; it only contained a single line which handled a `GET` request to the `/hello` path. For this example, we'll use the following routes file (`step01.routes` in the `routes` folder):

```
POST     /tasks        controllers.Step1.createTask
GET      /tasks        controllers.Step1.getTasks
GET      /tasks/:id    controllers.Step1.getTask(id: Long)
DELETE   /tasks/:id    controllers.Step1.deleteTask(id: Long)
PUT      /tasks/:id    controllers.Step1.updateTask(id: Long)

GET         /*path     controllers.Step1.notImplemented(path: String)
POST        /*path     controllers.Step1.notImplemented(path: String)
PUT         /*path     controllers.Step1.notImplemented(path: String)
DELETE      /*path     controllers.Step1.notImplemented(path: String)
```

In this route file, we define a number of different HTTP verbs, paths, and actions to execute. If we look at each line, we first see the HTTP verb, which we want to match. You can use this set of HTTP verbs with Play 2: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`. Next, we see the path that we want to match. Play 2 provides a number of different constructs you can use in that position:

Path constructs	Description
`/tasks`	This matches the path, but doesn't extract any information.
`/tasks/:id`	This matches the path and extracts a segment, which is passed into the action with the provided name (`id` in this case).
`/*path`	Play 2 also supports wildcards. With `*`, you can match a path, which is extracted, assigned to the provided name (`path` in this case), and passed into the action.
`/$id<[0-9]+>`	If you start a path with `$`, Play 2 interprets this as a regular expression. The match is extracted, assigned to the name (`id`), and passed into the action.

The last part of each line in the routes file is the name of the action to invoke (see the next code fragment). When we extract a value, we can also specify the type we expect (we will see more on this in the next section), for instance, `controllers.Step1.getTask(id: Long)` converts the extracted `id` to a `Long`, and if we don't specify anything, it is passed into the action as a `String`.

In the following code fragment, we show the controller that provides the actions referring from the routes file:

```
package controllers

import play.api.mvc._
```

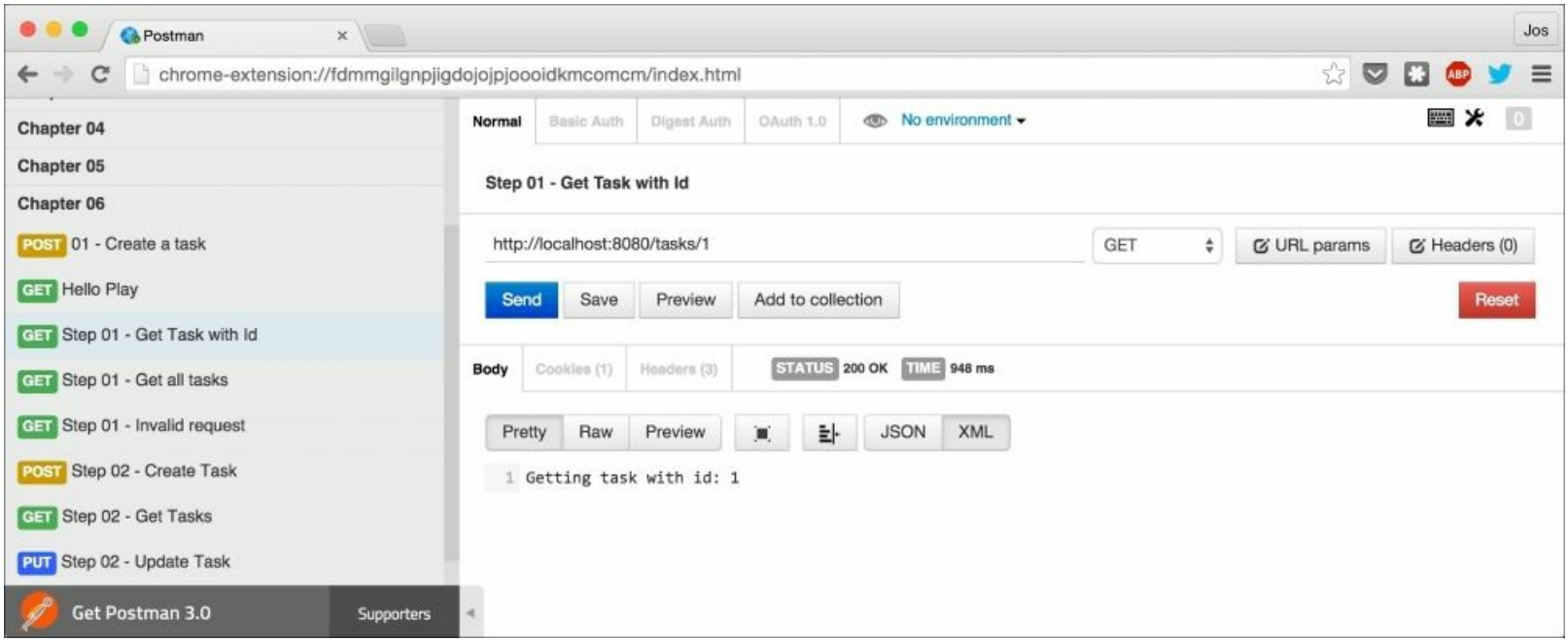
```
object Step1 extends Controller {

  def createTask = Action { request =>
    val body: Option[String] = request.body.asText
    Ok(s"Create a task with body:
         ${body.getOrElse("No body provided")}")
  }

  def getTasks = Action {
    Ok("Getting all tasks")
  }

  def getTask(id: Long) = Action {
    Ok(s"Getting task with id: $id")
  }

  def deleteTask(id: Long) = Action {
    Ok(s"Delete task with id: $id")
  }

  def updateTask(id: Long) = Action { request =>
    val body: Option[String] = request.body.asText
    Ok(s"Update a task with body:
         ${body.getOrElse("No body provided")}")
  }

  def notImplemented(path: String) = Action {
    NotFound(s"Specified route not found: $path")
  }
}
```

In the actions we defined in this `Controller`, we don't do anything special. We just specify the HTTP status, which is in the form of `Ok`, `NotFound`, or `BadRequest`, and set the body to send back to the client. Play 2 provides standard results for all the defined HTTP codes. Look in the `play.api.mvc.Results` object for all the possibilities.

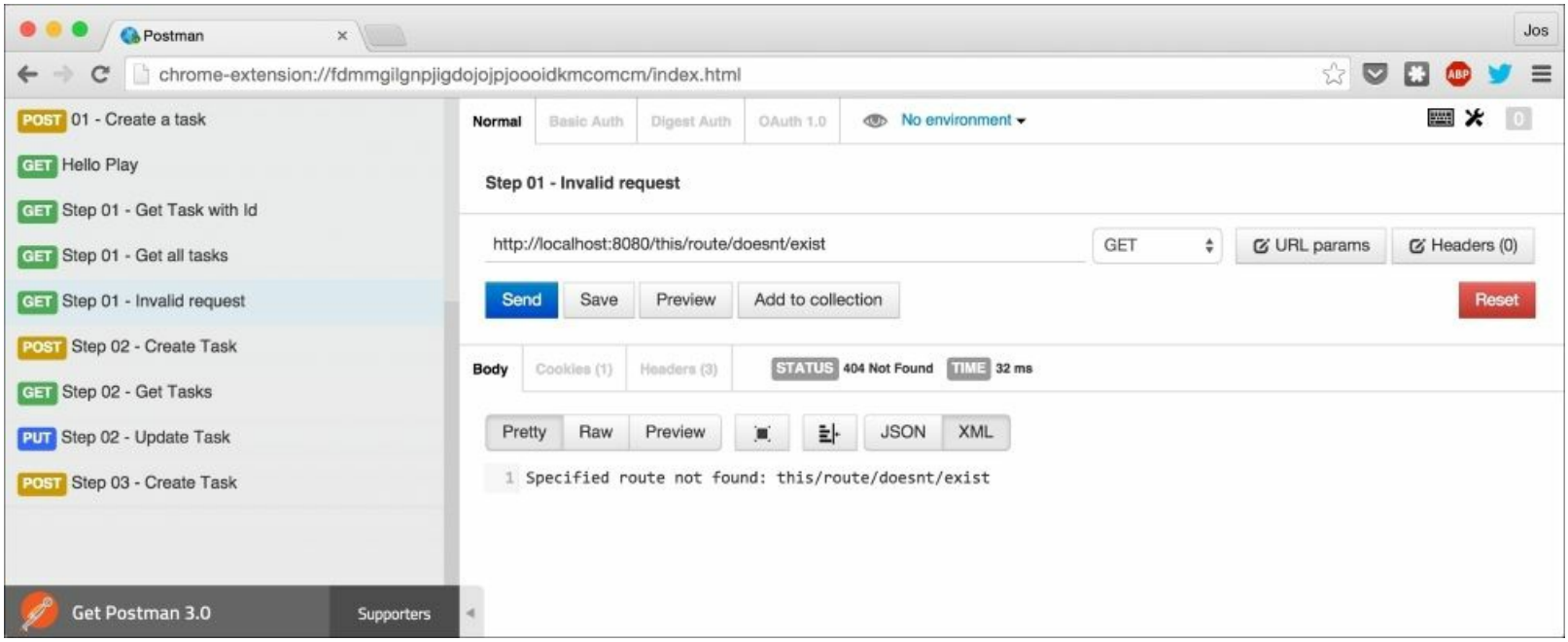
Now open up Postman, and we'll test some of the routes we just implemented. Start up the Play 2 service with the `sbt runCH06-runCH06Step1` command:

```
$ sbt runCH06-runCH06Step1
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)

--- (Running the application, auto-reloading is enabled) ---

[info] p.a.l.c.ActorSystemProvider - Starting application default Akka system:
application
[info] p.c.s.NettyServer$ - Listening for HTTP on /0:0:0:0:0:0:0:0:8080

(Server started, use Ctrl+D to stop and go back to the console...)
```

In Postman, from the `Chapter 06` folder, execute the **Step 01 – Get Task with Id** request:



In the response you can see that the ID is extracted from the URL path. If you specify a value that can't be converted to a long, the route won't match, and the action won't be executed. In the response code of this request, you can see **200 OK**, which reflects the `Ok("...")` function we used in the controller. Next, let's execute a request which is matched by one of the wildcard matchers, that is, **Step 01 - Invalid** request:



This time, you can see that the response code is **404 Not Found**, which is a result of the `NotFound("...")` function in our controller.

In the next section, we'll expand on this example and add support for `Future`. We will also look at more details of the incoming request and the ways you can customize the response.

