Learning
Ratpack

SIMPLE, LEAN, AND POWERFUL WEB APPLICATIONS

Dan Woods

Learning Ratpack

Build robust, highly scalable reactive web applications with Ratpack, the lightweight JVM framework. With this practical guide, you'll discover how asynchronous applications differ from more traditional thread-per-request systems—and how you can reap the benefits of complex non-blocking through an API that makes the effort easy to understand and adopt.

Author Dan Woods—a member of the Ratpack core team—provides a progressively in-depth tour of Ratpack and its capabilities, from basic concepts to tools and strategies to help you construct fast, test-driven applications in a semantic and expressive way. Ideal for Java web developers familiar with Grails or Spring, this book is applicable to all versions of Ratpack 1.x.

- Configure your applications and servers to accommodate the cloud
- Use Ratpack testing structures on both new and legacy applications
- Add advanced capabilities, such as component binding, with modules
- Explore Ratpack's static content generation and serving mechanisms
- Provide a guaranteed execution order to asynchronous processing
- Model data and the data access layer to build high-performance, data-driven applications
- Work with reactive and functional programming strategies
- Use distribution techniques that support continuous delivery and other deployment tactics

Dan Woods is an open source enthusiast and a member of the Ratpack core team. His professional life focuses on building and architecting scalable distributed systems for cloud runtimes.

" I wish *Learning Ratpack* had been available when I wrote my first Ratpack production application 18 months ago. The clear examples and explanations would have saved me a lot of time. This book will be sitting on my desk for a long time. *Learning Ratpack* is required reading for anyone on a team that I lead."

—Kyle Boon
Technical Lead/Senior Software Engineer at SmartThings

" This book includes clear and practical examples that guide you through the framework. It's not only a great read cover to cover but also serves as a rock solid reference guide. It is definitely a must-read for anyone interested in building high-performance web applications in Ratpack."

—Craig Burke
Assistant Director of Systems & Software Development at Carnegie Mellon University

Twitter: @oreillymedia
facebook.com/oreilly

Learning Ratpack
Simple, Lean, and Powerful Web Applications

Dan Woods

Learning Ratpack

by Dan Woods

Printed in the United States of America.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

This book is dedicated to my wonderful wife, Ashley. Without her continued support and patience, Learning Ratpack would never have been possible. Thank you, Ashley, for the many wonderful years we have shared together and the many, many more to come.

This book is also for my children, Natasha and Ethan. Your love and smiles continue to drive me every single day. I could not have done this without you two; you are my world.

Table of Contents

www.allitebooks.com

Foreword

If you want to know Ratpack, you have come to the right place. This book is the best and most comprehensive learning resource out there, and Dan is an integral part of the team and community behind Ratpack. I am thrilled that this book is now available.

There are many tools for writing web applications on the JVM, of which Ratpack is one. I set out to make Ratpack excel at scaling in all relevant ways. While performance and efficiency are crucial, the ability to start simple for greenfield tasks yet scale toward solving more challenging problems is equally important. You should feel as empowered by your tools after two years as you did after two days. This is Ratpack's mission, which Dan eloquently conveys.

We are now building, testing, and deploying web infrastructure differently than we were just a few years ago. Developers and users are, rightfully, more demanding than ever. Applications must integrate with and exist within an ever-shifting landscape of tools, practices, and platforms. Ratpack takes this to heart in several ways: it is a pure runtime in that it does not impose a proprietary mechanism for build automation, nor does it require proprietary plugins for IDE integration; it treats deep testability as a first-class concern; and it favors integration over abstraction, allowing more direct use of complementary technologies such as persistence, marshalling, and templating. This is a conscious trade-off of out-of-the-box magic for long term flexibility and control. Developer freedom is a key tenet of Ratpack. Admittedly, this may not be the most pressing concern for new (and simple, which all new projects inherently are) projects, but it's important when things get real later in an application's life. And of course, there is performance.

The term *performance* can mean many things when it comes to web applications, and the most relevant meanings for the term are also shifting over time. We are now asking backend web applications to deal with more HTTP connections than ever, and to do it faster than ever. Moreover, as we deploy more and more of our applications to platforms where you pay for what you use, making efficient and predictable use of

computing resources is becoming a primary concern for more teams. Ratpack uses event-driven I/O and asynchronous programming for this reason. More efficient and predictable use of computing resources means lower costs and reduced risk of emergency rearchitecting due to scaling limits.

Asynchronous programming brings its own set of challenges—a key feature of Ratpack is the way in which it makes asynchronous programming more palatable. For many new to the paradigm, this is an area where guidance when getting started is particularly useful. This book does a great job in demystifying the *what* and the *why* of this, which is reason alone to read it.

As the creator of Ratpack, I want other developers to enjoy the same level of satisfaction I do when using it. More specifically, I want other developers to feel empowered but not constrained by it. This requires an understanding of what Ratpack can do for you, why and how it works, and what it leaves up to you.

— Luke Daley

Preface

Ratpack has had a long and wonderful history as an open source web framework for the Java Virtual Machine (JVM). As a high-performance, reactive web framework, it brings forth the fundamental concept that building succinct, lightweight web applications is only truly serviced when matched with performance, efficiency, productivity, and testability as first-class features.

As systems infrastructure moves increasingly to the cloud, the need has never been more apparent for applications to be built in a way that utilizes system resources as efficiently as possible. Ratpack's emphasis on performance and efficiency means that robust web applications can be built to deliver high performance and low memory utilization. In this way, Ratpack aims to maximize your investment in compute resources, while facilitating a developer experience that is focused on productivity and risk reduction.

The need for high performance and efficient web applications means building asynchronous programming fixtures atop a nonblocking networking layer. The paradigms in asynchronous programming can be difficult for unacquainted developers to grasp, making web frameworks that build on these concepts unapproachable.

Indeed, even for the most seasoned of developers, asynchronous programming introduces a level of complexity that means the benefits must be carefully weighed against the increased cognitive overhead. With Ratpack, asynchronous programming is presented in a way that is meant to be digestible, with a drastically reduced cognitive load from developers.

Reducing complexity while increasing approachability also means allowing applications to be built in a semantic and expressive way. Web application concepts are easily expressed in Ratpack through a concise programming interface that works with web nomenclature that developers will be familiar with. To that extent, the simplest Ratpack application can be written in just a few lines of code, as we will explore at the beginning of Chapter 1.

Throughout *Learning Ratpack*, you will be taken on a progressively in-depth tour of the framework and its capabilities. With comprehensive demonstrations, you will leave every chapter with a working knowledge of the subject matter, and a foundation for a deeper understanding in subsequent chapters. You will be exposed to many aspects of the framework throughout, and by the end of the book, you will have the confidence to build production-grade web applications with Ratpack.

Who Is the Target Audience for This Book?

Learning Ratpack makes only a few assumptions about the background and experience of the reader. Generally speaking, the target audience for this book is Java web developers, who have experience developing servlet-based web applications and beyond-superficial exposure to the Groovy programming language. Specifically, developers familiar with Grails and/or Spring Framework will find this book and its concepts beneficial.

Although Groovy is not strictly required for building applications with Ratpack, its ability to express concepts in a semantically concise way makes it an excellent choice for succinctly demonstrating the capabilities of the framework. As you will find, the vast majority of example listings in this book will utilize Ratpack's integration with Groovy to functionally depict a concept. Where appropriate, a demonstration will distinctly or correspondingly be shown with the equivalent Java code. Beyond the specific usage of the Ratpack Groovy domain-specific language, Java developers should be able to easily follow most of the Groovy examples.

What Is This Book Not Trying to Accomplish?

There is so much discussion to be had on the underlying reasons as to why Ratpack is such a compelling web framework for the JVM, it would be difficult for any one text to cover that ground. To that extent, this book is not aiming to be an exhaustive conversation on the rationale of Ratpack. Though deep discussion takes place in many parts of the book (particularly Chapter 7), *Learning Ratpack* is intended as a guide for understanding how to use the framework and become productive with it.

Ratpack is a living web framework, meaning that new improvements and additions are added every single day. This book will expose you to the foundational aspects of the framework, and many of its higher-level features that will give you the understanding necessary to follow the framework as it grows. While this book will open the door to your journey of learning Ratpack, it is important to keep tabs on the project's continued activity to know what features are being added.

No Breaking Changes!

In light of the fact that Ratpack will continue to evolve and add new features and capabilities, it should be noted that the framework has a core tenet that no binary-incompatible breaking changes will be introduced in a minor or patch version release. This book is baselined on Ratpack 1.3.3, and the principle of not introducing breaking changes will mean that the code and demonstrations contained herein will be applicable to all versions of Ratpack in the 1.x.x line.

While new methods will be added to classes, new features added to the infrastructure, and new extension modules will be made available for your projects, your project will continue to be compatible with subsequent releases in the same major version. Following this core tenet in Ratpack means that developers can be assured that updating a framework dependency will not result in breaking changes to their code.

Staying in Touch

Ratpack has been privileged over the years to have an engaging and vibrant community. The project's website houses the manual and user guide for all versions, past and present, and links to the issue tracker and discussion forum. The code is freely available on GitHub. Furthermore, the community engages in more synchronous conversations via its community Slack channel. You are encouraged to join and ask questions!

Announcements, news, blog posts, and conference presentations are posted through the project's official Twitter account, @ratpackweb. Community discussions are tracked through the Twitter hashtag #ratpackweb.

Ratpack is also privileged to have a group of highly active core maintainers (Twitter handles provided):

- Luke Daley (project lead; @ldaley)
- John Engelman (@johnrengelman)
- Danny Hyun (@Lspacewalker)
- Rus Hart (@rus_hart)
- David Carr (@varzof)
- Robert Zakrzewski (@zedar185)
- Jeff Beck (@beckje01)
- Marcin Erdmann (@marcinerdmann)
- Dan Woods (myself; @danveloper)

There are many conferences in the Java and Groovy ecosystem where you will find presentations on Ratpack. Some of the more popular ones include:

- Gr8Conf
- No Fluff Just Stuff
- Greach
- DevNexus
- Devoxx

Whatever medium you choose to communicate, please find a way to keep in touch and give your feedback on using Ratpack. It is only through hearing our users that the framework will continue to improve.

Acknowledgments

Special thanks to the wonderful editors and editorial staff at O'Reilly, especially to Brian Foster and Nan Barber, who have held my hand and guided me along the way. I am deeply grateful for the expertise they provided in making this book a reality.

Many deep thanks to the technical reviewers of this book, Keith Conrad and Danny Hyun. I have valued their input considerably over the course of this project, and I have learned so much from them along the way. Also, thank you to Graeme Rocher, Grails project lead, for reviewing parts of this book.

Thank you to every member of the Ratpack community who has made the framework such a success. Thank you to all the wonderful individuals in the Groovy ecosystem, who have graciously welcomed newcomers and continued to raise the bar on what we can achieve as a community. I am proud to be a part of this group.

Last, but certainly not least, a warm and heartfelt thank you to Luke Daley. I have learned so much from Luke over the years, and his leadership with Ratpack has made so many people better developers. Without Luke's contributions to open source, the world would be far lesser a place. Thank you, Luke, for all you have done, and for your friendship.

Welcome to Ratpack

At its core, Ratpack is a high-performance framework that marries efficient processing with ease of use. In the modern state of web application development, developer productivity has a necessary emphasis in many web frameworks. Where Ratpack differentiates itself in this space is by continuing to focus on developer productivity, while also providing a foundation upon which performance and resource utilization are forefront considerations. Furthermore, the framework's concise and easy-to-use application programming interface (API) allows applications to be designed in a way where they can be semantically reasoned about without disjointed concepts or complexities that plague other Java-based web frameworks. Even for a novice developer, Ratpack applications can be quickly built and understood without a high barrier of spool up time. In every respect, Ratpack aims to make it easy to build web applications on the Java Virtual Machine (JVM) that foster an environment of productivity, performance, and efficiency.

Ratpack takes a lightly opinionated approach to the manner in which applications are built—in other words, as much that can be facilitated in a "one-liner" as possible is presented as such, and when the complexity of a project's implementation outgrows those facilities, the framework gets out of the way quickly. Ratpack's infrastructure is designed to be extended, and follows sensible patterns for doing so. As your web application's requirements grow, Ratpack will continue to be a powerful utility from which you can harness complex concepts—like reactive programming and deterministic asynchronous processing—in the way that most suitably fits your needs.

In the effort of supporting semantic APIs that are easy to use and understand, Ratpack employs many of the luxuries of Java that have more recently become available. Lambda expressions, functional programming interfaces, and method references are a few of the newer language features that the framework has come to adopt. Leverag-

ing these aspects of the language allows the framework to guide applications toward the simplest path possible for solving the problem at hand.

Ratpack also provides out-of-the-box support for newer versions of the Apache Groovy programming language. As a long-time language on the JVM, Groovy brings a lot to the table when designing APIs that are semantically concise, including its robust and inherent capability to provide flexible, domain-specific languages (DSLs) as the layer upon which applications are built. In recent years, Groovy has established itself as much more than a simple dynamic alternative to core Java, and has incorporated features into the language that make it a premier choice when building any modern application on the JVM. Static compilation, for example, gives applications built with Groovy similar performance levels to those built with Java. Furthermore, the ability to inform Groovy's compilation engine as to the structure of a DSL also serves as a benefit to using Groovy. The ability to coerce a closure to a single abstract method type is also utilized by Ratpack to work with closures as functional programming interfaces.

In supporting Groovy as a first-class language, Ratpack places the very best of the JVM's ecosystem at your disposal. However, as you will find throughout this book, Ratpack's integration with Groovy ends at reducing the visual verbosity of applications. As a capable dynamic language, Groovy opens the door for building systems that employ elaborate compile-time processing, code generation, and runtime method dispatching that can quickly become difficult to follow and debug. In Ratpack, you will find no "magic" that happens behind the scenes, and indeed the framework takes the approach of limiting complexity as much as possible, so that you always know what your application is doing.

Hello, World!

To better understand the nature of Ratpack, we can begin by taking a look at the simplest possible application type: a "Hello, World!" example. Here, we will start by showing how the Groovy integration can be leveraged to rapidly prototype applications and get a sense of their form and function. We can make use of Groovy's ability to act as a scripting language to bring in the necessary framework dependencies, define a simple application structure, and run it from the command line. If we start by looking at the example outlined in Example 1-1, we see the entirety of what is required to get a Groovy-based Ratpack application up and running.

Example 1-1. "Hello, World!" application for a Groovy-based Ratpack implementation

```
@Grab('io.ratpack:ratpack-groovy:1.3.3') ❶

import static ratpack.groovy.Groovy.ratpack ❷
```

```
ratpack {
  handlers {
    get { ❸
      render "Hello, World!" ❹
    }
  }
}
```

❶   We start our simple script by using Groovy's built-in dependency management system, known as "Grapes," to "grab" the necessary framework dependency and make it available to our runtime classpath.

❷   Statically importing the `Groovy.ratpack` method provides our script with the DSL within which we define our application's structure and features.

❸   The handler chain will be covered in more depth in the next section; for now, all you need to know is that the `get` handler is binding a processing block for incoming HTTP GET requests (like those from a browser).

❹   Within the application handler, we render a "Hello, World!" message back to the client.

This example represents all the code necessary to build your first Ratpack application! The designed simplicity makes it easy to follow and with just these few lines of code you are well on your way to becoming a proficient developer with Ratpack.

Running the Example

Groovy can run nearly everywhere that Java can, and if you do not already have the Groovy runtime at your disposal, you can download it from the project's website. For Mac OS X, Linux, or Cygwin users, you can alternatively choose to use SDKMAN! to install Groovy. Windows users can use the PowerShell-based package manager Posh-GVM. However you obtain Groovy, ensure that the language binaries are available in your runtime PATH.

Once you have Groovy in place on your system, you can run the "Hello, World!" application from the command line. If you place the contents of the example into an *app.groovy* file, and use the `groovy` command-line utility, you will see your application start in your console. The output presented in Example 1-2 shows what you will see.

Example 1-2. "Hello, World!" application output

```
$ groovy app.groovy
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
```

```
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
WARNING: No slf4j logging binding found for Ratpack, there will be no logging output.
WARNING: Please add an slf4j binding, such as slf4j-log4j2, to the classpath.
WARNING: More info may be found here: http://ratpack.io/manual/current/logging.html
Ratpack started (development) for http://localhost:5050
```

The most notable part of the output is the last line, which Ratpack outputs to indicate where the running application can be accessed. The default HTTP bind port in Ratpack is *5050*, so if you open a browser and navigate to *http://localhost:5050*, you will see the "Hello, World!" result.

Rapid Prototyping

In the interest of accommodating developer productivity, Ratpack applications can be quickly and easily modified, with changes reflected in real time. While you have the "Hello, World!" example application running, from another window you can open the *app.groovy* file and make changes to it. For example, if you change the text from "Hello, World!" to "Hello, Ratpack!" and refresh your browser, you will see the new message reflected. This core Ratpack concept allows developers to quickly test and vet changes or features without the need for a full application restart. This is undisputedly a powerful facility when getting the initial structure and features of an application layed out.

In the chapters that follow, we will tour the features of Ratpack that make it amenable to projects of all shapes and sizes. One such feature is its comprehensive integration with the Gradle build system, which makes it easy to get a full-featured project running and built. As will be demonstrated and discussed later, Ratpack's integration with Gradle allows it to leverage the build system's reloading capabilities that further the efforts of rapid productivity. All this is to say, the rapid prototyping capabilities offered by Ratpack extend beyond its simplest form, and are inherent at all levels of application complexity.

Arguably one of the most notable features of Ratpack—and one that you have undoubtedly already realized from running the "Hello, World!" application—is how quickly applications get up and running. Most applications sport sub-second startup times, which further benefits the process of rapid prototyping. When you make a significant number of complex changes or changes that otherwise require a full application restart, you are not left with a long process of waiting for an application container to get your code changes live, because startup times are short and quick. With developer productivity in mind, Ratpack is designed in a way that gets code started and running as fast as absolutely possible.

Handler Chain

The most important structure in the definition of a Ratpack application is undoubtedly the handler chain. If we dissect the code in Example 1-1, we see the `handlers` block, which is the area in which application request handlers are defined. The handler chain defines the edge of your application and the flow by which requests are processed. To draw a parallel to more traditional Java web development patterns, the concept of handlers in Ratpack are analogous to *filters* and *servlets* in servlet API terms. That is to say, one or more handlers can participate in the processing of a request and a handler is responsible for sending a response back.

Handlers are processed top-down, and can be defined to match on HTTP verb, requested URL, and other concepts that are provided by the chain API. As you explore the handler chain, you will find that, like the `get` method that was demonstrated in our example application, there are semantic methods for `post`, `put`, `delete`, `patch`, and `options` as well. Using these conveniently named binding methods, you can appropriate the request handling logic to the corresponding HTTP verb, making your application's request handling easy to follow and quick to understand.

In our example application, within the handler chain we can see that we are binding a request handler for the HTTP GET verb on the default route. Ratpack will route HTTP GET requests for the / URI to this handler, which is evidenced by the browser test that we conducted earlier. Through the handler chain API, we can see that similar binding methods exist for all HTTP verbs. As you build out the specification for your application, you need only define the request handlers inline in the handler chain.

URL Path Bindings

Each of the methods on the handler chain API for working with the different HTTP verbs allow you to specify the path to which the handler should be bound. If we consider again the "Hello, World!" application from earlier, we can extend it slightly to include a second handler, which binds to the `/foo` endpoint. The code in Example 1-3 shows the addition of the second handler. As you explore this sample code, note that in Ratpack, you should not specify the leading / when defining URI paths.

Example 1-3. "Hello, World!" application with /foo handler

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get {
      render "Hello, World!"
```

```
    }
    get("foo") { ❶
      render "Hello, Foo!"
    }
  }
}
```

❶ Here, we bind a request handler to `/foo`. Take note that we are able to leverage the fact that Groovy does not require us to wrap the final argument in the parens of a method call when that argument is a closure.

If we run this application and open a browser to *http://localhost:5050/foo*, we will find that we are greeted with the "Hello, Foo!" message, as we would expect. Applying handlers to URI routes in this way works in the same way as the corresponding methods of the various other HTTP verbs.

A special and important caveat to the handler chain is that *only a single handler can be bound to a given URI*. While it is said that requests *flow through* the handler chain, once a handler is found for a given URI path binding, no other handlers bound to that same path will be eligible for processing. This rule applies regardless of the HTTP verb to which multiple handlers are bound. In the case where you wish to have multiple handlers apply to one route (say, in a resource-oriented RESTful API), Ratpack provides a special mechanism for doing so.

There are two similar semantics for representing a handler type that are agnostic to a request's HTTP verb: `all` and `path`. These two types do the same thing, with the difference being that the latter takes a URI path as an argument and the former does not. In the handler chain, these handler types qualify for processing of any incoming request, and by making use of this, we can bind a handler to a given URI and define its capability of processing any HTTP verb type. For the purposes of understanding multiverb bindings, we will expand our prior example to make the `/foo` endpoint handle post GET and POST requests. To do so, we will utilize the `path` chain method and within its handler we will introduce the `byMethod` mechanism. Example 1-4 shows our expanded sample code.

Example 1-4. The byMethod specification

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get {
      render "Hello, World!"
    }
    path("foo") { ❶
```

```
      byMethod { ❷
        get { ❸
          render "Hello, Foo Get!"
        }
        post { ❹
          render "Hello, Foo Post!"
        }
      }
    }
  }
}
```

❶ Here, we have changed the binding from `get` to `path`.

❷ Within the handler logic, we use the `byMethod`, which is provided as part of the handler's context (this will be covered in more depth later in the book).

❸ The `byMethod` specification allows us to bind our verb-specific handlers. Here, we specify the handler for an HTTP GET request.

❹ Similarly, we define a handler for POST requests.

If we run this application and open our browser again to *http://localhost:5050/foo*, we are greeted with the "Hello, Foo Get!" message. Now, if we open a command line and use the cURL utility, we can perform a POST request to our application, as follows:

```
curl -XPOST http://localhost:5050/foo
```

The output from that call will result in seeing the "Hello, Foo Post!" message.

Prefixed Routes

The handler chain is all about building a contextual definition of your application's request-taking flow. As the edges of your application grow, so too does the complexity of understanding the various definitions. To that end, when you are building your application's handler chain, you can choose to prefix a set of handlers under a given route. This becomes incredibly valuable in conversations about designing RESTful APIs, but for now, just remember that building a prefixed chain is a capability that allows you to logically organize your application better.

Prefixed routes are fairly self-explanatory in practice, but to further your understanding, consider a scenario where you are building an ecommerce application, and you wish to have a set of endpoints dedicated to working with products. For the sake of discussion, let's suppose that we have a `list` endpoint, which lists all products; a `get` endpoint, for getting a specific product; and a `search` endpoint for looking up products. It would be repetitive to have to write each of these out as `product/list`, `product/get`, and `product/search`, and those definitions could get lost easily as the

complexity of your application grows. Instead, we will use the chain API's `prefix` method to wrap them all up in a *subchain* dedicated strictly for `products`. The code in Example 1-5 demonstrates using the `prefix` method.

Example 1-5. Using the prefix method

```groovy
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    prefix("products") {
      get("list") {
        render "Product List"
      }
      get("get") {
        render "Product Get"
      }
      get("search") {
        render "Product Search"
      }
    }
  }
}
```

To the `prefix` method, we specify the prefixed URI pattern; the closure supplied to the `prefix` method created a subchain, which acts with the exact same behavior as the handler chain otherwise, with the handlers being bound within the `/products` route.

Prefixed routes can be described at any level within a chain, meaning that they can be further nested within other `prefix` subchains. This can allow you to build complex request processing flows that are easy to get a handle on without the verbosity of defining route depth at each handler definition.

Path Tokens

When binding to a path, Ratpack provides a mechanism by which tokens can be defined and later extracted by the handler. This allows variable data, such as an ID, to be accessed from the request path and handled accordingly.

Consider the code in Example 1-6, which demonstrates a path handler that uses the token notation.

Example 1-6. Path tokens

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get("foo/:id?") { ❶
      def name = pathTokens.id ?: "World" ❷
      response.send "Hello $name!"
    }
  }
}
```

❶ Path tokens are prefixed with a colon and are named. The `?` at the end of the token indicates that this is an optional token. Without it, the `id` property will always be required.

❷ Within the handler logic, we use the `pathTokens.id` call to get access to the `id` path token.

The `pathTokens` type is an implementation of a `TypeCoercingMap`, which provides you with some assistance in translating path tokens to respective types. By default, the value will come out a string, but using the coercion methods available will simplify your code. For example, if we wanted to work with the `id` field from the example as a `Long` type instead of as a string, we could change the code to: `pathTokens.asLong('id')`. Similar coercion methods are available for `Boolean` (`asBool`), `Byte` (`asByte`), `Short` (`asShort`), and `Integer` (`asInt`).

Request Parameters

Request parameters are made available to handlers through the `request` object. Unlike path tokens, request parameters are not defined in the path binding. Example 1-7 demonstrates a handler that extracts a request parameter and handles the request accordingly.

Example 1-7. Using request parameters

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get {
      def name = request.queryParams.name ?: "Guest" ❶
      response.send "Hello, $name!"
```

```
        }
      }
    }
```

❶ Off the `request` object, we can access the `queryParams` map, which holds keys for the specifically named query parameters.

Running this application and navigating to *http://127.0.0.1:5050?name=John* will render the message "Hello, John!" in your browser, as we would expect from the handler code. Also as we would expect, leaving off the `?name=John` request parameter will yield the default "Hello, Guest!" message.

Request parameters will always be `String` types, so it is important to properly translate them to their corresponding type for use in external classes or services.

Parsing Request Data

Structured request data can be extracted from the request using the `parse` method in our handler. Through this interface, request data can be converted into a data structure for additional processing. Example 1-8 demonstrates this capability.

Example 1-8. Parsing request data

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack
import ratpack.form.Form

ratpack {
  handlers {
    all {
      byMethod {
        get { ❶
          response.send "text/html", """\
            <!DOCTYPE html>
            <html>
            <body>
            <form method="POST">
            <div>
            <label for="checked">Check</label>
            <input type="checkbox" id="checked" name="checked">
            </div>
            <div>
            <label for="name">Name</label>
            <input type="text" id="name" name="name">
            </div>
            <div>
            <input type="submit">
            </div>
```

```
        </form>
        </body>
        </html>
      """.stripIndent()
    }
    post {
      parse(Form).then { formData -> ❷
        def msg = formData.checked ? "Thanks for the check!" :
          "Why didn't you check??"
        response.send "text/html", """\
        <!DOCTYPE html>
        <html>
        <body>
        <h1>Welcome, ${formData.name ?: 'Guest'}!</h1>
        <span>${msg}</span>
        """.stripIndent()
      }
    }
  }
  }
  }
}
```

❶ This handler, while verbose, is fairly simple. All we are doing here is serving up
   an HTML form to work with in a simple view. This is not best practice, and nor-
   mally we would want this served from the project's assets. Serving content is cov-
   ered later in the book, so this is here only for demonstration's sake.

❷ Within the chain's post handler, we call the parse method and inform it that we
   want a Form object returned. The resulting formData gives us access to the data
   that was submitted from the get handler's HTML form. We can work with this
   like any other map.

This example is serving HTML directly out of the handlers, so pointing a web
browser to the application URL will this time show us a proper HTML form. Tog-
gling the form's checkbox and submitting will show the different behavior of the
POST handler, which converts the form data into a Form object before making a deci-
sion about what to render.

Content Negotiation in Handlers

We previously covered how to build method-agnostic handlers that have specialized
logic for processing different HTTP requests for the same URI binding. You will
recall that the mechanism by which this is accomplished is known as the byMethod
specification. In addition to specifying handler logic for different method types, Rat-
pack provides a specification with which handlers can apply specific logic based on
the request's content type.

Content type negotiation comes into play when designing request handlers that are responsible for sending data back to a client in a specified format. For example, your application may design a handler that renders an HTML page when the request specifies that it desires a `text/html` content type. That same handler may instead render a model as JSON data when a requested content type of `application/json` is specified. For managing this within your handler logic, Ratpack provides what is known as the `byContent` specification, and works similarly to the `byMethod` specification.

To illustrate this capability better, consider an application where we have a request handler bound to the `/users` endpoint. When a consumer of our application opens the endpoint in a browser, we want it to render back a list of `User` objects; when a client library accesses the endpoint and requests JSON or XML data, we want the data serialized as such. The `byContent` specification gives us flexibility to adapt the handler's response according to what the consumer is capable of receiving. The code in Example 1-9 demonstrates how this application would look.

Example 1-9. The byContent specification

```
@Grab('io.ratpack:ratpack-groovy:1.3.3')

import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class User { ❶
  String username
  String email
}

def user1 = new User( ❷
  username: "ratpack",
  email: "ratpack@ratpack.io"
)
def user2 = new User(
  username: "danveloper",
  email: "danielpwoods@gmail.com"
)

def users = [user1, user2] ❸

ratpack {
  handlers {
    get("users") {
      byContent { ❹
        html { ❺
          def usersHtml = users.collect { user ->
            """\
            |<div>
            |<b>Username:</b> ${user.username}
            |<b>Email:</b> ${user.email}
```

```groovy
            |</div>
        """.stripMargin()
    }.join()

    render """\
    |<!DOCTYPE html>
    |<html>
    |<head>
    |<title>User List</title>
    |</head>
    |<body>
    |<h1>Users</h1>
    |${usersHtml}
    |</body>
    |</html>
    """.stripMargin()
}
json { ❻
    render toJson(users)
}
xml { ❼
    def xmlStrings = users.collect { user ->
        """
        <user>
            <username>${user.username}</username>
            <email>${user.email}</email>
        </user>
        """.toString()
    }.join()
    render "<users>${xmlStrings}</users>"
}
    }
  }
 }
}
```

❶ We will start by envisioning a simple `User` model object with some simple properties.

❷ For demonstrative purposes, we will bootstrap a couple of test objects.

❸ To keep things simple, we can maintain a simple global list of our bootstrapped users. (Note that this is just for the purposes of demonstrating the example. Proper data-driven web applications will be covered in depth later in the book.)

❹ Within our handler, we can access the `byContent` method, which gives us some convenience methods similar to how the handler chain works. Within the closure we provide, we can specify handlers for the different content types our application is capable of providing.

❺   The `html` convenience method provides the ability to perform processing specifically when a `text/html` content type is requested.

❻   Similarly, `json` allows you to specify logic for an `application/json` content type. Note that here we are using Groovy's `JsonOutput` class to assist in serializing the user list to JSON. Ratpack has extensive support for JSON rendering that will be covered later in the book.

❼   Ratpack even provides convenience methods for working with XML content types. If your users request `application/xml`, this logic will be activated to respond to them.

If you run this script and navigate to *http://localhost:5050/users*, your browser will pull up a web page prominently displaying the list of users, as shown in Figure 1-1.
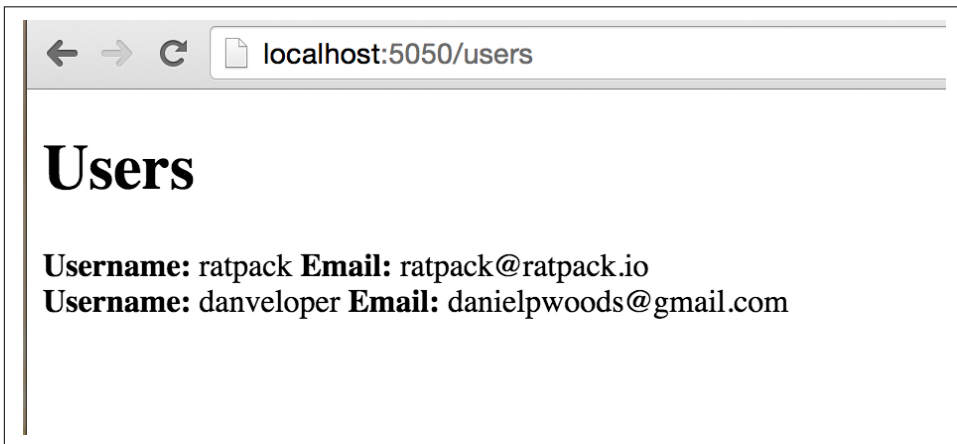

Figure 1-1. The byContent spec (HTML output)

Next, we can test JSON serialization. By again using the cURL command-line utility, we can issue a request that specifies that we want `application/json` content. Note that content type is specified through the request's `Accept` header. In cURL, we specify this header by supplying the `-H "Accept: application/json"` argument. The output in Example 1-10 shows running the request and the response from our application.

Example 1-10. The byContent spec (JSON output)

```
$ curl -H "Accept: application/json" localhost:5050/users
[{"username":"ratpack","email":"ratpack@ratpack.io"},{"username":"danveloper", ↵
"email":"danielpwoods@gmail.com"}]
```

Great! As you can see, the appropriate logic blocks provided to the `byContent` method are being activated accordingly. To be sure, we can run a similar test, shown in Example 1-11, this time checking that `application/xml` is working properly.

Example 1-11. The byContent spec (XML output)

```
$ curl -H "Accept: application/xml" localhost:5050/users
<users><user><username>ratpack</username><email>ratpack@ratpack.io</email></user> ↵
<user><username>danveloper</username><email>danielpwoods@gmail.com</email></user> ↵
</users>
```

Everything is working exactly as we would expect. It is important to know how `byContent` operates: when no content type is specified, the first handler is always activated. In our example, if we run the cURL command again, this time without specifying the header, we are returned HTML. As an exercise, if you move the `json` block before `html` and rerun the test, you indeed will see JSON output.

Using `byContent` also gives you the ability to specify custom content types. If your application requires rendering content in a specialized way according to a consumer's capabilities, you can build processing into your handler using the `type` method on `byContent`. If we change our demonstration slightly to also include a handler for the `application/vnd.app.custom+json` content type, we can reimagine a shortened version of our handler, such as that shown in Example 1-12.

Example 1-12. Custom content type

```
get("users") {
  byContent {
    html {
      // ... snipped for brevity ...
    }
    json {
      // ... snipped for brevity ...
    }
    xml {
      // ... snipped for brevity ...
    }
    type("application/vnd.app.custom+json") { ❶
      render toJson([
        some_custom_data: "my custom data",
        type: "custom-users",
        users: users
      ])
    }
  }
}
```

❶ Here, we apply a logic block to an application-specific content type. Within it, we render back a customized data structure with properties that consumers will expect.

If we run the cURL test again, this time specifying `application/vnd.app.custom +json`, we will see our custom data structure rendered, as shown in Example 1-13.

Example 1-13. The byContent spec (custom type output)

```
$ curl -H "Accept: application/vnd.app.custom+json" localhost:5050/users
{"some_custom_data":"my custom data","type":"custom-users", "users": ↵
[{"username":"ratpack","email":"ratpack@ratpack.io"},{"username":"danveloper", ↵
"email":"danielpwoods@gmail.com"}]}
```

This is exactly the output you would expect. This capability makes Ratpack a powerful choice for building applications where the content type serves as a way to drive the form of the models you render back to your consumers. But, what about when a consumer requests a content type that does not match anything we have built into the `byContent` block? For that, we need to leverage another feature of the specification, which is specifically designed for this. The `noMatch` method allows us to attach processing in this case. The shortened example shown in Example 1-14 demonstrates applying the `noMatch` block to your `byMethod` logic.

Example 1-14. The byContent spec (no match handling)

```
get("users") {
  byContent {
    html {
      // ... snipped for brevity ...
    }
    json {
      // ... snipped for brevity ...
    }
    xml {
      // ... snipped for brevity ...
    }
    type("application/vnd.app.custom+json") {
      // ... snipped for brevity ...
    }
    noMatch { ❶
      response.status 400
      render "negotiation not possible."
    }
  }
}
```

❶ Within the `noMatch` block, we will set the response status to `400` (Bad Request) and send back a plain-text message. Consumers will recognize the HTTP error code and realize there is a problem with the content type.

If we run the cURL test again, this time with an arbitrary content type string, we will see a "negotiation not possible" message returned. The output in Example 1-15 shows the output for this test.

Example 1-15. The byContent spec (no match output)

```
$ curl -H "Accept: application/nothing" localhost:5050/users
negotiation not possible.
```

You will recall that when no content type was specified, we defaulted to the first defined block (`html`); however, that may not be the desired behavior when an unknown content type is specified. We can further this example by saying that our application's requirements are that when no content type is matched, we want to render the JSON explicitly. To facilitate this, instead of specifying a closure to the `noMatch` method, we can provide a string with the content type to which the request processing should be routed. In Example 1-16, we specify that when no match is found, we instead want to treat the request as though it were `application/json`.

Example 1-16. The byContent spec (no match to JSON translation)