Adding the Future support and output writers

In this section, we add more functionality to the very simple service we created in the previous section. We're going to add the functionality to create tasks using request parameters and also connect our generic `TaskService`. First, we look at the route file (`step02.routes`):

```
POST   /tasks            controllers.Step2.createTask(
                              title: String, person: Option[String],
                              status: String ?="New")
GET    /tasks            controllers.Step2.getTasks
GET    /tasks/:id        controllers.Step2.getTask(id: Long)
DELETE /tasks/:id        controllers.Step2.deleteTask(id: Long)
PUT    /tasks/:id        controllers.Step2.updateTask(
                              id: Long, title: String,
                              person: Option[String],
                              status: String ?="New")
```

Not that much has changed here, except that we now pass in more arguments to the actions defined in the `Controller` class. If you look a bit closer at the first line, you can see that we pass in a number of parameters. So far, we've only seen parameters that were extracted from the URL path. If Play 2 encounters a parameter in the signature of the `Action` method that it doesn't know, it assumes it is passed in as a request parameter. So in the case of the first route, we expect a `title`, an optional `person`, and a `status` parameter. The `title` parameter is required, the `person` parameter is optional, and for the `status` parameter, we set a default value using the `?=` syntax. In the following paragraphs, we'll look at how these routes map to the actions defined in the `Step2` controller.

Before we look at the actions, we'll have a quick look at a couple of `Writeable` implicit values:

```
object Step2 extends Controller {

  // simple implicit to convert our tasks to a simple string for now
  implicit def wTask: Writeable[Task] =
       Writeable(_.toString.getBytes, Some("application/text"))
  implicit def wListTask: Writeable[List[Task]] =
       Writeable(_.mkString("\n").getBytes, Some("application/text"))
```

Play 2 uses a `Writeable` object to determine how to convert some types to an HTTP response. In the previous sample, we just converted each task to a string and returned that. We needed to do this in each of the actions. By defining a `Writeable[Task]` object (and a `Writeable[List[Task]]` for a list of `Tasks`), we can just return an `Ok(task)` , and Play 2 will check whether an implicit `Writeable` of that type is available in scope, and use that to convert the `Task` to an output (an `Array[Byte]` instance) it understands. In the following code fragment, we see that we just return a `Task`, instead of converting it in the `Action` block:

```
  def createTask(title: String, person: Option[String],
                 status: String) = Action.async { request =>

    val body: Option[String] = request.body.asText
```

```
    val createdTask = TaskService.insert(Task(
      -1,
      title,
      body.getOrElse("No body provided"),
      person.map(Person(_)),
      List[Note](),
      MStatus(status)))

    createdTask.map(Ok(_))
  }
```

The `createTask` action is called when we make a `POST` with the required parameters. In this function, we just use the provided information and the request's body to create and store a new `Task` instance. If you look at the signature of the `TaskService.insert`, you can see that this function returns a `Future[Task]` instance. If you use the `Action {...}` function, you'll get a compile message since we want to return a `Result[Future[Task]]` instance, which Play 2 doesn't understand. This is very easy to fix. All we need to do is use `Action.async {…}` instead of `Action {…}`. Now we can just return a `Future[T]`, and Play 2 will know what to do (as long as there is an implicit `Writeable[T]` in scope).

Before we move on to the next action, we'll look a bit closer at the request object used in this example. If you add `request =>` to your action block, you can access the incoming request. In this action, we use it to access the body, but this object allows access to more request information. The most important ones are listed here:

Property	**Description**
`id`	This is the ID of the request. This starts at 1 for the first request when the application is started and increments with each request. The type of this object is a long.

`tags`	This property contains information about the route and returns a `Map[String, String]`. This information looks similar to this:
	`Map(ROUTE_COMMENTS -> this is a comment,`
	`ROUTE_PATTERN -> /tasks,`
	`ROUTE_CONTROLLER -> controllers.Step2,`
	`ROUTE_ACTION_METHOD -> createTask,`
	`ROUTE_VERB -> POST)`

`uri`	This contains the full URI of the request. This is the path and query string together. This is of type, string.
`path`	This is the path part of the URI. This returns a string
`method`	This is the method part (the verb) used to make the request. This is a string.
`version`	This is the HTTP versions of this request. This is also returned as a string.

`queryString`	This is the parsed query string in the form of a `Map[String, Seq[String]]` object, which contains the query parameters. If you want to access the raw query string, you can use the `rawQueryString` property.
`headers`	This accesses all the provided headers from the request. This object is of type, `Headers`.
`remoteAddress`	This returns the clients address as a string.
`secure`	This property is true if the client used HTTPS; it is false otherwise.
`host`	This is the host part of the URI as a string.
`domain`	The domain part of the URI as a string.
`cookies`	The cookies send along the request. This object is of type, `Cookies`.

Besides these, the `request` object also provides some additional properties to access specific headers: `acceptLanguages`, `acceptedTypes`, `mediaType`, `contentType`, and `charSet`.

Now that we've seen what we can do with the incoming request, in the following code fragment, we'll look at how we can customize the response that gets sent back to the client:

```
  def getTasks = Action.async {
    TaskService.all.map(
      Ok(_)
        .as("application/text")
        .withCookies(new Cookie("play","cookie"))
        .withHeaders(("header1" -> "header1value")))
  }
```

In this fragment, we make a call to the `TaskService.all` function, which returns a `Future[List[Task]]` object. Since we've defined a `Writeable` for `List[Task]`, all we need to do is transform the `Future[List[Task]]` to a `Future[Result]`, which we do by calling map and returning `Ok(_)`. When we return the result, we use a couple of additional functions to customize the result. We use as to set the content-type of the result, and with `withCookies` and `withHeaders`, we add a custom cookie and header. Play 2 provides the following functions for this:

Function	**Description**
`withHeaders`	This adds the provided `(String, String)` tuples as a header to the response.
`withCookies`	This adds the provided `Cookies` instance to the response.
`discardingCookies`	It is also possible to remove specific cookies from the response with this function. All the cookies, which match the provided `DiscardingCookies`, will be removed.

as	This sets the resulting content-type to the provided string value.

The following code fragment shows the last couple of actions we implemented for this step:

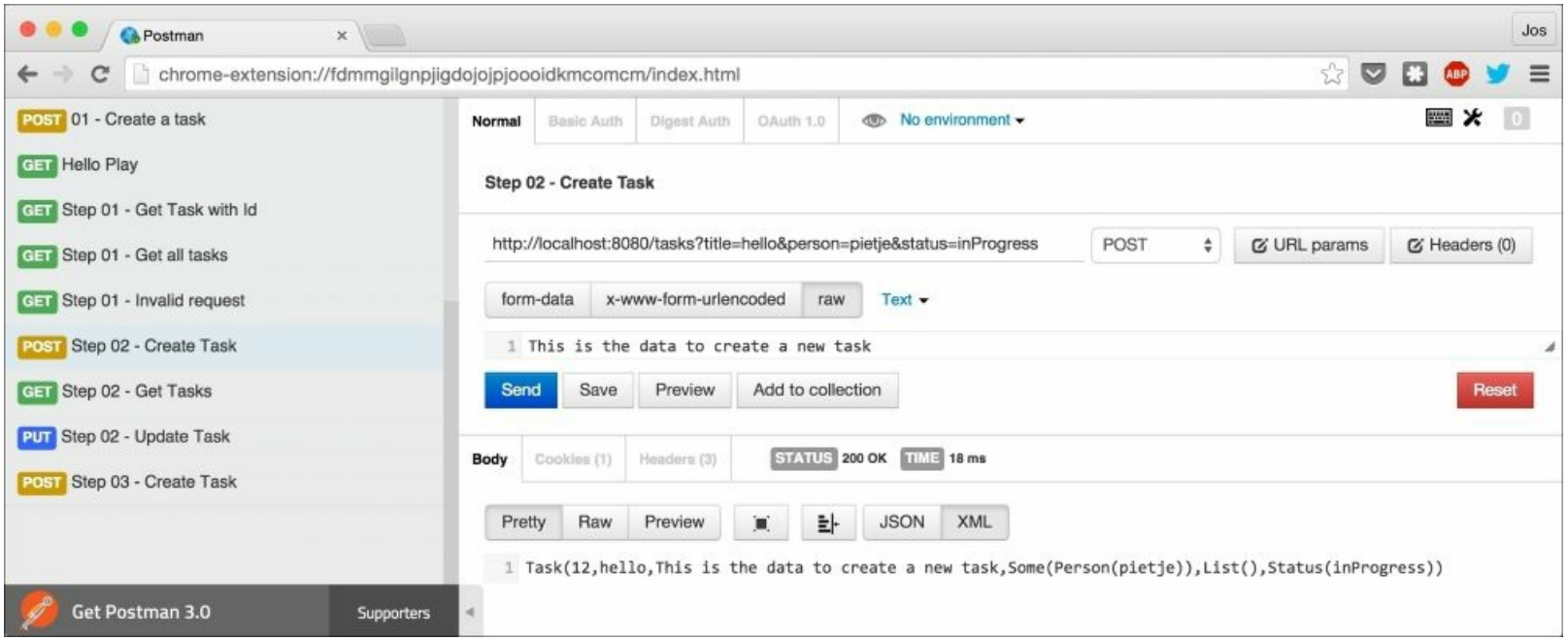
```
  def getTask(id: Long) = Action.async {
    val task = TaskService.select(id);
    task.map({
      case Some(task) => Ok(task)
      case None => NotFound("")
    })
  }


  def updateTask(id: Long, title: String,
                 person: Option[String],
                 status: String) = Action.async { request =>
    val body: Option[String] = request.body.asText

    val updatedTask = TaskService.update(id, Task(
      id,
      title,
      body.getOrElse("No body provided"),
      person.map(Person(_)),
      List[Note](),
      MStatus(status)))

    updatedTask.map({
      case Some(task) => Ok(task)
      case None => NotFound("")
    })
  }
```

As you can see, these last two actions are much the same as we've already seen. We just map over the `Future` and use pattern matching on the containing option to return either `Ok` (200) or `NotFound` (404).