```
get("users") {
  byContent {
    html {
      // ... snipped for brevity ...
    }
    json {
      // ... snipped for brevity ...
    }
    xml {
      // ... snipped for brevity ...
    }
    type("application/vnd.app.custom+json") {
      // ... snipped for brevity ...
    }
    noMatch "application/json"
  }
}
```

Running the test again with the `application/nothing` content type, we will notice that this time we are met with the same output that we saw earlier in our JSON test. Giving your application the ability to direct processing according to a request's acceptable content type makes it easy to build powerful APIs and multiuse handlers in a concise way. Your understanding of content type negotiation in Ratpack will serve

as an excellent foundation for the examples throughout this book. When we get into discussions of building APIs and data-driven applications, content type negotiation becomes a paramount consideration.

Chapter Summary

In this introductory chapter, we have opened the door on your understanding of how to build web applications with Ratpack. From covering the basic "Hello, World!" demonstration at the beginning, to working through a basic understanding of the handler chain, and how to work with requests, you now have the necessary exposure to get off the ground with Ratpack. This chapter's content will serve as a building block for the text to come. The concepts demonstrated here will prove ever-useful as we expand the conversation, and introduce you to the more advanced and robust concepts and features that Ratpack provides.

Getting Started

Getting started in playing around with Ratpack is really easy, as was demonstrated in the previous chapter. Up until now, you have been exposed to Ratpack's ability to easily get a prototype off the ground with just a few lines of Groovy. As you come to better understand working with Ratpack, it is important to note that while we can do everything we need to in a Groovy script, a collaborative and tested project will require proper application structure.

When building a new project, Ratpack's nature as a lightly opinionated framework means that some guidance is provided. As we will discuss in this chapter, Gradle is the preferred build and dependency management system for Ratpack applications, and with it you will garner many simplicities and benefits that otherwise would not be available. In the spirit of not strictly imposing its opinions, the framework and its component feature sets are provided simply as a set of Java libraries, so no matter your favored build tool you will certainly find the means to integrate Ratpack.

As we dig into how the framework facilitates extensibility, you will find that your existing experience building and working in Java-based projects is immediately applicable, and not to be replaced with a new framework-specific understanding. Indeed, while Ratpack provides a foundation for building web applications, it—and its optional feature set—is made available to you as a set of libraries to be incorporated simply as project dependencies. There are no framework binaries or Ratpack command-line utilities you need to learn to get started—a simple understanding of building Java projects is all that is needed.

Ratpack does not require a web application container at runtime, so there are no implicit project-level dependencies that you need to be concerned with. This means that the same framework libraries that you build with on your development system are the ones that will be used in your production environment. In fact, packaged Rat-

pack applications are built into Java Archive (JAR) files that are able to be self-contained and runnable in a standalone fashion.

In Ratpack's spirit of efficiency, only the minimally required dependency set need be included in your project. For example, if you are building a Java-based Ratpack application that does not require Groovy support, then your project does not need to inherit the Groovy dependencies along the way. Likewise, if your project has no need for authentication, then you will have no need to bring in the Pac4j dependency either. This conscious design decision to the structure of Ratpack ensures that your projects are as lightweight as possible, and that you are limiting your exposure to incorporating conflicting dependency versions as much as possible. Furthermore, having an explicit understanding of the framework capabilities your project includes further aids in quickly getting a grasp of what a project is trying to accomplish.

In this chapter, as we explore building a real-world project with Ratpack, you will be exposed to the means by which the framework and its component parts are made available to you. We will delve into building a standalone project, including an explanation of a project's structure, and an in-depth exploration of Ratpack's integration with the Gradle build system. Your understanding of Ratpack will be carried further by covering the means by which applications can be built from a runnable main class, deeper coverage of the handler chain, and how to build standalone request handlers.

This chapter provides comprehensive coverage of Ratpack's framework structure. By the end of this chapter, you will be able to set up and build new projects, and will have an understanding of how to work in an application's project structure. The knowledge that you carry forward from this chapter will serve as the foundation for later discussions on the many parts to Ratpack that make it such a compelling framework for the JVM.

Library Structure

The framework features that Ratpack provides are composable by incorporating the various framework libraries as project-level dependencies. For example, Ratpack projects that are built on Groovy must include the `ratpack-groovy` library in order to use the Groovy DSL. Similar library coordinates are available for all of the framework's features.

Taking the toolkit approach to framework development means that Ratpack projects only need to include the aspects of the framework they intend to use. All Ratpack libraries are homed in the `io.ratpack` group of Maven coordinates, and each library follows a sensible naming pattern that corresponds to its function in the framework. Support for HTTP sessions comes from the `ratpack-session` library; likewise, sup-

port for the reactive programming library RxJava is accomodated by including the `ratpack-rx` library.

All libraries in Ratpack's framework ecosystem extend from the `ratpack-core` library, which provides a Java 8–based fabric for building web applications. The `core` library will be a transitively included dependency for any project utilizing higher-level features.

The Maven coordinates for some of the framework libraries are as follows:

- Core: `io.ratpack:ratpack-core:1.3.3`
- Groovy: `io.ratpack:ratpack-groovy:1.3.3`
- Guice: `io.ratpack:ratpack-guice:1.3.3`
- Hystrix: `io.ratpack:ratpack-hystrix:1.3.3`
- RxJava: `io.ratpack:ratpack-rx:1.3.3`
- Session: `io.ratpack:ratpack-session:1.3.3`

Ratpack's ability to separate its component features allows the framework to grow in a cohesive and independent way. As new features are added, they extend the framework's core functionality. In that respect, Ratpack is better able to address the evolving needs of web applications without fundamentally changing its underlying form and function.

Project Structure

Anything beyond the most trivial of Ratpack applications needs to be organized into a proper project structure. To accomplish this, Ratpack projects need to make use of a build system, and it is recommended to use one that supports dependency management. As noted at the beginning of this chapter, Ratpack favors the Gradle build system, and in doing so provides helpful utilities and shortcuts that make it easy to resolve framework dependencies, and build and package applications.

While Gradle is the preferred and supported build system, because Ratpack is a framework built as a set of libraries, any build and dependency management system for the JVM will work just fine. This means that developers wishing to utilize Ant, Maven, or SBT for their projects can easily do so by simply incorporating the appropriate framework dependencies for their project. Out-of-the-box support is not provided for these build systems, but given Ratpack's library-oriented layout, it should not be hard to get up and running in these environments.

When considering the structural design of a Ratpack project, you need not think of your web application as anything more than a typical Java project. Ratpack projects do not deviate from the standard structure adopted by many JVM-based projects

(there is one exception to this that we will cover momentarily). While other JVM web frameworks may force your project into a particular structure, Ratpack applications are designed in a way that enables accustomed JVM developers to quickly be effective without needing to learn a new structural convention.

A typical project structure for a Ratpack application looks something like the tree shown in Example 2-1.

Example 2-1. Typical project filesystem structure

```
.
└── src
    ├── main
    │   └── java
    │       └── tld
    │           └── company
    │               └── app
    │                   └── Main.java
    └── test
        └── java
            └── tld
                └── company
                    └── app
                        └── ApplicationTest.java
```

The single noted exception to this project structure is where—while not required—Groovy applications can be structured with a script as the application entry point. The script is typically placed into a *src/ratpack/Ratpack.groovy* file (although the ability to override this location is available). Example 2-2 shows the minimum project structure for a Groovy-based project.

Example 2-2. Minimum project structure for Groovy

```
.
└── src
    └── ratpack
        └── Ratpack.groovy
```

Ratpack Gradle Plugin

As noted before, Ratpack's preferred and supported build system is Gradle. It can be downloaded using the SDKMAN! utility, in the same way as Groovy. After you have downloaded SDKMAN!, simply issue the `sdk use gradle` command. After issuing the command, you will be asked whether you want to install Gradle. Go ahead and do that to get Gradle downloaded and installed on your system. Example 2-3 shows the output from SDKMAN!.

Example 2-3. Installing Gradle with SDKMAN!

```
$ sdk use gradle

Stop! gradle 2.11 is not installed.
Do you want to install it now? (Y/n): Y

Downloading: gradle 2.11

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
    0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
    0     0    0   355    0     0    195      0 --:--:-- 0:00:01 --:--:--   230
100 44.7M  100 44.7M    0     0   900k      0  0:00:50 0:00:50 --:--:-- 1178k

Installing: gradle 2.11
Done installing!

Setting gradle version 2.11 as default.

Using gradle version 2.11 in this shell.
```

With the Gradle binaries installed, we can begin to build out our project. Ratpack's integration with Gradle alleviates some of the time necessary to bootstrap a new project, by providing sensible defaults in the project's build configuration. For instance, consider the *build.gradle* build script shown in Example 2-4. This build script is all that is necessary to get a new Ratpack Groovy application off the ground.

> Gradle 2.0 or higher is required for use with the Ratpack Gradle plugin, but for best performance and feature set, Gradle 2.6 or higher is recommended.

Example 2-4. Minimal Gradle build script

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'  ❶
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'  ❷

repositories {
  jcenter()
}
```

❶ Here, we add the `ratpack-gradle` plugin to the build script's classpath. This gives configuration to the build that offers sensible defaults to standard projects.

❷ We apply the `ratpack-groovy` plugin, which allows Ratpack to configure our build with the necessary project-level dependencies, including adding the required `ratpack-groovy` framework library to the build, as well as including the necessary Groovy libraries.

If we create an empty directory for our new project and place the contents shown in Example 2-4 into the *build.gradle* file, we can start to build out our project structure. For a simple starting place, we can follow the Groovy project structure outlined in the prior section. The tree listing for this simple project looks like the one shown in Example 2-5.

Example 2-5. Groovy project structure

```
.
├── build.gradle
└── src
    └── ratpack
        └── Ratpack.groovy
```

If you're familiar with Maven project structures, you'll undoubtedly recognize that the demonstrated project structure does not conform to standards. Indeed, Ratpack's Gradle integration is specially designed to include the *src/ratpack* tree as a project resource set. This is a convenience that comes for free when using Gradle, and makes it explicit as to where your project's Ratpack files live, versus the rest of the project's code. As we will explore later, the *src/ratpack* convention is used in conjunction with standard *src/main/groovy* and *src/main/java* structures.

If we populate the *Ratpack.groovy* file with a simple application definition, similar to what we saw in Chapter 1, we can realize a full-fledged application in our newly created project structure. Filling it in with the code shown in Example 2-6, we will get started.

Example 2-6. Simple Groovy Ratpack application

```groovy
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

When working in a project structure, it's not necessary to include the `@Grab` annotation, as we did in the Groovy script at the beginning of the book. This is because Gradle is now managing our classpath dependencies, so we have no need to use Grapes at this point.

From the project root, we can now run our application in development mode. To do so, simply issue `gradle run` on the command line, from the project root (i.e., where the *build.gradle* file lives). With this, we will see a string of output text similar to that shown when we ran the Groovy script before, as shown in Example 2-7.

Example 2-7. Gradle run output

```
$ gradle run
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources
:classes
:configureRun
:run
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
WARNING: No slf4j logging binding found for Ratpack, there will be no logging output.
Ratpack started (development) for http://localhost:5050
WARNING: Please add an slf4j binding, such as slf4j-log4j2, to the classpath.
WARNING: More info may be found here: http://ratpack.io/manual/current/logging.html
> Building 83% > :run
```

The last line to the `run` task will stay until you terminate the process. This is using Gradle to act as our project runner, so that you can see your application running live without it having to be packaged into a distribution. Again, the most notable line here is the one starting with `Ratpack started`, which shows us how to access our application. If you navigate to *http://localhost:5050*, you will find your "Hello, World!" message displayed in your web browser.

While Groovy is undoubtedly the easiest way to get started with Ratpack, the framework itself is designed around Java. As such, there are facilities available within the Gradle plugin to best accommodate those applications that choose to build on Java alone. Let's consider a Ratpack Java application by starting with an empty project directory. As before, we begin building the project by creating a *build.gradle* file. This time, we will apply the `io.ratpack.ratpack-java` plugin, as shown in Example 2-8.

Example 2-8. Ratpack Java build script

```
buildscript {
  repositories {
```

```
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-java' ❶

mainClassName = 'tld.company.app.Main' ❷

repositories {
  jcenter()
}
```

❶ Here, we see the application of the `io.ratpack.ratpack-java` plugin, which
    ensures the `ratpack-core` dependency is brought in and that the build is prop-
    erly configured for a Java-based project.

❷ When building from a Java-based project that is not using the Groovy DSL file,
    we need to specify the entry point to our application. Ratpack integrates with the
    Gradle Application plugin for building and development-time running. What-
    ever class in the project that will act as the application runner must be specified
    as the `mainClassName` build script directive. Note that the class name is arbitrary,
    though it is probably a good design decision to choose something obvious, like
    `Main` or `Application`.

With this build script in place, the project structure now conforms to standard Maven
conventions. The minimum filesystem tree looks like the one shown in Example 2-9.

Example 2-9. Java project structure

```
.
├── build.gradle
└── src
    └── main
        └── java
            └── tld
                └── company
                    └── app
                        └── Main.java
```

We will more fully discuss building Ratpack applications from a main class later in
this chapter.

Beyond simply running from the command line, the plugin bootstraps and config-
ures the aspects of Gradle that make the project ready for packaging and distribution.
For example, the project can be built into a distribution by issuing `gradle distZip`

or `gradle distTar` for generating a *.zip* or *.tar* file, respectively. This command will create the deployment structure, including the project's dependencies, and will generate "start" scripts for Linux and Windows machines with all the appropriate classpath configurations for your project. This is accomplished by leveraging Gradle's Application plugin.

As noted earlier in the chapter, Ratpack's nature as a component library framework means that your project need only include the facets that it needs, therein eliminating dependency bloat from the framework. This, however, means that it can be difficult for project developers to properly align the different versions of the various framework pieces. To alleviate this pain point, the Ratpack Gradle plugin provides a shortcut for resolving different libraries of the framework, and locks them according to the plugin version.

The code in Example 2-10 incorporates RxJava support into the project by simply including the `compile ratpack.dependency('rx')` line within the build script's `dependencies` block. From here, the plugin will inform Gradle as to the appropriate coordinates for the RxJava library.

Example 2-10. Gradle build script with RxJava dependency

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency('rx')
}
```

As shown at the beginning of the chapter, Ratpack's library structure is such that each framework module is named following a convention that prefixes "ratpack-" to each artifact name. Using the plugin's `ratpack.dependency(..)` mechanism, any of the framework libraries can be resolved by name, which is simply the artifact name with the prepended "ratpack-" omitted.

After a framework dependency is added to the project, it needs to be wired into your application. Ratpack's modular structure is powerful, flexible, and efficient, but is also a departure from many other JVM frameworks, such as Grails, that use a plugin system. Adding and working with modules in Ratpack is covered in depth in Chapter 5.

From here, the project can be integrated with the rest of the Gradle ecosystem, which includes IDE support and a vast array of plugins for nearly every task.

Gradle Wrapper

Gradle has the ability to produce a *wrapper* script, which serves as a convenient runner for your application's build. This script can be included as part of the project's source control, thereby ensuring that anyone working on the project has the ability to build to the project. For those without Gradle installed, the wrapper will download Gradle and pass commands to it when they are issued. Use of the wrapper is a powerful utility for ensuring that developers can always build and run your project.

To get started with creating the wrapper, within the root of your project (as noted earlier, the project root is where the *build.gradle* file lives) simply issue the `gradle wrap per` command. Upon doing so, you will be met with output such as that shown in Example 2-11.

Example 2-11. Creating the Gradle wrapper

```
$ gradle wrapper
:wrapper

BUILD SUCCESSFUL

Total time: 5.371 secs
```

Now within your project directory, you will see a *gradle/* directory and two new files in the root: *gradlew* and *gradlew.bat*. The former is a BASH script for Linux and Mac OS X systems; the latter is a batch file for Windows systems. The *gradle/* directory contains the wrapper JAR file and a corresponding properties file. This directory can generally be left alone, though if you wish to include the wrapper in your project's source control, the directory must also be included. With the wrapper in place, your project structure now looks like the tree listing in Example 2-12.

Example 2-12. Project structure with Gradle wrapper

```
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
```

```
|        └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── src
    └── ratpack
        └── Ratpack.groovy
```

Going forward, instead of issuing commands using the `gradle` command-line utility, you instead can call your *gradlew* script. In doing so, you ensure that you are using the project's specific version of Gradle instead of your system's version of Gradle. Using the wrapper will ensure that developers are always building and running with the appropriate version of Gradle for your project.

Now, if we run the command `./gradlew run` for the first time, you will likely be met with an output indicating that the wrapper is downloading and installing the appropriate version of Gradle. SDKMAN! and Gradle do not share the same filesystem locations for installed versions, so even if you have the appropriate version installed on your system, you may find that it is downloading again. Once the download is complete, you will see the same output that you saw before, as shown in Example 2-13.

Example 2-13. Gradle run output from wrapper

```
$ ./gradlew run
Downloading https://services.gradle.org/distributions/gradle-2.11-bin.zip
...............................................................
...............................................................
...............................................................
...............................................................
...............................................................
...............................................................
... [many more dots] ...
Unzipping /Users/danw/.gradle/wrapper/dists/gradle-2.11-bin/452syho4l32rlk2s8ivd ↵
jogs8/gradle-2.11-bin.zip to /Users/danw/.gradle/wrapper/dists/gradle-2.11-bin/ ↵
452syho4l32rlk2s8ivdjogs8
Set executable permissions for: /Users/danw/.gradle/wrapper/dists/gradle-2.11-bin/ ↵
452syho4l32rlk2s8ivdjogs8/
gradle-2.11/bin/gradle
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:configureRun
:run
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
WARNING: No slf4j logging binding found for Ratpack, there will be no logging output.
Ratpack started (development) for http://localhost:5050
```

```
WARNING: Please add an slf4j binding, such as slf4j-log4j2, to the classpath.
WARNING: More info may be found here: http://ratpack.io/manual/current/logging.html
> Building 83% > :run
```

Your project now has a standardized way of building and running, and the wrapper is the means by which all developers collaborating on the project can ensure they are working with the necessary Gradle version.

Hot Reloading

As noted in the first chapter, rapid prototyping and hot reloading of Ratpack applications is accomplished by integrating with Gradle's "continuous build" feature. This capability of the build system allows applications to be started in a mode where Gradle supervises the project structure and is capable of recognizing changes to classes and files within your project. Changes are transparently made available to the running JVM.

Nothing more is needed to get hot reloading and continuous build capabilities than to change your `./gradlew run` command to `./gradlew -t run`. This will automatically inform Gradle that you intend to reflect changes in real time, and you are free to develop and make changes on the fly.

A caveat to the continuous build integration is that the task *must* be started through Gradle directly. This can be accomplished either through the Gradle command-line utility or through invoking the Gradle task within your IDE. If you are developing your application from within an IDE and wish to leverage IDE debugging features (like breakpoints), then you should start your application with the command `./gradle -t run --debug-jvm`. This will start the application and wait for a remote debugger to connect to it. From within your IDE of choice, you can create a remote debugging run configuration to attach to your running application. For most applications, the default configuration will be fine.

Lazybones

Lazybones is a project generator, which eliminates the need for bootstrapping a typical boilerplate project structure. It takes it a step further with projects like Ratpack by providing a bare-bones structure and application from which to get started. For developers new to Ratpack, use of Lazybones is highly encouraged.

Lazybones can be downloaded through SDKMAN!, by issuing the `sdk install lazybones` command. Once installed, bootstrapping a new Ratpack project can be accomplished by running `lazybones create <template name> <template version> <target directory>`, which will download the versioned template and generate the project into the specified target directory. For example, to set up a new Ratpack project, you can run `lazybones create ratpack my-app`, where *my-app* is the name

of the application you wish to create. This will bootstrap the project structure for a Groovy-based project with a simple index page and Gradle build configuration.

Named templates for Lazybones are stored on Bintray, and templates for Ratpack applications can be found in the *ratpack/lazybones* repository. However, different template locations can be explicitly issued to the `lazybones create` command, allowing teams and organizations to pre-define common project structure according to their requirements. For example, running `lazybones create` `https://dl.bintray.com/ratpack/lazybones/ratpack-template-1.3.3.zip` *my-app* will create the project structure for a basic Ratpack Groovy application. The helpful output from this command is shown in Example 2-14.

Example 2-14. Output from running lazybones create

```
Creating project from template https://dl.bintray.com/ratpack/lazybones/ ↵
ratpack-template-1.3.3.zip (latest) in 'my-app'

Ratpack project template
----------------------------

You have just created a basic Groovy Ratpack application. It doesn't do much
at this point, but we have set you up with a standard project structure, a
Guice back Registry, simple home page, and Spock for writing tests (because
you'd be mad not to use it).

In this project you get:

* A Gradle build file with pre-built Gradle wrapper
* A tiny home page at src/ratpack/templates/index.html (it's a template)
* A routing file at src/ratpack/Ratpack.groovy
* Reloading enabled in build.gradle
* A standard project structure:

    <proj>
      |
      +- src
         |
         +- ratpack
         |    |
         |    +- Ratpack.groovy
         |    +- ratpack.properties
         |    +- public // Static assets in here
         |         |
         |         +- images
         |         +- lib
         |         +- scripts
         |         +- styles
         |
         +- main
         |   |
```

```
      |    +- groovy
                 |
                 +- // App classes in here!
      |
      +- test
          |
          +- groovy
                 |
                 +- // Spock tests in here!
```

That's it! You can start the basic app with

```
    ./gradlew run
```

but it's up to you to add the bells, whistles, and meat of the application.

```
  Project created in my-app!
```

As you can see from the output, the project structure is created as we would expect, and even some basic functionality is in place! The output even shows using the Gradle wrapper to run the project. If we follow the output and issue the `./gradlew run` command, we see output indicating that the application is starting, as shown in Example 2-15.

Example 2-15. Gradle output from Lazybones application

```
$ ./gradlew run
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources
:classes
:configureRun
:run
[main] INFO ratpack.server.RatpackServer - Starting server...
[main] INFO ratpack.server.RatpackServer - Building registry...
[main] INFO ratpack.server.RatpackServer - Ratpack started (development) ↵
for http://localhost:5050
> Building 83% > :run
```

As we have seen before, Ratpack conveniently outputs the location where our application is running. Opening a browser and navigating to *http://localhost:5050* reveals the template landing page (as shown in Figure 2-1).
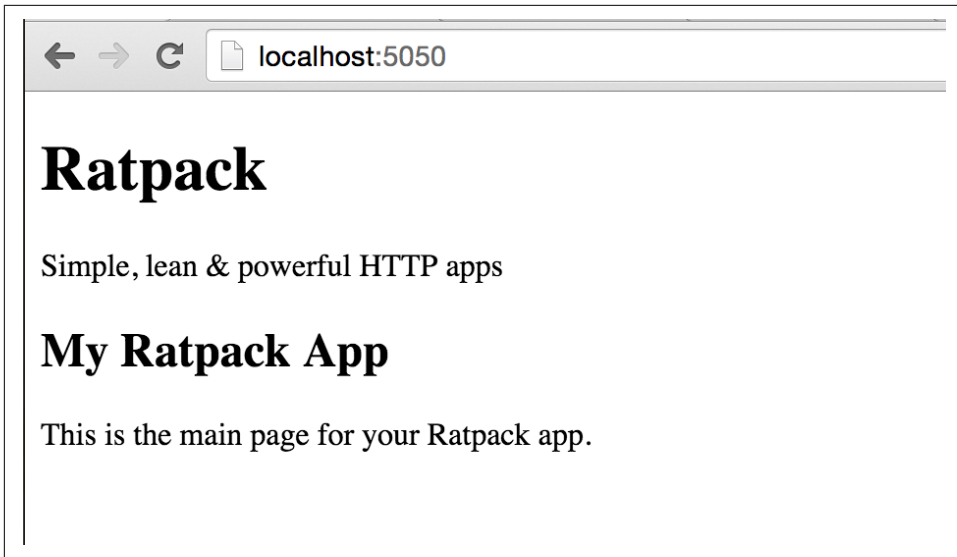
Figure 2-1. Lazybones template page

This templated project structure serves as a convenient starting place for your project. From here, you can begin to build out the requirements of your project and use provided Gradle build integration to package a runnable distribution.

Building from a Main Class

Ratpack's support for Groovy makes it easy to get started building complex applications using the *src/ratpack/Ratpack.groovy* file as an entry point into the application. However, not all Ratpack applications will be built with Groovy, or may not otherwise be able to follow the out-of-the-box Groovy DSL. In those cases, a project needs to have an entry point into the application. That is to say, some class needs to be provided that is runnable and does the work to stand up the Ratpack application.

In the case of Groovy applications that utilize *Ratpack.groovy* as their entry point, Ratpack is taking an opinionated approach to bootstrapping the application behind the scenes. Indeed, there is a runnable Java class that sits behind this to stand up the Ratpack server and configure the pieces necessary to get the web application runtime in place.

Java-based projects, for example, must provide a runnable Java class similar to the code shown in Example 2-16.

Example 2-16. Ratpack Java Main class

```java
package app;

import ratpack.server.RatpackServer;
import ratpack.server.ServerConfig;

public class Main {

  public static void main(String[] args) throws Exception {
    RatpackServer.start(spec -> spec
        .handlers(chain -> chain
          .get(ctx -> ctx.render("Hello, World!"))
        )
    );
  }
}
```

In this example, the Ratpack server is explicitly bootstrapped using the `Ratpack Server.start(..)` faculty, which takes a *definition* of the application, including the server configuration, any component bindings, and the handler chain. The definition's API is not dissimilar from the Groovy DSL, with the exception that the handler chain must build off of the `Chain` object that is provided to the `handlers` method. In Groovy, the closure supplied to this method uses *delegation*, which changes the lexical scope during execution. We do not have that same luxury with lambda expressions, so we must make explicit method calls to the handler chain's API.

The `Main` class is now entirely self contained and runnable. Running the project will start the server and bind to the default listening port. Navigate to *http://localhost:5050* and you will see the "Hello, World!" message displayed in your browser.

Use of a `Main` class is not restricted to only Java-based applications. Indeed, any language that builds on top of the JVM can leverage Ratpack's `Main` class paradigm as an entry point to a runnable Java application. In fact, Groovy applications that wish to utilize an explicit `Main` class instead of the *src/ratpack/ratpack.groovy* file can do so, and still employ the Groovy DSL, as shown in Example 2-17.

Example 2-17. Groovy chain DSL

```groovy
package app

import ratpack.groovy.Groovy
import ratpack.server.RatpackServer

class MainGroovy {

  public static void main(String[] args) {
    RatpackServer.start { spec -> spec
      .handlers(Groovy.chain {
```

```
      get {
        render "Hello, World!"
      }
    })
  }
 }
}
```

The part that matters most in Example 2-17 is the `Groovy#chain` call, which exposes the same `handlers` API as would be found in the *src/ratpack/ratpack.groovy* file.

Working with Handlers

A cursory overview of the handler chain, including its semantic API, was given in the previous chapter, and for many use cases, the information contained in that section is perhaps all that is needed. However, there are scenarios where the simplicity of inline handler declarations is not suited for the use case, and for that, Ratpack makes it easy to grow to a more robust layout. To extend beyond the basics, it is important to discuss what is actually happening behind the scenes when a handler is defined.

Understanding the Chain API Interactions in Groovy and Java

To start with, understand that most of the examples in this book emphasize the use of the Groovy handler chain DSL for defining the structure of a Ratpack application. The previous section exposed you to the handler chain semantics using the Java API. As you can see, the differences between using Groovy and using Java are fairly limited. For Ratpack, that is an intentional design decision.

All of the Groovy DSL in Ratpack is an affixture to the underlying pure Java API, meaning that the Groovy DSL is little more than a decoration for syntax sugar. To that extent, the semantics of the Groovy examples translate near-literally to the Java API, by understanding simply that in Java you are working with lambda expressions instead of closures. Whether you are using Java or Groovy, behind the scenes, Ratpack is taking the code segment that you have defined in the handler chain and coercing it to a `ratpack.handling.Handler` object.

The `Handler` class is the functional interface (in Java 8 terms) upon which all request handlers in Ratpack must be built. It exposes a single method, `handle`, which takes a `ratpack.handling.Context` object as an argument; it is through the `Context` that a handler interacts with a request. In Java 8, lambda expressions are seamlessly coerced to a functional interface type, without any need for explicit casting within the code. Groovy closures are similarly able to be cast to functional interface types. This allows the handler chain API to define a signature of `Chain#get(Handler)`—which, as we've already seen, is the semantic binding for an HTTP GET request—and have the Groovy representation of `chain.get { <handler logic> }` be symmetrical to the

corresponding Java 8 code of `chain.get(ctx -> <handler logic>)`. In both examples, the handler code segment is being automatically coerced into the `Handler` type, and the code will be executed when the `Handler#handle(Context)` method is invoked (as it is for matching requests).

The main difference to recognize between the Groovy and Java examples is that the Groovy examples do not need to specify the variable argument that is passed into them. This is because behind the scenes Ratpack utilizes Groovy's method dispatching mechanism to instruct the closure that it is to resolve any method calls to the scope of the `Context` object (before then reaching to the outer scope). This makes it shorter to write code dealing with request processing, by surfacing those method calls directly in the handler code block. To understand this better, consider that the Groovy code `chain.get { render "Hello, World!" }` is technically equivalent to `chain.get(ctx -> ctx.render("Hello, World!"))` through the Java API. (The difference of parentheses after the `render` method call is due to the fact that Groovy does not require variable arguments to be wrapped in parens.) Indeed, the Groovy closure could specify the `ctx` variable argument to the closure using the arrow notation, and the logic would be no different beyond being slightly more explicit. For instance, `chain.get { ctx -> ctx.render "Hello, World!" }` is equivalent to the previous Groovy example.

The scoping of Groovy closures to a working type does not start at handlers, however. As you may have noticed from the prior examples, the call to register a `get` handler does not require you to expand it out to its full invocation of: `chain.get { ... }`. That is because within the `handlers` section of the Groovy chain DSL, method calls to the `Chain` are implicitly dispatched. Similar to handlers and the context variable argument, the `chain` variable argument can be specified and it would be no more technically correct or incorrect. Using the format of `handlers { chain -> chain.get { ctx -> ctx.render "Hello, World!" } }` is equivalent to the more concise version of `handlers { get { render "Hello, World!" } }`.

Ratpack is able to leverage lower-level aspects of the Groovy compilation engine that inform it as to the intended scope of a coerced closure. This allows the handler and chain code segments to get aid in autocompletion from IDEs and even allows the Groovy DSL code blocks to be statically compiled. Ratpack's design decision to align the Groovy DSL with the underlying Java API is geared toward making the interactions with the framework more explicit, and therein reduces complexity when needing to understand or debug the behavior of a particular segment of code.

Standalone Handlers

It is indisputable that the easiest way to get started with Ratpack is by inlining the handler logic into the application's handler chain—as has been depicted in every

example up until now. But, while using closures and lambda expressions is a fast and expressive way to define a handler's functionality, it may not be the most logical approach for handlers with reasonably complex logic. In this scenario, extracting the handler logic to its own class may aid in the encapsulation and extended development of request processing logic.

As noted earlier in this chapter, the use of closures and lambda expressions is really just an expressive shortcut for giving Ratpack a `Handler` object. There are many scenarios where it might make sense to have your handler logic organized into its own class. A standalone handler can help to facilitate unit testing and separate the code for a handler's logic. Indeed, it is probably wise for any nontrivial handler logic to be defined in its own class.

Writing a standalone handler starts by simply implementing the `ratpack.han dling.Handler` interface and its corresponding `handle(ratpack.handling.Con text)` method. It can then be attached to the handler chain through the chain API, as demonstrated in the prior examples. Consider the code in Example 2-18, which shows a standalone handler class, and its corresponding association in the handler chain.

Example 2-18. The standalone handler class

```groovy
import static ratpack.groovy.Groovy.ratpack
import ratpack.handling.Handler
import ratpack.handling.Context

class DefaultRouteHandler implements Handler {
  private final String message

  DefaultRouteHandler(String message) {
    this.message = message
  }

  @Override
  void handle(Context context) {
    context.render message
  }
}

ratpack {
  bindings {
    add(new DefaultRouteHandler("Hello, World!"))
  }
  handlers {
    get(DefaultRouteHandler)
  }
}
```

This example introduces the most important caveat to standalone handlers: they must be present in the Ratpack component registry to be usable from the chain. The registry will be covered in depth in Chapter 5; for now, just keep in mind that the `bindings` block and its corresponding `add` method are the appropriate means for registering a standalone handler.

The `DefaultRouteHandler` class has a single parameter constructor that allows the message that is to be sent to the client to be configurable. We register it in the bindings block by constructing it with the "Hello, World!" message, and we refer to it in the handler chain by class name. As you can see from the handler chain, we are able to bind our standalone handler to the default route through the `get(DefaultRouteHandler)` handler method.

Standalone handlers provide greater reusability of common logic. We may, for example, want to bind a standalone handler to different routes, where its logic does different things according to attributes of the request. To understand this better, see the logic in Example 2-19, which is a slight modification to the prior example, this time depicting a reusability scenario.

Example 2-19. A standalone handler serviceable to different routes

```groovy
import static ratpack.groovy.Groovy.ratpack
import ratpack.handling.Handler
import ratpack.handling.Context

class DefaultRouteHandler implements Handler {
  private final String defaultMessage

  DefaultRouteHandler(String message) {
    this.defaultMessage = message
  }

  @Override
  void handle(Context context) {
    if (context.pathTokens.containsKey("name")) {
      context.render "Hello, ${context.pathTokens.name}!"
    } else {
      context.render defaultMessage
    }
  }
}

ratpack {
  bindings {
    add(new DefaultRouteHandler("Hello, World!"))
  }
  handlers {
    get(DefaultRouteHandler)
    get(":name", DefaultRouteHandler)
```

```
    }
}
```

This time, the handler is actually processing attributes of the request to make a judg-
ment as to what data is returned to the client. The logic is simple enough, but it
requires that the handler be bound to both the default route, and a route that has a
corresponding path token (path bindings and path tokens were discussed in Chap-
ter 1). We are able now to use the registered DefaultRouteHandler across many dif-
ferent bindings in the handler chain.

The preceding examples were contrived "Hello, World!" demonstrations to get you
familiar with the concept of standalone handlers, while keeping the code as easy to
follow as possible. A real-world scenario in which a standalone handler provides a
great deal of benefit is when many Ratpack applications need to share the same logic
—most likely for a top-level or entry handler. This handler logic can be written once,
packaged into a standalone artifact, and brought into one or many projects as a
dependency.

The need for this type of common logic has various use cases. Consider, for example,
the increasingly common scenario of a microservice platform architecture, which
publishes a client library for consumers. Each microservice in the stack can utilize
common handler logic to inspect a request and make intelligent rendering decisions
based on the version of the client, as defined by the incoming "User Agent" header.
This can allow a microservice to render a model according to the object structure that
is known to correspond with the requesting client.

The Ratpack application in Example 2-20 demonstrates the use of a standalone han-
dler for request introspection and handling according to the incoming User-Agent.

Example 2-20. Complex standalone handler beyond "Hello, World"

```
import ratpack.handling.Context
import ratpack.handling.Handler
import ratpack.registry.Registry

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

class UserAgentVersioningHandler implements Handler {
  private static final String ERROR_MSG = "Unsupported User Agent"

  enum ClientVersion {
    V1("Microservice Client v1.0"),
    V2("Microservice Client v2.0"),
    V3("Microservice Client v3.0")

    String versionString
```

```
      ClientVersion(String versionString) {
        this.versionString = versionString
      }

      static ClientVersion fromString(String versionString) {
        for (val in values()) {
          if (val.versionString == versionString) {
            return val

          }
        }
        null
      }
    }

  @Override
  void handle(Context context) {
    def userAgent = context.request.headers.get("User-Agent") ❶
    def clientVersion = ClientVersion.fromString(userAgent)
    if (!clientVersion) {
      renderError(context)
    } else {
      context.next(Registry.single(ClientVersion, clientVersion)) ❷
    }
  }

  private static void renderError(Context context) {
    context.response.status(400)
    context.byContent { spec ->
      spec.json({
        context.render(json([error: true, message: ERROR_MSG]))
      }).html({
        context.render("<h1>400 Bad Request</h1><br/><div>${ERROR_MSG}</div>")
      }).noMatch {
        context.render(ERROR_MSG)
      }
    }
  }
}

ratpack {
  handlers {
    get(new UserAgentVersioningHandler()) ❸

    get("api") { UserAgentVersioningHandler.ClientVersion clientVersion -> ❹
      if (clientVersion == UserAgentVersioningHandler.ClientVersion.V1) {
        render "V1 Model"
      } else if (clientVersion == UserAgentVersioningHandler.ClientVersion.V2) {
        render "V2 Model"
      } else {                  // it must be V3 at this point, as the versioning
        render "V3 Model"  // handler has figured out the request qualifies
      }
```

```
      }
    }
}
```

❶ In the `UserAgentVersioningHandler`, the request processing starts by extracting the `User-Agent` header from the request. Note that this line of code is using Groovy's dot notation on a `Map` object; the equivalent Java code is simply `context.getRequest().getHeaders().get("User-Agent")`. From here, it checks if the `User-Agent` is valid, and if not it renders a polite error message back to the client; if so, the `User-Agent` string is tokenized and the relevant version attribute is extracted. The version string is then processed through the `ClientVersion` enumeration, where it is turned into a representative object. In this example, a `User-Agent` string of `Microservice Client v1.0` will by represented as the `ClientVersion.V1` enumerated value. Likewise, `v2.0` and `v3.0` will be mapped to their respective values. If no enumerated value is able to represent the `User-Agent`, then the client is provided a "Bad Request" response.