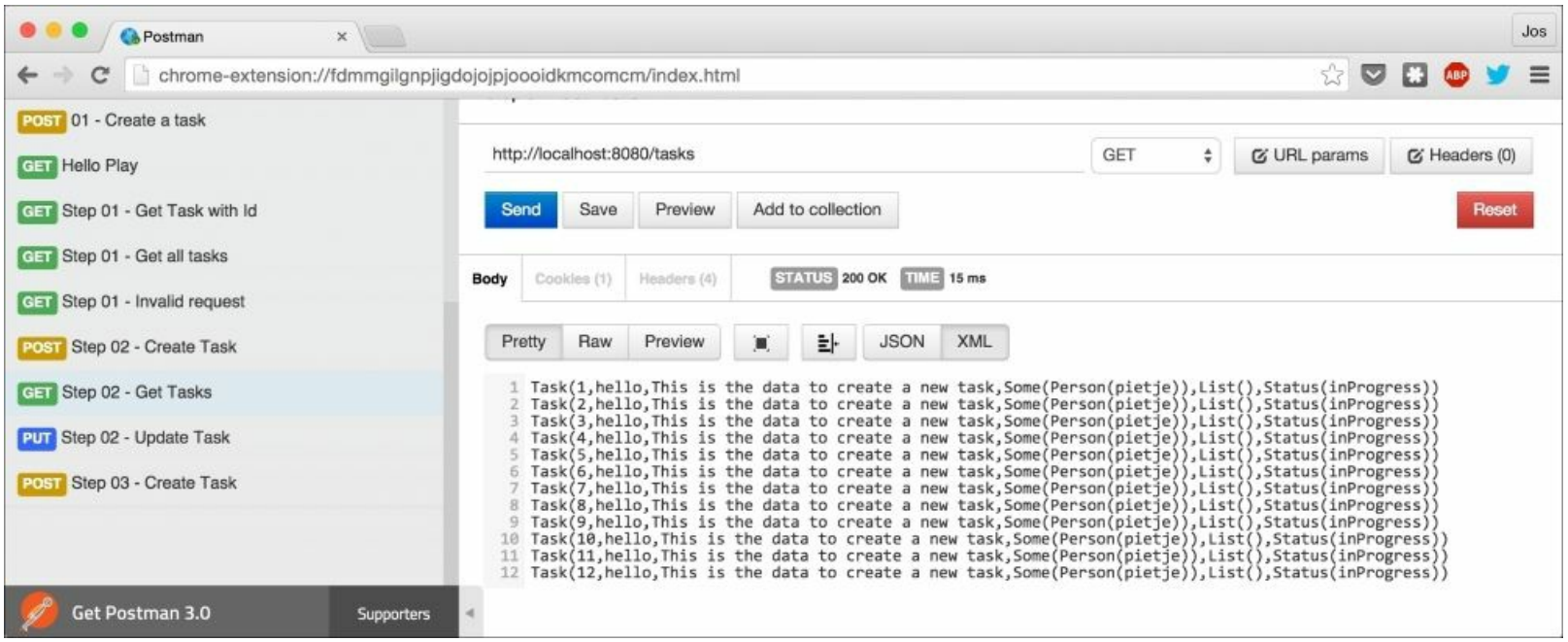
Now that we've explored the complete example, we'll check how it looks in Postman. First start up the example for this step by running `sbt runCH06-runCH06Step2`:

```
$ sbt runCH06-runCH06Step2
[info] Loading project definition from /Users/jos/dev/git/rest-with-
scala/project
[info] Set current project to rest-with-scala (in build
file:/Users/jos/dev/git/rest-with-scala/)

--- (Running the application, auto-reloading is enabled) ---

[info] p.a.l.c.ActorSystemProvider - Starting application default Akka system:
application
[info] p.c.s.NettyServer$ - Listening for HTTP on /0:0:0:0:0:0:0:0:8080

(Server started, use Ctrl+D to stop and go back to the console...)
```

Once started, first execute the **Step 02 - Create Task** request a couple of times:



As you can see, the `Task` is created asynchronously and the created `Task` is returned. Now when we run the **Step 02 – Get all tasks** request, you should see a list of all the requests, separated by line breaks:



In the next section, we'll show you how you can add JSON marshalling and error handling.

Adding JSON marshalling, validations, and error handling

As we've seen with the other frameworks, an important feature of a REST framework is the support for JSON. With Play 2, adding JSON is very easy. All it takes is defining the following implicits:

```
implicit val fmtNote = Json.format[Note]
implicit val fmtPerson = Json.format[Person]
implicit val fmtMStatus = Json.format[MStatus]
implicit val fmtTask = Json.format[Task]
```

With these implicits in scope, we can convert the relevant case classes to and from JSON by using the following two functions:

- `Json.toJson(obj)`: This converts the provided `obj` instance to JSON. This works if we've defined a `Format` object for that case class as we've done earlier.
- `request.body.asJson.map(_.as[Task])`: Converting from JSON to a case class is just as easy. Use `asJson` to convert the incoming body to a `JsValue` and then use as to convert it to a supported case class.

While this approach works great for simple scenarios, it doesn't allow you to validate the input values before you create your case objects. If you want to add validations when parsing the incoming JSON, you have to explicitly write out how a specific JSON field is mapped to a property of the case class.

To convert from and to JSON, we need to define a `Reads[T]` and a `Writes[T]` object for each class. A `Reads` object defines how we convert from the incoming JSON to a case class, and a `Writes` defines the other way around. If the `to` and `from` are symmetrical, you can also use a single `Format` implicit instead of defining the `Reads` and `Writes` separately (note that this doesn't work with case classes with only one parameter). If we look at the mapping of the `Note` case class, we can use a `Format` approach:

```
implicit def notesFormat: Format[Note] = (
   (JsPath \ "id").format[Long] and
   (JsPath \ "content").format[String])
     (Note.apply, unlift(Note.unapply))
```

In this code fragment, we map the first parameter of the case class to the `id` JSON field as a long and the second parameter to a string with name content. For the `Status` case class, we define a separate `Writes` and `Reads` and add validation:

```
implicit def statusReads: Reads[MStatus] =
   ((JsPath \ "status").read(
     minLength[String](3) andKeep
       filter(ValidationError("Status must be either New,
                              In Progress or Closed"))
```

```
        ((b: String) => List("New", "In Progress",
                             "Closed").contains(b))
   ))
   .map(MStatus(_))

 implicit def statusWrites: Writes[MStatus] = ((JsPath \
           "status").write[String])
           .contramap((_.status))
```

Note that in the import, we've aliased our `Status` case class to `MStatus` to avoid naming conflicts with the `Status` provided by Play 2. In the `statusReads` definition, you can see that we map the incoming status JSON field to our case class. We've also added two validation checks to this property. It needs to have a minimum length of `3` (`minLength`), and we check with a custom filter whether the value of `status` is one of `New`, `In Progress`, or `Closed`. For writing out JSON, we create a simple `Writes[MStatus]` instance.

In the `Reads` of `Task`, we also add some simple validations:

```
implicit def taskReads: Reads[Task] = (
     (JsPath \ "id").read[Long] and
       (JsPath \ "title").read(minLength[String](3)
                         andKeep maxLength[String](10)) and
       (JsPath \ "content").read[String] and(JsPath \
"assignedTo").readNullable[Person] and
       (JsPath \ "notes").read[List[Note]] and
       (JsPath \ "status").read[MStatus])(Task.apply _)
```

We want the title to be at least three characters and have a max of 10. Note that we've used `readNullable` in this `Reads` definition. With `readNullable`, we get an `Option[T]` object, which implies that the JSON field is also optional. The following table shows the different validation checks you can make:

Validation	Description
`max`	This checks for the maximum value of a numeric JSON property.
`min`	This checks for the minimum value of a numeric JSON property.
`filterNot`	This checks against a custom predicate. If the predicate returns true, a validation error will be created.
`filter`	This checks against a custom predicate. If the predicate returns false, a validation error will be created.
`maxLength`	This checks for the maximum length of a string JSON property.
`minLength`	This checks for the minimum length of a string JSON property.
`pattern`	This checks whether the JSON property matches the provided regular expression.

`email`	This checks whether the JSON property is an e-mail address.

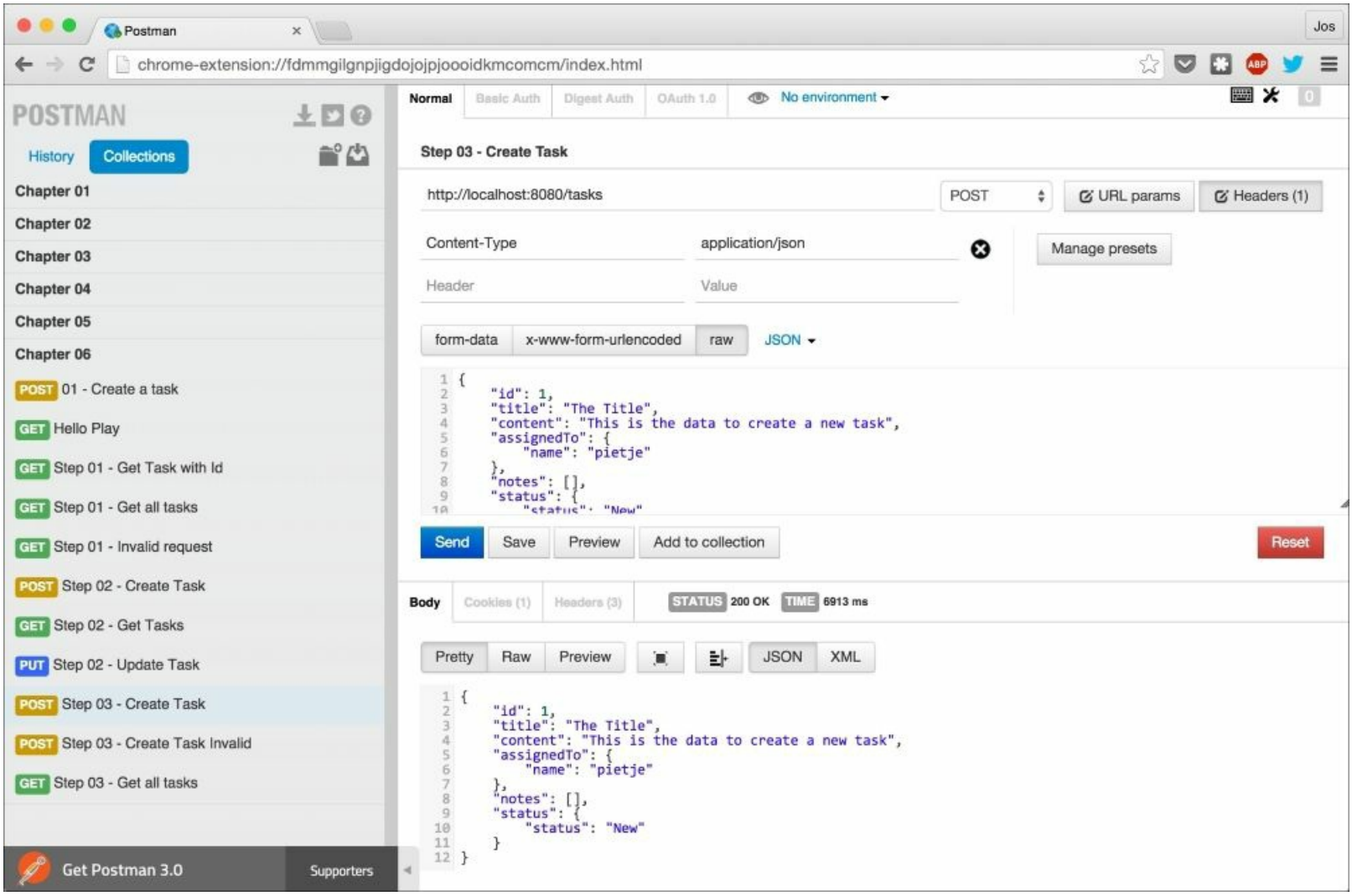
You can also add multiple checks to a single JSON property. With `andKeep` and `keepAnd` (there is a small difference in semantics of these two functions, but when used with validations, they work in exactly the same manner), both checks need to succeed, and with `or`, at least one of the checks must succeed.

Now that we've defined how to convert to and from JSON, let's look at how to use this in our actions:

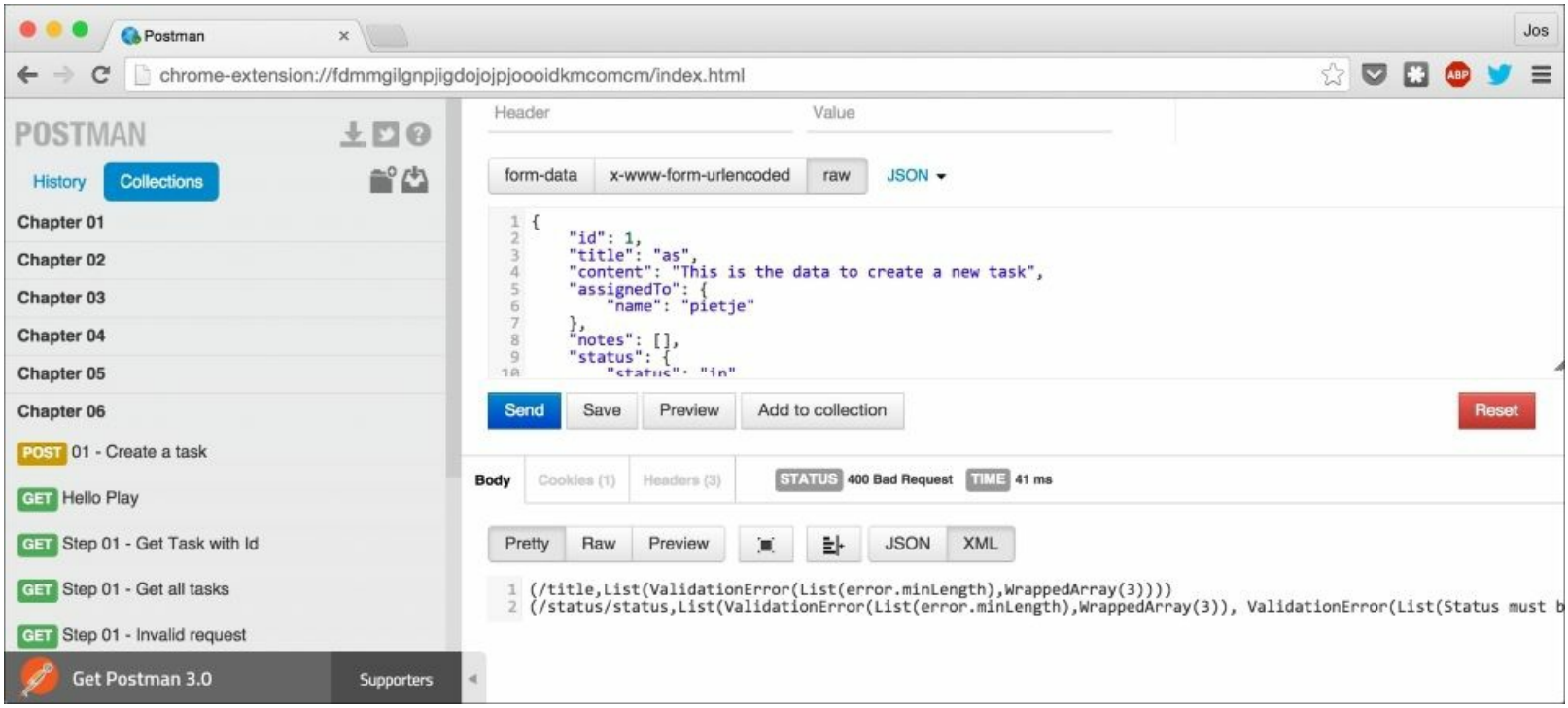
```
  def createTask = Action.async { request =>

    val body: Option[JsResult[Task]] =
             request.body.asJson.map(_.validate[Task])

    // option defines whether we have a Json body or not.
    body match {
      case Some(task) =>
        // jsResult defines whether we have failures.
        task match {
          case JsSuccess(task, _) => TaskService.insert(task).
                           map(b => Ok(Json.toJson(b)))
          case JsError(errors) =>
                          Future{BadRequest(errors.mkString("\n"))}
        }
      case None => Future{BadRequest("Body can't be parsed to JSON")}
    }
  }
```

In this code fragment, we first convert the body of the incoming request to JSON with the `asJson` function. This returns an `Option[JsValue]` object, which we map to an `Option[JsResult[Task]]` instance using the `validate` function. If our option is `None`, it means we couldn't parse the incoming JSON, and we'll return a `BadRequest` result. If we've got validation errors we get a `JsError` and respond with a `BadRequest` showing the errors; if the validation went well, we get a `JsSuccess` and add the `Task` to the `TaskService`, which responds with `Ok`.

Now open Postman and check what the resulting JSON looks like. First add a couple of `Tasks` with the **Step 03 – Create Task** request:



The JSON used in this request complies with the validations we added, so the request is processed without errors. If you execute the **Step 03 – Create Task Invalid** request, you'll see that the validations are triggered and a bad request is returned:

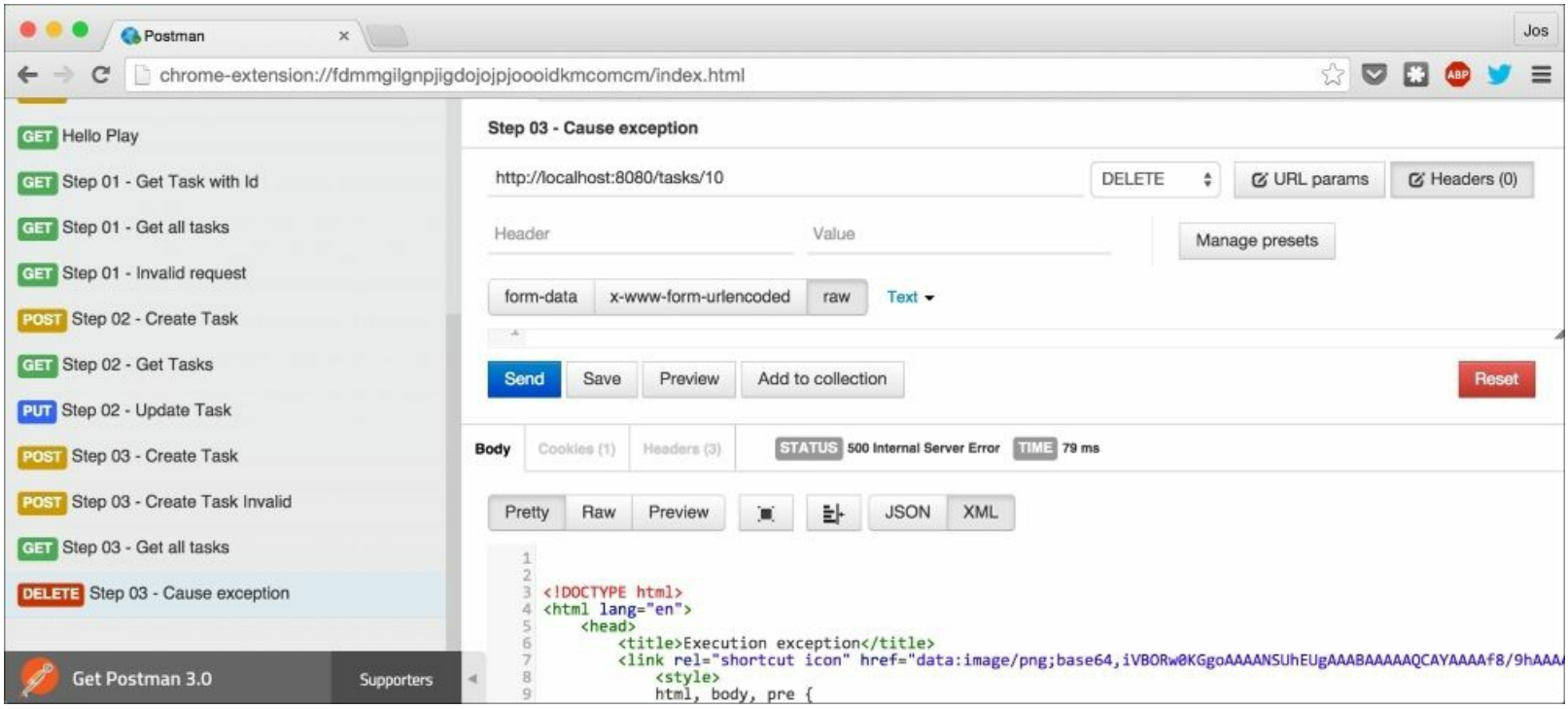


So far in this chapter, we've only handled functional errors. Incorrect data resulted in a `BadRequest` and a `Task` which couldn't be found returned a `Not Found` response. Play 2 also provides a generic way of handling unexpected errors. In the next section, we'll show you how you can add a custom error handler to your Play 2 service to process unhandled exceptions.