❷ This is perhaps the most important part of the flow, as this is the portion of code responsible for getting the `ClientVersion` enum value to the downstream API handler. (Contextual objects will be covered in Chapter 5.) This is also the line of code that is responsible for informing Ratpack to continue request processing down the chain.

❸ At the very beginning of the handler chain is where we have applied the `UserAgentVersioningHandler`. This allows the client versioning logic to exist upstream and provide for the API handler what version the client is using.

❹ This line of code represents the handler for the `api` endpoint. In this example, the variable argument to the handler closure is extracting the `ClientVersion` object from the context registry, making it available for processing as shown. The registry and its corresponding accessor methods will be covered in depth in Chapter 5.

Many microservices in the application stack may be accessed by a single client, and it should be clear at this point that Ratpack's ability to introduce standalone handlers into the chain is quite powerful.

Chapter Summary

As you move forward from this chapter, you take with you a firm grasp of the basics of Ratpack, including a great understanding of how to create and work in a full-fledged project structure. The chapters that follow will lean heavily on the knowledge that you have gained from this chapter, from working in project structures, to creat-

ing Gradle build scripts, and working with standalone handlers. You certainly by now have the fundamental understanding necessary to build robust applications with Ratpack. Now that you know how to build these applications, we must next show you how Ratpack makes it easy to test them. Your experience from this chapter will prove fundamentally important to the conversation of building applications that are well tested.

Testing Ratpack Applications

Now that we have covered how to build proper project structure and work with requests a little more completely, it is important to understand how to go about writing tests for your application. As a developer-first framework, Ratpack understands the need to make testing applications easy, and thus provides many test fixtures and utilities to aid in test-driven development and writing functional and integration tests.

Getting started with testing in Ratpack is as easy as including a test-scoped dependency in your project. If you are making use of the Gradle integration, then the process of adding the dependency will look very similar to how we imported framework components into the project in the previous chapter. If you are using a dependency and build management tool besides Gradle, then the `ratpack-test` framework module needs to be brought into the test time's compile and runtime classpath.

When using the Gradle plugin, you can add the `ratpack-test` framework module through the `dependencies` block of your project's *build.gradle* file. This time, however, it will need to add the dependency to the `testCompile` build configuration, so that the test fixtures are properly isolated from the regular compile and runtime classpaths. In addition to the `ratpack-test` module, you need to include a testing framework. Ratpack's test fixtures do not enforce any opinions about the testing infrastructure your project uses, but the Spock Framework is the preferred test framework. There are many examples of testing Ratpack applications with Spock within Ratpack's code base, and this chapter will draw heavily on building tests in that way.

The build script in Example 3-1 is an extended version of the one we worked on in Chapter 2. This time, the `ratpack-test` and Spock Framework dependencies are

added to the `testCompile` build configuration. Note that for complete object mocking and spying, Spock requires the `objenesis` and `cglib` dependencies as well.

Example 3-1. Build script with test dependencies

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  testCompile ratpack.dependency('test') ❶
  testCompile 'org.spockframework:spock-core:1.0-groovy-2.4' ❷
  testCompile 'cglib:cglib:2.2.2' ❸
  testCompile 'org.objenesis:objenesis:2.1' ❹
}
```

❶ Here, we add the `ratpack-test` dependency to the project's build in the *testCompile* configuration. Dependencies in this configuration are used at build time, but are not included in the final artifact.

❷ As noted, you are free to use whatever test framework is suitable to your project's requirements, but in this chapter we will largely demonstrate working with Spock, so here we add the Spock dependency.

❸ `cglib` is an optional dependency to Spock, though largely needed for providing mock object types.

❹ Likewise, `objenesis` is optional, but useful if you intend to mock concrete types, so we add it here for completeness.

Now that the dependencies are added, you will need to ensure that your project setup is structured for your test classes. Simply creating a new directory structure that follows common Maven-style project source set configuration will allow you to create tests that can utilize the `testCompile` configuration. The project structure should look similar to that found in Example 3-2. If you used Lazybones to build your project, you will already have the *src/test/* directory structure in place.

Example 3-2. Project structure with test source set

```
.
└── src
    ├── main
    │   └── groovy
    │       └── tld
    │           └── company
    │               └── app ❶
    ├── ratpack
    │   └── Ratpack.groovy
    └── test
        └── groovy
            └── tld
                └── company
                    └── app ❷
```
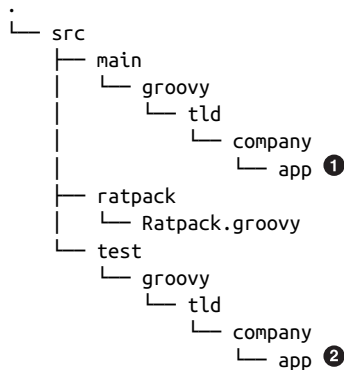
❶   Project classes go here.

❷   Test classes go here.

As you can see from the structure, the *src/test/* source set mirrors the *src/main/* tree. If you are already familiar with common Java project structures, this will be nothing compelling. If you haven't seen this before, just keep in mind that source code in *src/test/* gets both the project's compile and runtime dependency set, as well as the dependencies from the `testCompile` configuration.

With the test structure now in place, you can run your tests from either the command line or from within the IDE. The Gradle Java and Groovy plugins will run tests within their respective test directories every time the project is built. To simply run the tests from the command line, issue the `./gradlew test` command from the project root.

Spock Test Structure

Before diving into testing strategies in Ratpack, it is important to take a quick detour to understand a few key points about Spock. If you have never used Spock, or come from a background of writing tests in pure Java (with JUnit, for example), then some of the constructs brought about by Spock may at first appear foreign. Arguably the most important thing to understand about Spock is that at its most fundamental level, it is just a test runner on top of JUnit. Therefore, Spock tests are able to be run in the same way that typical JUnit tests would be run, and running them from within the IDE or from the command line is supported. Spock tests and JUnit tests are able to exist alongside one another within a project, without any risk of overlapping problems.

Spock tests employ a *behavior-driven development* (BDD) structure, and are referred to as *specifications*. The idea behind this is that a test structure is built around a fea-

ture specification, and should be as human readable as possible. Specifications consist of many *features*, and these features are the actual tests that will be run. A specification is a *test class*, and the features are methods within that class.

Following the BDD approach, features are defined in different blocks, which denote the component pieces of a feature's test. These blocks are organized by their function in the order of:

setup
> Used to set up some test data for the rest of the feature.

stimulus
> The invocation of the feature code under test—a method call to a service, for example. This block may also be used to capture repsonse data from application code and store it in a local variable for the assertion block.

assertion
> An assertion about the behavior observed from the stimulus block. This may be as simple as asserting that some data returned from a service call equals a calculated value. It is important to note that within this block, all statements are implicit assertions. Unlike JUnit tests, which must explicitly perform an assertion, each line of code in this segement is treated as an assertion (and therein respected as a boolean result).

cleanup
> Provides a segment to run some code after the feature test has completed. This is useful for cleaning up any test data that was bootstrapped in the setup block, and this code segment is always run, regardless of assertion success or failure.

Example 3-3 shows a simple service class that we will use to illustrate testing with Spock.

Example 3-3. The MyService class

```
package tld.company.app

class MyService {

  String doServiceCall() {
    "service was called"
  }

  void shutdown() {
    // stub implementation
  }
}
```

Example 3-4 demonstrates the corresponding specification for the MyService class.

Example 3-4. Specification for MyService

```groovy
package tld.company.app

import spock.lang.Specification

class MyServiceSpec extends Specification { ❶

  void "service calls should return proper response"() { ❷
    setup: ❸
    "Set up the service for testing"
    def service = new MyService()

    when: ❹
    "Perform the service call"
    def result = service.doServiceCall()

    then: ❺
    "Ensure that the service call returned the proper result"
    result == "service was called"

    cleanup: ❻
    "Shutdown the service when this feature is complete"
    service.shutdown()
  }
}
```

❶ The `MyServiceSpec` class extends Spock's `Specification` type. This is needed to ensure that Spock's test runner properly captures this class as a test.

❷ Here, we define the feature, and we do so using a human-readable definition of what this feature is aiming to accomplish.

❸ Within the `setup` block, we construct a new instance of `MyService`.

❹ The `when` block is the stimulus phase of the test, where we actually invoke the feature-under-test.

❺ The `then` block is where we make assertions about the result of the service call.

❻ And finally, we use the `cleanup` block to perform any after-test work that needs to happen to ensure the object and its dependencies are disposed of properly.

As you can see from the specification, the different blocks can be optionally labeled with a descriptive text of their role in the feature test. This can help future developers in understanding the intention of the feature's implementation. In this specification, the setup, stimulus, assertion, and cleanup blocks are denoted by the `setup`, `when`, `then`, and `cleanup` labels, respectively. The labeling structure provides a human-

readable flow to the structure of the test. Matching that with the description text makes the feature test readable from beginning to end.

The `setup` block creates the instance of the `MyService` class for use within the test. The `when` block—or the stimulus—is then responsible for performing the function of the test and capturing the result. The `then` block performs the assertion that the `MyService` implementation has returned the proper value, and the `cleanup` block runs the finalization operation—in this case, a stub function on the implementation.

From the Example 3-4 demonstration, you can understand a few more important aspects of Spock specifications. The first is that features within a specification are able to be written as methods, and Spock leverages the fact that Groovy allows you to create a method name as a string value. This allows a feature test's implementation within a specification to provide descriptive text about the feature directly within the method signature.

String method names and the ability to segment code blocks within a feature method are capabilities that are heavily leveraging Groovy to provide Spock's BDD capabilities. To that end, specifications must always be written in Groovy, as Spock utilizes AST transformations to convert the readable specification into a JUnit runnable test. Using Groovy also reduces the verbosity of the code in the specification, aiding in the human readability of the overall specification. This allows the emphasis of a Spock specification to remain of the feature test, without having to too deeply decipher the test's code.

The Spock framework provides many more capabilities, and it is worth taking the time to explore the documentation to further understand its features. In addition, there are many examples of practical and advanced Spock usage within the Ratpack code base, and you can get further understanding of testing Ratpack applications with Spock by going through some of the tests in Ratpack's GitHub repository.

Functional Testing

Functional testing is a testing strategy by which your application starts under a test configuration, without any mocks being applied to replace actual service implementations. A functional test configuration may, for example, choose to use ephemeral or test-dedicated datasources for data access objects. That is to say, if your application uses MySQL as a datasource, then your functional test configuration may point to a MySQL host that can be truncated after each run.

Ratpack makes it easy to write functional tests, because the Ratpack server is inherently embeddable. To make writing functional tests even easier, Ratpack provides fixtures that simplify the process of embedding an existing application in a test.

For the scenario of a Groovy-based Ratpack application, a functional test can leverage the `GroovyRatpackMainApplicationUnderTest` fixture to perform requests against a running version of the application. Considering the simple "Hello, World!" application demonstrated previously, the test class in Example 3-5 shows how to utilize Ratpack's test fixture to make an assertion about the response data from the default handler.

Example 3-5. Groovy FunctionalSpec

```groovy
package app

import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import spock.lang.Specification

class FunctionalSpec extends Specification {

  void "default handler should render Hello, World!"() {
        setup:
    def aut = new GroovyRatpackMainApplicationUnderTest() ❶

    when:
    def response = aut.httpClient.text ❷

    then:
    response == "Hello, World!" ❸

    cleanup:
    aut.close() ❹
  }
}
```

❶ Here, we set up the `ApplicationUnderTest`. Because we're working on a Groovy project, the `GroovyRatpackMainApplicationUnderTest` will be used to start the application and provide our test with fixtures to aid in different testing scenarios.

❷ The `httpClient` is one such fixture that we can use to access our application.

❸ In the `then` block, we make the assertion that the test returned the proper response.

❹ Finally, we close the application, which will properly shut down the application server and services.

All `ApplicationUnderTest` test fixture implementations in Ratpack provide an HTTP client, which provides a DSL for performing requests against the running application. In this example, the `aut.httpClient.text` call is performing an HTTP GET request to the default endpoint. The `when` block is capturing the response text

from the GET request, and the `then` block is making an assertion that we got the "Hello, World!" message as we expect.

Spock can also be used to test Ratpack applications that are not based on Groovy. Those applications, which build from a main class, can use the `MainApplicationUnderTest` to specify the entry point to the application under test. Given the main class in Example 2-16, from earlier in the book, the Example 3-6 specification shows this capability.

Example 3-6. Main class FunctionalSpec

```groovy
package tld.company.app

import ratpack.test.MainClassApplicationUnderTest
import spock.lang.Specification

class FunctionalSpec extends Specification {

  void "default handler should render Hello, World!"() {
    setup:
    def aut = new MainClassApplicationUnderTest(Main)

    when:
    def response = aut.httpClient.text

    then:
    response == "Hello, World!"

    cleanup:
    aut.close()
  }
}
```

The features for the functional specification of a Ratpack application need not always define a new `ApplicationUnderTest` within their `setup` block. Instead, a functional specification can define the `ApplicationUnderTest` object at the class level, and write many features against that single object. This allows an application with many handlers to only be started once within the specification, while providing many assertions about the application's function.

Consider again a more complex Ratpack application, like the one demonstrated earlier in Example 2-20, which utilized a standalone handler to discriminate on the `User-Agent` header of the request. We can write a functional specification with different feature methods to describe the behavior we expect for the different `User-Agent` versions our application recognizes. The specification in Example 3-7 provides feature tests for the different client versions the application is capable of handling.

Example 3-7. FunctionalSpec with multiple features

```groovy
package tld.company.app

import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import spock.lang.AutoCleanup
import spock.lang.Specification

class FunctionalSpec extends Specification {

  @AutoCleanup
  def aut = new GroovyRatpackMainApplicationUnderTest()

  void "should properly render for v1.0 clients"() {
    when:
    def response = aut.httpClient.requestSpec { spec ->
      spec.headers.'User-Agent' = ["Microservice Client v1.0"]
    }.get("api").body.text

    then:
    response == "V1 Model"
  }

  void "should properly render for v2.0 clients"() {
    when:
    def response = aut.httpClient.requestSpec { spec ->
      spec.headers.'User-Agent' = ["Microservice Client v2.0"]
    }.get("api").body.text

    then:
    response == "V2 Model"
  }

  void "should properly render for v3.0 clients"() {
    when:
    def response = aut.httpClient.requestSpec { spec ->
      spec.headers.'User-Agent' = ["Microservice Client v3.0"]
    }.get("api").body.text

    then:
    response == "V3 Model"
  }
}
```

In this specification, we move the `GroovyRatpackMainApplicationUnderTest` object
to the class level, and annotate it with `@AutoCleanup`. The annotation will inform
Spock that the `close()` method should be called on the object when all feature tests
have finished running. We can now omit both the `setup` and `cleanup` blocks and
each feature test describes the stimulus and assertion blocks. When the specification
is instantiated for running, it will start the application just once, and each feature is
able to perform requests against that running application. When all the features have

been run, the application will be gracefully stopped, allowing subsequent specifications to also start the application without any risk of overlap.

This specification also introduces the test HTTP client's `requestSpec()` method, which allows the feature to configure the attributes of the outgoing request. In this example, we are able to use the `RequestSpec` to add the `User-Agent` header, according to what each feature test is asserting. Once the request is properly configured, the `.get('api')` call is made, which performs the HTTP GET request against the `/api` endpoint. This returns a `Response` object, which is where the `body` property lives. Calling `body.text` extracts only the content from the response, as `aut.httpClient.text` did in the previous example. Note that when performing the `get('api')` call, no preceding slash is specified in the URI.

Example 3-7 provided a verbose demonstration of the ability to run multiple feature tests against a single `ApplicationUnderTest`. The specification can be rewritten more simply by employing Spock's ability to *rollup* test values into a single feature. The code in Example 3-8 demonstrates a cleaner implementation of the same specification.

Example 3-8. Refactored FunctionalSpec with rollups

```
package tld.company.app

import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import spock.lang.AutoCleanup
import spock.lang.Specification
import spock.lang.Unroll

class FunctionalSpec extends Specification {

  @AutoCleanup
  def aut = new GroovyRatpackMainApplicationUnderTest()

  @Unroll
  void "should render #expected for #userAgent clients"() {
    when:
    def response = aut.httpClient.requestSpec { spec ->
      spec.headers.'User-Agent' = [userAgent]
    }.get("api").body.text

    then:
    response == expected

    where:
    userAgent                | expected
    "Microservice Client v1.0" | "V1 Model"
    "Microservice Client v2.0" | "V2 Model"
    "Microservice Client v3.0" | "V3 Model"
  }
}
```

Notice the `where` block at the bottom of the feature specification. This allows us to specify a *data table* of variables that can be applied to the stimulus and assertion blocks. This feature test will be run as three standalone tests, while the feature code is able to be more concisely implemented.

The test HTTP client is a powerful utility in the test fixtures that Ratpack provides. In addition to the prior demonstrations, it also supports:

- The ability to read the response status code through a call such as `aut.httpClient.get("api").statusCode`
- The ability to read response headers through a call such as `aut.httpClient.get("api").headers.'Content-Type'`
- The ability to perform GET/POST/PUT/PATCH/DELETE operations through their corresponding (lowercased) method names
- The ability to manipulate the request body for methods supporting it through the `RequestSpec.Body`

Bootstrapping Test Data

Any test type will often need to have some data bootstrapped into the application so that complex data responses can be observed. Functional tests can make standing up test data difficult, since you want the application to run in as pure a runtime as possible, without manipulating its startup sequence. To that extent, tests may start a feature by POSTing data to an API endpoint that stores it for retrieval. This approach has two benefits, in that you will be testing both the application's ability to properly store the data, as well as its ability to render that data back to clients.

To demonstrate this approach more effectively, consider the application in Example 3-9. For demonstration's sake, the application provides a `post` handler on the `api` endpoint, which parses the incoming request data into a usable object structure and stores it in a global list. A call to the `get` handler on the `api` endpoint will in turn render the objects back to the caller.

Example 3-9. Bootstrapping test data

```
import groovy.json.JsonSlurper

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

class User {
  String username
  String email
}
```

```
List<User> userStorage = []
JsonSlurper jsonSlurper = new JsonSlurper()

ratpack {

  handlers {
    path("api") {
      byMethod {
        post {
          request.body.map { body ->
            jsonSlurper.parseText(body.text) as Map
          }.map { data ->
            new User(data)
          }.then { user ->
            userStorage << user
            response.send()
          }
        }
        get {
          response.send(toJson(userStorage))
        }
      }
    }
  }
}
```

Example 3-10 shows the application's corresponding tests.

Example 3-10. Bootstrapping test data FunctionalSpec

```
package tld.company.app

import groovy.json.JsonSlurper
import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import spock.lang.AutoCleanup
import spock.lang.Specification

import static groovy.json.JsonOutput.toJson

class FunctionalSpec extends Specification {

  private static final JsonSlurper jsonSlurper = new JsonSlurper()

  @AutoCleanup
  def aut = new GroovyRatpackMainApplicationUnderTest()

  void "bootstrap data and properly render it back"() {
    setup:
    def user = [username: "danveloper", email: "danielpwoods@gmail.com"]

    when:
```

```
    def response = aut.httpClient.requestSpec { spec ->
      spec.body { b ->
        b.text(toJson(user))
      }
    }.post('api')

    then:
    response.statusCode == 200

    when:
    def json = aut.httpClient.get('api').body.text

    and:
    def users = jsonSlurper.parseText(json) as List

    then:
    users == [user]
  }
}
```

The `FunctionalSpec` for this example introduces some new concepts, such as the ability to have multiple stimulus and assertion blocks within a single feature test. It also shows the use of the `and` block to logically provide multiple stimuli prior to an assertion (the `then` block). This fluid capability of Spock's allows you to effectively organize your test to bootstrap data, assert that the data was properly bootstrapped, then read it back out.

Architecting for Improved Testability

The example application in this section achieves its storage and retrieval through the use of a globally scoped list, `userStorage`. This works great for demonstrative purposes and to concisely understand how Ratpack handlers and their corresponding tests are able to be written, but in a real-world example, we would want the storage mechanism to be contained in the application and resolved from interfaces. This will add benefit to our application and the ability to read and write data in a test scenario.

If we rearchitect the application from Example 3-9 to instead resolve the `userStorage` from the application's user registry (more on this in "Extending Ratpack with Registries" on page 104), we can have much more granular control over how the application stores and accesses data. The prior example specification demonstrated a "round trip" of posting data to the `api` endpoint, and then subsequently reading it back out. We may, instead, want to have different feature tests that demonstrate the component parts of that process.

To start with, we should begin by creating a `UserService` interface that will act as our service or data access layer for storing and retrieving the posted user data. When considering testing, programming to interfaces for aspects of code that manage data

interaction is always a good idea. Example 3-11 depicts the `UserService` interface. It should also be noted that at this point, we should move these and the `User` class out to the *src/main/groovy/* directory in our project structure. This way, they are still available to our application, but are not sloppily embedded in the *src/ratpack/ ratpack.groovy* script.

Example 3-11. The UserService interface

```groovy
package tld.company.app

import ratpack.exec.Promise

interface UserService {
  Promise<Void> save(User user)
  Promise<List<User>> getUsers()
}
```

Example 3-12 provides a demonstration for a `UserService` implementation.

Example 3-12. UserService implementation

```groovy
package tld.company.app

import ratpack.exec.Promise

class DefaultUserService implements UserService {
  private final List<User> storage = []

  @Override
  Promise<Void> save(User user) {
    storage << user
    Promise.sync { null }
  }

  Promise<List<User>> getUsers() {
    Promise.sync { storage }
  }
}
```

These examples also demonstrate a concept known as the *execution layer*, which will be covered in more depth later in the book. Having the `UserService` interface contracts defined to return a `Promise` allows the concrete implementation to employ the appropriate strategy for accessing its `User` storage. In the preceding example, we are simply pushing the user data into a list in the `save` method, and equally as simply returning that whole list from the `getUsers` method. A more robust implementation may need to employ blocking mechanisms, which are offered through the execution. For our current purposes, defining service contracts as promises is a good idea.

With our service contract and concrete implementation now defined, we need to make a few changes to our *src/ratpack/Ratpack.groovy* script to make use of the `User Service` within our application. To do this, we will *bind* the `DefaultUserService` to the `UserService` interface in the registry, and then *inject* it for use within the `api` handler. The code in Example 3-13 shows this change.

Example 3-13. Working with DefaultUserService

```groovy
import tld.company.app.DefaultUserService
import tld.company.app.User
import tld.company.app.UserService
import groovy.json.JsonSlurper

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings { ❶
    bindInstance UserService, new DefaultUserService()
    bindInstance JsonSlurper, new JsonSlurper()
  }
  handlers {
    path("api") { JsonSlurper jsonSlurper, UserService userService -> ❷
      byMethod {
        post {
          request.body.map { body ->
            jsonSlurper.parseText(body.text) as Map ❸
          }.map { data ->
            new User(data)
          }.flatMap { user ->
            userService.save(user)
          }.then {
            response.send()
          }
        }
        get {
          userService.getUsers().then { users -> ❹
            response.send(toJson(users))
          }
        }
      }
    }
  }
}
```

❶ Within the `bindings` block, we specify the binding of the `DefaultUserService` to the `UserService` interface.

❷   Here, we define the /api endpoint, and at this point we use the closure variable-argument syntax to specify that we want the UserService and JsonSlurper injected into the handler.

❸   Within the post logic of the handler's byMethod block, we use the JsonSlurper to consume the request body as a Map type.

❹   Within the get logic, we work with the Promise returned from the getUsers call to render back the user list.

Our application is also now working with the Promise objects returned from the User Service. The essence to the underlying *reactive programming* technique will be covered in depth in Chapter 10, but for the sake of understanding here, it is important to understand that a Promise represents a "promise for the requested data." When a promise is *subscribed to* through the then method, the call for the data is made and subsequently returned. As you can see in the handler's get logic, the resulting data is made available to the then method so we can render properly back to the client.

At this point, we can rerun the FunctionalSpec from before to ensure that we indeed did not break anything through the refactor, and indeed we will find that nothing has. This use of functional testing within your Ratpack applications provides you with a comprehensive test footprint from the very beginning of your project. As your requirements grow, you will undoubtedly find needs and uses for integration and unit testing, and when you do, Ratpack continues to support you in those pursuits.

Integration Testing

When it comes to testing your application, it may not always be desirable to require a feature test to perform the "round trip" described in the prior sections. Instead, you want may to have different feature tests for posting and retrieving data. It may be desirable in this scenario to make use of a *mock implementation* of the UserService to assert that the appropriate data *will* be saved through the post handler, or to send back mock test data from the get handler. Doing so gives you more granular control over testing how data is stored and represented by your application.

If you are not familiar with mocks as a testing strategy, it is important to understand that a mock is a proxy object that emulates the *mocked type* and captures interactions. This allows your test to ensure that your application code is interacting with its service layer in the way that you would expect. Spock gives you the capability to inspect the object that is sent to a service call to ensure that the data meets a feature's requirement. In this section, we will demonstrate mocking the UserService and ensuring that a properly constructed User object is sent to the save method.

We can leverage the same `ApplicationUnderTest` fixture from the prior section to override the `DefaultUserService` binding, and instead replace it with a mock implementation for our test's sake. We must also introduce another powerful test fixture that Ratpack provides: the `ExecHarness`. The `ExecHarness` is an assistant to tests that rely on the execution layer for managing data flow in their applications.

Example 3-14 shows an integration specification for our application.

Example 3-14. Integration testing with Ratpack and Spock

```groovy
package tld.company.app

import ratpack.exec.Promise
import ratpack.registry.Registry
import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import ratpack.test.exec.ExecHarness
import ratpack.impose.ImpositionsSpec
import ratpack.impose.UserRegistryImposition
import spock.lang.AutoCleanup
import spock.lang.Specification

import static groovy.json.JsonOutput.toJson

class IntegrationSpec extends Specification {
  UserService mockUserService = Mock(UserService)   ❶

  @AutoCleanup
  ExecHarness harness = ExecHarness.harness()

  @AutoCleanup
  def aut = new GroovyRatpackMainApplicationUnderTest() {
    @Override
    protected void addImpositions(ImpositionsSpec impositions) {   ❷
      impositions.add(UserRegistryImposition.of(
        Registry.of { r ->
          r.add(UserService, mockUserService)
        }
      ))
    }
  }

  void "should convert and save user data"() {
    setup:
    def userMap = [username: "danveloper", email: "danielpwoods@gmail.com"]

    when:
    aut.httpClient.requestSpec { spec ->
      spec.body { b ->
        b.text(toJson(userMap))
      }
    }.post('api')
```

```
      then:
      1 * mockUserService.save(_) >> { User user ->
        assert user.email == userMap.email
        assert user.username == userMap.username
        Promise.sync { null }
      }
  }

  void "should render user list as json"() {
    setup:
    def users = [
        new User(username: "danveloper", email: "danielpwoods@gmail.com"),
        new User(username: "kenkousen", email: "ken@kousenit.com"),
        new User(username: "ldaley", email: "ld@ldaley.com")
    ]

    when:
    def response = aut.httpClient.get("api")

    then:
    1 * mockUserService.getUsers() >> Promise.sync { users }
    response.body.text == toJson(users)
  }
}
```

❶ This time, we have created a mock implementation of the UserService.

❷ Using the GroovyRatpackMainApplicationUnderTest's addImpositions method, we are able to instruct Ratpack that we prefer our mock over any previously bound implementations. (Note that the registry API is slightly different from that of the bindings block from the application section.)

We can now separate the test facets responsible for ensuring data is properly saved and retrieved. Instead of the round-trip style from the FunctionalSpec shown earlier, this time we have two feature methods, one for each part of the process. The "should convert and save user data" feature test is responsible for ensuring the posted data is properly coerced to a user object and that the properties of that user object are populated as expected. The "should render user list as json" feature test ensures that a list of users is rendered back as JSON.

Using the Spock interaction notation of 1 * mockUserService.save(_), we can ensure that the save method is only called once. In the then block of the feature test, we can then capture the User object that was sent from the handler and ensure that the email and username fields match what we would expect. Note that within an interaction we need to perform explicit assertions, as this segment of code does not correspond to Spock's assertion-per-statement rule that normally applies to the then

block. From here, we can leverage the `ExecHarness` test fixture to return a promise to the handler.

Within the "`should render user list as json`" feature test, we can use the `setup` block to construct a list of sample `User` objects that we expect to be returned from the handler. This is a nice alternative to manually bootstrapping data through the round-trip method demonstrated earlier in the chapter. We also utilize the `ExecHarness` test fixture here to return a `Promise` for our test users. The test ensures that the `getUsers` method is called and that the response matches a JSON-serialized version of our test data.

Unit Testing

Functional and integration tests in Ratpack get you pretty far, as the cost of starting your application in an embedded fashion within the specification is so low. But there are scenarios (such as when employing test-driven development) in which you will want to write unit tests for some feature of your code. Ratpack supports that scenario through the test fixture `ExecHarness`, which we discussed briefly in the previous section.

The `ExecHarness` test fixture, put simply, allows you to work with Ratpack's threading and execution model, and make assertions about asynchronous operations in a synchronous manner. This is valuable in unit testing, because your test specification will require that you have captured the result of an asynchronous call within the stimulus block of your feature test prior to being able to make an assertion about that result. Behind the scenes, it is actually building the working operation.

Consider again the `DefaultUserService` example from the prior sections, and you may recall that when we rearchitected our application for better testability, we moved the `UserService` interface contracts to return `Promise` objects. Not having known about the `ExecHarness`, you may have tried to write a unit test that made assertions about the result of the `getUsers()` call, but you quickly would have realized that you are not able to work with a `Promise` outside of a "managed thread." Ratpack's threading and execution model will be covered in depth in Chapter 7, but for our current purposes, it is enough to understand that a promise instructs Ratpack as to how the code contained within it should be executed. Without being on a managed thread, Ratpack has no way of knowing that.

In a normal application runtime, you never have to think about managed threads or Ratpack executions at all. In fact, your code will always be run within the context of a managed thread, so it is never an issue. Indeed, this is part of the reason why functional and integration testing in Ratpack is so appealing. Because in those testing scenarios you are starting your application from within your specification, you do not need to do any additional work to ensure your code is bound to a managed thread.

Unit testing differs here, as you are directly working with the *subject* of your specification, without Ratpack's underlying infrastructure.

Luckily, the `ExecHarness` test fixture provides the minimally required infrastructure for you to work within the context of a managed thread, but not have to do too much work that deviates your specification from its intended test logic. Example 3-15 shows a `UserServiceUnitSpec`, which utilizes the `DefaultUserService` concrete implementation to test the intended functionality of the `save` and `getUsers` methods.

Example 3-15. Unit test for UserService

```groovy
package tld.company.app

import ratpack.test.exec.ExecHarness
import spock.lang.AutoCleanup
import spock.lang.Shared
import spock.lang.Specification
import spock.lang.Subject

class UserServiceUnitSpec extends Specification {
  private static final def users = [
      new User(username: "danveloper", email: "danielpwoods@gmail.com"),
      new User(username: "kenkousen", email: "ken@kousenit.com"),
      new User(username: "ldaley", email: "ld@ldaley.com")
  ]

  @AutoCleanup
  ExecHarness execHarness = ExecHarness.harness() ❶

  @Subject @Shared
  UserService userService = new DefaultUserService()

  void "should save and return user list"() {
    given:
    execHarness.yield { ❷
      users.each { user -> userService.save(user) }
    }

    expect:
    execHarness.yieldSingle { ❸
      userService.getUsers()
    }.value == users
  }
}
```

❶ The most notable portion of the test is the call here to `ExecHarness.harness()`. This is all the code necessary to bootstrap the underlying Ratpack-managed infrastructure for your test. From within our feature test, we can now leverage that infrastructure for working with the `save` and `getUsers` methods. (Note that

in the "`should save and return user list`" feature test, we are utilizing Spock's `given` and `expect` blocks this time to setup, stimulus, and assertion blocks. Please refer to the Spock documentation for more details on this.)

❷ Within the feature test, we are using the `yield` call on `execHarness` to wrap the calls to the `DefaultUserService`. This is necessary to support the underlying infrastructure for `Promise` types. The call to `yield` will subscribe to and capture the result of the produced promise; in the case of the call to `save`, we are leveraging the fact that `yield` will wait until the call is complete to move on to the `expect` block.

❸ In the case of the call to `getUsers`, we use the call to `yieldSingle` to capture the list of users. The corresponding call to `.value` captures the result's value explicitly, and we can assert the service returned the appropriate data set.

The key benefit to testing this way is that it is extremely fast to stand up the minimally required Ratpack underpinnings, without the need to start the application entirely. Even though the latter case is acceptably fast, the use of `ExecHarness` costs a negligible amount of time to bootstrap.

Unit Testing Standalone Handlers

As we discussed in Chapter 2, you can improve your application's testability by building standalone request handlers. Standalone handlers are able to be distinctly unit tested without the need for `EmbeddedApp` or `ExecHarness`. Using the `RequestFixture` test helper, you can write succinct test specifications that allow you to emulate requests and make assertions about your handler's responses.

Let's start by demonstrating a test against the simplest possible standalone request handler, as shown in Example 3-16.

Example 3-16. The simplest standalone request handler

```
package tld.company.app

import ratpack.handling.Handler
import ratpack.handling.Context

class AppHandler implements Handler {

  @Override
  void handle(Context ctx) {
    ctx.response.send("ok")
  }
}
```

As you can see, the handler simply sends an "ok" message back as the response. Using `RequestFixture`, we can activate this handler in a standalone manner and make an assertion that it is properly sending back the message. Example 3-17 demonstrates the use of `RequestFixture` for this purpose.

Example 3-17. The AppHandler test specification

```
package tld.company.app

import ratpack.test.handling.RequestFixture
import spock.lang.Specification

class AppHandlerSpec extends Specification {

  def handler = new AppHandler() ❶

  void "should render proper response"() {
    given:
    def result = RequestFixture.handle(handler) {} ❷

    expect:
    result.bodyText == "ok" ❸
  }
}
```

❶ Here we create a new instance of the `AppHandler` for use throughout the specification.

❷ Using the `ratpack.test.handling.RequestFixture` class, we call the `handle` method against the `AppHandler` instance. The trailing closure is provided an instance of `RequestFixture` for further customization of the request details.

❸ The `result` provides you with the details of the response, allowing you to make assertions about the data that will be sent back.

When the `RequestFixture#handle` method is called, a test `Context` is created and the normal request-response lifecycle is perfectly emulated. To better demonstrate the importance of this capability, let's look at a more complex standalone handler, as shown in Example 3-18.

Example 3-18. A complex standalone request handler for unit testing

```
package tld.company.app

import ratpack.handling.Handler
import ratpack.handling.Context
```

```
import static ratpack.jackson.Jackson.json

class ProductHandler implements Handler {

  @Override
  void handle(Context ctx) {
    ProductService productService = ctx.get(ProductService) ❶

    def id = ctx.pathTokens.asLong("id") ❷
    if (id != null) {
      productService.getProduct(id).then { product -> ❸
        if (product) {
          ctx.render(json(product)) ❹
        } else {
          ctx.response.status(404).send() ❺
        }
      }
    } else {
      ctx.response.status(400) ❻
      ctx.render(json([status: "error", message: "product id is required"])) ❼
    }
  }
}
```

❶ The `ProductHandler` starts its processing by getting a `ProductService` from the `Context` registry.

❷ Next, it looks at the `pathTokens` to get a `Long` value for the `id` token of the request path.

❸ If an `id` value is available, then the `getProduct` method is called.

❹ If a product is returned, we render it back as JSON.

❺ If no product is returned by the `productService`, then we send back a 404 (Not Found) status to the client.

❻ If no `id` value is available, then here we set the response status to 400 (Bad Request).

❼ Given the error, we send back a helpful message.

To fully illustrate this demonstration, imagine that the `ProductService` interface (and its corresponding `Product` model) is defined as shown in Example 3-19.

Example 3-19. The ProductService interface

```groovy
package tld.company.app

import groovy.transform.Immutable
import ratpack.exec.Promise

interface ProductService {

  Promise<Product> getProduct(Long id)

  @Immutable
  static class Product {
    Long id
    String name
    String description
    BigDecimal price
  }
}
```

From here, we can build a succinct specification with feature tests for all the capabilities of the `ProductHandler`. Example 3-20 shows the comprehensive unit test specification for the `ProductHandler`.

Example 3-20. The unit test specification for the ProductHandler

```groovy
package tld.company.app

import ratpack.exec.Promise
import ratpack.jackson.JsonRender
import ratpack.test.handling.RequestFixture
import spock.lang.Specification

class ProductHandlerSpec extends Specification {

  def handler = new ProductHandler()               ❶
  def productService = Mock(ProductService)        ❷
  def product = new ProductService.Product([       ❸
      id: 1,
      name: "Learning Ratpack",
      description: "Simple, lean, powerful web applications",
      price: 49.99
  ])
  def requestFixture = RequestFixture.requestFixture()  ❹
      .registry { r ->
        r.add(ProductService, productService)
      }

  void "should properly render valid product requests"() {  ❺
    when:
    "a valid product is requested"
```

```
    def result = requestFixture.pathBinding([id: product.id.toString()]) ❻
        .handle(handler)

    then:
    "it should be properly rendered back"
    1 * productService.getProduct(product.id) >> Promise.sync { product } ❼
    result.rendered(JsonRender).object == product ❽
}

void "should return not found for an invalid product"() { ❾
    when:
    "an invalid product is requested"
    def result = requestFixture
        .pathBinding([id: "0"]) ❿
        .handle(handler)

    then:
    "a 404 status should be sent back"
    1 * productService.getProduct(0) >> Promise.sync { null } ⓫
    result.status.code == 404 ⓬
}

void "should return bad request for an invalid request"() { ⓭
    when:
    "no id is specified"
    def result = requestFixture.handle(handler) ⓮

    then:
    "a 400 status should be sent back"
    0 * productService.getProduct(_) ⓯
    result.status.code == 400 ⓰
}
}
```

❶   Here we create an instance of the `ProductHandler`.

❷   As our handler uses the `ProductService` and we wish to make assertions about
its interactions, we provide a specification-level mock of the interface.

❸   We will also create a valid `Product` object for use in our feature tests.

❹   Also at the specification level, we create the `RequestFixture` for use within the
feature tests.

❺   Our specification's first feature unit test is ensuring that valid product requests
respond with the proper type.

❻   Within this feature test, we set the path binding for the `id` path token, as the han-
dler is expecting.

❼ Here we define an interaction on the `ProductService`. At this statement, we say that the controller should call `getProduct` one time, and the result will be a `Promise` of the `Product` type that we defined at the specification level.

❽ Using the `rendered` method on the result from the `RequestFixture`, we can assert that the object rendered from the handler is the same as our `Product`.

❾ Our next feature test is checking that a request for an invalid product `id` returns a 404 (Not Found) status code.

❿ We set the `id` path token to `0`.

⓫ Here we define the interaction that the call to `getProduct` for `id` `0` will return a `Promise` of a `null` value. Recall that the handler logic interprets a `null` value as a 404 result.

⓬ In this unit test, we finally assert that the handler did indeed return a 404 status code.

⓭ Our final feature test checks that the handler returns a 400 (Bad Request) code when there is no `id` path token available.

⓮ We call the `handle` method on `RequestFixture`, this time without any `pathBinding` values.

⓯ Here we assert that the `getProduct` method on the `ProductService` was never called.

⓰ Finally, we ensure the handler returned the 400 status code.

For unit testing, the `RequestFixture` test helper enables unparalleled coverage of your request handling code. The examples shown in this section should serve as a good introduction, and you can use them as a base to explore further. Indeed, `RequestFixture` provides all the faculties for adding headers, modifying and working with cookies, testing against different HTTP verbs, and so forth. In terms of unit testing, you will undoubtedly find the `RequestFixture` a valuable tool in your toolbelt.

Other Testing Scenarios

Ratpack's emphasis and focus on testing and testability scales even beyond Ratpack applications. Indeed, many of the test fixtures for Ratpack can be used to ameliorate test coverage of existing code in non-Ratpack applications. Specifically, the `ratpack-`

test module provides an `EmbeddedApp` test fixture, which can be employed outside of Ratpack projects to bring improved test coverage to existing code by leveraging the Ratpack application and testing infrastructure.

To understand this better, consider a scenario that is familiar to so many readers, where you have a legacy application that is tightly bound, does not make proper use of dependency injection, and lacks comprehensive (or any) test coverage. Your initial reaction is to completely rewrite and rearchitect the application in a proper, test-friendly way, but you understand the liability of doing so. Before knowing about Ratpack, you may have gone down the route of building extensive test suites and verbose fixtures to test even the simplest of operations in this application. Ratpack takes this otherwise arduous task and allows you to wrap the existing service implementations into a test-only Ratpack runtime, providing for you all the test fixtures that make testing Ratpack applications so easy and complete.

For demonstration's sake, assume that the legacy application you want to test is a servlet-based application that provides `doPost` and `doGet` implementations that store and retrieve `User` objects accordingly. In this case, consider that the original architects of your legacy application have tightly bound the storage of users to the servlet, making it extremely difficult to test. Example 3-21 shows roughly what this servlet implementation might look like.

Example 3-21. Legacy servlet implementation

```java
package tld.company.app;

import com.fasterxml.jackson.databind.ObjectMapper;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.BufferedReader;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class UserServlet extends HttpServlet {

  private static final String DB_HOST = System.getProperty("db.host");

  private Connection getConnection() throws SQLException {
    return DriverManager
      .getConnection("jdbc:mysql://" + DB_HOST + "/db", "root", "");
  }

  @Override
  public void doPost(HttpServletRequest request, HttpServletResponse response) {
    try {
```

```java
      StringBuffer jb = new StringBuffer();
      String line = null;
      BufferedReader reader = request.getReader();
      while ((line = reader.readLine()) != null) {
        jb.append(line);
      }

      ObjectMapper objectMapper = new ObjectMapper();
      User user = objectMapper.readValue(jb.toString(), User.class);

      Connection c = getConnection();
      PreparedStatement pstmt = c
        .prepareStatement("insert into users (username,email) values (?, ?)");

      pstmt.setString(1, user.getUsername());
      pstmt.setString(2, user.getEmail());
      pstmt.execute();
      pstmt.close();
      c.close();
      response.setStatus(200);
    } catch (Exception e) {
      try {
        response.sendError(500, e.getMessage());
      } catch (Exception e1) {
        throw new RuntimeException("error throwing error", e1);
      }
    }
  }
}

@Override
public void doGet(HttpServletRequest request, HttpServletResponse response) {
  try {
    Connection c = getConnection();
    Statement stmt = c.createStatement();
    ResultSet rs = stmt.executeQuery("select username,email from users");

    List<User> users = new ArrayList<>();
    while (rs.next()) {
      String username = rs.getString(1);
      String email = rs.getString(2);
      User user = new User();
      user.setUsername(username);
      user.setEmail(email);
      users.add(user);
    }
    rs.close();
    c.close();
    response.setStatus(200);
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.writeValue(response.getOutputStream(), users);
  } catch (Exception e) {
    try {
```

```
          response.sendError(500, e.getMessage());
      } catch (Exception e1) {
          throw new RuntimeException("error throwing error", e1);
      }
    }
  }
}
```

The legacy application in Example 3-21 does not offer any ability to test the servlet's interaction with the database on any level. Everything, right down to the SQL statements, is embedded in the respective doPost and doGet methods. This type of code is extremely fragile and difficult to refactor, because even the slightest alteration can result in a complete processing failure.

The challenge is to get some functional tests around this servlet code so that we can test a round trip of storage and subsequent retrieval. Before exploring how Ratpack can help test this type of code, it is important to understand that the use of servlet API code is simply for demonstrating what a legacy application might look like. There is no faculty in Ratpack for mapping servlets onto handlers, or for working with them in any practical way outside of an embedded test scenario. Indeed, the best we can offer in this scenario is a blocking call to the servlet from within a handler to emulate what an incoming request might look like.

The code in Example 3-22 demonstrates using the EmbeddedApp within a feature test to stand up the servlet within an embedded Ratpack application, and run the round-trip test on the code.

Example 3-22. Demonstrating EmbeddedApp

```
package tld.company.app

import groovy.json.JsonSlurper
import groovy.sql.Sql
import org.apache.catalina.ssi.ByteArrayServletOutputStream
import ratpack.func.Action
import ratpack.test.embed.EmbeddedApp
import spock.lang.Specification

import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse
import java.sql.DriverManager

import static groovy.json.JsonOutput.toJson

class UserServletFunctionalSpec extends Specification {
  private static final String DB_HOST = "localhost"
  private static final Integer DB_PORT = 3306
  private static final Sql sql =
      new Sql(
```

```
        DriverManager.getConnection(
          "jdbc:mysql://${DB_HOST}:${DB_PORT}/db",
          "root",
          ""
        )
      )

  static {
    System.setProperty("db.host", DB_HOST)
    System.setProperty("db.port", DB_PORT.toString())
  }

  UserServlet userServlet = new UserServlet()
  JsonSlurper jsonSlurper = new JsonSlurper()

  EmbeddedApp app = EmbeddedApp.of({ spec -> ❶
    spec.handlers { chain ->
      chain.all { ctx ->
        def servletRequest = Stub(HttpServletRequest) ❷
        def servletResponse = Stub(HttpServletResponse) ❸
        def outputStream = new ByteArrayServletOutputStream()
        def responseStatus = 0
        def bodyPromise = ctx.request.body.map { body ->
          servletRequest.getReader() >> {
            new BufferedReader(body.inputStream.newReader()) ❹
          }
        }
        servletResponse.setStatus(_) >> { int status -> ❺
          responseStatus = status
        }
        servletResponse.sendError(_, _) >> { int status, String msg -> ❻
          responseStatus = status
          outputStream.write(msg.bytes)
        }
        servletResponse.getOutputStream() >> outputStream
        ctx.byMethod { b ->
          b.post({
            bodyPromise.then {
              userServlet.doPost(servletRequest, servletResponse) ❼
              ctx.response.status(responseStatus) ❽
              ctx.response.send(outputStream.toByteArray()) ❾
            }
          })
          b.get({
            bodyPromise.then {
              userServlet.doGet(servletRequest, servletResponse) ❿
              ctx.response.status(responseStatus) ⓫
              ctx.response.send(outputStream.toByteArray()) ⓬
            }
          })
        }
      }
    }
```

```
      }
    } as Action)

    def setupSpec() { ⓭
      trunc()
    }

    def cleanupSpec() { ⓮
      trunc()
    }

    private static void trunc() {
      sql.execute("delete from users")
    }

    void "should save user data and list results"() { ⓯
      setup:
      def user = [username: "danveloper", email: "daniel.p.woods@gmail.com"]

      when:
      def postResponse = app.httpClient.requestSpec { spec ->
        spec.body { b ->
          b.text(toJson(user))
        }
      }.post()

      then:
      postResponse.statusCode == 200

      when:
      def getResponse = app.httpClient.get()

      then:
      getResponse.statusCode == 200
      (jsonSlurper.parseText(getResponse.body.text) as List) == [user]
    }
}
```

❶ The `EmbeddedApp` test fixture is initialized through the `EmbeddedApp.of()` assembly method, where we define the structure of the test application using the same handler chain API that we would see from a regular main class implementation.

❷ Here, we create a `Stub` for the request object. `Stub` types work similarly to `Mock` types, except that there are no assertions as to how many times an object was interacted with. However, we can continue to capture interactions to provide responses for method calls.

❸ Similarly, we create a `Stub` for the response.

❹   We capture an interaction on the request stub which is designed to construct a `BufferedReader` for use within the servlet. At this point in the data flow, we have access to Ratpack's `Context` object, so we can directly translate the request body onto the `BufferedReader`. Note that here we have to retrieve the request body from within a `Promise` type, since Ratpack lazily reads request bodies to improve performance.

❺   We capture an interaction on the response stub to store the status sent by the servlet into a local variable, which we later apply to the Ratpack response object.

❻   We capture an interaction on the response stub to store any error status and message sent by the servlet.

❼   Here, we begin the code necessary to translate the interaction of the servlet's `doPost` method. Within the handler's `post` logic, we call out to the `doPost` method on the servlet using the stubbed request and response objects.

❽   We translate the status, as captured from the response stub, and map it onto the Ratpack response.

❾   Here, we use the `send` method on `response` to send the data back that was captured from the servlet interactions.

❿   Within the handler's `get` logic, we call out to the `doGet` method on the servlet.

⓫   Here, we translate the status onto the Ratpack response.

⓬   Similar to the `post` logic, we send back the captured output.

⓭   Because the servlet did not give us much flexibility on the MySQL datasource connection, we must have a test MySQL instance set up and bound on localhost (more details on getting started with MySQL can be found in the MySQL documentation). This is a test database instance, so we can safely clean out the `users` table between test runs. Within the `setupSpec` method, we place a call to delete all the users from the database. This will ensure that we are working with a clean slate between runs.

⓮   Similarly, and for completeness, we also call the delete method from the `cleanup Spec` portion of the specification, which will be executed as the last phase of feature tests.

⓯   With the setup and adaptation out of the way, we are left only to write our round-trip feature test. The feature test shown here should look similar to the examples

shown previously in this section. The `EmbeddedApp` fixture offers a `TestHttp Client`, similar to how `ApplicationUnderTest` works. Using this mechanism we can write a feature test that writes some data through the servlet's `doPost` handler, and subsequently retrieve it from the `doGet` handler, all as though we were writing a functional feature test against an actual Ratpack application.

To make all of this work, from within the legacy application, we need only add the `testCompile` dependencies of `ratpack-test` and Spock, as shown earlier. This method of using Ratpack to assist in testing a legacy code base is an excellent way to get started with using Ratpack right away. If you work in an environment where you do not have the freedom to start a "greenfield" Ratpack project or you cannot immediately rewrite an existing legacy application in Ratpack, you can gain a plethora of experience with Ratpack by introducing tests that leverage the `EmbeddedApp` fixture into an existing code base.

Chapter Summary

Test-driven development is at the forefront of modern web development, and the fixtures that Ratpack provides for testing make it easier than ever to build well-tested web applications. As this chapter has shown you, from creating unit tests, to working with integration and functional testing constructs, Ratpack's charter as a developer-first framework could never be more present than in the conversation of testing. Furthermore, this chapter has shown you how Ratpack's test structures can be utilized in existing or legacy projects to bring higher levels of confidence to those projects. Indeed, its nature as a lightweight web framework makes it an excellent utility in your extended testing arsenal. As you move forward with learning Ratpack, you should keep the principles and concepts covered in this chapter close to mind in order to ensure that you are always building systems that account for testability from the start.

Application Configuration

Due to the trend of system architectures increasingly moving to cloud infrastructure, modern applications need the ability to incorporate configurations from one or many sources to accommodate the cloud's ephemeral nature. Ratpack provides a modern, easy-to-use system and semantic API for consuming and working with configuration details. Application and server configuration can be derived from files within the project, inside a library, or from the filesystem, system properties, and environment variables. In every respect, Ratpack empowers developers to follow the principles outlined by the Twelve Factor App. Regardless of your deployment infrastructure, Ratpack's configuration system is ready and able to support your application's needs.

The configuration system allows you to take configuration from one or many sources and map it to typed model objects. Those model objects are then able to be used throughout your application. The use of the `serverConfig` method on the application definition provides the means for specifying the sources to your configuration. Consider a Groovy application that derives its database connection information from configuration. Example 4-1 outlines an application with a `DatabaseConfig` class that holds the properties necessary for specifying the connection details.

Example 4-1. Application with DatabaseConfig

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class DatabaseConfig {  ❶
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}
```

```
ratpack {
  serverConfig { ❷
    json "dbconfig.json" ❸
    require("/database", DatabaseConfig) ❹
  }
  handlers {
    get("config") { DatabaseConfig config -> ❺
      render toJson(config)
    }
  }
}
```

❶ The `DatabaseConfig` class properties are set with reasonable defaults. These will
    be overridden with explicitly defined values during the configuration mapping.

❷ Here, we introduce the use of the `serverConfig` block, wherein we can specify
    sources of configuration and map the configuration onto our configuration
    model.

❸ The semantic API in the `serverConfig` block gives us appropriately named
    methods for consuming configuration in different formats (in this case, JSON).

❹ After we specify the source of the configuration, we use the `require` method to
    map the */database* *configuration path* to our model object. After being hydrated
    by the configuration system, the `DatabaseConfig` class will become usable
    throughout our application.

❺ To demonstrate configuration, we provide a `get` endpoint on */config* that gets a
    hold of the mapped `DatabaseConfig` object, and we render it back to the caller.

Remembering the simplest project structure for a Ratpack Groovy application, if we
place this example into the *src/ratpack/Ratpack.groovy* file, run the project, and access
the */config* endpoint (Figure 4-1), we will find that we are met with a block of JSON
that corresponds perfectly to the default properties on the `DatabaseConfig` class.
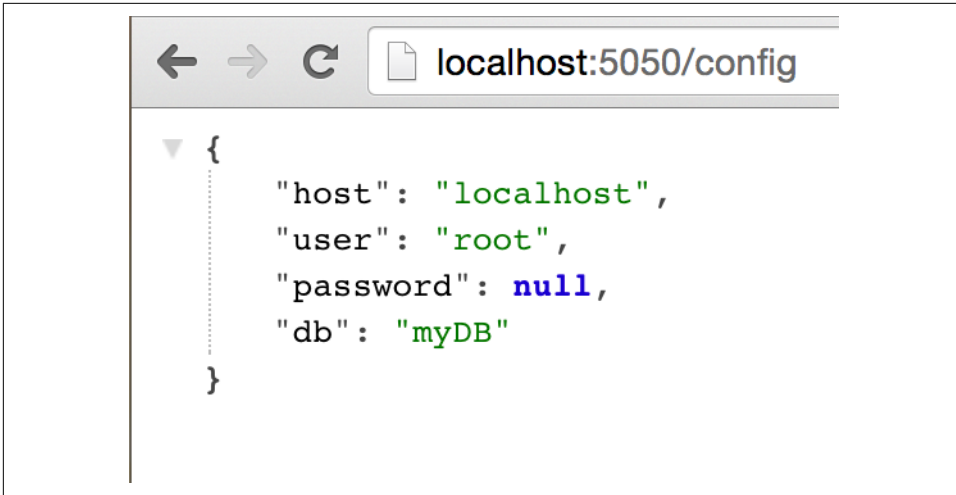
Figure 4-1. DatabaseConfig JSON output

Next, let's take a look at adding the *dbconfig.json* file into the project. Consider the project structure shown in Example 4-2.

Example 4-2. Project structure

```
.
├── build.gradle
└── src
    └── ratpack
        ├── dbconfig.json
        └── Ratpack.groovy
```

Because our application specifies that we want to map the `/database` configuration path to our model, we must ensure that the directives in the *dbconfig.json* file are specified under that structure. The text shown in Example 4-3 shows the form of this file.

Example 4-3. The dbconfig.json file

```
{
  "database": {  ❶
    "host": "mysql001.dev.company.com",
    "user": "ratpack",
    "password": "l3arn!ngR@tpack"
  }
}
```

❶ Note here that we are structuring the properties that will be mapped onto the `DatabaseConfig` class under the `database` key, as the application is expecting.

Reloading configuration takes place during the server initialization, so you will want to restart the process to get the configuration from *dbconfig.json* properly initialized. Once the application is restarted, if we again load the `/config` endpoint (Figure 4-2), we will now see that the `DatabaseConfig` model is properly hydrated.

Figure 4-2. Configured DatabaseConfig JSON

At this point, it is important to note that Ratpack takes a highly cautious approach to loading files. When you specify a string referencing a file to the configuration system, it resolves that file relative to the project's base directory. In this case, the base direc-tory is defined as the *src/ratpack* root (where the *Ratpack.groovy* script is homed). Even if we were to change the string path to */etc/dbconfig.json*, it would attempt to resolve that file from *src/ratpack/etc/dbconfig.json*. This is done as a means of protec-tion and security to remove the possibility of files being unsafely or erroneously loaded from outside the project.

When you are building an application for deployment, however, it may be your inten-tion to load files from the filesystem, so in that case you must explicitly specify that you wish to do so. This is accomplished by specifying a `java.nio.file.Path` refer-ence instead of a string. We can reimagine our example, this time loading the *dbcon-fig.json* file from the */etc* directory instead, as shown in Example 4-4.

Example 4-4. Loading dbconfig.json from outside the project

```
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson
```

```
import java.nio.file.Paths

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json Paths.get("/etc/dbconfig.json")  ❶
    require("/database", DatabaseConfig)
  }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶ Here, we use the `Paths` helper class to reference our configuration file from the filesystem path */etc/dbconfig.json*.

We are not restricted to loading configuration from within the project or off the filesystem. As noted in the introductory section, we can also load configuration files from libraries or other classpath dependencies. To load configuration from the classpath, we must provide a `java.net.URL` reference to the file we wish to load. For this, we can make use of the `Class#getResource` method to get a handle on our config. To illustrate this, if we were to say that our *dbconfig.json* file comes from the */config* folder of a JAR file that is on the project's classpath, we can load it as shown in Example 4-5.

Example 4-5. Loading dbconfig.json from classpath

```
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json Class.getResource("/config/dbconfig.json")  ❶
    require("/database", DatabaseConfig)
```

```
    }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶ We are now calling the `Class#getResource` method. Note that we can shorten this to `getResource`, as Groovy scripts implicitly provide us with the static methods on `Class`.

Ratpack's configuration system also acts with the ability to overlay configurations, where later references win the mapping. We may have the requirement to load the *dbconfig.json* file from our project, then from the classpath, and finally from the filesystem. In doing so, we can allow more specific configurations to take precedence, which would thereby accommodate different deployment environments. It is intuitive to follow the calls that perform the loading of these configuration files, as shown in Example 4-6.

Example 4-6. Overlaying dbconfig.json configurations

```
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

import java.nio.file.Paths

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json "dbconfig.json" ❶
    json Class.getResource("/config/dbconfig.json") ❷
    json Paths.get("/etc/dbconfig.json") ❸
    require("/database", DatabaseConfig)
  }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶ We start by loading the *dbconfig.json* from our project, which lives in the *src/ratpack* directory.

❷ Then we can overlay the classpath configuration.

❸ And finally, we incorporate configuration that comes from the */etc/dbconfig.json* file.

Inspecting what each of the respective *dbconfig.json* files look like will help paint the picture of how overlaying works. We start with the *dbconfig.json* file we worked with earlier in Example 4-3 and overlay the */config/dbconfig.json* from the classpath; this file may have contents like those shown in Example 4-7.

Example 4-7. Classpath dbconfig.json contents

```
{
  "database": {
    "user": "app-user"  ❶
  }
}
```

❶ The only value the classpath's *dbconfig.json* file provides is the user property.

The user property from the classpath takes precedence over the same value from our project's *dbconfig.json* file. At this point, the configuration model looks like the JSON depicted in Example 4-8.

Example 4-8. Project and classpath merged dbconfig.json

```
{
  "database": {
    "host": "mysql001.dev.company.com",
    "user": "app-user",  ❶
    "password": "l3arn!ngR@tpack"
  }
}
```

❶ Here, app-user is favored over the previous value of ratpack.

Next, we bring in the */etc/dbconfig.json* from the filesystem. Let's consider that its contents look like those in Example 4-9.

Example 4-9. Filesystem dbconfig.json

```
{
  "database": {
    "host": "ratpack-mysql-0441.prod.internal",  ❶
```

```
      "password": "kcapt@Rgn!nra3l" ❷
  }
}
```

❶   The filesystem *dbconfig.json* overrides the database hostname.

❷   It also specifies a new password for the database user.

Now the three *dbconfig.json* configurations have been merged, and we end with the configuration representation that is depicted in Example 4-10.

Example 4-10. Fully merged dbconfig.json

```
{
  "database": {
    "host": "ratpack-mysql-0441.prod.internal", ❶
    "user": "app-user", ❷
    "password": "kcapt@Rgn!nra3l" ❸
  }
}
```

❶   The host value was last specified in the */etc/dbconfig.json* file, so it wins in the final representation.

❷   The user value was captured from the classpath's */config/dbconfig.json*, so it is favored here.

❸   password also came in from the filesystem, though if it had not been specified, the password field from our project's *dbconfig.json* would be shown here.

Ratpack's ability to overlay configuration sources means that your application can support a robust runtime environment, where configuration directives are able to be specified at different layers of responsibility, and merged into a cohesive representation.

Configuration files in JSON format are not the only possibility. There are matching methods on serverConfig for working with YAML and Java properties files. Furthermore, these different file types can be used together when building the final configuration representation. If the application's requirement was to begin by loading the *dbconfig.json* file from within the project, followed by loading a *dbconfig.yml* file from the */etc* directory of the filesystem, we could realize an application structure like the one shown in Example 4-11.

Example 4-11. Mixing JSON and YAML configuration files

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

import java.nio.file.Paths

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json "dbconfig.json"   ❶
    yaml Paths.get("/etc/dbconfig.yml")   ❷
    require("/database", DatabaseConfig)
  }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶ Here, we load the *dbconfig.json* file from our project.

❷ Now we use the `yaml` method to load the */etc/dbconfig.yml* file.

If the */etc/dbconfig.yml* file has contents like those depicted in Example 4-12, then we can begin to realize how the merged configuration will be represented.

Example 4-12. The dbconfig.yml file

```yaml
database:
  host: ratpack-mysql-0441.prod.internal
```

Because the */etc/dbconfig.yml* file was loaded last, its properties get the highest order of precedence, meaning that the fully merged configuration will look like the JSON structure shown in Example 4-13.

Example 4-13. Merged config with JSON and YAML

```json
{
  "database": {
    "host": "ratpack-mysql-0441.prod.internal",   ❶
    "user": "ratpack",
    "password": "l3arn!ngR@tpack"
```

```
      }
}
```

❶   As you can see, the `host` is favored from the YAML file, while all other properties
     are inherited from the project's *dbconfig.yml*.

Java properties files are equally as easy to incorporate with the configuration system.
If we expand the example (Example 4-14), we might say that now our requirements
are that *dbconfig.json* is first loaded from the project, next a */config/dbconfig.proper-
ties* file is loaded from the classpath, and finally */etc/dbconfig.yml* is loaded from the
filesystem. When loading Java properties files, we make use of the `props` method in
the `serverConfig` block.

Example 4-14. Overlaying JSON, properties, and YAML files

```
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

import java.nio.file.Paths

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json "dbconfig.json" ❶
    props Class.getResource("/config/dbconfig.properties") ❷
    yaml Paths.get("/etc/dbconfig.yml") ❸
    require("/database", DatabaseConfig)
  }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶   Again, the project's *dbconfig.json* file is loaded.

❷   Next, the */config/dbconfig.properties* file is loaded from the classpath.

❸   And finally, the */etc/dbconfig.yml* file is loaded from the filesystem.

The classpath resource file */config/dbconfig.properties* may specify contents like those shown in Example 4-15.

Example 4-15. The dbconfig.properties file

```
database.user=app-user ❶
```

❶ Here, we specify the configuration path, `/database`, using dot-notation, and we override the `user` value for the configuration.

Given this additional configuration source, the final configuration will look like the one shown in Example 4-16.

Example 4-16. Fully merged configuration from JSON, properties, and YAML files

```
{
  "database": {
    "host": "ratpack-mysql-0441.prod.internal", ❶
    "user": "app-user", ❷
    "password": "l3arn!ngR@tpack" ❸
  }
}
```

❶ The `host` value is brought in from the */etc/dbconfig.yml* file.

❷ `user` is specified in the */config/dbconfig.properties* classpath file.

❸ And the `password` value comes from our project's *dbconfig.json*.

Configuring with Environment Variables and System Properties

Modern Java web applications—especially those designed to run on cloud infrastructure—need the ability to derive configuration from the system environment, in the form of environment variables and Java system properties. In support of that effort, Ratpack provides conventions by which directives can be specified by those sources, and incorporated into your application. If we revisit the example application, we can see how easy it is to allow these sources to participate in the configuration system. The code in Example 4-17 shows the use of the `env` and `sysProps` methods in the `serverConfig` block.

Example 4-17. Environment variables and system properties configuration

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

ratpack {
  serverConfig {
    json "dbconfig.json"
    env() ❶
    sysProps() ❷
    require("/database", DatabaseConfig)
  }
  handlers {
    get("config") { DatabaseConfig config ->
      render toJson(config)
    }
  }
}
```

❶ This method call is all that is necessary to include configuration from environment variables.

❷ It is similarly easy to include configuration from system properties.

When including configuration from these sources, it is important that you follow the key naming conventions when defining directives. By default, environment variables that are prefixed with RATPACK_ will be captured as eligible configurations. Likewise, when specifying Java system properties, the property needs to be prefixed with `rat pack.` to be included. Prefixes to both of these methods can be overridden according to your application's requirements by supplying the prefix string to the respective methods.

Configuring with Environment Variables

To better illustrate defining configuration with environment variables, consider a scenario where we want to specify the `host` property of the `DatabaseConfig` in the form of an environment variable. On our system, the `RATPACK_DATABASE__HOST` environmental variable must be exported prior to running the application. The console output in Example 4-18 shows how we go about doing this.

Example 4-18. Setting environment variables for the project

```
$ export RATPACK_DATABASE__HOST=ratpack-mysql001.prod.internal ❶
$ ./gradlew run ❷
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:configureRun
:run
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
WARNING: No slf4j logging binding found for Ratpack, there will be no logging output.
WARNING: Please add an slf4j binding, such as slf4j-log4j2, to the classpath.
WARNING: More info may be found here: http://ratpack.io/manual/current/logging.html
Ratpack started (development) for http://localhost:5050
> Building 83% > :run
```

❶ Here, we issue the `export` command with our environment variable.

❷ Next, we run the project, and the variable will be available to our application's process.

If we again access our `/config` handler from a browser (Figure 4-3), we will now see that the database host specified by our environment variable is favored over the one specified in our project's *dbconfig.json* file.

Figure 4-3. Configuration JSON with environment config

An important caveat with environment variable configuration is that depth must be denoted by a double underscore. Note in the environment variable's key, `RATPACK_DATABASE__HOST`, between the `DATABASE` and `HOST` parts, there are two underscores, which translates to the equivalent property reference of `database.host`.

Most modern cloud platforms provide references to databases and even passwords in the form of system environment variables. Furthermore, with the rising use of containerized runtimes, like those built with Docker, injecting configuration through environment variables is quickly becoming the preferred mechanism for deployments. It is a good idea to start building your application with the mindset that at some point you will need the ability to pull configuration from environment variables. Even if you do not immediately need it, it is inconsequential to application startup performance to simply add the `env()` method to your `serverConfig` block.

Configuration with System Properties

Before we demonstrate using Java system properties, we must first make a small change to our project's *build.gradle* file. Gradle does not implicitly pass system properties on to the `run` task, so we must tell it that we want those properties passed on to our application. We can make this change by configuring the `run` task in our build script, as shown in Example 4-19.

Example 4-19. Configuring Gradle run task

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

run {
  systemProperties System.getProperties() ❶
}
```

❶ Within the closure that configured the `run` task, we simply pass all of the system properties from the Gradle process on to our application.

Let's again demonstrate overriding the database host field, this time with system properties. The console output in Example 4-20 shows the passing of the rat pack.database.host value on to the run task, and thus our application.

Example 4-20. Setting system property configuration

```
$ ./gradlew -Dratpack.database.host=mysql004.dev.company.com run ❶
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources
:classes
:configureRun
:run
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
WARNING: No slf4j logging binding found for Ratpack, there will be no logging output.
WARNING: Please add an slf4j binding, such as slf4j-log4j2, to the classpath.
Ratpack started (development) for http://localhost:5050
WARNING: More info may be found here: http://ratpack.io/manual/current/logging.html
> Building 83% > :run
```

❶ Here, we use the `-Dratpack.database.host=mysql004.dev.company.com` argument to set the database host value.

Again accessing our application's /config endpoint (Figure 4-4) will now show the database host from the system property.

Figure 4-4. Configuration JSON from system properties

Like environment variables, system properties can act as a way to configure your application according to the system environment. To that end, it is also advisable to

start out building your application with the intent to support configuration from system properties.

Nested Configuration Models

It may be desirable for your application's configuration model to be a complex structure with nested object types. If you consider the application depicted in Example 4-21, you can see that the `ApplicationConfig` class holds properties for the `DatabaseConfig` type that we used earlier, and now also has a `LandingPageConfig` type as well.

Example 4-21. Mapping configuration to nested models

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

class LandingPageConfig {
  String welcomeMessage = "Welcome to Ratpack!"
  String analyticsKey = "ua-12345678"
}

class ApplicationConfig {  ❶
  DatabaseConfig database = new DatabaseConfig()
  LandingPageConfig landing = new LandingPageConfig()
}

ratpack {
  serverConfig {
    json "dbconfig.json"
    env()
    sysProps()
    require("", ApplicationConfig)  ❷
  }
  handlers {
    get("config") { ApplicationConfig config ->
      render toJson(config)
    }
  }
}
```

❶ Here, we specify the `ApplicationConfig` class with `database` and `landing` fields that correspond to the `DatabaseConfig` and `LandingPageConfig` classes, respectively.

❷ Because we want to map all configuration onto the top-level `ApplicationConfig` class, we specify an empty string as the configuration path here.

Without making any changes to our project's *dbconfig.json* file from earlier, if we start this application, we will find that the file's configuration is properly mapped to the `DatabaseConfig` within the `ApplicationConfig` (see Figure 4-5).


```
localhost:5050/config

{
    "landing": {
        "analyticsKey": "ua-12345678",
        "welcomeMessage": "Welcome to Ratpack!"
    },
    "database": {
        "password": "l3arn!ngR@tpack",
        "user": "ratpack",
        "db": "myDB",
        "host": "mysql001.dev.company.com"
    }
}
```
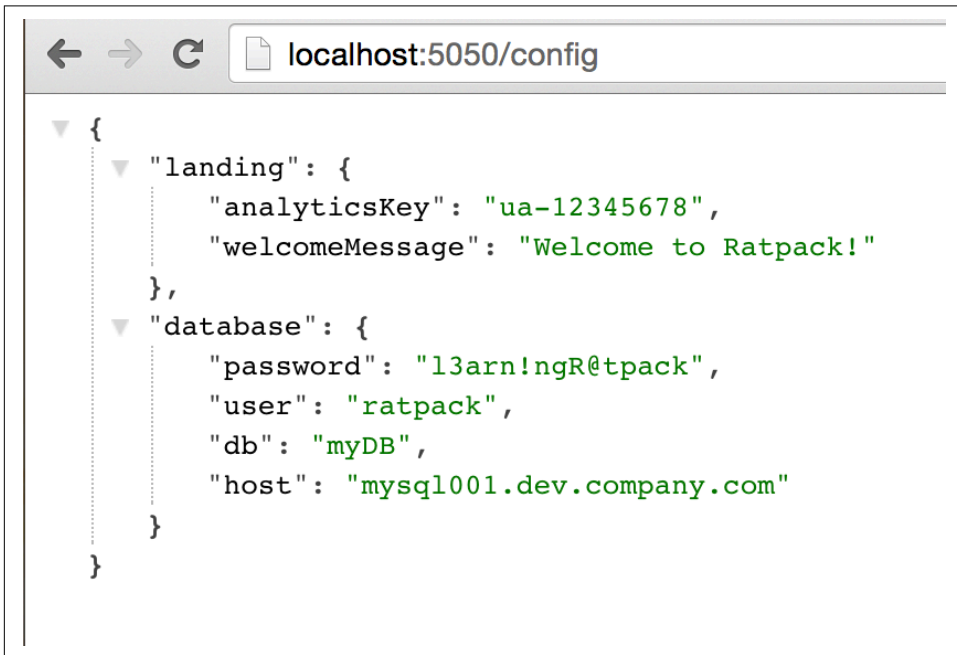
Figure 4-5. ApplicationConfig JSON output

If we add a new configuration file to our project for the `LandingPageConfig`, then we can demonstrate instituting configuration for it as well. The YAML configuration shown in Example 4-22 shows explicit configuration for the `LandingPageConfig` properties.

Example 4-22. The landingpage.yml file

```
landing:
  analyticsKey: "ua-abcdefgh" ❶
```

❶ Here, we will use this opportunity to override the default `analyticsKey` configuration value.

We place this file into our *src/ratpack* directory and modify our *Ratpack.groovy* to include this configuration, as shown in Example 4-23.

Example 4-23. Application with landingpage.yml config included

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

class DatabaseConfig {
  String host = "localhost"
  String user = "root"
  String password
  String db = "myDB"
}

class LandingPageConfig {
  String welcomeMessage = "Welcome to Ratpack!"
  String analyticsKey = "ua-12345678"
}

class ApplicationConfig {
  DatabaseConfig database
  LandingPageConfig landing
}

ratpack {
  serverConfig {
    json "dbconfig.json"
    yaml "landingpage.yml"  ❶
    env()
    sysProps()
    require("", ApplicationConfig)
  }
  handlers {
    get("config") { ApplicationConfig config ->
      render toJson(config)
    }
  }
}
```

❶ We incorporate the *landingpage.yml* configuration, which has the configured value for the `analyticsKey` property.