First, let's see what the default behavior of Play 2 is when an exception occurs. For this, we've changed the `Delete` action to the following:

```
def deleteTask(id: Long) = Action.async {
  val task = TaskService.delete(id);

  // assume this task does something unexpected and throws
  // an exception.
  throw new IllegalArgumentException("Unexpected argument");

  task.map({
    case Some(task) => Ok(task)
    case None => NotFound("")
  })
}
```

The result, when we call this action, is the following:



Play 2 responds with a `500 Internal Server` error and a lot of HTML explaining the error (note that when you run in production mode, you see a slightly different error, but still in HTML form). While this is great when developing web applications, it isn't that useful when creating a REST service. To customize the error handling, we have to provide Play 2 with an implementation of `HttpErrorHandler`. We could implement this interface from scratch, but an easier approach is to extend the default error handler (`DefaultErrorHandler`) provided by Play 2. The handler we'll use is shown here:

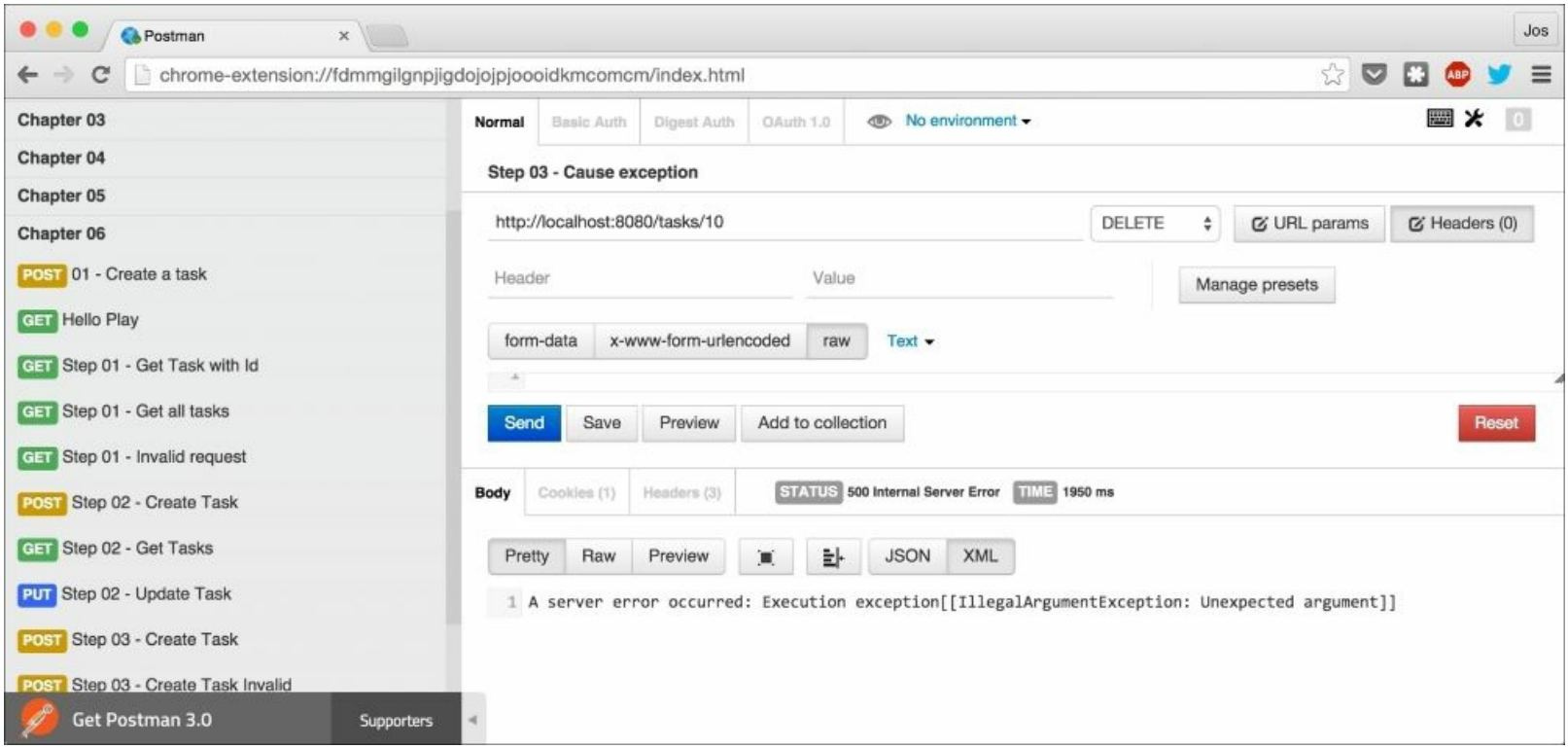
```
class ErrorHandler @Inject() (env: Environment,
                              config: Configuration,
                              sourceMapper: OptionalSourceMapper,
                              router: Provider[Router]
                              ) extends DefaultHttpErrorHandler(env, config,
sourceMapper, router) {

  override def onDevServerError(request: RequestHeader,
                                exception: UsefulException) = {
    Future.successful(
      InternalServerError("A server error occurred: " +
                          exception.getMessage)
    )
  }

  override def onProdServerError(request: RequestHeader,
                                 exception: UsefulException) = {
    Future.successful(
      InternalServerError("A server error occurred: " +
                          exception.getMessage)
    )
  }
```

```
}
```

The following functions can be overridden to define custom error handling:

Function	Description
`onClientError`	This is called when an error occurs in the 4xx range. Depending on the error, the default handler delegates to any of the following three functions.
`onBadRequest`	This is called when a bad request (400) is made.
`onForbidden`	This is called when a request is made to a forbidden resource (403).
`onNotFound`	This is called when a resource isn't found (404).
`onServerError`	This is called when a server error occurs. This function delegates to one of the following two functions.
`onDevServerError`	When in development mode, this function is called when a server error occurs.
`onProdServerError`	When in production mode, this function is called when a server error occurs.

Now we'll get a much simpler error message when we make a request that causes an internal server error:



At this point, we've discussed the most important features of the Play 2 framework.

Summary

In this chapter, we walked you through the main features of Play 2 (at least those relating to the REST part). We started with a simple service where we introduced the route file, the controller and the actions. After that, we looked at how to retrieve information from the request, parse path segments, and access query parameters. Play 2 also makes it simple to customize the response to an action. It provides standard case classes for all possible HTTP response code, and provides additional functionality to add headers and cookies to the response. Working with futures is supported by Play 2 through the `async` function. This way we can work with futures, just like we do with normal objects. Finally, we looked at JSON support and validations. Play 2 has standard functionality to convert incoming JSON to case classes and convert case classes back to JSON. When converting JSON to a case class, you can also add validations that check whether the supplied values are valid, before the case class is created.

With this chapter on Play 2, we finished the discussion of the different REST frameworks featured in this book. In the last chapter of this book, we'll look at some advanced REST framework features, such as HATEOAS, authentication, and client support.

Chapter 7. JSON, HATEOAS, and Documentation

In this last chapter, we're going to look a bit deeper into a couple of important parts of REST. We'll start with a more in-depth explanation of the different JSON libraries that are available, and after that, we'll explore the HATEOAS concept and explain how you can apply that principle to the frameworks explained in this book.

Let's start with JSON.

Working with JSON

There are many different JSON frameworks available for Scala. In this chapter, we'll look at four of the most important and most used frameworks. The following table gives a short introduction to the frameworks that we'll use:

Framework	Description
Argonaut	Argonaut is an extensive JSON library that offers a functional approach to working with JSON. It has a very extensive JSON traversal and search functionality. You can find out more about it at http://argonaut.io.
Json4s	Json4s is a library that provides a standard way of parsing and rendering JSON. It provides a standardized interface on top of the existing libraries, such as lift-json and Jackson. You can find out more about it at http://json4s.org/.
Play JSON	Play JSON provides JSON support for the Play 2 framework. This library, however, can also be used in a standalone manner, and provides a very easy to use way of working with JSON. You can find out more about it at https://www.playframework.com/documentation/2.4.x/ScalaJson.
spray-json	spray-json is a lightweight JSON framework, which provides some basic functions for processing JSON. It was part of the Spray framework, but can also be used as a standalone. You can find out more about it at https://github.com/spray/spray-json.

For each framework, we will look at how you can accomplish the following steps:

- **Parse from string to JSON object**: The input for this step is a string. We'll show you how you can use the JSON frameworks to convert this string into a JSON object.
- **Output JSON object as a string**: When you've got a JSON object, a common requirement is to print it as a string. All the frameworks provide support for this functionality.
- **Create a JSON object by hand**: Sometimes, you want to create a JSON object by hand (for example, when serializing part of a complex object). In this step, we'll explain how to do this.
- **Query a JSON object**: After converting a string in to a JSON object, a common requirement is to get specific values out of the JSON string. In this step, we'll show you the different ways in which this is supported.
- **Converting to and from a case class**: In the previous chapters, we already saw how case classes can be converted to and from JSON. All the frameworks provide explicit support for this functionality, and we'll explore how this works in this step.

Before we look at the code, we first have to make sure we have all the required libraries. Each of the JSON libraries can be added just by including a single SBT dependency. The following set of dependencies adds all the libraries:

```
val backendDeps = Seq (
  "io.argonaut" %% "argonaut" % "6.0.4",
  "org.json4s" %% "json4s-native" % "3.2.10",
  "io.spray" %%  "spray-json" % "1.3.2",
```

```
    "com.typesafe.play" %% "play-json" % "2.4.0"
  )
```

The first step we explain for each of the frameworks is how to convert a string to a JSON object. For each of the frameworks, we'll use the following input string:

```
  val json = """{
    "id": 1,
    "title": "The Title",
    "content": "This is the data to create a new task",
    "assignedTo": {
      "name": "pietje"
    },
    "notes": [],
    "status": {
      "status": "New"
    }
  }"""
```

You can also directly run the code from the various libraries. You can find the sources in the `chapter7` folder and run the examples by running SBT. From SBT, run the following commands:

```
> chapter07/run-main chapter7.json.PlayJson
> chapter07/run-main chapter7.json.Argonaut
> chapter07/run-main chapter7.json.Json4S
> chapter07/run-main chapter7.json. SprayJson
```

Now let's look at the first JSON library, Json4s.