If we run this application, we see that the configuration (Figure 4-6) from *landingpage.yml* was indeed properly mapped onto our configuration.
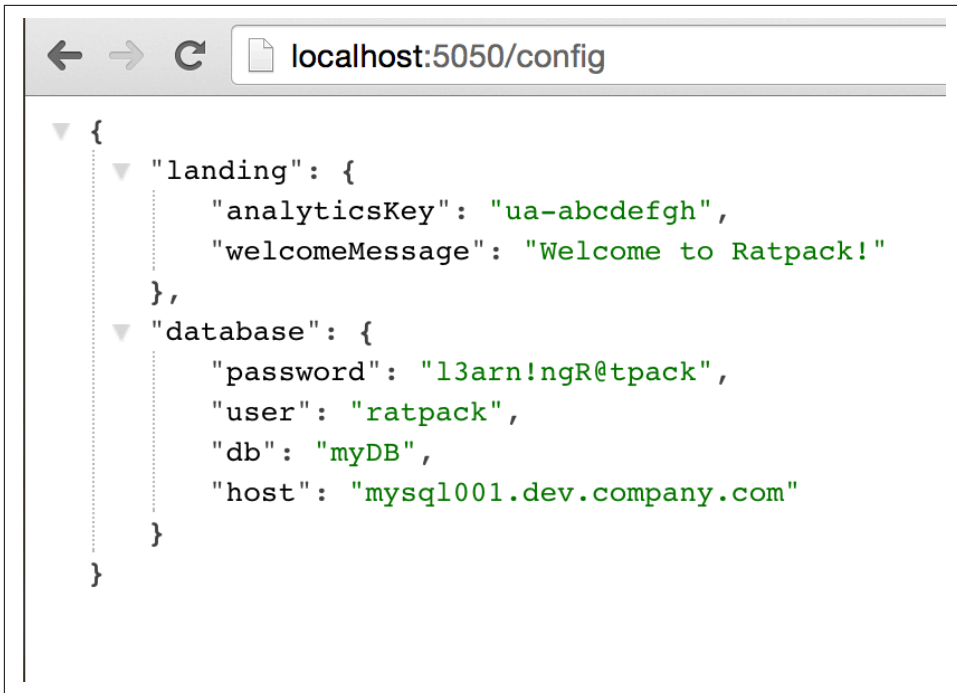
Figure 4-6. ApplicationConfig JSON output

An important thing to make note of at this point is that within both the *dbconfig.json* and *landingpage.yml* files, the configuration directives were placed within the objects `database` and `landing`, respectively. The names of these values matter, as they correspond to the field names of the corresponding model on the `ApplicationConfig` class. That is to say that if within the *landingpage.yml* file we changed the `landing` key to something different, the configuration system would not know how to map that onto the `ApplicationConfig` model.

Nesting configuration models does not stop at just a single depth. Configuration models of varying complexity and nested object depth can be described in declarative configurations and mapped to the corresponding object structure in your application.

Custom Configuration Source

Ratpack's configuration system is designed to be completely extensible. While most use cases are satisfied with its helper methods for loading JSON, YAML, Java properties, environment variables, and Java system properties, there still may be a time when your application needs to load its configuration from a nonstandard source. For example, your project's requirements may dictate that a centralized configuration server needs to be used to hydrate the application configuration. Or, you may just

want your application to have more granular control over how configuration is loaded into the system.

To facilitate custom configuration sources, Ratpack provides the `ratpack.con fig.ConfigSource` interface, which you can apply to the configuration system via the `add` method on the `serverConfig` block. Let's demonstrate this with an adaptation to the prior examples that introduces a `CustomConfigSource` that is responsible for explicitly setting the `host` and `analyticsKey` properties on the `DatabaseConfig` and `LandingPageConfig` classes. Consider the code shown in Example 4-24 to get an understanding of how we accomplish this.

Example 4-24. Customizing configuration mapping

```
package app

import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.node.ObjectNode
import ratpack.config.ConfigSource
import ratpack.file.FileSystemBinding

class CustomConfigSource implements ConfigSource { ❶

  @Override
  ObjectNode loadConfigData(ObjectMapper objectMapper,
                            FileSystemBinding fileSystemBinding)
                            throws Exception { ❷
    ObjectNode node = objectMapper.createObjectNode() ❸
    DatabaseConfig databaseConfig = new DatabaseConfig( ❹
      host: "my-database-host.dev.company.com"
    )
    LandingPageConfig landingPageConfig = new LandingPageConfig( ❺
      analyticsKey: "ua-learningratpack"
    )

    node.set("database", objectMapper.valueToTree(databaseConfig)) ❻
    node.set("landing", objectMapper.valueToTree(landingPageConfig)) ❼

    node ❽
  }
}
```

❶ Our class must implement the `ratpack.config.ConfigSource` interface.

❷ This interface defines a `loadConfigData` method, which provides us a Jackson `ObjectMapper` and the `FileSystemBinding` that corresponds to the project's base directory. Behind the scenes of Ratpack's configuration system, Jackson is utilized to perform the mapping of configuration data. From the `loadConfigData`

method, we must return a Jackson `ObjectNode`, which holds the property mappings and will be incorporated into the configuration loading process.

❸ We use the `ObjectMapper` to create the `ObjectNode` that we will return.

❹ Here, we demonstrate constructing the `DatabaseConfig` object and explicitly overriding its `host` field.

❺ We do the same here for the `analyticsKey` on the `LandingPageConfig` object.

❻ Here we set the `database` value on the `ObjectNode` to a tree of our `DatabaseConfig` object.

❼ We do the same here for the `landing` value and the `LandingPageConfig`.

❽ Finally, we return the fully hydrated `ObjectNode`.

This example is intentionally contrived for the sake of simplicity and to demonstrate capabilities. Real-world implementations will need to do more intelligent work, but this demonstration should serve as a good foundation of understanding.

We need to modify our `Ratpack.groovy` class to incorporate the `CustomConfig Source` into the configuration system. Before we do that, however, let's take a quick look at the project's new structure. For the purposes of this demonstration, the configuration classes have been moved out of the *Ratpack.groovy* file and into the project's source tree structure. There is also where we will find the `CustomConfigSource`. The project structure is depicted in Example 4-25.

Example 4-25. Revised project structure for CustomConfigSource

```
.
├── build.gradle
└── src
    ├── main
    │   └── groovy
    │       └── app
    │           ├── ApplicationConfig.groovy
    │           ├── CustomConfigSource.groovy
    │           ├── DatabaseConfig.groovy
    │           └── LandingPageConfig.groovy
    └── ratpack
        ├── dbconfig.json
        ├── landingpage.yml
        └── Ratpack.groovy
```

Next, see how the updated *Ratpack.groovy* script now looks, with the addition of the `CustomConfigSource` and accounting for the use of the project structure, as shown in Example 4-26.

Example 4-26. Updated Ratpack.groovy file

```groovy
import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

import app.* ❶

ratpack {
  serverConfig {
    json "dbconfig.json"
    yaml "landingpage.yml"
    add new CustomConfigSource() ❷
    env()
    sysProps()
    require("", ApplicationConfig)
  }
  handlers {
    get("config") { ApplicationConfig config ->
      render toJson(config)
    }
  }
}
```

❶ Our configuration models and the `CustomConfigSource` now live in the `app` package, so we import those resources here.

❷ Here, we add the `CustomConfigSource` into the mix. You will realize from what you know by now that the `host` and `analyticsKey` values will be favored from those we set in the `CustomConfigSource`, as it is further down the chain than our JSON and YAML configuration files.

If we run this application and again hit the `/config` endpoint, we will see that indeed the configuration (see Figure 4-7) specified by the `CustomConfigSource` is applied to the rendered `ApplicationConfig` model.
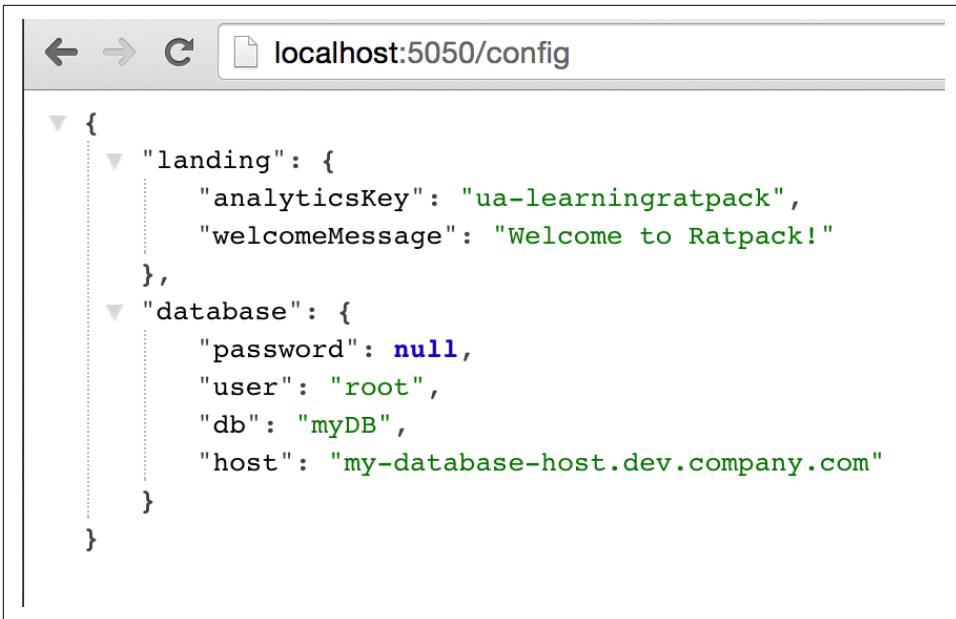
Figure 4-7. ApplicationConfig JSON output

Being able to extend the configuration system for your application's specific needs serves as another example of how Ratpack helps guide you when needing to do things that are beyond the framework's core capability.

Setting Server Configuration

Ratpack's underlying web server has a corresponding configuration model object in the `ratpack.server.ServerConfig` object. Ratpack will automatically map Java system properties and environment variables over top of the default configurations. To that extent, you can override server-level configuration, like the server's port, by providing the corresponding runtime values. Table 4-1 shows a description of `ServerConfig` fields and their corresponding Java system property and environment variable keys to override the default values.

Table 4-1. Table of ServerConfig properties

Field	Description	System Property	Environment Variable
port	The port that the Ratpack server binds to. Default: 5050. Example overrides to port 8080	-Dratpack.port=8080	RATPACK_PORT=8080
development	Whether Ratpack runs in development mode or not. Default: true. Example overrides to false	-Dratpack.development=false	RATPACK_DEVELOPMENT=false
threads	The number of request-taking threads that Ratpack uses. Default: 2 * number of CPUs available to the Java runtime. Example overrides to a static 4	-Dratpack.threads=4	RATPACK_THREADS=4
maxContentLength	The max number of bytes a request body can be. Default: 1048576 (1MB). Example overrides to 2MB	-Dratpack.maxContentLength=2097152	RATPACK_MAX_CONTENT_LENGTH=2097152
connectTimeoutMillis	The connection timeout of a request in milliseconds. Default: indefinite. Example overrides to 1 second	-Dratpack.connectTimeoutMillis=1000	RATPACK_CONNECT_TIMEOUT_MILLIS=1000
maxMessagesPerRead	The maximum number of bytes per read, as defined by the SO_RCVBUF [a] socket options. Default is unlimited. Example overrides to 1KB	-Dratpack.maxMessagesPerRead=1000	RATPACK_MAX_MESSAGES_PER_READ=1000
writeSpinCount	The maximum number of loops before a write operation returns zero. Default is writing until all the bytes have been written. Example overrides to 100	-DwriteSpinCount=100	RATPACK_WRITE_SPIN_COUNT=1000
requireClientSslAuth	Specifies whether to require HTTP client SSL authorization. An SSLContext must be specified. Default: false. Example overrides to true	-DrequireClientSslAuth=true	RATPACK_REQUIRE_CLIENT_SSL_AUTH=true

[a] *http://bit.ly/so_rcvbuf*

Many of the configuration values specified by the `ServerConfig` object can be tuned to improve performance depending on the runtime environment. The most commonly modified properties are the `port`, `development`, and `threads` values. In a cloud-based runtime environment, like Heroku, the application's server port may need to be mapped according to some value provided by the system environment. In a production environment, it is valuable to set the `development` property to a `false` value. The `threads` server config value can be tuned according to a server's environment to provide better performance. Some virtualization infrastructures, for example, will advertise more or fewer CPUs than actually exist on the host, so this property can be tuned to align an application more appropriately with the underlying system.

Chapter Summary

This chapter has exposed you to Ratpack's concise fixtures for working with configuration. From project and library to external and environment configuration sources, all the way through to demonstrating a custom `ConfigSource` implementation, and tuning framework-level settings, you now have a comprehensive understanding of how Ratpack's configuration system works and how you can leverage it in your projects.

Ratpack Modules

Ratpack's framework features are separated into different modules that can be made available to an application by bringing a module of interest onto a project's classpath. This allows the framework to be feature rich, while remaining lightweight and resource efficient. For instance, for applications that do not need support for authentication or sessions, those supported modules need not be included in the project.

It is important to note that modules differ slightly from the plugins you find in other popular web application frameworks. In those frameworks, there is often an internal plugin manager that framework code will reach out to in order to decorate an application with some functionality. Ratpack has no plugin infrastructure, but instead allows modules to affect an application's functionality by providing the framework with classes that serve to decorate or extend core functionality.

The component model for modularity and extensibility has added benefits in that functionality is provided through the exact same mechanism through which an application's components are provided. This gives more granular control to application developers in the case where they need to adapt a framework feature to their specific use case. Ratpack takes it a step further by giving control to the application in ascertaining what component is resolved and when. In most cases, you will not need to adjust a module's behavior, but in a scenario where you want to override the specific opinions of a feature's functionality, the ability to do so is there.

Before we can jump into a discussion about the specific modules that Ratpack offers and how they are used, we must first explore the infrastructure that supports its modularity: its registries.

Extending Ratpack with Registries

Registries in Ratpack are a central and intricate concept, and are leveraged in nearly every aspect of its function. From the startup sequence to the request flow, all the way into application code, registries are an ever-present thing. They need to be, because it is through the registry that Ratpack gets access to the services and support objects that aid in its operation.

In its essence, a registry is a fairly mundane object type, resembling the function of a Map, but it is through its usage that the framework-level concept becomes complex. Registries within Ratpack are a means for facilitating extensibility and flexibility. They provide a contract that can be utilized by applications to emulate a sort of abstract dependency injection mechanism. On that same token, their extensibility can be leveraged, as it is with the ratpack-guice and ratpack-spring-boot modules, to incorporate advanced DI frameworks into your application.

Registries are able to be layered, or "joined," to provide a flexibility that allows for a child-to-parent resolution process for components. As registries are built, they can be overlayed onto another registry to provide a more appropriate component binding for the needs of your application. They provide a boundary for registering different class implementations within an isolated scope of your application. In contrast, registries can be leveraged within your application to facilitate data flow between boundaries that would not normally be able to intercommunicate.

Whether or not it is apparent at first, the very first thing that most Ratpack applications will do is interact with a registry to provide a binding for their service layer. This happens within the application definition and is provided through a concise API that allows you to specify the service and support class implementations that you will use throughout your application. Consider the code in Example 5-1, which shows a Java-based main class that binds a UserService implementation for use within a handler.

Example 5-1. Main class with UserService binding

```
package app;

import ratpack.server.RatpackServer;

public class Main {

  public static void main(String[] args) throws Exception {
    RatpackServer.start( spec -> spec
        .registryOf(r -> r ❶
          .add(UserService.class, new DefaultUserService()) ❷
        )
        .handlers(chain -> chain
```

```
              .get(ctx -> {
                UserService userService = ctx.get(UserService.class); ❸
                userService.list().then(users -> {
                  StringBuilder sb = new StringBuilder();
                  sb.append('[');
                  for (User user : users) {
                    sb.append(jsonify(user));
                  }
                  sb.append(']');
                  ctx.getResponse().contentType("application/json");
                  ctx.render(sb.toString());
                });
              })
          )
      );
    }

    private static String jsonify(User user) {
      return "{ \"username\": \""
          +user.getUsername()+"\", \"email\": \""
          +user.getEmail()+"\" }";
    }
}
```

❶   Within the application definition, we can build out the *user registry* by making use of the `registryOf` mechanism. Within this configuration call, we can now provide a binding for later user in the application.

❷   Here, we create a binding to the `UserService` interface with a new instance of the `DefaultUserService` class.

❸   When a request comes into our handler logic, we can retrieve the binding using the `ctx.get(UserService.class)` call.

In this example, a sort of primitive dependency injection capability is apparent, because we are able to resolve the `UserService` from the `Context` object that was provided to the handler. It is important to understand that Ratpack takes no opinion as to how your application employs dependency injection. Indeed, the entire core of the framework utilizes registries in its own function, so there is no base-level requirement for a DI framework. Ratpack applications can be built completely and entirely without the integration of external DI frameworks.

Robust dependency injection through third-party libraries, such as Guice and Spring, is well supported. It is entirely up to the application developer to ascertain whether her application has the requirement for the kind of mature DI support provided by those libraries. In the case where it is a requirement, Ratpack's registry serves as a key integration interface, allowing registry implementations to be backed into those DI frameworks. This gives a bridge from the DI framework to Ratpack, and allows you

to write your handler code through the registry's contract, while still employing your DI framework of choice behind the scenes.

Use of registries in this manner opens the door for a lot of possibilities and complex requirements. As noted earlier, a registry acts as a boundary for component resolution, so it is practical to have a Ratpack application that leverages multiple DI frameworks behind the scenes. Indeed, both Guice and Spring can be leveraged within the same Ratpack application, affording you the opportunity to interface with a broad range of ecosystems that leverage those frameworks.

The registry defined during the application definition is just the first place where your application can provide bindings for your handler logic. As shown, the context provides an interface for your handlers to resolve services, but that is because the `Context` object is itself a registry. Every request that comes into your handler chain has a new `Context` object created, and thus a new registry, so the context can serve as a way for upstream handlers in your chain to communicate downstream. When the `Context` object is initially created, it is joined to the upstream registries, so within your handlers you are able to resolve services that you bound as part of your application definition.

Consider a scenario where your application needs to perform security authorization on a request prior to serving data. Assume that an access token will be supplied as a header in the request, and from within a resource's handler we need to ensure that the client has access to the data. Multiple handlers will likely need to perform this type of authorization check, so a good practice in this case is to create a top-level handler that looks up the user's profile based on the provided token, and provides that profile to downstream handlers. Example 5-2 shows a demonstration of this scenario.

Example 5-2. Component delegation sample code

```java
package app;

import ratpack.handling.Context;
import ratpack.registry.Registry;
import ratpack.server.RatpackServer;

public class Main {

  private static final String AUTH_HEADER = "x-auth-token";

  public static void main(String[] args) throws Exception {
    RatpackServer.start(spec -> spec
            .registryOf(r -> r
              .add(UserService.class, new DefaultUserService())
            )
            .handlers(chain -> chain
              .all(ctx -> { ❶
```

```java
            if (ctx.getRequest().getHeaders().contains(AUTH_HEADER)) { ❷
              String token = ctx.getRequest().getHeaders().get(AUTH_HEADER);
              UserService userService = ctx.get(UserService.class);
              userService.getProfileByToken(token).then(profile -> ❸
                ctx.next(Registry.single(profile)) ❹
              );
            } else {
              unauthorized(ctx);
            }
          })
          .get("users/:username", ctx -> {
            UserProfile profile = ctx.get(UserProfile.class); ❺
            if (profile.isAuthorized("showuser")) { ❻
              UserService userService = ctx.get(UserService.class);
              userService.getUser(ctx.getPathTokens().get("username"))
                  .then(user -> {
                    ctx.getResponse().contentType("application/json");
                    ctx.render(jsonify(user));
                  });
            } else {
              unauthorized(ctx);
            }
          })
        )
    );
  }

  private static void unauthorized(Context ctx) {
    ctx.getResponse().status(401);
    ctx.getResponse().send();
  }

  private static String jsonify(User user) {
    return "{ \"username\": \""
        + user.getUsername() + "\", \"email\": \""
        + user.getEmail() + "\" }";
  }
}
```

❶ We start out by attaching an `all` handler, which inspects every incoming request.

❷ Within this handler, we check to make sure the authorization token is provided.

❸ If it is, we look up the `UserProfile` by the specified token.

❹ Once we get the user's profile, we can manage data flow to the downstream handler by creating a single object registry and joining it to the context registry through the `ctx.next(Registry.single(profile))` call.

❺ In this downstream `get` handler, we can now extract the `UserProfile` from the context registry.

❻ Let's assume the `UserProfile` class has an `isAuthorized` method that performs some calculation to ensure that the requested resource and provided parameters align to provide an authorization in the form of a `boolean` response. In this case, if the user is allowed to perform the `showuser` function, then we process accordingly; if they are not allowed, we send back an HTTP Unauthorized status code.

The new single-object registry that we attach at the beginning of the chain acts as a child registry. The registry contract dictates that objects will be first resolved from child registries; if the object does not exist in a child registry, then the object will be requested from the parent's registry, and so forth. This pattern is what allows the context to remain an immutable registry, while still facillitating data flow through the handler chain.

Managing data flow through the handler chain is critically important for any nontrivial application, and it is entirely because of Ratpack's registry paradigm that this flexibility exists. Note that because handlers are functionally isolated members of your application, they are not appropriate places to store state. Even standalone handlers should not store state during the request processing chain. Instead, it is better practice to make use of the context registry when you need to build some mutable state to flow through the handler chain.

Google Guice

Now that you have a comprehensive understanding of Ratpack registries, we can continue discussing its framework modules. As noted earlier, the registry paradigm is what provides the extensibility and flexibility for dependency injection frameworks to interoperate at the framework level. Ratpack's various libraries provide many components that decorate your application with advanced capabilities. Instead of asking you to bind each of those components individually through your application definition, instead Ratpack provides its features as Guice modules. These modules are incorporated into your application through Ratpack's Guice support, which provides a mechanism for easily creating a Guice-backed registry.

Google Guice is a programmatic dependency injection library, and is the favored DI engine for Ratpack. Its concept of *modules* and ability to programmatically construct component bindings without the need for classpath scanning or other opinionated conventions make it an excellent choice for Ratpack applications.

To start with, we need to ensure that the `ratpack-guice` dependency is available within our project. If you are building a Groovy-based Ratpack project and utilizing the `ratpack-groovy` Gradle plugin, then the dependency is already applied. For Java-based projects, you need to explicitly include it as part of your Gradle build script. The `ratpack.dependency(..)` mechanism was demonstrated in Chapter 2, and Example 5-3 shows a Java-based build script with the Guice dependency applied.

Example 5-3. Java build script with Guice dependency

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-java'

mainClassName = 'app.Main'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency("guice")
}
```

In the prior section, we demonstrated using the `registryOf` method in the application definition to bind components to the user registry. The definition also supports the ability for you to supply a `Registry` object, which will be used in lieu of having Ratpack create one on your behalf. When employing the Guice support, we can make use of a factory method off of the `ratpack.guice.Guice` class to construct a Guice `Injector` and build the registry implementation that will resolve components through it. The code in Example 5-4 shows our application like before, this time using Guice to bind our `UserService` class.

Example 5-4. Java application with Guice-backed registry

```
package app;

import ratpack.guice.Guice;
import ratpack.server.RatpackServer;

public class Main {
```

```
  public static void main(String[] args) throws Exception {
    RatpackServer.start( spec -> spec
        .registry(Guice.registry(r -> r ❶
          .bindInstance(UserService.class, new DefaultUserService()) ❷
        ))
        .handlers(chain -> chain
              .get(ctx -> {
                UserService userService = ctx.get(UserService.class); ❸
                userService.list().then(users -> {
                  StringBuilder sb = new StringBuilder();
                  sb.append('[');
                  for (User user : users) {
                    sb.append(jsonify(user));
                  }
                  sb.append(']');
                  ctx.getResponse().contentType("application/json");
                  ctx.render(sb.toString());
                });
              })
        )
    );
  }

  private static String jsonify(User user) {
    return "{ \"username\": \""
        +user.getUsername()+"\", \"email\": \""
        +user.getEmail()+"\" }";
  }
}
```

❶ Here, we create a new Guice-backed registry using the `Guice.registry(..)` factory method. You can also see that we are now making use of the `BindingsSpec` API to add our components to the registry. Within it, we get some additional features that pertain specifically to Guice, such as the ability to specify an interface to an implementation.

❷ In this example, we specifically bind a new instance of the `DefaultUserService` class to the `UserService` interface.

❸ Now, we can retrieve the class that is bound to the `UserService` interface, and our handler logic never needs to be concerned with concrete types.

Even though we have changed the underlying infrastructure that supports our application's *user registry*, we do not have to change the means by which we access components within our code. Regardless of the fact that we are using Guice, we continue to use the `Context` to retrieve component bindings. Most applications will find benefit in using dependency injection, but Ratpack's lightly opinionated nature leaves it entirely up to you as to when the time is appropriate to incorporate DI.

BindingsSpec in Groovy

If you are building a Groovy-based application, then there is out-of-the-box support for Guice. The Groovy DSL takes a slightly more structured approach to defining your application, so it provides a `bindings` block, as discussed briefly in Chapter 2. This block delegates to the `BindingsSpec` to make wiring your components from the *src/ratpack/ratpack.groovy* script into a Guice-backed registry a concise process.

The `bindings` block exposes several helper functions to simplify the steps necessary to bind your components through Guice. Generally speaking, the Ratpack philosophy is to make dependency injection a "one-liner," so the interfaces for creating injected components are fairly straightforward.

Perhaps the most commonly used function in the `bindings` block is the `bindInstance` method, which we have seen previously. This allows you to provide a concrete binding to an interface type, therein giving you the flexibility of implementation. To recap the `bindInstance` usage, consider the application depicted in Example 5-5.

Example 5-5. Groovy BindingsSpec sample

```groovy
import app.DefaultUserService
import app.UserService

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bindInstance(UserService, new DefaultUserService()) ❶
  }

  handlers {
    all { UserService userService -> ❷
      userService.list().then { users ->
        render(toJson(users))
      }
    }
  }
}
```

❶ Here, we bind the `UserService` interface to an instance of the `DefaultUserService` implementation.

❷ Our handler is able to inject components by their bound types, instead of having to deal with concrete implementation classes.

Use of Guice-backed registries means that Ratpack applications are able to garner all the benefits afforded by Guice's injection system. If, for example, the `DefaultUserSer`

vice wanted to make use of a data access object within its implementation, then we could simply provide that as a field on the class and annotate it with @Inject. Inspect the DefaultUserService code snippet in Example 5-6 to understand this better.

Example 5-6. DefaultUserService implementation

```groovy
package app

import ratpack.exec.Blocking
import ratpack.exec.Operation
import ratpack.exec.Promise

import javax.inject.Inject

class DefaultUserService implements UserService { ❶

  @Inject ❷
  UserDAO dao

  @Override
  Promise<List<User>> list() {
    Blocking.get {
      dao.listUsers() ❸
    }
  }
}
```

❶ As you can see, the DefaultUserService is an implementation of the UserService, which is the type to which we bound in the bindings block of our application.

❷ We can leverage the javax.inject.Inject annotation to inform Guice that the UserDAO object should be automatically injected.

❸ Our method code can make use of the UserDAO object as it normally would.

Now we can update the Ratpack Groovy script to also bind the UserDAO implementation, as shown in Example 5-7.

Example 5-7. Binding UserDAO in Guice-backed registry

```groovy
import app.DefaultUserService
import app.MySqlUserDAO
import app.UserDAO
import app.UserService

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack
```

```
ratpack {
  bindings {
    bindInstance(UserDAO, new MySqlUserDAO()) ❶
    bindInstance(UserService, new DefaultUserService())
  }

  handlers {
    all { UserService userService ->
      userService.list().then { users ->
        render(toJson(users))
      }
    }
  }
}
```

❶ When we bind the `UserDAO` object to the `MySqlUserDAO` implementation, it
   becomes participant in the Guice lifecycle. The subsequent binding of the `Defaul`
   `tUserService` allows Guice to provide the necessary binding to its `dao` field.

This mechanism of backing the registry with DI gives your applications a plethora of
extensibility. You can get even deeper with DI through the `BindingsSpec` by utilizing
the `bind` method to provide class-level bindings directly. If we refactor the `DefaultU`
`serService`, as shown in Example 5-8, we can utilize Guice's ability to provide con‐
structor variable argument dependency injection and remove the necessity for a
mutable field-level dependency.

Example 5-8. Refactored DefaultUserService

```
package app

import ratpack.exec.Blocking
import ratpack.exec.Operation
import ratpack.exec.Promise

import javax.inject.Inject

class DefaultUserService implements UserService {

  private final UserDAO dao

  @Inject
  DefaultUserService(UserDAO dao) {
    this.dao = dao
  }

  @Override
  Promise<List<User>> list() {
    Blocking.get {
```

```
      dao.listUsers()
    }
  }
}
```

From here, we can update our Ratpack Groovy script to bind directly to the class and let Guice handle the construction of the object on our behalf. This is shown in Example 5-9.

Example 5-9. Binding the DefaultUserService without constructor

```
import app.DefaultUserService
import app.MySqlUserDAO
import app.UserDAO
import app.UserService

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bindInstance(UserDAO, new MySqlUserDAO())
    bind(UserService, DefaultUserService) ❶
  }

  handlers {
    all { UserService userService ->
      userService.list().then { users ->
        render(toJson(users))
      }
    }
  }
}
```

❶ The `DefaultUserService` can now maintain a private and final field variable, and we pass the onus of properly constructing that object on to Guice. This allows the application code to operate in a more modular way, and frees it from the burden of constructing new objects, and therein properly utilizes the DI engine.

Using the Guice binder in Ratpack

The Guice integration in Ratpack extends even further, and for more advanced requirements, you can get direct access to Guice's binding configuration. Exposed through the `binder` method on the `BindingsSpec`, Ratpack allows you to interface directly with the DI engine to perform any more advanced configuration that you may need, as shown in Example 5-10. It is fairly uncommon that you would need to

dig into this layer from within your Ratpack application, but it's good to know that the capability is there.

Example 5-10. Accessing the Guice binder

```
import app.DefaultUserService
import app.MySqlUserDAO
import app.UserDAO
import app.UserService
import com.google.inject.Scopes

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    binder { b -> ❶
      b.bind(UserDAO).to(MySqlUserDAO).in(Scopes.SINGLETON)
      b.bind(UserService).to(DefaultUserService).asEagerSingleton()
    }
  }

  handlers {
    all { UserService userService ->
      userService.list().then { users ->
        render(toJson(users))
      }
    }
  }
}
```

❶ Here, we are hooking into the Guice `binder` configuration to get direct access to the Guice APIs for binding our components. It's not necessary to change anything related to our interaction with the DI-backed components from the registry; this is simply a way to perform more specific binding configuration than the `BindingsSpec`'s `bindInstance` and `bind` methods provide you.

From the code, you can see that we are now specifying that we wish for the `MySqlUserDAO` to be bound to the `UserDAO` interface as a singleton from the Guice binder. This means that the `MySqlUserDAO` will be constructed once, and only once, when it is injected into its dependent components. Likewise, we are binding the `DefaultUserService` to the `UserService` interface as an *eager singleton*, ensuring that its instance is constructed and ready to go at application startup time.

Ratpack's integration with Guice at this layer opens the door for your application to have a plethora of extensibility, and enables the mechanism by which Ratpack's framework modules are incorporated in your application's runtime. If your applica-

tion has modular considerations, you can utilize the same mechanism that the framework leverages to provide components through modular programmatic bindings.

If we take the examples from before and refactor them slightly to take a more modular approach, we can bind them into our application with just a single line of code. Consider the Guice module in Example 5-11, which takes the UserDAO and UserService bindings and incorporates them from a Guice module implementation.

Example 5-11. Binding in a Guice module

```groovy
package app

import com.google.inject.AbstractModule
import com.google.inject.Scopes

class ApplicationModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(UserDAO).to(MySqlUserDAO).in(Scopes.SINGLETON)
    bind(UserService).to(DefaultUserService).asEagerSingleton()
  }
}
```

By subclassing the com.google.inject.AbstractModule, we are given the ability to implement a configure() method, which has protected access to the bind(..) method that we saw previously. This class can exist in a submodule to your project, enabling you to define proper service class boundaries within your architecture. From here, we can leverage the module method on the BindingsSpec within the Ratpack Groovy script to properly incorporate the bindings into our application, as shown in Example 5-12.

Example 5-12. Applying a Guice module

```groovy
import app.ApplicationModule
import app.UserService

import static groovy.json.JsonOutput.toJson
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(ApplicationModule) ❶
  }

  handlers {
    all { UserService userService -> ❷
      userService.list().then { users ->
        render(toJson(users))
```

```
          }
        }
      }
}
```

❶ Providing the `ApplicationModule` to the `BindingsSpec` allows Guice to perform its binding operation through the module's `configure` method.

❷ The bindings contained therein are made available to your application as they normally would.

Guice modules are the means by which Ratpack provides extensibility to applications. Nearly all framework features are incorporated into applications through the use of Guice modules. While it is not strictly required for your application to rely on Guice for dependency injection, when you need to include Ratpack features, then Guice will likely be required.

Framework Modules

Framework modules in Ratpack are built as Guice modules, and thus implement the `AbstractModule` abstract class that was shown in the prior section. This is the preferred method of extensibility in Ratpack, though this path serves merely as a means to an end. That is to say, the goal of framework modules is to get their offered components into the user registry for use throughout your application. Because Guice provides a convenient system of modularity, the door is open for extensibility through this mechanism.

It is a two-step process to incorporate a framework feature into your Ratpack application. By now, these parts have been laid out at various points, so the process should look familiar. The first step is to incorporate the appropriate dependency into your project. When making use of the Ratpack Gradle plugin, it is as simple as employing the `ratpack.dependency(<module qualifier>)` helper within the `dependencies` block of your build script. For example, the build script in Example 5-13 shows the inclusion of the `ratpack-session` module as a project dependency. Note that the module qualifier is denoted as the artifact identifier, without the leading `ratpack-`.

Example 5-13. Gradle build script with session module

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}
```

```
apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency('session')
}
```

The second step to incorporating a framework feature is to bind its associated module through the BindingsSpec. You can make use of the module(..) method within the bindings block of your application specification to accomplish this. The Ratpack Groovy script in Example 5-14 shows bringing in the ratpack.session.SessionModule using this mechanism.

Example 5-14. Incorporating the SessionModule

```
import ratpack.session.Session
import ratpack.session.SessionModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(SessionModule) ❶
  }

  handlers {
    // ... [snip] ...
  }
}
```

❶ For most of the framework modules, this single line is sufficient for incorporating a module's functionality.

Behind the scenes, a module may bind framework-level components, like Parsers, Renderers, and Handlers, but this will generally be transparent. There are, however, exceptions to the "one-liner" rule, where some framework modules will require you to provide additional configuration directives to help properly incorporate them in your application. These types of modules are known as ConfigurableModules, and provide a means for you to programmatically influence the default configuration of a particular feature.

Configurable Modules

Many of Ratpack's framework modules will require you to provide some additional configuration as part of their function. The most notable example of this is the `ratpack-hikari` module, which provides connection pooling for database connections. When utilizing Hikari within your application, you will need to provide it with configuration directives, including the database connection URL and the datasource driver class name.

To accommodate configurability of modules, the `BindingsSpec` offers an interface for providing configuration values to the configuration model object exposed by those modules. To demonstrate this capability, consider the incorporation of the `HikariMod ule` in the `BindingsSpec`. Example 5-15 shows bringing in Hikari through the `mod ule(HikariModule)` call, this time with the added change of influencing its configuration model.

Example 5-15. HikariModule (configuration sample code)

```
import ratpack.hikari.HikariModule
import javax.sql.DataSource
import ratpack.exec.Blocking

import static ratpack.jackson.Jackson.json
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(HikariModule) { c ->            ❶
      c.dataSourceClassName = "com.mysql.jdbc.jdbc2.optional.MysqlDataSource" ❷
      c.addDataSourceProperty "URL", "jdbc:mysql://localhost:3306/db" ❸
      c.username = 'root'                  ❹
    }
  }

  handlers {
    prefix("users") {
      get { DataSource dataSource ->       ❺
        // ... [snip for brevity] ...
      }
    }
  }
}
```

❶ The closure that you supply to `module` is provided the module's configuration model, similar to how the Groovy DSL handler chain works. This means that you can contextually influence the configuration properties of the model object directly within your closure's code.

❷ In the case of the `HikariModule` specifically, its configuration object provides a property setter for `dataSourceClassName`, and here you can see the assignment.

❸ Additionally, methods on the configuration object can be called as you would expect. Here, we are calling the `addDataSourceProperty` method and supplying it with a `URL` parameter.

❹ We must also set the username we will connect to. For the case of a local demonstration, using `root` without a password will be sufficient.

❺ Here, you can make use of the `javax.sql.DataSource` object provided by the module, and know that it will be fully configured according to the properties you specified when adding the module to the user registry.

In the case of `ConfigurableModule`, you can provide a closure to the `module` method, and you will be given the opportunity to influence its configuration model prior to any of its component bindings. This is critically important for modules that need to supply some user-configured detail during their binding. Most of Ratpack's framework modules are provided as `ConfigurableModule` types, though they are generally preconfigured with reasonable defaults to make it easy to get the most common configurations off the ground. Only in the cases where you need to make changes should you have to affect binding at this level.

As expected, the Java API for working with `ConfigurableModule` is not dissimilar. The major difference you will see is that the lambda expression that we supply to the `module` method takes the configuration object as a parameter. The code shown in Example 5-16 demonstrates the same capability as the prior example, this time using the Java API.

Example 5-16. HikariModule (Java configuration)

```java
package app;

import javax.sql.DataSource;
import ratpack.server.RatpackServer;
import ratpack.hikari.HikariModule;
import ratpack.guice.Guice;
import ratpack.exec.Blocking;
import java.sql.*;
import java.util.List;
import java.util.ArrayList;

import static ratpack.jackson.Jackson.json;

public class Main {
```

```java
public static void main(String[] args) throws Exception {
  RatpackServer.start(spec -> spec
    .registry(Guice.registry(b -> b ❶
      .module(HikariModule.class, config -> { ❷
        config.setDataSourceClassName
        ("com.mysql.jdbc.jdbc2.optional.MysqlDataSource"); ❸
        config.addDataSourceProperty("URL", "jdbc:mysql://localhost:3306/db");
        config.setUsername("root");
      })
    ))
    .handlers(chain -> chain
      .prefix("users", pchain -> pchain
        .get(ctx -> {
          DataSource dataSource = ctx.get(DataSource.class); ❹
          // ... [snip] ...
        })
      )
    )
  );
}

}
```

❶ We create a Guice-backed registry to be able to work with the `BindingsSpec`.

❷ Here, we add the `HikariModule`, and to the second parameter we supply a lambda expression that takes the module's config object.

❸ Similar to the Groovy example, we call setters and methods on the config object within this block.

❹ Within our handler logic, we can now use the `Context` to get access to the `Hikari Module`'s fully configured component bindings.

The `ConfigurableModule` mechanism is an excellent way for users to be able to participate in the bootstrapping of module bindings. Though not every module requires configuration, in the case where they do, Ratpack makes it simple and concise to get those configurations in place. `ConfigurableModule` types are not reserved strictly for framework modules either. If your applications or libraries are using modules to provide component bindings, they too can implement this type.

Consider again the `ApplicationModule` example from earlier, and say that you want to provide the `DefaultUserService` with a string as part of its construction. The scenario may be that the implementation is given a unique qualifier from the environment. In that case, instead of directly reading the value from the environment, we can leverage Ratpack's configuration mechanism and `ConfigurableModule` to ensure the

class gets the right value. The code in Example 5-17 demonstrates the changes to our module.

Example 5-17. Configurable ApplicationModule

```
package app

import ratpack.guice.ConfigurableModule
import com.google.inject.Scopes
import com.google.inject.Provides

class ApplicationModule extends ConfigurableModule<ApplicationModule.Config> { ❶

  static class Config { ❷
    String nodeName
  }

  @Override
  void configure() { ❸
    bind(UserDAO).to(MySqlUserDAO).in(Scopes.SINGLETON)
  }

  @Provides
  UserService userService(UserDAO dao, Config config) { ❹
    new DefaultUserService(dao, config.nodeName)
  }
}
```

❶   Here, we extend the `ratpack.guice.ConfigurableModule` class, and to its parameterization we provide the configuration class this module works with.

❷   Many Ratpack framework modules define their configuration classes statically within the module itself. This is a good pattern to follow, as it keeps things concise and easy to follow, however there is no strict requirement to do this.

❸   Like before, we can continue to use the `configure` method to perform bindings.

❹   Here, we are using the `@Provides` faculty of Guice to provide the binding for the `UserService`. The arguments to this method will be provided by Guice, so we can safely get an instance of the `UserDAO` and `Config` to supply to the `DefaultUserService` constructor.

Within module methods annotated with `@Provides`, the method arguments are pulled from the Guice bindings, so components previously bound by Guice can be used in the construction of later bindings. In this case, we are injecting the `Config` object to the `userService` method and we use that in the construction of the `Defaul`

tUserService. Next, we can demonstrate working with this module in our application, as shown in Example 5-18.

Example 5-18. Using the configurable ApplicationModule

```
import app.ApplicationModule
import app.UserService

import static ratpack.groovy.Groovy.ratpack
import static groovy.json.JsonOutput.toJson

ratpack {
  serverConfig {
    env() ❶
  }
  bindings {
    moduleConfig( ❷
      ApplicationModule,
      serverConfig.get("/user", ApplicationModule.Config)
    )
  }
  handlers {
    get("users") { UserService userService -> ❸
      // ... [snip] ...
    }
  }
}
```

❶ The desire is for the application to pull the config's `nodeName` string from the environment, so here we specify that we want to bring configuration from environment variables.

❷ When adding the `ApplicationModule`, we can map the module's `Config` object from Ratpack's configuration mechanism. Note that here we use the `moduleConfig` method, since we are providing the already populated `Config` object, instead of configuring manually through a closure, as we have seen with the `module` method.

❸ And, as we would expect, we can continue to work with the `UserService` binding as before.

As you can see from the example, in addition to being able to configure the module via a closure, we can also provide an already-mapped `Config` object. Using this approach, an environment variable of `RATPACK_USER__NODE_NAME` will be mapped onto the `nodeName` property of the `ApplicationModule.Config` class and provided to the `DefaultUserService`.

Modular Object Rendering in Ratpack

Another mechanism for modularity in Ratpack comes from its support for decoupling the rendering of model objects from your handler logic. Binding via the framework's `ratpack.render.Renderer` interface type allows you to implement your rendering logic in a standalone fashion. This can assist you in better organizing the logic for how your application provides data back to callers. Complex business rules for ascertaining what data is returned from model objects can be isolated to your renderer's implementation, allowing your handlers to remain focused on handling a request. This form of decoupling can play a critical role in applications that wish to provide layers of authorization. It can also help to massage data into forms that better fit a caller's needs.

Making use of renderers in Ratpack is as easy as implementing the `Renderer` interface and binding it in the user or context registry. For example, the code in Example 5-19 shows the `UserRenderer` implementation, which is responsible for rendering `User` objects.

Example 5-19. Rendering the User type

```groovy
package app

import ratpack.handling.Context
import ratpack.render.Renderer

import static groovy.json.JsonOutput.toJson

class UserRenderer implements Renderer<User> {
  @Override
  Class<User> getType() {
    User
  }

  @Override
  void render(Context context, User user) throws Exception { ❶
    def showAll = context.request.queryParams.containsKey("showAll") && ❷
        context.request.queryParams.showAll == "true"

    if (showAll) { ❸
      context.render(toJson(user))
    } else {
      context.render(toJson([username: user.username]))
    }
  }
}
```

❶ The `render` method is given access to the request's `Context` object, so within the `UserRenderer`, you are given the opportunity to inspect the request as part of your processing flow.

❷ In this example, the renderer checks if the user has provided the `?showAll=true` query parameter.

❸ If so, the full set of properties from the `User` object is rendered back as JSON; if not, then only the username is rendered back.

For the `UserRenderer` to be made available in rendering `User` objects, you need only add it to the user registry. The application code shown in Example 5-20 depicts adding the renderer and making use of it.

Example 5-20. Binding the UserRenderer

```groovy
import app.DefaultUserService
import app.MySqlUserDAO
import app.UserDAO
import app.UserRenderer
import app.UserService

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bind(UserDAO, MySqlUserDAO)
    bind(UserService, DefaultUserService)
    bind(UserRenderer) ❶
  }

  handlers {
    prefix("users") {
      get(":username") { UserService userService ->
        userService.getUser(pathTokens.username).then { user ->
          render(user) ❷
        }
      }
    }
  }
}
```

❶ The call to `bind(UserRenderer)` incorporates it in the user registry, and it is made available to Ratpack's rendering infrastructure.

❷ When the call to `context.render(User)` is made Ratpack will make user of the `UserRenderer` to materialize the response for the caller.

Ratpack's rendering infrastructure serves as an excellent means by which the logic for producing structured model data to consumers can be decoupled from the request handling logic. Following this paradigm allows the requirements for how consumers use your application to evolve independently of its models and request handling life-cycle.

Rendering with Content Type

Renderers play an important role in ensuring that the response's content type is appropriately set when sending data back to a client. Up until now, most of the demonstrations in this book have shown simply rendering a string value back to the client. This is a good strategy when prototyping to ensure that the flow and configuration of your application is working as you would expect. It does not work well, however, when you intend to get your application in front of consumers that require the data that your service provides to come with an associated content type.

There are two ways to ensure that your response's content type is properly set. The first, and perhaps most common way, is to explicitly set the content type before your handler sends the response. This can be accomplished succinctly, as demonstrated by the handler code snippet in Example 5-21.

Example 5-21. Explicitly setting contentType in handler

```
get {
  response.contentType("application/json")
  response.send(toJson(new User(username: "dan", email: "danielpwoods@gmail.com")))
}
```

The call to `response.contentType("application/json")` ensures that the response content type header is properly set, regardless of the type of data that you are sending. Indeed, in all the handler examples we've looked at thus far, the code is simply sending back a string. Ratpack's internal renderers will capture the fact that the handler is sending back a string, and if no content type has been explicitly specified, then the "text/plain" content type will be applied. The exception to this rule is when your handler logic employs the `byContent` mechanism, the response content type will be set to that of the supplied "Accept" header (except when using the `noMatch` method, which requires you to explicitly specify the content type).

When making use of a `Renderer`, your application can set the content type explicitly from within the `render` method. This means that when your rendering logic must account for many different response formats, your handler logic can remain concise and isolated from the setting of content types and mechanisms by which your models are serialized back to clients.

Rendering JSON Data

It is indisputable that JSON has become the de facto serialization format for the Web. Up until this point, the JSON examples in this book have utilized Groovy's `JsonOutput#toJson` method to serialize objects to JSON. As noted in Chapter 4, using the `toJson` method within a call to `render` causes Ratpack to interpret the response as "text/plain," so additional steps are necessary to ensure that the response content type is set appropriately. Additionally, the `JsonOutput` helper class does not provide flexibility and extensibility in how your JSON data is serialized back to clients.

To provide rich support for working with JSON, Ratpack provides a JSON renderer implementation that utilizes Jackson for serializing your models. Jackson provides a high level of flexibility for working with JSON data, including an extensive system of modularity for ensuring that objects are serialized and deserialized according to your data's requirements. Jackson is a core dependent of Ratpack's, and the `JsonRenderer` implementation in Ratpack is provided out of the box to help facilitate working with JSON data.

Rendering an object as JSON with Jackson is as simple as constructing a `JsonRenderer` and supplying it to Ratpack's rendering infrastructure. You can make use of the `json` factory method on the `ratpack.jackson.Jackson` class to construct the `JsonRenderer`, and once created, you can supply it to the `render` method within your handler. The code in Example 5-22 demonstrates this capability.

Example 5-22. Rendering JSON

```
import app.DefaultUserService
import app.MySqlUserDAO
import app.UserDAO
import app.UserService

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    bind(UserDAO, MySqlUserDAO)
    bind(UserService, DefaultUserService)
  }

  handlers {
    prefix("users") {
      get(":username") { UserService userService ->
        userService.getUser(pathTokens.username).then { user ->
          render(json(user)) ❶
        }
      }
    }
```

}
}

❶ When rendering the `user` as JSON, we can wrap the object in a `Renderable` type, which is facilitated through the `ratpack.jackson.Jackson.json` static call.

The render call is now delegated to the `JsonRenderer`, which ensures that the response content type is properly set and that the object is appropriately serialized to JSON. Furthermore, your application can utilize Jackson's structures—including its annotations, serializers, and deserializers—for ensuring your data is properly serialized. In most cases, the code in Example 5-22 is sufficient for properly sending JSON back to clients. Using the `JsonRenderer` means that the most common use cases of responding with JSON data do not require you to build your own renderers.

It is important to note that although Jackson support is provided by default, it is in no way an inextensible component. Indeed, like all components in Ratpack, the latest registry binding will always win precedence. This means that if you have a scenario where you need to provide a customized version of the `ObjectMapper`, then a specific binding can be provided in place of the default one.

Special Rendering Scenarios

There are two special rendering scenarios in Ratpack where the framework will take an opinionated approach to your handler's response. The first special rendering situation is when you attempt to render back a `null` object. In this case, Ratpack will set the response status code to 404. If a `null` object in your application denotes anything other than a "Not Found" status, then your handler logic will need to account for null values.

The second scenario is when you attempt to render a `Promise` object. In this scenario, Ratpack will fulfill the promise and pass the resulting data on to the rendering infrastructure for further processing. The code in Example 5-23 demonstrates this scenario by removing the call to `then`, as shown in the prior examples.

Example 5-23. Rendering a Promise

```
import app.*

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bind(UserDAO, MySqlUserDAO)
    bind(UserService, DefaultUserService)
    bind(UserRenderer)
  }
```

```
handlers {
  prefix("users") {
    get(":username") { UserService userService ->
      render(userService.getUser(pathTokens.username)) ❶
    }
  }
}
}
```

❶ In this example, we pass a `Promise<User>` to the `render` method. Ratpack's rendering infrastructure will fulfill this promise, and the resulting user object will be handed off to the `UserRenderer` for final processing.

The ability to render `Promise` types directly provides a simplification for code that fulfills a promise and hands the resulting object to the `render` command. This shortcut keeps your code concise and directed.

Chapter Summary

With the information provided in this chapter, you are prepared to develop applications that make use of Ratpack's more advanced capabilities of component binding, modularity, and dependency injection. Additionally, you are now equipped to better architect your code for proper service and component boundaries, as well as make use of advanced configuration directives. The understanding provided up until this point will prove critically useful for the conversation in subsequent chapters. Building on the discussion that has taken place up until this point, the remaining chapters will expose you to using Ratpack's faculties for serving static assets, creating data-driven applications, and building robust microservice architectures.

Serving Web Assets

Although much of the conversation and example code in this book thus far has focused on building applications that send simple responses back to clients, Ratpack also has an extensive and flexible infrastructure for serving and generating web content. In addition to being able to serve static content from within a project, Ratpack also supports the ability to generate dynamic content on a per-request basis. This chapter will walk you through working with Ratpack's content serving and generation mechanisms to build full-featured web applications.

Web content, including static assets and templates, can be served from a directory within the project, a JAR on the classpath, or from a directory on the filesystem. To accommodate serving content, Ratpack's `ServerConfig` must be supplied with a base directory through the `baseDir` property. With a Ratpack Groovy project, the base directory will be automatically discovered based on the location of the *Ratpack.groovy* file (*src/ratpack* unless explicitly configured otherwise). For main class Ratpack applications, the application base directory can also be automatically discovered by placing a *.ratpack* marker file within your application's project structure.

Serving Static Content

Before you begin serving static content from within your application, you should create a subdirectory within the base directory of your project from which static content will be derived. The naming of this directory is arbitrary to its use, but it is advisable to name it something like *static* to denote that the files contained within represent nondynamic content. The directory tree in Example 6-1 shows a project structure for a Ratpack Groovy application with an *src/ratpack/static* directory that contains the application's static HTML, JavaScript, and CSS files. Note again that the base directory in this structure is automatically resolved to *src/ratpack*.

Example 6-1. Project structure with static resources

```
.
├── build.gradle
└── src
    └── ratpack
        ├── Ratpack.groovy
        └── static
            ├── css
            │   └── app.css
            ├── index.html
            └── js
                └── app.js
```

Ratpack main class applications can be similarly structured, but it is important to note that these types of applications make no assumption as to the location of the base directory. That is to say, when building the application definition, a `ServerConfig` object must be supplied, with the base directory properly specified. Consider the directory tree shown in Example 6-2, which depicts a common configuration for Java-based applications.

Example 6-2. Project structure with specified base directory

```
.
├── build.gradle
└── src
    └── main
        ├── java
        │   └── app
        │       └── Main.java
        └── resources
            ├── .ratpack
            └── static
                ├── css
                │   └── app.css
                ├── index.html
                └── js
                    └── app.js
```

Within this project, the *.ratpack* marker file is placed directly within the *src/main/resources* directory. The *src/main/resources/static* directory is where the static files will be served for this application.

As you are already familiar with the handler chain in Ratpack, it should be understood that serving content comes by way of a built-in handler. By making use of the `files` method on the `Chain` API, we can configure a handler to derive static content from a directory relative to the base directory. Example 6-3 shows the usage of the `files` method from the Groovy DSL.

Example 6-3. The files method on the Chain API

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    files { ❶
      dir("static").indexFiles("index.html")
    }
  }
}
```

❶ The `files` call is made, which takes a closure to configure the handler. Within that closure, the `dir("static")` call is made indicating that the handler should serve assets from the *src/ratpack/static* directory. Finally, the call to `.index Files("index.html")` is used to indicate that when serving a directory, the *index.html* file should be served as the directory's root path.

The same API exists for serving static content within a Java main class application. You can see in Example 6-4 that the application's `ServerConfig` object is configured by way of the `serverConfig` call on the application definition. Within the handler chain, the `chain.files(..)` call is made in the exact same way as the prior example.

Example 6-4. The files method on Java chain

```
package app;

import ratpack.server.BaseDir;
import ratpack.server.RatpackServer;

public class Main {

  public static void main(String[] args) throws Exception {
    RatpackServer.start(spec -> spec
            .serverConfig(c -> c.baseDir(BaseDir.find()).build()) ❶
            .handlers(chain -> chain
                    .files(files -> files
                            .dir("static").indexFiles("index.html")
                    )
            )
    );
  }
}
```

❶ Here we use the `BaseDir#find` method to indicate that Ratpack should discover the base directory by locating the marker file.

If you run either of these applications and navigate to *http://localhost:5050*, your browser will display a "Hello, World!" message from the *index.html* file from the project's *static* directory.

Caveats to the FileHandler

The `FileHandler` that is added to the chain through the `files` method is a *terminal handler*, which means that it will take the responsibility of resolving a requested static resource and sending it back to the client. In the case where a static resource is requested, but it cannot be found within the `dir(..)` supplied to the `FileHandler` `Spec`, the handler will respond to the client with a 404 HTTP status code.

Given its terminal nature, any use of the `files` method on the handler chain *must be* placed at the end of the chain to allow calls that are not destined for a static asset to be eligible for processing. This rule applies to subchains as well, but if you wish to serve all of your static assets out of a particular route—say, `/static`—then it is sufficient to make the call to `files` within a subchain provided to the `prefix` handler.

To depict these caveats of the `FileHandler` better, consider the code in Example 6-5. As demonstrated, in order for requests to `/api` to succeed, the `files` handler must be specified *after* the `prefix("api")` call.

Example 6-5. Files and routes

```
ratpack {
  handlers {
    prefix("api") {
      // ... API routes snipped ...
    }
    files {
      dir("static").indexFiles("index.html")
    }
  }
}
```

It is generally considered best practice for the `files` handler to be placed at the end of the chain. This ensures that any potential handling overlap does not accidentally get served by static content when you really intended your nonstatic content handler to respond to the request. Example 6-6 demonstrates an alternative scenario where `files` can be included elsewhere.

Example 6-6. Static files prefixed

```
ratpack {
  handlers {
    prefix("static") {
```

```
      files {
        dir("static").indexFiles("index.html")
      }
    }
    prefix("api") {
      // ... API routes snipped ...
    }
  }
}
```

It is important to remember when following this best practice that handlers may create a *subchain*, so adding a `files` handler at the end of the subchain of a `prefix` handler would continue to serve best practices.

Using FileSystemBinding to Customize Asset Resolution

The moment your application is started, Ratpack builds a `FileSystemBinding` object that is used to represent resources, including static content, within your application. The `FileSystemBinding` acts as the interface to all files within your project, and provides you with a convenient mechanism to work with project-level resources directly from within a handler's logic.

If you have the need to access a project-level file from within a handler, the provided `Context` object supplies a shortcut method for doing so. With a call to `Context#file`, you can get access to any file *relative to the current filesystem binding*. The emphasis in the last sentence will be explained in depth momentarily, but it is enough to say that the filesystem binding for a particular request can be modified according to the attributes of the request.

Consider the Groovy project structure depicted in Example 6-7, and envision a scenario where you have a handler that needs to make a decision about whether to serve the file *html/foo.html* or *html/bar.html*.

Example 6-7. Project structure with HTML resources

```
.
├── build.gradle
└── src
    └── ratpack
        ├── html
        │   ├── bar.html
        │   ├── error.html
        │   └── foo.html
        └── Ratpack.groovy

3 directories, 5 files
```

For the sake of demonstration, let's make it easy and say that the discriminating factor for determining what file to serve is a request query parameter, `file=<file>`. Given that requirement, we can craft a handler that inspects the query parameter and uses the `Context#file` method to read the file and serve it directly back to the client. If the request fails the requirement, then we can serve back the *html/error.html* file. The code in Example 6-8 shows the *ratpack.groovy* file with the corresponding handler logic.

Example 6-8. The Ratpack.groovy file

```
ratpack {
  handlers {
    all {
      def fileParam = request.queryParams.file ❶

      if (fileParam == "foo" || fileParam == "bar") { ❷
        render(file("/html/${fileParam}.html"))
      } else { ❸
        response.status(404)
        render(file("/html/error.html"))
      }
    }
  }
}
```

❶ We start by capturing the `?file=<file>` query param.

❷ Here, we check to make sure the captured parameter is `foo` or `bar`. If it is, then we use the `file` method to render back the correspondingly named HTML file.

❸ Otherwise, we set the status to `404` and render back the *error.html* file.

As noted earlier, the `FileSystemBinding` can be used to access any resource *relative to the current filesystem binding*. Within this example, the handler chain has made no adjustment to the filesystem binding, so files will be resolved relative to the *default filesystem binding*, which is the base directory. In this case, that resolves to *src/ratpack*.

Expanding on the prior example, we can make use of `fileSystem` method on the handler chain API to directly specify the filesystem binding from which files should be resolved. This can aid greatly in reusability of handler code for different file resolution requirements. To demonstrate this capability, let's start by extracting the handler logic into its own class, as shown in Example 6-9.

Example 6-9. File rendering—standalone handler

```groovy
package app

import ratpack.handling.Context
import ratpack.handling.Handler

class FooBarFileHandler implements Handler {

    @Override
    void handle(Context ctx) {
        // capture the "?file=<file>" query param
        def fileParam = ctx.request.queryParams.file

        // check to make sure it was either "foo" or "bar"
        if (fileParam == "foo" || fileParam == "bar") {
            // if so, then use the Context.file(..) call to read the requested resource
            ctx.render(ctx.file("/html/${fileParam}.html"))
        } else {
            // if not, then set the status to 404 and render back the error page
            ctx.response.status(404)
            ctx.render(ctx.file("/html/error.html"))
        }
    }
}
```

Next, let's envision that the requirements are a bit different than before, and now we need to serve a different *foo.html*, *bar.html*, and *error.html* according to what *client site* the user is visiting. That is to say, based off of whether the request is coming in to *www.client1.com* or *www.client2.com*, we should resolve the HTML files from the relative directories of *client1* and *client2* accordingly. The tree shown in Example 6-10 depicts the project structure for this example.

Example 6-10. Client-specific resources tree

```
.
├── build.gradle
└── src
    ├── main
    │   └── groovy
    │       └── app
    │           └── FooBarFileHandler.groovy
    └── ratpack
        ├── Ratpack.groovy
        ├── client1
        │   └── html
        │       ├── bar.html
        │       ├── error.html
        │       └── foo.html
        └── client2
```

```
        └── html
            ├── bar.html
            ├── error.html
            └── foo.html
```

```
9 directories, 9 files
```

Given the project structure with client-specific resources, and our `FooBarFileHan` `dler` extracted, we can update the handler chain to make use of the `fileSystem` chain method, which will scope the `FooBarFileHandler` according to what site the request is accessing. The *ratpack.groovy* script in Example 6-11 shows the updated chain.

Example 6-11. Handler chain with fileSystem

```
ratpack {
  bindings {
    bind FooBarFileHandler
  }
  handlers {
    host("www.client1.com") { ❶
      fileSystem("client1") { ❷
        all(FooBarFileHandler)
      }
    }
    host("www.client2.com") { ❸
      fileSystem("client2") { ❹
        all(FooBarFileHandler)
      }
    }
  }
}
```

❶ Use the `host(..)` handler to create a subchain when the `Host` header matches `www.client1.com`.

❷ Change the `fileSystem` binding to `client1`.

❸ Same as the previous `host` handler, but for `www.client2.com`.

❹ Same as the previous `fileSystem` call, but for `client2`.

In each of the subchains created within the `host` handler, the `fileSystem` call is employed to change the relative filesystem binding. When the `FooBarFileHandler` uses the `ctx.file("/html/${fileParam}.html")` call to resolve the appropriate HTML file, it will therefore be resolving that file within the appropriate project subdirectory.

Use of the `fileSystem` method on the handler chain API can serve as a powerful mechanism when supporting a multitenant environment that has different static content according to what site is being accessed. The use of the `FileSystemBinding` and `Context#file` method, however, are not specifically designed for multitenancy, and indeed can be used wherever it is necessary to access project-level resources. All of Ratpack's provided `Renderer` and `Handler` implementations, for example, make use of the `FileSystemBinding` or `Context#file` method to resolve configurations, templates, or other static assets within the project.

Serving Dynamic Content

Any web application of reasonable complexity that serves HTML content will likely need the ability to dynamically generate views according to the attributes of a particular request. This can be accomplished in a number of ways, given that Ratpack has robust integration with multiple dynamic templating engines, including Handlebars.js, Thymeleaf, Groovy Markup Templates, and Groovy Text Templates.

Support for Groovy Markup and Text Templates comes for free in Groovy-based Ratpack projects. Handlebars.js and Thymeleaf support is optionally included through the `ratpack-handlebars` and `ratpack-thymeleaf` modules accordingly. As demonstrated earlier in the book, if your project is built with Gradle, you can incorporate these modules by utilizing the `ratpack.dependency(..)` mechanism of the Ratpack Gradle plugin. The code in Example 6-12 shows the `dependencies` block of a Gradle build script with Handlebars.js and Thymeleaf support.

Example 6-12. Gradle build script with Handlebars.js and Thymeleaf

```
dependencies {
  compile ratpack.dependency("handlebars")
  compile ratpack.dependency("thymeleaf")
}
```

Each of the available templating engines offers the ability to resolve a template from within your project and provide it with a *view model* that will be used to hydrate the dynamic content. No matter what templating strategies you choose to employ within your project, it should be understood that the support for dynamically rendering templates comes from an integration with Ratpack's rendering infrastructure. That is to say, each of the template engines provides a static method to build a renderable object, which can then be passed to the `render` method within your handler.

To understand template rendering better, consider the project structure in Example 6-13, which shows a typical Groovy Ratpack application.

Example 6-13. Project structure with Groovy template

```
.
├── build.gradle
└── src
    └── ratpack
        ├── Ratpack.groovy
        └── templates
            └── welcome.html

3 directories, 3 files
```

Note that this time the project structure includes the *src/ratpack/templates/welcome.html* file. We will use this file to dynamically render HTML content from a view model as a Groovy text template. The template code in Example 6-14 demonstrates how we can use Groovy Strings (GStrings) to substitute variable content inline. (Note that all variable properties stem from the `model` variable.)

Example 6-14. Groovy text template (welcome.html)

```html
<!DOCTYPE html>
<html>
<head>
<title>${model.title}</title>  ❶
</head>
<body>
${model.welcomeMessage}  ❷
<footer>
${model.footerMessage}  ❸
</footer>
</body>
</html>
```

❶ Derive the `title` attribute from the view model.

❷ Insert the `welcomeMessage` variable into the `body`.

❸ Fill the footer with the `footerMessage`.

This template can be rendered as shown in Example 6-15.

Example 6-15. Rendering welcome.html

```groovy
import ratpack.groovy.template.TextTemplateModule

import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
```

```
  bindings {
    module(TextTemplateModule)
  }
  handlers {
    get {
      render(groovyTemplate(
                [title: "Hello, Ratpack!",
                 welcomeMessage: "Welcome to Learning Ratpack!",
                 footerMessage: "Ratpack is Great!"], "welcome.html"))
    }
  }
}
```

Running this application and opening a browser to *http://localhost:5050* will produce the HTML shown in Example 6-16.

Example 6-16. Rendered welcome.html

```
<!DOCTYPE html>
<html>
<head>
<title>Hello, Ratpack!</title>
</head>
<body>
Welcome to Learning Ratpack!
<footer>
Ratpack is Great!
</footer>
</body>
</html>
```

Groovy text templates have additional templating support, such as iterations and conditionals, that can satisfy relatively simplistic dynamic content needs. As your project requirements grow, however, you may find more robust templating support in the integrations offered with Handlebars.js and Thymeleaf.

Handlebars.js Support

As noted earlier, support for rendering Handlebars.js templates is provided as an optional module. As such, once the dependency is made available to your project, you will need to incorporate the `HandlebarsModule` into your application. The application shown in Example 6-17 demonstrates this.

Example 6-17. A ratpack.groovy file with HandlebarsModule

```
import ratpack.handlebars.HandlebarsModule

import static ratpack.groovy.Groovy.ratpack
import static ratpack.handlebars.Template.handlebarsTemplate
```

```
ratpack {
  bindings {
    module HandlebarsModule
  }
  handlers {
    get {
      render(handlebarsTemplate("welcome", [
        title: "Hello, Ratpack!",
        welcomeMessage: "Welcome to Learning Ratpack!",
        footerMessage: "Ratpack is Great!"], "text/html"))
    }
  }
}
```

Handlebars templates will by default be resolved from *$baseDir/handlebars*, but this can be overridden by configuring the HandlebarsModule in the bindings block, as shown in Example 6-18.

Example 6-18. Configuring the HandlebarsModule

```
ratpack {
  bindings {
    module(HandlebarsModule) {
      templatesPath "otherTemplatePath"  ❶
    }
  }
  handlers {
    // ... snipped ...
  }
}
```

❶ This path will be resolved relative to the base directory.

For the sake of simplicity, let's leave the templatesPath alone, and work from the default template directory. Now, we can imagine building on a project structure like the one shown in Example 6-19.

Example 6-19. Project structure with Handlebars template

```
.
├── build.gradle
└── src
    └── ratpack
        ├── handlebars
        │   └── welcome.hbs
        └── Ratpack.groovy

3 directories, 3 files
```

The *welcome.hbs* file can be constructed using Handlebars.js template expressions, as shown in Example 6-20.

Example 6-20. The welcome.hbs file

```
<!DOCTYPE html>
<html>
<head>
<title>Learning Ratpack</title> ❶
</head>
<body>
 ❷
<footer>
 ❸
</footer>
</body>
</html>
```

❶ This expression inserts the `title` variable from the handler's `render` call into the view.

❷ The body of the HTML page will be filled with the `welcomeMessage` we supplied.

❸ The footer will include the `footerMessage` key that we provided.

Ratpack's Handlebars.js integration provides an excellent foundation for web applications that need rich dynamic content capabilities. Even more than that, however, it can also provide a footprint for single-page applications (SPAs) that desire an isomorphic view layer. That is to say, web applications that provide some mixture of client-side and server-side rendering can make use of the same Handlebars.js templates.

Thymeleaf Support

Thymeleaf is a templating engine for the JVM that provides you with the ability to concisely craft markup using taglib-like notation within your view layer. In that respect, its functionality parallels that of JavaServer Pages (JSP), so it can be thought of as a comfortable adoption for those Java developers accustomed to developing JSP views.

As noted earlier, like the Handlebars.js support, Thymeleaf templating support is incoporated into your project as an optional dependency. Once the dependency is included in your project, you will need to incorporate the `ThymeleafModule` into your application. The application in Example 6-21 demonstrates incorporating Thymeleaf and a simple handler for rendering a *welcome.html* template.

Example 6-21. ratpack.groovy with ThymeleafModule

```groovy
import ratpack.thymeleaf.ThymeleafModule

import static ratpack.groovy.Groovy.ratpack
import static ratpack.thymeleaf.Template.thymeleafTemplate

ratpack {
  bindings {
    module ThymeleafModule
  }
  handlers {
    get {
      render(thymeleafTemplate([
              title: "Hello, Ratpack!",
              welcomeMessage: "Welcome to Learning Ratpack!",
              footerMessage: "Ratpack is Great!"], "welcome"))
    }
  }
}
```

By default, Thymeleaf templates will be resolved from the *$baseDir/thymeleaf* direc-
tory within your project. Like the HandlebarsModule, the ThymeleafModule can be
customized for a different template directory. The updated bindings code is shown
in Example 6-22.

Example 6-22. Configuring the ThymeleafModule

```groovy
ratpack {
  bindings {
    module(ThymeleafModule) {
      // Given this directive, Thymeleaf templates will
      // be resolved from $baseDir/otherTemplatesPath
      templatesPrefix "otherTemplatesPath"
    }
  }
  handlers {
    // ... snipped ...
  }
}
```

If we leave the templatesPrefix configuration directive as its default, then we will be
working with a project structure like the one shown in Example 6-23.

Example 6-23. Project structure with Thymeleaf templates

```
.
├── build.gradle
└── src
    └── ratpack
```

```
        ├── Ratpack.groovy
        └── thymeleaf
            └── welcome.html
```

```
3 directories, 3 files
```

The *welcome.html* file can then be constructed using the Thymeleaf notation, as shown in Example 6-24.

Example 6-24. Thymeleaf welcome.html

```html
<!DOCTYPE html>
<html>
<head>
<title th:text="${title}" />
</head>
<body th:text="${welcomeMessage}">
<footer th:text="${footerMessage}" />
</body>
</html>
```

In Thymeleaf, the `th:text` key is what is used to fill content into the element. As you can see from the example, the values to these expressions are able to be filled using the `${..}` notation, similar to Groovy text templates.

Thymeleaf templates have robust support for building views, including support for iterations, expressions, and internationalization. If you are considering Thymeleaf as the templating engine for your dynamic content, it is worth reviewing the project's documentation to see all the features it offers.

Groovy Markup Templates

When building a Groovy Ratpack project, in addition to Groovy text templates, you can elect to build dynamic content using the Groovy `MarkupTemplateEngine`, which provides a Groovy DSL for building markup. No additional project dependency is required to make use of `MarkupTemplate` types, however you must be sure to include the `MarkupTemplateModule` in your `bindings` block.

Like Groovy text templates, Groovy markup templates will be resolved from the *$baseDir/templates* directory. The syntax for rendering a Groovy markup template is similar to that of text templates, as is shown in Example 6-25.

Example 6-25. ratpack.groovy with MarkupTemplate

```groovy
import ratpack.groovy.template.MarkupTemplateModule

import static ratpack.groovy.Groovy.ratpack
import static ratpack.groovy.Groovy.groovyMarkupTemplate
```

```
ratpack {
  bindings {
    module(MarkupTemplateModule)
  }
  handlers {
    get {
      render(groovyMarkupTemplate([
              title: "Hello, Ratpack!",
              welcomeMessage: "Welcome to Learning Ratpack!",
              footerMessage: "Ratpack is Great!"], "welcome.gtpl"))
    }
  }
}
```

Given this application code, the corresponding project structure will look similar to the one depicted in Example 6-26.

Example 6-26. Project structure with MarkupTemplate

```
.
├── build.gradle
└── src
    └── ratpack
        ├── Ratpack.groovy
        └── templates
            └── welcome.gtpl

3 directories, 3 files
```

The *welcome.gtpl* file can be crafted using the `MarkupTemplateEngine`'s Groovy DSL. As such, to dynamically render the simple HTML page shown before, the contents of the template would look like that shown in Example 6-27.

Example 6-27. The welcome.gtpl file

```
html {
  head {
    title(title)
  }
  body {
    yield welcomeMessage
    footer {
      yield footerMessage
    }
  }
}
```

The `MarkupTemplateEngine` provides robust support for programmatic construction of content material using regular Groovy syntax. Additionally, it provides the ability

to include and render other templates, so you can build complex layouts using modular view components. If you are interested in building your dynamic content with the `MarkupTemplateEngine`, it is highly recommended that you review the documentation.

Conditionally Serving Content

Ratpack's handler chain makes it easy to conditionally serve content according to the attributes of a request. Scenarios where you may find this beneficial are those where you need to, for example, elect to serve a resource endpoint according to the security authorization of the requesting user. Another scenario may be where you want to serve different static assets according to the `User-Agent` header of an incoming request. For example, your project may have different static asset sets that are best suited to be served to Internet Explorer versus Firefox, or Chome and Safari. In reality, the programmatic flow of the handler chain can be leveraged in a wide variety of circumstances to choose what assets or endpoints are made available to clients.

Conditionally Scoping Resources

Consider the scenario where your application wants to make decisions as to what endpoints and resources are available to the currently logged-in user. In this case, a user named "admin" may need access to privileged resources that we would otherwise not want to make available to regular users. To start with demonstrating this example, we will need to include the framework's optional support for HTTP sessions. To accomplish this, we must incorporate the `ratpack-session` module into our project. The snippet in Example 6-28 shows a Gradle build script's `dependencies` block with the session support added.

Example 6-28. Gradle build script with ratpack-session

```
dependencies {
  compile ratpack.dependency("session")
}
```

With the session support added, we can build a `/login` endpoint that validates a user against a hardcoded username and password combination (proper application security will be covered in detail later in the book). The code in Example 6-29 demonstrates the simple login handler and stores the user in the current session.

Example 6-29. Admin login handler

```
import ratpack.session.SessionModule
import ratpack.session.Session
import ratpack.form.Form
```