Working with Json4s

First, we'll show how to parse the string value we just saw:

```
  val parsedJson = parse(json);
```

All you have to do is call the `parse` function on the string and the result is a `JValue` object. If the string can't be parsed, a `ParseException` will be thrown. The result, when printed, looks similar to this:

```
JObject(List((id,JInt(1)), (title,JString(The Title)), (content,JString(This is
the data to create a new task)),
(assignedTo,JObject(List((name,JString(pietje))))), (notes,JArray(List())),
(status,JObject(List((status,JString(New)))))))
```

As you can see, the string is parsed into a set of Json4s-specific classes. Now that we have got a `JValue` object, we can also convert it back to a string again:

```
  pretty(render(parsedJson)); // or compact
```

By calling `pretty(render(parsedJson)`, you get a `pretty` printed string value. If you want a compact string, you can call `compact(render(parsedJson))` instead. The result of the `pretty` function is shown here:

```
{
  "id":1,
  "title":"The Title",
  "content":"This is the data to create a new task",
  "assignedTo":{
    "name":"pietje"
  },
  "notes":[],
  "status":{
    "status":"New"
  }
}
```

The next step is creating a JSON object (a `JValue` object) by hand. Json4s provides a very convenient way to do this:

```
val notesList = Seq[Note](Note(1,"Note 1"), Note(2, "Note 2"))
  val jsonManually =
    ("id" -> 1) ~
    ("title" -> "title") ~
    ("content" -> "the content") ~
    ("assignedTo" ->
      ("name" -> "pietje")) ~
    ("notes" ->
      notesList.map { note =>
            (("id" -> note.id) ~
             ("content" -> note.content))}) ~
```

```
    ("status" ->
      ("status" -> "new"))
```

As you can see, you only need to specify the key and provide a value; Json4s will automatically create the corresponding JSON objects. And when we print it, this time using the `compact` function, we will see the following:

```
{"id":1,"title":"title","content":"the content","assignedTo":
{"name":"pietje"},"notes":[{"id":1,"content":"Note 1"},{"id":2,"content":"Note
2"}],"status":{"status":"new"}}
```

To query a JSON object, Json4s provides two approaches. You can use an XPath-like expression, as we have in the following code fragment, or you can use a comprehension (more info on that can be found at the Json4s website):

```
  println(jsonManually \\ "content") // all the content
  println(jsonManually \ "assignedTo" \ "name") // single name

  // allows unboxing

  println(jsonManually \\ "id" \\ classOf[JInt])
```

In this code fragment, we use the `\` and `\\` operators to search through the JSON object. With the `\` operator, we select a single node, and with the `\\` operator, we search through all the children. The result of the previous `println` statement is this:

```
JObject(List((content,JString(the content)), (content,JString(Note 1)),
(content,JString(Note 2))))
JString(pietje)
List(1, 1, 2)
```

Besides these operators, Json4s also provides a number of functions to search through a JSON object. You can see the available functions by looking at the `MonadicJValue` class. The final feature we look at is how to convert case classes to and from JSON. If we've already got a Json4s JSON object, we can use the extract function:

```
  implicit val formats = DefaultFormats
  val task = jsonManually.extract[Task]
```

The result of this is a `Task` instance. You can also serialize it directly to and from a string value:

```
  import org.json4s.native.Serialization
  import org.json4s.native.Serialization.{read, write}
  implicit val autoFormat = Serialization.formats(NoTypeHints)

  val taskAsJson: String = write(task)
  val backToTask: Task = read[Task](taskAsJson)
}
```

Using Json4s is really very straightforward. It provides the core functionality to easily create, query, and serialize JSON. Next up is Argonaut.

Working with Argonaut

Argonaut follows a more functional approach to create and parse JSON as you'll see in the following examples. We start again by converting a string object in to a JSON object:

```
 val parsed = json.parse // returns a scalaz disjunction
 val parsedValue = parsed | jString("Error parsing")
```

Argonaut extends the string object with a `parse` function. The result of this function is a `\/` instance:

```
\/-({"id":1,"status":{"status":"New"},"content":"This is the data to create a
new task","notes":[],"title":"The Title","assignedTo":{"name":"pietje"}})
```

This is similar to an `Either` object, but where an `Either` object isn't right or left-biased, the `\/` instance is right-biased (which means you can also easily use it in `for` comprehensions). To get the value out of the `\/` instance, we use the `|` operator.

Once we have a JSON value, we can easily convert it in to a string:

```
  println(parsedValue.spaces4)
```

This results in the following output:

```
{
    "id" : 1,
    "status" : {
        "status" : "New"
    },
    "content" : "This is the data to create a new task",
    "notes" : [

    ],
    "title" : "The Title",
    "assignedTo" : {
        "name" : "pietje"
    }
}
```

For output that is more compact, you can also use `spaces2` or `nospaces`. Argonout also provides a flexible way of creating JSON objects manually:

```
  val notesList = List[Note](Note(1,"Note 1"), Note(2, "Note 2"))
  val jsonObjectBuilderWithCodec: Json =
      ("status" := Json("status" := "New")) ->:
      ("notes" := notesList.map(
                  note => Json("id" := note.id,
                               "content" := note.content)) ) ->:
      ("assignedTo" := Json("name" := "Someone")) ->:
      ("content" := "This is the content") ->:
      ("title" := "The Title") ->:
      ("id" := 1) ->: jEmptyObject
```

Argonaut provides a number of operators you can use to build a JSON object:

Operator	Description
`->:`	This prepends the given value to a JSON object.
`->?`	This prepends the given optional value to a JSON object if it is set.
`-->>:`	This prepends the given value to a JSON array.
`-->>:?`	This prepends the given optional value to a JSON array if it is set.

On the Argonaut website, a couple of alternative ways of creating JSON objects can also be found.

Querying objects with Argonaut can also be done in a couple of different ways. For our example, we'll query using a lens:

```
val innerKey2StringLens = jObjectPL >=>
    jsonObjectPL("notes") >=>
      jArrayPL >=>
      jsonArrayPL(0) >=>
        jObjectPL >=>
        jsonObjectPL("id") >=>
          jStringPL
```

With this piece of code, we have defined a lens that matches a specific element in a JSON object. We always start with a `jObjectPL` function, which selects the root node of a JSON object. Next, we use the `jsonObjectPL("notes")` function to select the value of the `"notes"` key. By using `jArrayPL`, we convert the value to an array and use `jsonArrayPL(0)` to select the first element of the array. Finally, we use `JObjectPL` again to convert it in to an object, on which we query the `"id"` key and finally, convert it in to a string. Now that we have a lens, we can use it on a specific JSON object to extract the value (as an `Option[String]` instance):

```
val res = innerKey2StringLens.get(jsonObjectBuilderWithCodec))
```

Argonaut, of course, also supports converting to and from case classes. The first thing we have to do is define a codec. A codec defines how a case class maps to the keys of a JSON object:

```
object Encodings {

    implicit def StatusCodecJson: CodecJson[Status] =
      casecodec1(Status.apply, Status.unapply)("status")
    implicit def NoteCodecJson: CodecJson[Note] =
      casecodec2(Note.apply, Note.unapply)("id", "content")
    implicit def PersonCodecJson: CodecJson[Person] =
      casecodec1(Person.apply, Person.unapply)("name")
    implicit def TaskCodecJson: CodecJson[Task] =
      casecodec6(Task.apply, Task.unapply)("id", "title",
```

```
                    "content", "assignedTo", "notes", "status")
  }

  import Encodings._
```

Note that we make the codecs implicit. This way, Argonaut will pick them up when it needs to convert a string in to a case class and back again:

```
  val task = new Task(
    1, "This is the title", "This is the content",
    Some(Person("Me")),
    List[Note](Note(1,"Note 1"), Note(2, "Note 2")), Status("new"))

  val taskAsJson: Json = task.asJson

  val taskBackAgain: Task =
              Parse.decodeOption[Task](taskAsJson.spaces4)
```

When you've defined a codec for a specific case class, you can just call the `asJson` function to convert the case class in to a `Json` object. To convert from a JSON string to a case class we can use the `Parse.decodeOption` function (Argonaut also provide a `decodeEither` function, a `decodeValidation` and a `decodeOr` if you want other wrappers instead of `Option`).

In the chapter on Akka HTTP, we already mentioned that, for JSON support, we use the spray-json library. In the following section, we'll dive a bit deeper into this library.

Working with spray-json

spray-json provides a very easy way to work with JSON strings. To parse a string to a JSON object you can just call `parseJson` on the string value.

```
  import spray.json._

  val parsed = json.parseJson
```

The result, when printed, looks similar to this:

```
{"id":1,"status":{"status":"New"},"content":"This is the data to create a new
task","notes":[],"title":"The Title","assignedTo":{"name":"pietje"}}
```

Of course, we can also convert a JSON object back to a string by using either the `prettyPrint` or `compactPrint` functions:

```
println(parsed.prettyPrint) // or .compactPrint
```

The result from `prettyPrint` looks similar to this:

```
{
  "id": 1,
  "status": {
    "status": "New"
  },
  "content": "This is the data to create a new task",
  "notes": [],
  "title": "The Title",
  "assignedTo": {
    "name": "pietje"
  }
}
```

When you want to create a JSON object by hand, spray-json provides a number of basic classes you can use for that (`JsObject`, `JsString`, `JsNumber`, and `JsArray`):

```
val notesList = Seq[Note](Note(1,"Note 1"), Note(2, "Note 2"))
  val manually = JsObject(
    "id" -> JsNumber(1),
    "title" -> JsString("title"),
    "content" -> JsString("the content"),
    "assignedTo" -> JsObject("name" -> JsString("person")),
    "notes" -> JsArray(
      notesList.map({ note =>
        JsObject(
        "id" -> JsNumber(note.id),
        "content" -> JsString(note.content)
        )
      }).toVector),
   "status" -> JsObject("status" -> JsString("new"))
  )
```

The result is pretty much the same object we saw in the first example of the section:

```
{"id":1,"status":{"status":"new"},"content":"the content","notes":
[{"id":1,"content":"Note 1"},{"id":2,"content":"Note
2"}],"title":"title","assignedTo":{"name":"person"}}
```

The next step we'll look at is how we can query a JSON object for a specific field. This is something for which spray-json doesn't provide specific functions or operators. The only way to access a specific field or value is by using the `getFields` or `fields` functions:

```
println(manually.getFields("id"));
println(manually.fields)
```

The `getFields` function returns a `Vector` object, containing all the fields on the current object that match this name. The fields function returns a `Map` object of all the fields.

The last feature we look at in each of the frameworks is how we can use it to convert from case classes to JSON and back again. In the chapter on Akka HTTP, we already showed you how to do this with spray-json:

```
val task = new Task(
    1, "This is the title", "This is the content",
    Some(Person("Me")),
    List[Note](Note(1,"Note 1"), Note(2, "Note 2")), Status("new"))

  object MyJsonProtocol extends DefaultJsonProtocol {
    implicit val noteFormat = jsonFormat2(Note)
    implicit val personFormat = jsonFormat1(Person)
    implicit val statusFormat = jsonFormat1(Status)
    implicit val taskFormat = jsonFormat6(Task)
  }

  import MyJsonProtocol._
  val taskAsString = task.toJson

  // and back to a task again
  val backToTask = taskAsString.convertTo[Task]
```

What we do is extend the `DefaultJsonProtocol` and define, for each of our case classes, how they should be mapped through and from JSON. spray-json provides us with a very convenient helper function called `jsonFormat#(object)`, where the `#` corresponds to the number of arguments of the case class. With this function, we can define the default marshalling for our case classes, like we did in the preceding example. To use these implicit conversions, all we have to do is bring them in scope, and we can use the `toJson` function on our case classes to serialize them to `Json`, and use `convertTo` to convert JSON back to our case class.

Working with Play JSON

The final JSON library is also one we have touched upon in the previous chapters—Play JSON. As you'll see in the code, the way this library works closely resembles the spray-json library. Let's start by looking at converting from a string into a JSON object:

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

val fromJson = Json.parse(json)
```

Very simple, just call the `parse` function and provide the JSON, and the result is the following JSON object (when printed):

```
{"id":1,"title":"The Title","content":"This is the data to create a new
task","assignedTo":{"name":"pietje"},"notes":[],"status":{"status":"New"}}
```

To convert the JSON object to a string, we can either call the `stringify` function directly on the JSON object or use the `Json.prettyPrint` function:

```
println(Json.prettyPrint(fromJson))
```

The `prettyPrint` function returns the following result:

```
{
  "id" : 1,
  "title" : "The Title",
  "content" : "This is the data to create a new task",
  "assignedTo" : {
    "name" : "pietje"
  },
  "notes" : [ ],
  "status" : {
    "status" : "New"
  }
}
```

Nothing too special so far. The same goes for creating JSON objects manually. Just like spray-json, Play JSON provides you with a set of base classes (`JsObject`, `JsNumber`, `JsString`, `JsObject`, and `JsArray`) that you can use to create your JSON object:

```
// 3. Create JSON object by hand.
val notesList = Seq[Note](Note(1,"Note 1"), Note(2, "Note 2"))
val manually = JsObject(Seq(
  "id" -> JsNumber(1),
  "title" -> JsString("title"),
  "content" -> JsString("the content"),
  "assignedTo" -> JsObject(Seq("name" -> JsString("person"))),
  "notes" -> JsArray(
    notesList.map({ note =>
      JsObject(Seq(
```

```
        "id" -> JsNumber(note.id),
        "content" -> JsString(note.content)
      ))
    })),
  "status" -> JsObject(Seq("status" -> JsString("new")))
))
```

Now, let's go to querying. This is where Play JSON provides us with a couple of very useful operators:

```
println(manually \\ "content")
println(manually \ "assignedTo" \ "name")
println((manually \\ "id" )(2))
```

With the `\\` operator, we look for all the fields in the complete tree matching the field and return that as a `List` object, and with the single `\` operator, we look for a field in the current object. What makes this very easy to use is that these operators can easily be nested as you can see from the previous code fragment. When we look at the output, we see the following:

```
List("the content", "Note 1", "Note 2")
JsDefined("person")
2
```

Converting to and from case classes is also very straightforward with this library. We first define a set of implicit conversions, simply by calling `Json.format[T]`:

```
object Formats {
  implicit val noteFormat = Json.format[Note]
  implicit val statusFormat = Json.format[Status]
  implicit val personFormat = Json.format[Person]
  implicit val taskFormat = Json.format[Task]
}
```

And with these implicit conversions defined, we can use the `toJson` and `fromJson[T]` functions to convert our case classes to and from JSON:

```
import Formats._

val task = new Task(
  1, "This is the title", "This is the content", Some(Person("Me")),
  List[Note](Note(1,"Note 1"), Note(2, "Note 2")), Status("new"))

val toJson = Json.toJson(task)
val andBackAgain = Json.fromJson[Task](toJson)
```

Let's quickly recap the frameworks before we move on to the next subject.

JSON frameworks summary

So, which JSON framework is the best? Well, the general answer, of course, is it depends. All the frameworks have their advantages and disadvantages. If I do have to make a choice, I'd say that for simple JSON needs, Json4s is a really great choice. It provides a very easy way to create JSON objects from scratch, has an intuitive way of querying data, and allows you to easily convert to and from case classes. If you have more complex requirements, Argonaut is a very interesting choice. It provides a very functional way of JSON processing and has a number of interesting features for creating new JSON objects and querying the existing ones.

HATEOAS

In the first chapter, we looked at the definition of what a RESTful service is. Part of that definition is that a REST service should use HATEOAS, which is an acronym for Hypertext As The Engine Of Application State. What this means is that to be truly RESTful, our services doesn't just need to provide more information than simply the JSON representation of a resource, but it should also provide information about the state of the application. When we talk about HATEOAS, we have to deal with the following two main principles:

- **Hypermedia/mime-types/media-types/content-types**: The hypermedia of a resource describes the current state of a resource. You can look at this as a sort of contract that describes the resource we're working with. So, instead of setting the type of a resource to `application/json`, you define a custom content-type like `application/vnd.restwithscala.task+json`.
- **Links**: The second part of HATEOAS is that a resource representation needs to have links to other states of the resource and actions that can be executed on that resource.

For instance, the following code provides information about the current response through the `self` link and uses media-types to indicate what to expect from these links:

```
{
  "_links" : [ {
    "rel" : "self",
    "href" : "/tasks/123",
    "media-type" : "application/vnd.restwithscala.task+json"
  }, {
    "rel" : "add",
    "href" : "/project/123/note",
    "media-type" : "application/vnd.restwithscala.note+json"
  } ],
  "id" : 1,
  "title" : "This is the title",
  "content" : "This is the content",
  "assignedTo" : {
    "name" : "Me"
  },
  "notes" : [ {
    "id" : 1,
    "content" : "Note 1"
  }, {
    "id" : 2,
    "content" : "Note 2"
  } ],
  "status" : {
    "status" : "new"
  }
}
```

Since media-types are an important part of the resource, we don't just need to be able to set the

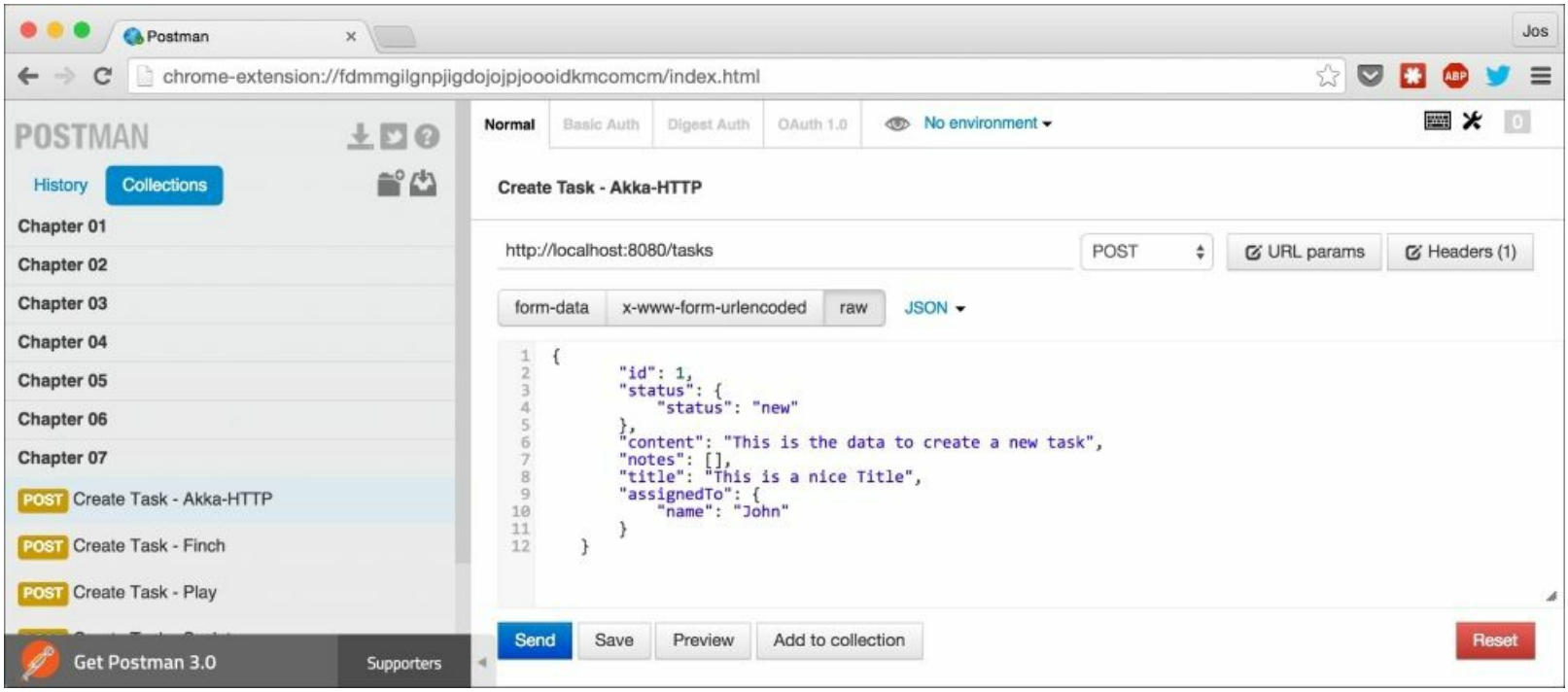
media-type on the response, but also filter based on the incoming media-type, since a different media-type on a specific endpoint can have a different meaning.

Handling media-types

Now let us walk through the frameworks discussed in this book and see how they handle media-types. You can, of course, run all these examples from the code provided with this book. You can use the following commands to start the various servers:

```
sbt runCH07-Finch
sbt runCH07-Unfiltered
sbt runCH07-Scalatra
sbt runCH07-akkahttp
sbt runCH07-play
```

We have also provided a number of requests in Postman that you can use to test whether media handling works. You can find these in the **Chapter 07** collection:



The first framework we'll explore is Finch.

Handling media-types with Finch

To handle media-types with Finch, we're going to create a filter. This filter, and the code to glue everything together, is shown here:

```
val MediaType = "application/vnd.restwithscala.task+json"
val filter = new SimpleFilter[HttpRequest, HttpResponse] {
  def apply(req: HttpRequest,
            service: Service[Request, HttpResponse])
                                    : Future[HttpResponse] = {
    req.contentType match {
      case Some(MediaType) => service.apply(req).map({ resp =>
```