```groovy
import ratpack.registry.Registry

import static ratpack.groovy.Groovy.ratpack

class User {
  String username

  boolean isAdmin() {
    return username == "admin"
  }
}

ratpack {
  bindings {
    module SessionModule
  }
  handlers {
    post("login") { Session session ->          ❶
      parse(Form).flatMap { form ->              ❷
        session.getData().map { sessionData ->   ❸
          if (form.username == "admin" && form.password == "password") {
            sessionData.set("username", "admin")      ❹
          } else {
            sessionData.set("username", "anonymous")   ❺
          }
        }
      }.then {
        redirect "/"   ❻
      }
    }
  }
}
```

❶ Build a POST handler to the `/login` endpoint and access the `Session` object from the registry.

❷ Parse the URL-encoded form variables.

❸ Access the `Session`'s data (the `getData()` call returns a `Promise`, because it may involve a blocking operation).

❹ Check if the user is an admin, and if so set the appropriate value of the `username` session key.

❺ If the user is not an admin, set the `username` session key to `anonymous`.

❻ Once the work is done on the session, redirect to /.

The example does not do much at this point, and indeed is incomplete in the flow (the call to redirect / will not match any handler in the chain). But given that we now have access to the `username` value in the session, we can do some work lower in the chain to discriminate on who the user is and to what resources they should have access. Building on this further, the handler chain snippet in Example 6-30 shows the downstream handlers that interpret the session data and make conditional routing decisions based on whether the user is an admin or not.

Example 6-30. Handler chain with conditional resources

```groovy
class User {
  String username

  boolean isAdmin() {
    return username == "admin"
  }
}

ratpack {
  bindings {
    module SessionModule
  }
  handlers {
    post("login") { Session session ->
      parse(Form).flatMap { form ->
        session.getData().map { sessionData ->
          if (form.username == "admin" && form.password == "password") {
            sessionData.set("username", "admin")
          } else {
            sessionData.set("username", "anonymous")
          }
          sessionData
        }
      }.then { sessionData ->
        redirect "/"
      }
    }
    all { Session session ->                              ❶
      session.getData().then { sessionData ->
        def usernameOption = sessionData.get("username")
        def user = new User(username: usernameOption.present ?   ❷
                                      usernameOption.get() :
                                      "guest"
        )
        next(Registry.single(User, user))                ❸
      }
    }
    when {                                               ❹
      def user = get(User)
      user.isAdmin()
```

```
  } {
    prefix("adminApi") { ❺
      // ... API handlers for admin user ...
    }
    files { ❻
      dir("admin").indexFiles("index.html")
    }
  }
  when { ❼
    def user = get(User)
    !user.isAdmin()
  } {
    files { ❽
      dir("user").indexFiles("index.html")
    }
  }
  }
 }
}
```

❶ Attach an `all` handler in the chain to process the `sessionData`.

❷ Apply the `username` field to a domain object that has some logic for informing if the user is an admin or not. Note that `SessionData#get` returns an `Optional` type, so here we also check if there is a value, and if not, we set the username to `guest`.

❸ Place the `User` object into the registry and delegate down the chain.

❹ Here, we use the `when` request flow handler to route to a new subchain if the user is an admin.

❺ In this subchain, we can define privileged resources for the admin user.

❻ We can also choose to serve static assets from a specific directory.

❼ If the user is not an admin, then we will delegate to this subchain.

❽ In the nonadmin subchain, we can choose an entirely different set of static resources that we will use.

To emphasize a point, this is a contrived example to demonstrate conditionally serving resources and assets according to some attribute of the request. This should not be considered a good example of how to secure your Ratpack applications. Chapter 12 will comprehensively cover Ratpack's support for authentication, authorization, and general security architecture.

This example can, however, be applied to a variety of use cases that involve introspecting the request as it flows through the chain, and making downstream decisions about what endpoints and assets are available to a request. The ability for the handler chain to make programmatic decisions about the flow of a request to the various endpoints opens a lot of opportunities for applications that are security centric or wish to have specific bindings for different client types.

Conditionally Serving Assets Based on Request Attributes

Obscuring resources for the sake of security is not the only use case where you would want to make a programmatic decision about what assets are served and when. As noted at the beginning of the section, you may find it valuable to bundle your static assets according to the capabilities of a client's browser. For example, consider a scenario where you may wish to have your view assets modularized and served depending on what `User-Agent` header is supplied as part of the request.

Given the demonstration in the prior subsection, it should be quite easy to recognize how to accomplish this using the `when` handler. For each of the `User-Agents` that we wish to support, we can define a condition in the predicate block of the `when` handler to ensure routing to the appropriate asset bundles. The handler chain in Example 6-31 is a depiction of this example.

Example 6-31. Handler chain with User-Agent asset routing

```
handlers {
  // ... application handlers go first ...

  when {
    request.headers.'user-agent' ==~ /.*MSIE.*/ ❶
  } {
    files {
      dir("msie").indexFiles("index.htm") ❷
    }
  }

  when {
    request.headers.'user-agent' ==~ /.*Chrome\\/[.0-9]* Mobile.*/ ❸
  } {
    files {
      dir("mobile/chrome").indexFiles("index.html")
    }
  }

  files { ❹
    dir("default").indexFiles("index.html")
  }
}
```

❶ Using the `when` handler, the predicate can perform a regular expression match against the contents of the `User-Agent`.

❷ If the `User-Agent` matches, in this case `MSIE`, then we elect to serve static content from the `$baseDir/msie` directory.

❸ The handler is used again, but this time to perform a regular expression match on a Chrome Mobile `User-Agent`.

❹ If none of the conditions matched, then we will serve static assets from the `default` directory.

With the inspect-and-route strategy for conditionally serving static assets, we can build robust backend and frontend applications that are designed to give the user the best experience possible.

Sending Files from Handlers

There may be scenarios where you find that you want to send a file that exists within your project's base directory, without it necessarily being served through the `files` method. A perfect example of this is demonstrated in , where in the case of a 404 response you want to explicitly send a *404.html* page back to the client. Within this section, we will cover the use of `Context#file` to statically serve project resources.

As you already know, every handler is given access to the `Context` object, which serves as the interface for working with the request's lifecycle. In addition to that, the `Context` also provides you with a shortcut to the `FileSystemBinding` interface so that you can succinctly work with your project-level resources.

Similar to the `files` handler chain method, the `Context#file` method allows you to resolve assets relative to the current filesystem binding. The application shown in Example 6-32 demonstrates using the `Context#file` method to respond to a request with a file from within the project.

Example 6-32. Sending a file from a handler

```
ratpack {
  handlers {
    get {
      render file("static/welcome.html")
    }
  }
}
```

It is really as simple as that. With the project structure shown in Example 6-33, the application will render the *src/ratpack/static/welcome.html* back to a caller.

Example 6-33. Project structure with welcome.html

```
.
├── build.gradle
└── src
    └── ratpack
        ├── Ratpack.groovy
        └── static
            └── welcome.html

3 directories, 3 files
```

Note that files can be sent as a response to requests using the `render` method on the `Context` object. The rendering infrastructure will inspect the file and set the appropriate `content-type` header on the response. Given that, *any* type of file is able to be served using the `Context#file` method in a handler, not just HTML resources. This gives your application a great deal of flexibility in understanding a request's properties to make a decision about what static content should be sent back.

Customizing 404 Behavior

Ratpack ships with default implementations of the `ratpack.error.ClientErrorHan dler` class for both development and production runtimes. You can provide your own implementation of this interface and add it to the server or context registry to customize the behavior when 404 responses are sent. You may wish to simply render a custom error page (the default development `ClientErrorHandler` renders a Ratpack-branded HTML response) or you may desire more robust handling logic that takes into account content and media types.

Consider a concise example of a basic Groovy Ratpack application, which has a single handler bound to the default route and serves a *welcome.html* page for visitors. The application in Example 6-34 shows this basic setup, but note that the `bindings` block now incorporates a `CustomErrorHandler` into the application.

Example 6-34. Simple ratpack.groovy with CustomErrorHandler

```groovy
import app.CustomErrorHandler
import ratpack.error.ClientErrorHandler
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    // Binds the ClientErrorHandler interface to the
```

```
    // project's CustomErrorHandler implementation.
    bindInstance(ClientErrorHandler, new CustomErrorHandler())
  }
  handlers {
    get {
      render(file("static/welcome.html"))
    }
  }
}
```

The `CustomErrorHandler` will be responsible for serving our project's specialized
404.html page. The implementation for this is shown in Example 6-35.

Example 6-35. CustomErrorHandler implementation

```
package app

import ratpack.error.ClientErrorHandler
import ratpack.handling.Context

class CustomErrorHandler implements ClientErrorHandler {
  @Override
  void error(Context ctx, int statusCode) throws Exception {
    ctx.render(ctx.file("static/404.html"))
  }
}
```

As you can see from the example, the `ClientErrorHandler` interface is not terribly
dissimilar from an ordinary `Handler`, and we get access to the `Context` object to han-
dle the request appropriately. The project structure for this demonstration is provided
in Example 6-36.

Example 6-36. Project structure with custom 404 logic

```
.
├── build.gradle
└── src
    ├── main
    │   └── groovy
    │       └── app
    │           └── CustomErrorHandler.groovy
    └── ratpack
        ├── Ratpack.groovy
        └── static
            ├── 404.html
            └── welcome.html

6 directories, 5 files
```

If you start this application and open a browser to *http://localhost:5050*, you will be immediately presented with the contents of the *static/welcome.html*. If you then navigate to *http://localhost:5050/foo*, you can see the `CustomErrorHandler` is invoked, and the project's *static/404.html* page is rendered instead.

Cache Control

It is important to allow static assets that do not change to be cached on the client side. Subsequent page loads will then be free to use assets from the browser's cache, thus speeding page loads and reducing network traffic. To understand the application of the cache headers a little better, examine the application shown in Example 6-37.

Example 6-37. Application with cache headers

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    // ... Application handlers go first ...

    all { ❶
      int cacheTime = 60 * 60 * 24 * 365 // one year
      response.headers.add("Cache-Control", "max-age=$cacheTime, public")
      next()
    }
    files { ❷
      dir("static").indexFiles("index.html")
    }
  }
}
```

❶ Add an `all` type handler into the chain to be invoked with every request that makes it through to the `files` handler.

❷ The `files` handler, as always, will come last in the chain.

With the cache headers set, any request to serve static content from the application will have the appropriate `Cache-Control` header applied. In this example, we set the cache time to one year, so future requests for the same static content will not incur any network traffic (provided the browser has them stored).

For demonstration's sake, we can say that the project structure for this application is like the one shown in Example 6-38.

Example 6-38. Cache-Control project structure

```
.
├── build.gradle
└── src
    └── ratpack
        ├── Ratpack.groovy
        └── static
            ├── img
            │   └── ratpack.png
            └── index.html

4 directories, 4 files
```

The *index.html* file can be simple enough to demonstrate the browser cache, and may look like that shown in Example 6-39.

Example 6-39. The index.html file

```html
<!DOCTYPE html>
<html>
<head>
    <title>Welcome!</title>
</head>
<body>
Welcome to the application!
<br/>
<img src="/img/ratpack.png">
</body>
</html>
```

If you start this application and open a web browser to *http://localhost:5050*, you will see the *index.html* welcome page and the *ratpack.png* image. Opening your browser's development tools and refreshing the page, you will see that the refresh did not need to again request the *index.html* or the *ratpack.png*, because the server has responded with the Cache-Control headers.

Note that your browser's configuration may disable cache on localhost pages. If this is the case, then you will need to explicitly unset the "Disable Cache" option, as the lack of a check in the box demonstrates in Figure 6-1.
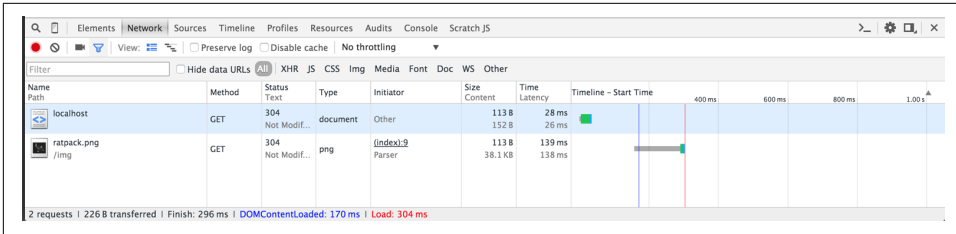
Figure 6-1. Chrome Dev Tools showing cached response

Asset Pipeline

The Asset Pipeline is a third-party module for Gradle and Ratpack that provides the ability to precompile, digest, and serve assets from within your Ratpack applications. For any application that has requirements for generating and serving static content, it should be considered a best practice to incorporate the Asset Pipeline into your project.

The Asset Pipeline Gradle plugin gives your application the ability to pre-process static content from higher-level formats. Example 6-40 demonstrates incorporating it into your Ratpack project. In the case of CSS resources, the Asset Pipeline is capable of compiling assets built with SASS, SCSS, or LESS into CSS files. Similarly, JavaScript resources that are built with CoffeeScript and Handlebars are able to be compiled and used as regular JS files. The Asset Pipeline additionally provides URL replacement within your static content to ensure that links are properly resolved when assets are used. The Asset Pipeline gives you a high level of flexibility when developing content for the Web.

Example 6-40. Gradle build script with Asset Pipeline dependencies

```
buildscript {
  ext {
    assetPipelineVer = "2.7.2"
  }
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
    classpath "com.bertramlabs.plugins:asset-pipeline-gradle:${assetPipelineVer}" ❶
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'
apply plugin: 'asset-pipeline' ❷

repositories {
```

```
    jcenter()
}

dependencies {
  compile "com.bertramlabs.plugins:ratpack-asset-pipeline:${assetPipelineVer}" ❸
}
```

❶ The first part to incorporating asset compilation into your project's build is to
    add the `asset-pipeline-gradle` build script dependency shown here.

❷ With this in place, here we apply the `asset-pipeline` plugin. This will give our
    project all the tasks necessary to build our static assets.

❸ Finally, we add the `ratpack-asset-pipeline` dependency to the project. This
    provides the necessary module and components to add to our application.

Example 6-41 shows a simple application using Asset Pipeline.

Example 6-41. Asset Pipeline ratpack.groovy file

```
import asset.pipeline.ratpack.AssetPipelineModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(AssetPipelineModule) { c ->  ❶
      c.sourcePath "../../../src/assets"  ❷
    }
  }
  handlers {
    get {
      redirect("/assets/index.html")  ❸
    }
  }
}
```

❶ With the necessary dependencies in place, we will add the `AssetPipelineModule`
    to our project.

❷ Note that we use the module's configuration to specify where the source assets
    exist.

❸ This line demonstrates redirecting an incoming request of `/` to the project's
    index.html file.

The `AssetPipelineModule` applies the `AssetPipelineHandler` to your handler chain
for serving the compiled assets. By default, the asset handler chain adds an `/assets`

endpoint to your application, but this can be configured as part of the `module` defini-
tion in the `bindings` block. For example, the code in Example 6-42 demonstrates
overriding the `url` field on the configuration class to serve assets directly from the
root of the application.

Example 6-42. Configuring the AssetPipelineModule

```groovy
import asset.pipeline.ratpack.AssetPipelineModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(AssetPipelineModule) {
      url "/"
    }
  }
  handlers {
    // ... Application handlers ...
  }
}
```

The Asset Pipeline Gradle plugin will, by default, look for assets in the *src/assets*
directory. Stylesheets will typically go in *src/assets/stylesheets*, JavaScript resources will
go in *src/assets/javascripts*, and HTML resources will go in *src/assets/htmls*. It is
important to note that the Asset Pipeline compiler will flatten this directory structure
and serve assets from those nested directory structures as top-level links. That is to
say, given the prior example where we overrode the `url` configuration to `/`, a Java-
Script file homed at *src/assets/javascripts/application.js* will be accessible at the URI */
application.js*. If your application requires additional structure, you can create subdir-
ectories within each of the asset groups. For example, *src/assets/javascripts/js/applica-
tion.js* will be served as */js/application.js*.

It is also worth noting that development time asset compilation can be tricky depend-
ing on how your project is compiled and run. If you are running your application
with `gradle -t run`, for example, then the assets for your development runtime will
exist in a location within the *build/* tree of your project's root. The `DevelopmentAsse
tHandler` looks for assets in the directory specified in the `AssetPipelineModule` con-
fig class's `sourcePath` variable, so you may need to tune that property according to
your development runtime. For example, the *ratpack.groovy* file shown in
Example 6-43 demonstrates overriding the `sourcePath` to look for assets higher up in
the runtime tree.

Example 6-43. Overriding AssetPipelineModule sourcePath

```
import asset.pipeline.ratpack.AssetPipelineModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module(AssetPipelineModule) {
      url "/"
      sourcePath "../../../src/assets"
    }
  }
  handlers {
    // ... Application handlers ...
  }
}
```

For any real-world web application that consists of an extensive frontend, it is recommended to leverage the Asset Pipeline. Simply put, the flexibility and robustness it offers to compile, minify, uglify, and digest assets are capabilities that your application is going to want. Even if you do not use additional asset types, like SASS, LESS, or CoffeeScript, the build-time digesting will allow your assets to be served properly every time when you go to production.

Chapter Summary

This chapter has covered the wide range of features that Ratpack offers for building applications that provide and utilize static content. Ratpack's commitment to being a feature-rich web application framework is demonstrated by its ability to generate content dynamically, to serve assets from your project's base directory, to make programmatic decisions about what assets are served and when, to customize 404 behavior, and to integrate with powerful utilities like the Asset Pipeline. This chapter's coverage of working with static assets will set you on a path to developing rich, content-first web applications.

Asynchronous Programming, Promises, and Executions

Asynchronous frameworks for the JVM are plagued by the fact that Java has no language-level support for continuations. This means that when asynchronous APIs are employed, a callback handler (or "completion handler") must be provided to the receiver in order to return the data to the caller. This invoke-and-callback process introduces a nondeterministic *data flow*, where concurrency must be carefully considered when building and returning data. It also introduces nondeterminism to the processing *control flow*, where it can be difficult, if not impossible, for the framework to know whether an asynchronous operation is still processing.

Consider the code in Example 7-1, which outlines the contract for an asynchronous service, and shows the associated user code.

Example 7-1. An asynchronous demonstration using a callback

```java
public interface AsyncDatabaseService {
  void findByUsername(String username, Consumer<User> callback); ❶
}

RatpackServer.start(spec -> spec
  .registry(...)
  .handlers(chain -> chain
    .get(":username", ctx -> {
      AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
      String username = ctx.getPathTokens().get("username");
      db.findByUsername(username, user -> { ❷
        ctx.render(user);
      });
    })
```

```
  )
);
```

❶ Let's assume for the sake of conversation that the `AsyncDatabaseService` pro-vides the ability to look up a `User` from a database, based on the username. The contract for this specifies that callers provide the `username`, and also the callback function that will be invoked once the asynchronous call to the database has completed.

❷ Here, we gather the `username` argument from the handler's `pathTokens`, and we invoke the `findByUsername` call with the necessary arguments.

This is a simple enough example, and even looks somewhat similar to the Ratpack code that you have already seen thus far. The trained eye may be able to recognize that the response processing has been wrapped inside the `Consumer`, thereby taking the control flow away from Ratpack. Because there is no way to know when the `Consumer` will be invoked, Ratpack is completely unaware if a request is still being pro-cessed, or if there was an error in the `findByUsername` processing that will cause the callback to never be invoked. Without being able to manage control flow, the only option is for the framework to wait for a period of time, and if no response is sent out, then assume that the handler did not send one. This has practical problems in both development and end-user experience, but luckily Ratpack has an answer to this problem.

Before we get into the solution, let's make the problem a little more realistic and add some additional methods. It is not at all uncommon that the construction of a domain object makes multiple database calls to gather all the data necessary. If we consider again the `AsyncDatabaseService`, and suppose that we want to capture the `UserProfile` for a given username, but the relational structure of the database requires that we first capture the `User` object to get the profile's reference, then we can see how the processing becomes problematic.

Example 7-2 shows the modifications to the `AsyncDatabaseService` with the new contract added.

Example 7-2. AsyncDatabaseService updates

```
public interface AsyncDatabaseService {
  void findByUsername(String username, Consumer<User> callback);
  void loadUserProfile(Long profileId, Consumer<UserProfile> callback);
}
```

If you are coming from a synchronous programming background or are not familiar with the complexities of asynchronous data composition, your first logical approach to the problem may be to simply capture the `UserProfile` in the outer scope and

inline the calls to the database. Example 7-3 shows the handler code you might choose to implement this approach. Those familiar with asynchronous programming will likely see the problem here, which is that asynchronous calls return immediately, with the contract that they will invoke the provided callback once they have the data.

Example 7-3. AsyncDatabaseService in handler

```
get(":username", ctx -> {
  AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
  String username = ctx.getPathTokens().get("username");

  User user;
  db.findByUsername(username, u1 -> user = u1); ❶
  db.loadUserProfile(user.getProfileId(), p1 -> { ❷
    ctx.render(p1);
  });
})
```

❶ The call here, although capturing the returned User object and assigning it to the user variable in the outer scope, has no guarantee of fulfillment before the call to loadUserProfile is made.

❷ Because loadUserProfile depends on the user object being properly assigned, we will likely see a NullPointerException here.

To those initiated with asynchronous programming, the problem of scoping and callbacks is well known. A "solution" to this problem may result in nesting the second call within the Consumer of the first call. Although workable, this approach results in nested code blocks that are difficult to maintain and harder to debug when problems do arise. Example 7-4 depicts this strategy.

Example 7-4. Nested async calls

```
get(":username", ctx -> {
  AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
  String username = ctx.getPathTokens().get("username");

  db.findByUsername(username, u1 -> { ❶
    db.loadUserProfile(u1.getProfileId(), profile -> { ❷
      ctx.render(profile); ❸
    });
  });
})
```

❶ In this demonstration, the call to get the User object is made here.

❷ Upon its completion, the call to get the `UserProfile` is made.

❸ Upon the `loadUserProfile` call's completion, the call to render the profile is made.

Although we can be sure that the data will be properly scoped when it is needed, managing data flow through callback nesting presents a disjointed control flow. If you envision following this practice for three, four, or five asynchronous calls, you can begin to imagine how deep the nesting will go, making it difficult, if not impossible, to trace the data flow. Those with JavaScript development experience are likely already aware of the problems presented by asynchronous callbacks.

Promises: A Better Approach to Async Programming

As you have already been exposed to Ratpack `Promises` at this point, you can probably see how the prior examples can be ameliorated by changing the contracts on the `AsyncDatabaseService` to work with promises instead of callbacks. For completeness, let's outline what those changes look like here. If we change the `AsyncDataSer` `vice`, as depicted in Example 7-5, to return `Promise` objects for their respective asynchronous calls, we can begin to see how the control flow is simplified.

Example 7-5. AsyncDatabaseService using promises

```
public interface AsyncDatabaseService {
  Promise<User> findByUsername(String username)
  Promise<UserProfile> loadUserProfile(Long profileId)
}
```

With the contract on the service changed, you can modify the corresponding handler logic to instead *stream* the resulting data to processing functions, once it becomes available, as shown in Example 7-6. The first benefit that becomes quickly obvious is the code is clearer and easier to follow.

Example 7-6. Streaming data from promises

```
get(":username", ctx -> {
  AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
  String username = ctx.getPathTokens().get("username");

  db.findByUsername(username).flatMap(user -> {   ❶
    return db.loadUserProfile(user.getProfileId());   ❷
  }).then(profile -> {   ❸
    ctx.render(profile);
  });
})
```

❶ The call to get the `User` object is made here, and once that data is available, it is streamed to the provided function.

❷ The resulting data from the `findByUserName` call is made available so that we can use it to get the `UserProfile`. These calls use the `flatMap` operator on `Promise` to transform the value of the stream by chaining the two promises.

❸ Shown here, the subscription to that result is then yielded the resulting `profile`, which can be rendered.

At first, this may not look better than the nested callback approach, but if you consider a more complex stream that makes three or four asynchronous calls, we can see how much better the code reads when using `Promise` types. Example 7-7 shows what many more asynchronous calls looks like using the callback strategy from before.

Example 7-7. Async composition with callbacks

```
get(":username", ctx -> {
  AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
  String username = ctx.getPathTokens().get("username");

  db.findByUsername(username, u1 -> {
    db.loadUserProfile(u1.getProfileId(), profile -> {
      db.loadUserFriends(profile.getFriendIds(), friends -> {
        db.loadPhotosFromFriends(friends.getPhotoIds(), photos -> {
          ctx.render(photos);
        });
      });
    });
  });
})
```

As you can see, the depth of callbacks grows further, making the resulting handler logic obscure. The same composition of asynchronous calls using `Promise` types can be seen in Example 7-8.

Example 7-8. Async composition with promises

```
get(":username", ctx -> {
  AsyncDatabaseService db = ctx.get(AsyncDatabaseService.class);
  String username = ctx.getPathTokens().get("username");

  db.findByUsername(username).flatMap(user -> {
    return db.loadUserProfile(user.getProfileId());
  }).flatMap(profile -> {
    return db.loadUserFriends(profile.getFriendIds());
  }).flatMap(friends -> {
    return db.loadPhotosFromFriends(friends.getPhotoIds());
```

```
    }).then(photos -> {
      ctx.render(photos);
    });
})
```

The stream of executions is much more inlined now, making the code more readable and easier to debug. However, `Promise` types in Ratpack are not just abstractions over callbacks, as you may find with other frameworks. Indeed, they represent a processing construct that is much more aligned with the concept of a continuation. Each of the points where an asynchronous call is made denotes a frame of the continuation. During the invocation of each frame, the continuation is suspended until its operation returns a value (thus fulfilling the promise). The serialization of the asynchronous calls gives your application a deterministic control flow, where you can rely on the fact that the prior operation has completed before carrying on to the next.

Execution Model

Ratpack's execution model is an implementation of a continuation, and the `Promise` type is used to represent the distinct frames of the continuation. When a `Promise` type is used, a processing stream is created from the asynchronous call and the corresponding user code, which is denoted by the subscription, or simply the function supplied to the `then(..)` method. Each `Promise` and its corresponding subscription in an execution comprise a *stream* of processing, and the execution ensures that streams are processed in order.

The processing calls that make up a stream are placed into a stack, and that stack is correspondingly popped to invoke the respective code frame in order. An execution, as demonstrated by Example 7-8, is made up of many streams (each represented by a `Promise`). The streams are serialized within an *execution* to provide the deterministic asynchronous processing shown earlier. Example 7-9 shows the execution layout and processing flow from that example.

Example 7-9. Execution breakdown

```
Execution
|
|____Stream
   |
   |__end-of-frame marker
   |
   |__.then(p1 -> container.photos = p1)
   |
   |__db.loadPhotosFromFriends(friends.getPhotoIds())
|__Stream
   |
   |__end-of-frame marker
```

```
 |
 |__.then(f1 -> container.friends = f1)
 |
 |__db.loadUserFriends(profile.getFriendIds())
|__Stream
 |
 |__end-of-frame marker
 |
 |__.then(p1 -> container.profile = p1)
 |
 |__db.loadUserProfile(user.getProfileId())
|____Stream
 |
 |___end-of-frame marker
 |
 |___.then(u1 -> container.user = u1)
 |
 |__db.findByUsername(username)
```

In this tree, each element in a given *stream* is known as an *execution segment*. Streams are processed in last in, first out (LIFO) order, so the stream's *end-of-frame marker* is always placed at the beginning of a stream to inform the execution when a stream has reached completion. During processing, when an execution segment is "overrun" but has not yet fulfilled its value, the execution knows that the stream is still processing. When all streams are "drained," the execution knows that the processing work is finished.

By allowing Ratpack to manage control flow through its execution model, applications get the benefit of its awareness of the processing that is being done. The most poignant example of this awareness is Ratpack's ability to ascertain whether a request has sent a response or not. As noted earlier, in most asynchronous frameworks, the framework has no awareness as to whether user code has sent a response to a client or not. It simply has to rely on timeout techniques to determine when processing is completed or not.

In Ratpack, however, because the execution model is aware of the work that is being done by the handler, it can ascertain whether handler logic is still processing. This is most practically seen by Ratpack's ability to inform a client, in the form of an error response, that the handler logic did not send a response. In this paradigm, requests are able to be smartly handled based on data, and not on response-writing timeouts.

Scheduling Execution Segments for Computation or I/O

In addition to serializing execution segments, the execution gives the stream the ability to inform as to what type of work the execution is doing, so that work can be executed on the proper thread pool. As discussed earlier in the book, Ratpack is able to make *any* API asynchronous. This includes APIs that perform I/O-bound or blocking

operations. For performance reasons, it is of critical importance that blocking operations do not take place on request-taking threads, so Ratpack provides a `Promise` type data construct to represent a call to a blocking operation, through the `Blocking` class.

If we consider again the examples demonstrated earlier in this chapter, but this time modified to utilize a `BlockingDatabaseService`, we can see how the `Blocking` API can be used to turn those blocking database calls into asynchronous execution segments. Example 7-10 demonstrates the corresponding handler logic.

Example 7-10. Blocking API

```java
package app;

import ratpack.server.RatpackServer;
import java.lang.Exception;
import ratpack.exec.Blocking;
import ratpack.exec.Operation;

public class Main {

  private static class ObjectContainer {
    User user;
    UserProfile profile;
    Friends friends;
    Photos photos;
  }

  public static void main(String[] args) throws Exception {
    RatpackServer.start(spec -> spec
      .registryOf(...)
      .handlers(chain -> chain
       // ... other application handlers here ...

        .prefix("photos", pchain -> pchain
          .get(":username", ctx -> {
            BlockingDatabaseService db = ctx.get(BlockingDatabaseService.class);
            String username = ctx.getPathTokens().get("username");

            final ObjectContainer container = new ObjectContainer(); ❶

            Blocking.get(() -> db.findByUsername(username))
              .then(u1 -> container.user = u1);

            Blocking.get(() ->
              db.loadUserProfile(container.user.getProfileId())
            ).then(p1 -> container.profile = p1);

            Blocking.get(() ->
              db.loadUserFriends(container.profile.getFriendIds())
            ).then(f1 -> container.friends = f1);
```

```
        Blocking.get(() ->
          db.loadPhotosFromFriends(container.friends.getPhotoIds())
        ).then(p1 -> container.photos = p1);

        Operation.of(() -> ctx.render(container.photos)).then(); ❷
      })
    )
  )
  );
 }
}
```

❶ Note that here we have to wrap the objects of our composition in a container object, since Java lambdas require mutable variables to be effectively final. While we cannot modify the `ObjectContainer` class from within our lambda expressions, we can modify its properties directly.

❷ Our final call to render the `Photos` object is wrapped in an `Operation` type, which allows Ratpack's execution model to ensure this code is executed last. By doing so, we are given determinisim as to the availability of the `container.pho tos` property, since we know it will have been set by the prior execution segment.

The call to `Blocking.get(..)` that wraps each of the `BlockingDatabaseService` calls returns a `Promise` type that is specifically designed to inform the execution that any of the code contained therein should be executed on *blocking thread*. Ratpack manages two separate thread pools: one for computation and request taking, and another for I/O-bound and blocking operations. They are known as the *request taking* and *blocking* thread pools, respectively.

When the stream is created for the blocking call, it opens the door for the `Promise` that is generated by the `Blocking.get(..)` call to schedule the work on the blocking thread pool. When the blocking execution segment completes, the data returned is then scheduled back to the *originating computation thread*.

Rescheduling the subscription's execution segment to the originating thread is another important aspect to the execution model. For performance reasons, we will want to ensure that the subscription's execution segment that follows the blocking segment does not incur another context switch to resume computation processing. In that respect, Ratpack is said to provide *thread affinity* for computation execution segments.

Without specifying that an execution segment is going to perform a blocking operation, the segment will be scheduled by default to the computation thread pool. Computation and blocking operations can be seamlessly intermixed in an execution without the need for user code to interact with a separate thread pool.

This ability to provide blocking operations with a safe execution environment means that legacy code and third-party libraries that are not designed for asynchronous work can still be safely utilized in your Ratpack applications. Libraries that have no inherent asynchronous capabilities (e.g., ones that are built on JDBC) can still be utilized in an asynchronous way by your handler and service logic.

Leveraging Executions on Unmanaged Threads

Ratpack's execution model can still be leveraged on threads that are not part of the normal request processing flow. This is important for services that provide background processing capabilities within your Ratpack application. Consider a scenario where your application needs a background thread for caching or health checks, and you want to make use of services and APIs that utilize `Promise` type responses.

A perfect example of background processing that needs Ratpack's execution model is a periodic thread that uses the framework's nonblocking HTTP client library to call an external HTTP service and store the resulting data in local cache. The background thread will not already have an execution bound, so we can create one on our own and bind it accordingly. As is true with most things in Ratpack, the semantics for accomplishing this are concise.

Consider the code in Example 7-11, which depicts a service class that uses Ratpack's computation thread pool to schedule a periodic call to the external service and cache the results locally.

Example 7-11. Execution forking

```
public class UserService implements Service, Runnable { ❶
  static String USER_SERVICE_URI = "https://user-service.internal";

  private final HttpClient httpClient;
  private final ObjectMapper mapper;

  private final Map<String, User> userCache = Maps.newConcurrentMap();

  @Inject
  public UserService(HttpClient httpClient, ObjectMapper mapper) { ❷
    this.httpClient = httpClient;
    this.mapper = mapper;
  }

  @Override
  public void onStart(StartEvent event) {
    ExecController execController = event.getRegistry().get(ExecController.class); ❸
    execController.getExecutor()
        .scheduleAtFixedRate(this, 0, 60, TimeUnit.SECONDS); ❹
  }
```

```
  @Override
  public void run() {
    Execution.fork().start(e -> {  ❺
      httpClient.get(new URI(USER_SERVICE_URI+"/users")).onError(t ->
        t.printStackTrace()
      ).map(response -> {  ❻
        return (List<User>)mapper
          .readValue(response.getBody().getBytes(),
            new TypeReference<List<User>>() {});  ❼
      }).then(users -> {  ❽
        for (User user : users) {
          userCache.put(user.getUsername(), user);
        }
      });
    });
  }

  public User findByUsername(String username) {
    return userCache.containsKey(username) ? userCache.get(username) : null;
  }

  public List<User> list() {
    return Lists.newArrayList(userCache.values());
  }
}
```

❶ The UserService class starts out by implementing the ratpack.server.Service
   special type and the Runnable interface. We will use the class itself as the Runna
   ble for the periodic operation. Through the Service interface, Ratpack provides
   startup and shutdown hooks to when the application starts and stops, respec-
   tively.

❷ We know that we will need access to the ratpack.http.HttpClient and to Jack-
   son's ObjectMapper class to transform data from the external service, so here
   those resources are dependency injected with Guice. In the onStart method, we
   can get access to the server registry through the StartEvent.

❸ Via the registry, we get access to the ratpack.exec.ExecController. The Exe
   cController provides us with access to the *computation* ExecutorService,
   which is a ScheduledExecutorService.

❹ We use the executor to schedule the class's run method at a fixed rate.

❺ Because the thread that is produced by the executor is outside the context of a
   request's lifecycle, we need to explicitly start a new execution on it. Ratpack pro-
   vides a static method, Execution.fork(), to build an execution on the current

thread. When we call the `start` method on the `ExecutionBuilder`, the execution is tied to the current thread. We are given access to the execution object, denoted by the `e` variable, but we do not explicitly need it, as `Promise` types will automatically be tied to the thread's current execution.

❻ The `httpClient.get(..)` call here produces a `Promise`. We can use the `map` operator to transform the `ReceivedResponse` type, `response`, into a `List<User>` object by passing the bytes of the response body down the stream.

❼ Here, we use the Jackson `ObjectMapper` to transform the response into a list of `User` objects. The resulting list will then be passed downstream.

❽ Here, using the `then` method, we subscribe to the stream. The subscription's execution segment is then responsible for mapping the users into the `userCache`.

From here, handlers can get access to a `User` object or a list of `User` objects through the `findByUsername` and `list` methods, respectively. These calls will then pull the values from the local cache. All of the work has been done by the periodic background thread, where we continue to use `Promise` types and the execution model, even outside the normal processing flow.

Error Handling

Error handling is an important aspect to the flow of data. As you build increasingly complex processing streams, Ratpack will always ensure that errors are propagated appropriately. To that extent, error handlers can be defined in both the global context of an execution as well as the local context of a promise. To illustrate this, consider the first example of global error handling within the application. If we define a `ServerErrorHandler` in the user registry, then we can capture all unhandled exceptions throughout the application. Example 7-12 demonstrates this with a simple example.

Example 7-12. Adding ServerErrorHandler

```
import ratpack.error.ServerErrorHandler
import ratpack.handling.Context

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bindInstance ServerErrorHandler, new ServerErrorHandler() { ❶
      @Override
      void error(Context ctx, Throwable throwable) throws Exception { ❷
        ctx.response.status(500)
        ctx.render(ctx.file("errors/500.html"))
```

```
        }
      }
    }
    handlers {
      get {
        throw new RuntimeException() ❸
      }
    }
  }
}
```

❶ We bind the `ratpack.error.ServerErrorHandler` to a new instance.

❷ Within the `error` method, we are given access to the request's `Context` as well as the `Throwable` object that produced the error.

❸ For demonstration's sake, here we throw an unchecked exception.