```
          resp.setContentType(MediaType, "UTF-8")
          resp
        })
        case Some(_) => Future
          {BadRequest(s"Media type not understood, use $MediaType")}
        case None => Future
          {BadRequest(s"Media type not present, use $MediaType")}
      }
    }
  }

  val matchTaskFilter: Matcher = "tasksFilter"
  val createTask = CreateNewTask()
  val createNewTaskFilter = filter andThen createTask

  val taskCreateAPI =
     Post / matchTaskFilter /> createNewTaskFilter
```

You can create a filter by extending the `SimpleFilter` class. This filter provides access to the incoming `HttpRequest` instance and the outgoing `HttpResponse` instance. In this filter, we check whether the media-type is the correct one, and if this is the case, we process the request. If not, we return a `BadRequest` response. To give the client an indication of the type of response they are dealing with, we also set the media-type on the response object. With the filter defined, we create our route and invoke the `createNewTaskFilter` instance, which first calls the `filter` instance and then the `createTask` service. Now whenever a request with the correct media-type comes in, it is processed in the correct manner.

Handling media-types with Unfiltered

Filtering on media-types is very easy to do in Unfiltered. We use some basic Scala pattern matching to check whether the `POST` on a specific path contains the correct media-type:

```
val MediaType = "application/vnd.restwithscala.task+json"

case req @ Path("/tasks") => (req, req) match {
  case (req @ POST(_), (RequestContentType(MediaType))) =>
handleCreateTask(req).map(Ok ~> ResponseHeader("content-type",
                              Set(MediaType)) ~> ResponseString(_))
```

As you can see, all we do is pattern match two request properties, the verb (`POST`) and the content-type of the request; when those match, we process the request, and set the correct header on the response.

Handling media-types with Scalatra

The next one to discuss on the list of frameworks is Scalatra. As we have seen when discussing Scalatra, it provides a way to define a `before` function and an `after` function for a specific path. We use that functionality to check in the `before` function whether the media-type matches, and in the `after` function, we update the content-type:

```
before("/tasks") {
    (request.getMethod, request.contentType) match {
      case ("POST", Some(MediaType)) => // do nothing
      case ("POST", _) => halt(status = 400, reason = "Unsupported Mimetype")
      case (_,_) => // do nothing since it isn't a post
    }
  }

  after("/task") {
    request.getMethod match {
      case "POST" => response.setContentType(MediaType)
      case _ => // do nothing since it isn't a post
    }
  }
```

As you can see, we use pattern matching to match a specific verb and content-type. This means that if we have a `POST` verb and the correct content-type, we execute the request. If the verb matches but the content-type doesn't, we respond with a bad request and if the verb doesn't match, we just process it normally.

Handling media-types with Akka HTTP

Using media-types with Akka HTTP takes a bit more work. The reason is that Akka HTTP extracts the content-type out of the headers and adds it to the entity. This means we have to check the entity for the existence of a specific content-type instead of just checking a header. The first thing we do is define the content-type we're looking for, and a function, which we can use to transform the response object and set the correct type on the response:

```
  val CustomContentType =
  MediaType.custom("application/vnd.restwithscala.task+json",
                   Encoding.Fixed(HttpCharsets.`UTF-8`))

  def mapEntity(entity: ResponseEntity): ResponseEntity = entity match {
    case HttpEntity.Strict(contentType, data) =>
      HttpEntity.Strict(CustomContentType, data)
    case _ => throw new IllegalStateException(
       "Unexpected entity type")
  }
```

As you can see, the `mapEntity` function takes a `ResponseEntity` instance as its parameters, and returns a new one with the correct content-type. In the next code fragment, we'll show how you can check the incoming request for the correct content-type and use the previously-defined function to set the response:

```
            post {
                (entity(as[String]) & (extractRequest)) {
                  (ent, request) =>
                request.entity.contentType() match {
                  case ContentType(MediaType(
                      "application/vnd.restwithscala.task+json")
                      , _) =>
```

```
                    mapRequest({ req => req.copy(entity =
                      HttpEntity.apply(MediaTypes.`application/json`,
                      ent)) }) {
                      (entity(as[Task])) {
                        task => {
                          mapResponseEntity(mapEntity) {
                            complete {
                              TaskService.insert(task)
                            }
                          }
                        }
                      }
                    }
                  case _ => complete(StatusCodes.BadRequest,
                                     "Unsupported mediatype")
                }
            }
        }
```

A lot is happening here, so let's look at the directives we use here and why:

1. First, we extract the request and the entity of the request (the body and content-type) with the `entity` and `extractRequest` directives.
2. Next, we match the `request.entity.contentType` property of the entity, and if it matches, we create a new entity with the `application/json` content-type. We do this so that the standard JSON-to-case-class mapping of Akka HTTP still works.
3. Next, we convert the entity in to a `Task` instance, call the service, and create a response.
4. Before the response is returned, the `mapResponseEntity` function is called, which sets the content-type to our original value.

Note that instead of an approach using directives, we could also have redefined the required implicits to make the JSON conversion work with our own custom content-type.

Handling media-types with Play 2

To implement custom media-types in Play 2, we'll use an `ActionBuilder` approach. With an `ActionBuilder` approach, we can change the way an action is invoked. The following code shows what the `ActionBuilder` approach of this example looks like:

```
object MediaTypeAction extends ActionBuilder[Request] {

  val MediaType = "application/vnd.restwithscala.task+json"

  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result])
= {
    request.headers.get("Content-Type") match  {
      case Some(MediaType) => {
          block(request)
      }
      case _ => Future{BadRequest("Unsupported mimetype")}
    }
```

```
  }
}
```

Here, we define a new type of action, called a `MediaTypeAction`. When we use this action instead of the normal `Action` class, first the content-type of the incoming message is checked; if it matches, the request is processed; if not, the request is ignored and a `BadRequest` response is generated.

We can use this new `MediaTypeAction` in the following manner:

```
  def createTask = MediaTypeAction.async((parse.tolerantJson)) { request =>

    val body = request.body.validate[Task]

    // option defines whether we have a JSON body or not.
    body match {
        case JsSuccess(task, _) => TaskService.insert(task).map(
           b => Ok(Json.toJson(b)).as(MediaTypeAction.MediaType))
        case JsError(errors) => Future{BadRequest(errors.mkString("\n"))}
    }
  }
```

As you can see, we just replace the `Action` with `MediaTypeAction` and that's it. When this action is called, first the code from `MediaTypeAction` is executed, then the code is provided to the `Action`. To properly convert the incoming data to a JSON object, we need to make some small changes to the way we handle JSON. We use an explicit body parser (`parse.tolerantJson`) to parse the incoming JSON. With the function, we don't check whether the provided content-type is `application/json`, but just convert the body.

In the beginning of this section, we explained two important parts of HATEOAS: media-type handling and supporting links. In the next section, we'll show a way you can add links to your case classes easily.

Using links

For the links, we're going to create a very simple extension to our model. Instead of just sending the case class serialized to JSON, we're adding a `_links` object. This object can contain different links. For instance, it can not only define a link to the resource itself, but also can contain links to actions that can be executed on this resource. The final JSON we're aiming for looks similar to this:

```
{
  "_links" : [ {
    "rel" : "self",
    "href" : "/tasks/123",
    "media-type" : "application/vnd.restwithscala.task+json"
  } ],
  "id" : 1,
  "title" : "This is the title",
  "content" : "This is the content",
  "assignedTo" : {
    "name" : "Me"
  },
  "notes" : [ {
    "id" : 1,
    "content" : "Note 1"
  }, {
    "id" : 2,
    "content" : "Note 2"
  } ],
  "status" : {
    "status" : "new"
  }
}
```

As an example, we'll use Play JSON, but pretty much the same approach can be used with the other JSON libraries explored in this chapter. The first thing we do is define what the links will look like. For this, we define a trait and a case class:

```
import org.restwithscala.common.model.{Note, Person, Status, Task}
import play.api.libs.json._
import play.api.libs.functional.syntax._

object PlayJsonLinks extends App {

  trait HasLinks {
    val links: List[Links]
  }

  case class Links(rel: String, href: String, `media-type`: String)
```

When we create a new `Task`, we can extend from this trait to add links. As we have explained before, Play JSON uses implicit values to determine how to serialize a specific class to JSON. For this scenario, we define the following implicit values:

```
  trait LowPriorityWritesInstances {

    // use standard writes for the case classes
    implicit val statusWrites = Json.writes[Status]
    implicit val noteWrites = Json.writes[Note]
    implicit val personWrites = Json.writes[Person]
    implicit val taskWrites = Json.writes[Task]
    implicit val linkWrites = Json.writes[Links]

    // and a custom one for the trait
    implicit object hiPriorityWrites extends OWrites[HasLinks] {
      def writes(hi: HasLinks) = Json.obj("_links" -> hi.links)
    }
  }

  /**
   * The write instance which we include
   */
  object WritesInstances extends LowPriorityWritesInstances {
  implicit val taskWithLinksWrites = new Writes[Task with HasLinks] {
      def writes(o: Task with HasLinks) = {
        (implicitly[OWrites[HasLinks]].writes(o)) ++
                    taskWrites.writes(o).as[JsObject]
      }
    }
  }
```

What happens here is that we define the standard implicits for the various parts of our model, including our new `HasLinks` trait. Besides that, we also define a very specific implicit that matches `Tasks` that extend the `HasLinks` trait. So, when we call the `toJson` function on a `Task` that extends `HasLinks`, the `taskWithLinksWrites` will match. In the `writes` function, we first convert the `Links` object and combine this with the converted `Task`.

To use this, all we have to do is define a new `Task` and use with `HasLinks` to add any links we have:

```
val task = new Task(
    1, "This is the title", "This is the content",
    Some(Person("Me")), List[Note](Note(1,"Note 1"),
       Note(2, "Note 2")), Status("new")) with HasLinks {
         val links =
             List(Links("self",
                        "/tasks/123",
                        "application/vnd.restwithscala.task+json"))
  }

  // import the implicit convertors
  import WritesInstances._

  println(Json.prettyPrint(Json.toJson(task)))
}
```

Now, after importing the correct implicits, we can convert the `Task` with `HasLinks` to JSON, just like

we do other objects.

Summary

In this final chapter, we looked at a couple of important aspects of REST. We explored the various JSON libraries that can be used in your REST services for converting objects to and from JSON. Besides that, we looked at a very important aspect of REST called HATEOAS. The most important aspect of HATEOAS is the ability to detect and filter on media-type (content-type) and to add links to your resources to create self-describing APIs. We saw how to detect and work with media-types in the frameworks discussed in this book, and how to add links with one of the JSON frameworks explored in this chapter.

With this last section on adding links to a JSON response, we have reached the end of this book. In the various chapters, we explored the most important features of a number of REST frameworks that are available in the Scala ecosystem.

In the previous chapters, we tried to show you the most important features of these frameworks and explain how to use these features to create scalable, asynchronous, and maintainable REST services. Please keep in mind that each of these frameworks has many more features than we could explore in this book.

I hope you have had fun reading the book and experimenting with the examples. If you like them, feel free to use them, expand them, and share the results!

Index

A

C

D

E

F

G

H

I

J

M

O

P

R

S

U

V

Z