Running this application and accessing the default route (Figure 7-1) will generate the error and activate our error handler. In this example, the `ServerErrorHandler` sets the response status to `500` and accordingly sends back an HTML file. Without this handler, in development mode, you will see the stacktrace that was produced; in production, you will see a blank page. Obviously, for a production application you will want to give your users something more than a blank screen, so here we can specify that we want to send back an HTML file instead.
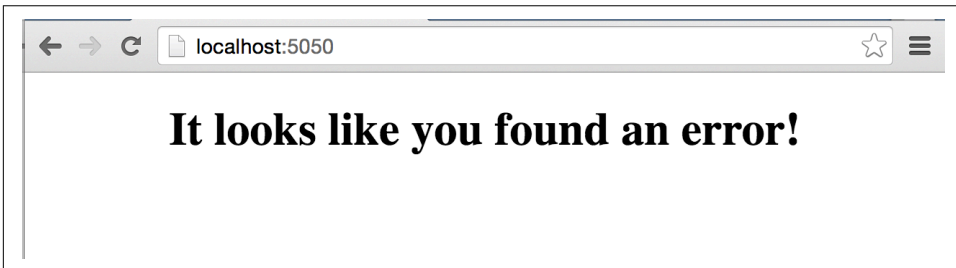
Figure 7-1. 500 error page

Given that global error handling takes place within the scope of the request-response lifecycle, you are able to perform more complex processing when an error is captured. For example, you could publish a metric, send an email, write to a database table, or all three. Additionally, you can choose to render more elaborate error screens using Ratpack's templating options.

Execution-Wide Error Handling

Error handlers can also be attached at the level of a forked execution. For example, if you have a background process that is forking executions, as discussed earlier, an

execution-wide error handler can be attached to process any exceptions that were unhandled, in a manner similar to how `ServerErrorHandler` works. The main difference between how these two strategies work is that the former handles errors that are outside the context of a typical request-response lifecycle.

When you go about forking your own execution, prior to the `start` call you can attach the execution-wide error handler using the `onError` method. The `Throwable` that was thrown is provided as an argument. If we expand Example 7-11 from earlier, we can see how Example 7-13 does this.

Example 7-13. Forked execution with error handling

```java
public class UserService implements Service, Runnable {
  static String USER_SERVICE_URI = "https://user-service.internal";

  private final HttpClient httpClient;
  private final ObjectMapper mapper;

  private final Map<String, User> userCache = Maps.newConcurrentMap();

  @Inject
  public UserService(HttpClient httpClient, ObjectMapper mapper) {
    this.httpClient = httpClient;
    this.mapper = mapper;
  }

  @Override
  public void onStart(StartEvent event) {
    ExecController execController = event.getRegistry().get(ExecController.class);
    execController.getExecutor()
        .scheduleAtFixedRate(this, 0, 60, TimeUnit.SECONDS);
  }

  @Override
  public void run() {
    Execution.fork().onError(t -> {   ❶
      t.printStackTrace();
    }).start(e -> { //
      httpClient.get(new URI(USER_SERVICE_URI+"/users")).map(response -> {
        return (List<User>)mapper
            .readValue(response.getBody().getBytes(),
                new TypeReference<List<User>>() {});
      }).then(users -> {
        for (User user : users) {
          userCache.put(user.getUsername(), user);
        }
      });
    });
  }
```

```
  public User findByUsername(String username) {
    return userCache.containsKey(username) ? userCache.get(username) : null;
  }

  public List<User> list() {
    return Lists.newArrayList(userCache.values());
  }
}
```

❶ Prior to the call to `start`, we attach our execution-wide error handler using the `onError` method.

In this demonstration, we are simply printing the stacktrace from the `Throwable`, but because error handling is applied in the context of the forked execution, you have the ability to perform more elaborate error processing.

Promise Error Handling

Global and execution-wide error handling should really only be used as a catch-all for uncaught exceptions. You will undoubtedly have the scenario where a `Promise` processing stream needs to have its own logic for error handling. Attaching error handling to a `Promise` type uses similar semantics to how we attached it to the execution, but this time the error is captured for the promise's localized stream. This gives you the ability to perform processing that is more specific to a promise's given function.

Example 7-14 demonstrates attaching error handling to a `Promise` type by using the `onError` method. Like before, the `Throwable` is provided to the error handling logic.

Example 7-14. Promise error handling

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    get {
      request.body.map { ❶
        throw new RuntimeException() ❷
      }.onError { t -> ❸
        render "There was an error."
      }.then { ❹
        render "There was not an error."
      }
    }
  }
}
```

❶ To illustrate this example, we will use the `getBody` method on `request`, which returns a `Promise`. We create a processing stream using the `map` method so that we can simulate raising an exception.

❷ We can simulate a failure at this point by throwing a `RuntimeException`.

❸ Here, we attach the error handler to the promise stream using the `onError` method. The argument provided is the `Throwable` type. Within the processing logic, here we choose to render back a simple message, but like before, more elaborate processing can take place in this block.

❹ We subscribe to promise as we normally would. Note that given this processing stream, we should never hit this block.

If we run this application with the simulated error and open a browser to the application, we will be met with the message that was rendered from the error handling block (Figure 7-2).
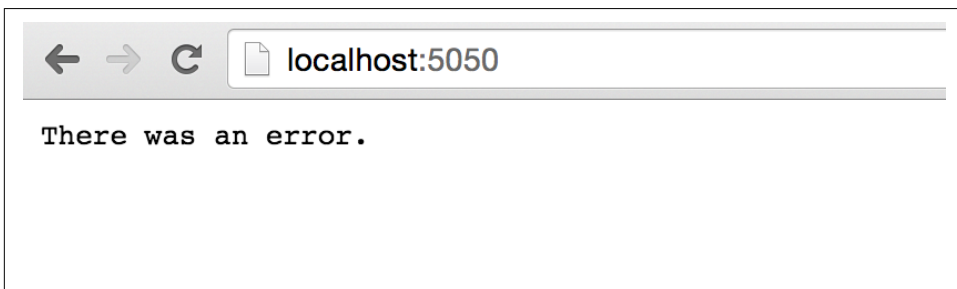
Figure 7-2. Promise error handling

It is important to note that if the error handling logic throws a new exception, the original exception will be wrapped, so as to provide proper stack tracing. Using `Promise`-level error handlers provides an excellent means to localize your error handling logic according to the operation that was underway. It is a good practice to ensure that you have error handling throughout your application, so adding a `Server ErrorHandler` that will respond to all uncaught exceptions in conjunction with `Promise`-level error handling ensures your application will always respond in a user-friendly way.

Creating Promises on Your Own

It will not always be that you are in a scenario where you are given an interface with a Ratpack `Promise` type. In fact, it is likely that most integrations with third-party libraries should be adapted to work under the premise of a `Promise`, especially those libra-

ries that have their own threading and execution model. For that, Ratpack provides a simple means by which promises can be created from both synchronous (computation) and asynchronous (I/O) calls that are not managed in the framework's own execution model.

Promises from Synchronous Calculations

The simplest way to think about creating a `Promise` type in a *synchronous* manner is by understanding that the underlying calculation to get the value that the promise will hold is computation-bound. That is to say, synchronous promises can be built when their value is not an I/O-bound operation that will block the existing thread. To demonstrate this clearly, consider the interface in Example 7-15.

Example 7-15. The NumberService interface

```
package app

import ratpack.exec.Promise

interface NumberService {

  Promise<Integer> getRandomNumber()
}
```

The `getRandomNumber` method returns a `Promise<Integer>`, which the concrete implementation can choose to either calculate computationally or perform a blocking operation to resolve. The reason for the latter case might be the need for high consistency in random number generation across multiple nodes. For the purposes of producing a synchronous promise, let's consider that we simply want to calculate the random number using `java.util.Random` as shown in Example 7-16.

Example 7-16. The SyncNumberService concrete implementation

```
package app

import java.util.Random
import ratpack.exec.Promise

class SyncNumberService implements NumberService {

  Promise<Integer> getRandomNumber() {
    Promise.sync {         ❶
      new Random().nextInt()  ❷
    }
  }
}
```

❶ Here, we use the `Promise#sync` method to create the `Promise` type, to which we supply a `Factory` that will generate the promise's value.

❷ Here we synchronously calculate the random value that will fulfill the promise.

> When building service interfaces or other APIs in your Ratpack application, you should always ensure you are using the `ratpack.exec.Promise` and `ratpack.exec.Operation` return types for calls that could potentially be made asynchronously. By coding to these return types from the outset, you can ensure that the concrete implementation will be flexible enough to leverage nonblocking networking when appropriate.

Synchronous promises behave no differently in terms of the execution model. Ratpack has no awareness as to whether or not they will be performing computation or I/O bound calculations, so they will always be executed in their serial order according to their position in the execution stack.

Promises from Asynchronous Calls

As noted, the situation will undoubtedly arise where you will need to integrate with libraries that have their own threading or execution models. To that end, you can still leverage Ratpack promises (and thus its execution model), but you will need to adapt their asynchronous calls to `Promise` types.

The simplest way to understand how to build `Promise` types from asynchronous calls is to consider again Example 7-2 from earlier in the chapter. To this service interface, we provide a callback that provides our processing logic with the data. If we assume that we have no control over the interface (i.e., it is inherited from an external library), we can build a new service interface that adapts these contracts to `Promise` types.

Example 7-17 shows leveraging the `AsyncDatabaseService` to produce results in the form of `Promise` types.

Example 7-17. The PromiseDatabaseService implementation

```java
package app;

import ratpack.exec.Promise;

public class PromiseDatabaseService {

  private final AsyncDatabaseService db;
```

```
  public PromiseDatabaseService(AsyncDatabaseService db) { ❶
    this.db = db;
  }

  public Promise<User> findByUsername(String username) { ❷
    return Promise.async(down ->
      db.findByUsername(username, user ->
        down.success(user)
      )
    );
  }

  public Promise<UserProfile> loadUserProfile(Long profileId) { ❸
    return Promise.async(down ->
      db.loadUserProfile(profileId, profile ->
        down.success(profile)
      )
    );
  }
}
```

❶ The `PromiseDatabaseService` class will leverage the `AsyncDatabaseService` and its underlying threading model to perform the calls to the database. Here, we set the instance as a class-level private field for use within our public methods.

❷ The `findByUsername` method now instead returns a `Promise<User>`. To accommodate this, we use the `Promise#async` method, to which we are given access to an object we can use to fulfill the promise once the data becomes available, shown here as `down.success(result)`.

❸ The `loadUserProfile` method now returns a `Promise<UserProfile>`, also using the `Promise#async` method to fulfill the promise from the mirroring call to the `AsyncDatabaseService`.

Let's expand on this demonstration a little further and consider that in addition to the `Consumer<User>` and `Consumer<UserProfile>` callbacks we provide to the `findByUsername` and `loadUserProfile` methods, we also supply a `Consumer<Throwable>` to each as well to help capture errors. The new service interface looks like that shown in Example 7-18.

Example 7-18. The updated AsyncDatabaseService with error handling

```
package app;

import java.util.function.Consumer;

public interface AsyncDatabaseService {
```

```
    void findByUsername(String username,
                        Consumer<User> callback,
                        Consumer<Throwable> error);

    void loadUserProfile(Long profileId,
                         Consumer<UserProfile> callback,
                         Consumer<Throwable> error);
}
```

We want to ensure that any errors thrown during the asynchronous processing of the AsyncDatabaseService are propagated back to the Promise that we return from our adapted service interface. To accommodate this, the fulfiller provided by the Promise#async method also allows us to call its error method. Consider the updated PromiseDatabaseService shown in Example 7-19.

Example 7-19. The PromiseDatabaseService updated with error handling

```
package app;

import ratpack.exec.Promise;

public class PromiseDatabaseService {

  private final AsyncDatabaseService db;

  public PromiseDatabaseService(AsyncDatabaseService db) {
    this.db = db;
  }

  public Promise<User> findByUsername(String username) {
    return Promise.async(down ->
      db.findByUsername(username, user ->
        down.success(user)
      , error ->
        down.error(error)
      )
    );
  }

  public Promise<UserProfile> loadUserProfile(Long profileId) {
    return Promise.async(down ->
      db.loadUserProfile(profileId, profile ->
        down.success(profile)
      , error ->
        down.error(error)
      )
    );
  }
}
```

As you can see now, the `Consumer<Throwable>` is provided to the `AsyncDatabaseSer vice` calls, and the corresponding errors are provided to the `Promise` type we return.

It is important to note that when adapting asynchronous libraries that have their own threading and execution model, once the data from *their thread of execution* fulfills the promise, the processing is returned to the originating thread of execution in Ratpack's execution model. While this will eventually make downstream processing of multiple data objects faster lower in the execution chain, you will likely experience a CPU context switch when moving data between threads.

As with synchronous promises, asynchronous promises behave no differently in terms of Ratpack's execution model. This mechanism, however, can be powerful when building an application that needs to leverage existing asynchronous services or datasources. Nothing else is needed than what is demonstrated to show adapting these interfaces into Ratpack's promise-based execution model.

Chapter Summary

In this chapter, you learned that Ratpack's execution model is one of the greatest— although generally transparent—benefits of the framework. By providing guaranteed execution order to the complex nature of asynchronous processing, any nondeterminism that asynchronicity would present to web development is now guaranteed determinism. Furthermore, being able to make use of the execution model through the succinctness of the `Promise` API makes it highly consumable for application developers.

With the execution model further providing supervision of the work that is being done within Ratpack applications, application developers are given a guarantee as to the timeliness and correctness of responses that are being sent to clients. There is no ticking clock on sending a response to a client when the execution model is leveraged; instead, when there is no more work to do, a response must be sent, or Ratpack will inform the client that a handler did not send a response.

Given the additional ability to easily schedule execution segments to the thread pool that is most appropriate for their work, Ratpack applications can employ any legacy or third-party API, without being concerned that those APIs will interfere with the performance of their request-taking pools. Furthermore, with Ratpack always having your application's performance interests in mind, you can be assured that any jumps between blocking and nonblocking operations will always take the more resource-efficient path.

Asynchronous programming is a difficult paradigm for many application developers, and can at first be off-putting, but the necessity of this style of development when employing a highly performant, nonblocking networking stack cannot be avoided. Ratpack aims to make this approach to applicaiton development easy by reducing the

complexities for application developers. By also giving the confidence of deterministic control and data flow to what would otherwise be nondeterministic, Ratpack developers can feel comfortable developing against asynchronous code.

Data-Driven Web Applications

Up until this point, the examples in this book have largely been based on contrived scenarios to demonstrate usability. Now that you have an in-depth understanding of Ratpack's execution model, and the insight to build robust web applications with Ratpack, we can discuss leveraging the framework to solve real-world problems. This chapter will show you how to model data and your data access layer for asynchronicity and high performance. Indeed, the discussion throughout this chapter will familiarize you with solutions to a problem space that you will face with nearly every web application that you build: working with databases and providing data as HTTP resources.

The Java ecosystem is rich with libraries and utilities to help you build comprehensive data-driven web applications. A conundrum for most asynchronous JVM web frameworks, however, is that the large majority of these libraries are built on synchronous APIs such as JDBC. Luckily, with Ratpack, this does not mean that you cannot still use those libraries to build your data access objects and service layers. The execution model's ability to seamlessly schedule synchronous, I/O-bound work to the blocking thread pool makes it the perfect foundation for working with database libraries with which you are likely already familiar. Furthermore, when the database call returns, processing is returned to the request-taking thread, affording your application optimum performance when working with data-driven models.

Groovy SQL Support

Groovy-based Ratpack applications have the benefit of being able to leverage Groovy SQL, Groovy's DSL for working with databases. Groovy SQL is a concise utility for data access layers that want the ability to read and write from a database without having an opinionated querying and modeling paradigm. The DSL that it provides allows queries and executions to be written with just a single method call. Beyond

that, accessing the rows of a database and their column data is presented with Groovy syntax that you will undoubtedly find familiar. The DSL's fixtures for working with databases allows you to write the queries you know you will need, and populate your model objects as you see fit.

It is a two-part process to start using Groovy SQL in your Ratpack application. First, we must establish a `javax.sql.DataSource` connection that the `Sql` class will use. To facilitate this, we will need a database driver as part of our application. For the sake of demonstration, let's make use of the H2 in-memory database library to get our example going. The Gradle build script in Example 8-1 shows the necessary additions.

Example 8-1. Gradle build script with H2 in-memory database

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile 'com.h2database:h2:1.4.190'
}
```

Next, we can build the `DataSource` directly in the application's `bindings` block, as shown in Example 8-2.

Example 8-2. Building the DataSource and binding it

```
import org.h2.jdbcx.JdbcDataSource
import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    bindInstance DataSource, new JdbcDataSource( ❶
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )
```

```
    }
  handlers {
    // nothing here for now...
  }
}
```

❶   We can use Groovy's `Map` constructors to set the properties directly as part of the constructor arguments. (Note that the format of the URL is structured with H2's `DB_CLOSE_DELAY=-1`, which instructs the database to keep all data as long as the JVM is alive.)

With the `DataSource` in place and bound properly, we can move on to the second part, which is to incorporate the `SqlModule` into our project. The `SqlModule` will use the bound `DataSource` and provide our application with access to the Groovy `Sql` object, which is the DSL we will use to interact with the database. Example 8-3 shows the incorporation of the `SqlModule`.

Example 8-3. Incorporating the SqlModule

```
import org.h2.jdbcx.JdbcDataSource
import ratpack.groovy.sql.SqlModule
import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module SqlModule ❶
    bindInstance DataSource, new JdbcDataSource(
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )
  }
  handlers {
      // nothing here for now...
  }
}
```

❶   It is no different than the process with which you are already familiar.

We can modify the `bindings` block slightly to additionally bind a `rat pack.server.Service` implementation, which we can use to bootstrap some data into the database. This will help us for demonstration purposes now, but even in a scenario where you are working with a nonvolatile database, you can leverage this technique to validate that the database is properly structured, or to make amendments to

the database schema prior to the application starting up. Example 8-4 shows how we can create a new database table and bootstrap it with some trivial data.

Example 8-4. Modified bindings block with database initialization

```groovy
import org.h2.jdbcx.JdbcDataSource
import ratpack.groovy.sql.SqlModule
import ratpack.server.Service
import ratpack.server.StartEvent
import groovy.sql.Sql

import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module SqlModule
    bindInstance DataSource, new JdbcDataSource(
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )

    bindInstance new Service() {
      void onStart(StartEvent e) {
        Sql sql = e.registry.get(Sql)                                          ❶
        sql.execute(
        "CREATE TABLE TEST(ID INT PRIMARY KEY AUTO_INCREMENT, NAME VARCHAR(255))"  ❷
        )
        sql.execute "INSERT INTO TEST (NAME) VALUES('Luke Daley')"            ❸
        sql.execute "INSERT INTO TEST (NAME) VALUES('Rob Fletch')"
        sql.execute "INSERT INTO TEST (NAME) VALUES('Dan Woods')"
      }
    }
  }
  handlers {
    // nothing here for now...
  }
}
```

❶ Here we use the `StartEvent` provided to the `onStart` method to get access to the server registry and resolve the `Sql` object.

❷ This is the first introduction to working with the `Sql` DSL, and as you can see, the relatively simple `execute` call simply runs the provided SQL statement against the database.

❸ From here on, we employ the `sql.execute(..)` call to bootstrap data directly into the TEST table.

The simplicity of working with databases through the `Sql` DSL is what makes it an appealing prospect for Groovy applications. It should be noted that the initialization that we are demonstrating in the `Service` is still performing blocking calls to the database. During the `onStart` call, it is not important that we schedule these calls to the blocking thread pool, because we will not be affecting request taking, as requests will not be accepted until all `Service` classes have been initialized.

When we plan to make database calls during request processing, however, we will want to ensure that blocking calls are scheduled to the blocking thread pool properly. As you are already aware of the `Blocking` fixture, you probably understand how we can use that within a handler to perform queries against the database. The updated code in Example 8-5 shows a `get` handler listing the values we have stored during initialization.

Example 8-5. Blocking database call in handler

```groovy
import groovy.sql.Sql
import org.h2.jdbcx.JdbcDataSource
import ratpack.exec.Blocking
import ratpack.groovy.sql.SqlModule
import ratpack.server.Service
import ratpack.server.StartEvent

import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module SqlModule
    bindInstance DataSource, new JdbcDataSource(
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        Sql sql = e.registry.get(Sql)
        sql.execute(
          "CREATE TABLE TEST(ID INT PRIMARY KEY AUTO_INCREMENT, NAME VARCHAR(255))"
        )
        sql.execute "INSERT INTO TEST (NAME) VALUES('Luke Daley')"
        sql.execute "INSERT INTO TEST (NAME) VALUES('Rob Fletch')"
```

```
            sql.execute "INSERT INTO TEST (NAME) VALUES('Dan Woods')"
        }
      }
    }
    handlers {
      get { Sql sql -> ❶
        Blocking.get { ❷
          sql.rows("select * from test").collect { ❸
            it["name"] ❹
          }
        } then { names ->
          render(json(names)) ❺
        }
      }
    }
  }
}
```

❶ We can inject the Sql type into the handler as we would any other dependency.

❷ Make sure that we wrap the blocking database calls in Blocking.get(..) to ensure they are executed on the blocking thread.

❸ The Sql DSL allows us to query many rows simply with the rows method.

❹ The resulting objects are ones that we can leverage the "getAt" bracket notation to access the columns of the row.

❺ The name columns are collected off of each row and returned as the value of the promise. Here, we can use the Jackson support to render the list of names as JSON.

As you can already see, using the Sql DSL to work with databases is concise and intuitive. There are many simple data access methods that allow you to query one or many rows and columns of data, including row and firstRow. These return a single row from the provided SQL statement or the first row of a multirow query, respectively. It is worth reading through the Sql class's comprehensive API documentation, which includes examples and discussion.

Alone, the simplistic and unopinionated nature of the Sql DSL is attractive enough, but there are also strong technical reasons to favor this approach over rolling your own. When you are constructing a SQL statement that will insert or update data in the database, the Sql class allows you to use GString variable notation within your statement, and it will transform your variables into a PreparedStatement type. This is absolutely necessary for web applications that will be inserting or updating data. Example 8-6 shows a post handler to the "create" URI, which creates a new record in the database for the provided name.

Example 8-6. Inserting data using GString notation

```groovy
import groovy.sql.Sql
import org.h2.jdbcx.JdbcDataSource
import ratpack.exec.Blocking
import ratpack.groovy.sql.SqlModule
import ratpack.server.Service
import ratpack.server.StartEvent
import ratpack.form.Form

import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module SqlModule
    bindInstance DataSource, new JdbcDataSource(
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )

    bindInstance new Service() {
      void onStart(StartEvent e) {
        Sql sql = e.registry.get(Sql)
        sql.execute(
          "CREATE TABLE TEST(ID INT PRIMARY KEY AUTO_INCREMENT, " +
            "NAME VARCHAR(255))"
        )
        sql.execute "INSERT INTO TEST (NAME) VALUES('Luke Daley')"
        sql.execute "INSERT INTO TEST (NAME) VALUES('Rob Fletch')"
        sql.execute "INSERT INTO TEST (NAME) VALUES('Dan Woods')"
      }
    }
  }
  handlers {
    get { Sql sql ->
      Blocking.get {
        sql.rows("select * from test").collect {
          it["name"]
        }
      } then { names ->
        render(json(names))
      }
    }
    post("create") { Sql sql ->
      parse(Form).then { f ->           ❶
        def name = f.name
        if (name) {                     ❷
          Blocking.get {                ❸
```

```
            sql.execute "INSERT INTO TEST (NAME) VALUES($name)" ❹
          } onError { t ->
            render(json([success: false, error: t.message])) ❺
          } then {
            render(json([success: true])) ❻
          }
        } else {
          response.status(400) //
          render(json([success: false, error: "name is required"]))
        }
      }
    }
  }
}
```

❶ We start the request handling by parsing the supplied data using the Form parser. This will give us access to the URL-encoded values that were sent to this handler.

❷ Check to make sure that the request data sent the name value.

❸ Wrap the blocking Sql call to ensure it gets scheduled properly.

❹ Using the execute method, we can construct the insert statement using the $name variable.

❺ If there was an error, inform the caller.

❻ Send the success message back to the caller.

The most important part to the entire POST handler is the INSERT statement that is constructed and called. To every Groovy developer, this looks like a simple variable replacement, but the Sql DSL takes this to the next level to ensure that your SQL statements are safe. In this case, instead of simply replacing the token value, Sql is storing the value, transforming the SQL statement into a PreparedStatement type, replacing the $name part with the necessary ?, and applying the indexed value appropriately. That is a lot of work that we do not have to do anymore when using Sql.

Connection Pooling with HikariCP Support

The use of Groovy SQL demonstrated in the prior section has taken a somewhat naive approach to building a data access layer. The underlying datasource in the examples is using but a single database connection for managing access to a database. This is fine for testing or prototyping, but when you are building your application for production, you will need the ability to pull connections from a *connection pool*. This allows your database calls to pull from a set of persistent connections, thereby

improving overall performance and allowing a higher number of concurrent and parallel requests to access your database.

Ratpack provides optional support for pooling database connections through integration with the HikariCP library. HikariCP is a high-performance connection pooling library for Java, and touts itself as a "zero-overhead production-quality connection pool." Indeed, its performance measurements place it well above competing libraries in the space.

To integrate HikariCP in your Ratpack application, you will need to start by including the ratpack-hikari dependency in your project's build script. Example 8-7 shows an amendment to the buildscript from the prior section, this time with HikariCP support added.

Example 8-7. HikariCP build script dependency

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile 'com.h2database:h2:1.4.190'
  compile ratpack.dependency('hikari') ❶
}
```

❶ Adds the ratpack-hikari dependency to the project.

With the dependency in place, we need to modify our application code to incorporate the HikariModule into the project. The HikariModule is a ConfigurableModule, so we can supply the module(HikariModule) call in the bindings block with a Closure that will allow us to configure the properties of the connection pool. The application code in Example 8-8 shows the modified application from the prior section, this time using HikariCP to manage the datasource.

Example 8-8. HikariCP and Groovy Sql

```groovy
import groovy.sql.Sql
import ratpack.exec.Blocking
import ratpack.form.Form
import ratpack.groovy.sql.SqlModule
import ratpack.hikari.HikariModule
import ratpack.server.Service
import ratpack.server.StartEvent

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module SqlModule
    module(HikariModule) { c ->               ❶
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'  ❷
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'  ❸
      c.username = 'sa'                        ❹
      c.password = ''                          ❺
    }

    bindInstance new Service() {
      void onStart(StartEvent e) {
        // ... bootstrapping logic from prior section here ...
      }
    }
  }
  handlers {
    // ... handlers from prior section here ...
  }
}
```

❶ We add the `HikariModule` and provide the configuration `Closure`.

❷ Specify the `dataSourceClassName`, which is the same class we used in the prior section.

❸ Specify the `URL` property, similar to how we did previously, but this time apply it with the `addDataSourceProperty` method on the configuration object.

❹ Sets the database username.

❺ We also need to set a password. In this case, since we are using the in-memory H2 database, we set it to an empty string.

Again, in this example we are using the H2 in-memory database for demonstration simplicity, but any JDBC-compliant datasource can be as easily utilized here. And

these changes are all that is necessary; the `HikariModule` will provide the `SqlModule` with the necessary `DataSource` to make its database calls.

If you are not building a Groovy Ratpack project, then you can still make use of the HikariCP support to provide your project with connection pooling. For example, if you wanted to utilize database connection pooling within a Java application, and use the HikariCP datasource directly in your application, that is equally as doable. Granted, the implementation of using the `DataSource` directly from within Java will be more verbose while accessing the necessary components is trivial. The Java application in Example 8-9 is the same Groovy application we have been demonstrating, rewritten in Java using the HikariCP datasource directly.

Example 8-9. Java HikariCP application

```java
package app;

import com.google.common.collect.Lists;
import com.google.common.collect.Maps;
import ratpack.exec.Blocking;
import ratpack.form.Form;
import ratpack.guice.Guice;
import ratpack.hikari.HikariModule;
import ratpack.server.RatpackServer;
import ratpack.server.Service;
import ratpack.server.StartEvent;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.List;
import java.util.Map;

import static ratpack.jackson.Jackson.json;

public class Main {

  public static void main(String[] args) throws Exception {
    RatpackServer.start(spec -> spec
      .registry(Guice.registry(bindingsSpec ->
        bindingsSpec
          .module(HikariModule.class, c -> {
            c.setDataSourceClassName("org.h2.jdbcx.JdbcDataSource");
            c.addDataSourceProperty("URL", "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1");
              c.setUsername("sa");
              c.setPassword("");
            })
            .bindInstance(new Service() {
              @Override
              public void onStart(StartEvent event) throws Exception {
```

```
            DataSource dataSource = event.getRegistry().get(DataSource.class);
            try (Connection connection = dataSource.getConnection()) {
              connection.createStatement()
                .execute("CREATE TABLE TEST(ID INT PRIMARY KEY " +
                         "AUTO_INCREMENT, NAME VARCHAR(255))");
              connection.createStatement()
                .execute("INSERT INTO TEST (NAME) VALUES('Luke Daley')");
              connection.createStatement()
                .execute("INSERT INTO TEST (NAME) VALUES('Rob Fletch')");
              connection.createStatement()
                .execute("INSERT INTO TEST (NAME) VALUES('Dan Woods')");
            }
          }
        })
  ))
  .handlers(chain -> chain
    .get(ctx -> {
      Blocking.get(() -> {
        DataSource dataSource = ctx.get(DataSource.class);
        List<Map<String, String>> personList = Lists.newArrayList();
        try (Connection connection = dataSource.getConnection()) {
          ResultSet rs = connection.createStatement()
            .executeQuery("SELECT * FROM TEST");
          while (rs.next()) {
            long id = rs.getLong(1);
            String name = rs.getString(2);
            Map<String, String> person = Maps.newHashMap();
            person.put("id", String.valueOf(id));
            person.put("name", name);
            personList.add(person);
          }
        }
        return personList;
      }).then(personList -> ctx.render(json(personList)));
    })
    .post("create", ctx ->
      ctx.parse(Form.class).then(f -> {
        String name = f.get("name");
        if (name != null) {
          Blocking.get(() -> {
            DataSource dataSource = ctx.get(DataSource.class);
            try (Connection connection = dataSource.getConnection()) {
              PreparedStatement pstmt = connection
                .prepareStatement("INSERT INTO TEST (NAME) VALUES(?)");
              pstmt.setString(1, name);
              pstmt.execute();
            }
            return true;
          }).onError(t -> {
            ctx.getResponse().status(400);
            ctx.render(json(getResponseMap(false, t.getMessage())));
          }).then(r ->
```

```
              ctx.render(json(getResponseMap(true, null)))
            );
        } else {
          ctx.getResponse().status(400);
          ctx.render(json(getResponseMap(false, "name not provided")));
        }
      })
    )
   )
  );
}

private static Map<String, Object> getResponseMap(Boolean status, String message) {
  Map<String, Object> response = Maps.newHashMap();
  response.put("success", status);
  response.put("error", message);
  return response;
}
}
```

The `HikariModule` configuration object allows you to tune every aspect of the connection pool, including pool sizes, connection timeout, validation timeout, and leak detection, among other properties. The project's GitHub page provides documentation for all of the properties that you can directly tune.

Ratpack and Grails GORM

Groovy SQL is not the only game in town if you are building data-driven, Groovy-based Ratpack web applications. Indeed, if you have operated in the Groovy ecosystem, you're likely familiar with and understand the power, simplicity, and usability of Grails' data mapping layer, GORM. Prior to Grails 3.x, making use of GORM outside of Grails was a highly complex process, but these days the library can be used exclusively outside of Grails. This opens the door for frameworks like Ratpack to allow developers to integrate the robust data modeling and interactions provided by GORM without the need for the Grails web framework.

Similar to working with Groovy SQL, to leverage GORM inside of Ratpack, we will need to make use of the `Blocking.get(..)` mechanism to ensure blocking database calls are appropriately scheduled to the blocking thread pool. There are some other added complexities that need to be handled, mostly stemming from the underlying implementation, which uses Hibernate.

The problems presented by Hibernate are specifically related to the fact that it uses `ThreadLocal` storage for binding a database `Session` to the current thread. As we've already discussed, Ratpack maintains a small request-taking thread pool that is reused across many different requests, so this is an unsuitable place to have request-specific `ThreadLocals`.

A thread from the blocking pool is fine for retrieving data with Hibernate, because it will be dedicated exclusively to the blocking processing. But Hibernate's manner of creating a proxy around domain objects in order to provide lazy references to related objects in a graph presents an issue when we want to query a domain object in a blocking thread, and then move processing of that data to the computation thread. Because Hibernate is still managing the proxied object, when the object is moved out of the blocking thread and back to the computation thread, the `ThreadLocal` reference to the `Session` will no longer be there, and Hibernate will throw an error.

To make using GORM (and thus Hibernate) more friendly, we need to ensure that the object is either fully initialized, unwrapped from its proxied type, or translated to a data transfer object type, prior to moving it off of the blocking thread. For some, this may prove a prohibitive tax for the sake of using the Hibernate-based ORM; for those who need the robust modeling and data access capabilities provided by GORM, it may serve simply as a strategy for accomplishing a means to an end. Your mileage will vary according to your comfort level and your application's requirements. The goal of this section of the book is to expose you to the possibilities you are open to when working with a framework as flexible, user friendly, and ecosystem friendly as Ratpack.

To start with integrating GORM into our project, we will need to include the GORM dependencies in the project's build script. The build script in Example 8-10 shows the necessary dependencies.

Example 8-10. Gradle build script with GORM dependencies

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile('org.grails:grails-datastore-gorm-hibernate4:5.0.0.RELEASE') { ❶
    exclude module: 'groovy' ❷
  }
  compile('org.grails:grails-spring:3.0.9') { ❸
    exclude module: 'groovy' ❹
  }
```

```
    compile 'com.h2database:h2:1.4.190' ❺
}
```

❶ This call adds the `grails-datastore-gorm-hibernate4` dependency to the project.

❷ We need to explicitly exclude the `groovy` transitive dependency from this project, as Ratpack will provide this with a known good version.

❸ The GORM infrastructure leverages the Spring Framework for transaction management, so we also need to incorporate this library.

❹ This library also ships with a `groovy` dependency, so we will exclude it here.

❺ For demonstration's sake, as with the prior section on Groovy SQL, we will include the H2 in-memory database driver. This can safely be replaced by MySQL, PostgreSQL, Oracle, or whatever JDBC driver your project requires.

With the GORM dependencies in place, let's take a look at what a simple `Person` domain class would look like, which is similar to the prior section's example. The class in Example 8-11 depicts the simple model object we will be using.

Example 8-11. Person GORM domain class

```
package app

import org.grails.datastore.gorm.GormEntity

class Person implements GormEntity<Person> {
  Long id
  Long version
  String name

  static constraints = {
    name blank: false
  }
}
```

A slight difference this example shows is that when using GORM in Ratpack, we want to ensure that we get the type safety and IDE assistance provided by GORM's extension methods. In Grails, a domain class does not need to explicitly implement the `GormEntity<T>` trait in order to get the rich domain model functionality it offers. Instead, these classes are generally annotated with `@grails.persistence.Entity`, and a Groovy AST (Abstract Syntax Tree) transform applies the trait. Instead of doing that, we can explicitly implement `GormEntity<T>` and get the benefits of GORM as well as compile-time type safety and IDE assistance.

With the `Person` domain class in place, we can incorporate GORM support into our application and bootstrap some simple data. Let's continue the demonstration by taking a look at what our basic application logic will look like. The code in Example 8-12, which includes the `bindings` block and a `Service` implementation, shows the basic setup.

Example 8-12. Basic setup of a Ratpack application with GORM

```groovy
import app.GormModule
import app.Person
import ratpack.service.Service
import ratpack.service.StartEvent
import ratpack.exec.Blocking
import grails.orm.bootstrap.HibernateDatastoreSpringInitializer

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module GormModule ❶

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        e.getRegistry().get(HibernateDatastoreSpringInitializer) ❷
        Blocking.exec { ❸
          Person.withNewSession { ❹
            new Person(name: "Luke Daley").save() ❺
            new Person(name: "Rob Fletch").save()
            new Person(name: "Dan Woods").save()
          }
        }
      }
    }
  }
  handlers {
    // ... nothing here for now.
  }
}
```

❶ For this example, the `GormModule` will come from within our project and will be demonstrated next. This module will be responsible for incorporating all of the components necessary to get GORM up and running.

❷ Here we explicitly request the `HibernateDatastoreSpringInitializer` to ensure that GORM's infrastructure was properly initialized. Doing this here will ensure that all component bindings have been properly created and that GORM is aware of our `Person` class.

❸ It is necessary to wrap the GORM calls using the `Blocking` execution fixture to ensure that the Hibernate session is safely bound to its own thread.

❹ We can utilize GORM's `DomainClass.withNewSession(..)` call to construct a new Hibernate session, and wrap the database calls within the supplied `Closure`.

❺ Construct a new `Person` object with the `name` property that we specify, and call the `save()` method on it.

The `withNewSession` and `save` methods come from the `GormEntity<T>` trait that the `Person` class explicitly inherited. These are the static and instance-level methods that allow GORM to work with domain objects as rich models.

The `GormModule` is the most important part to the whole equation. It is the incorporation of this that allows GORM to be used at all within our application. The Guice module logic in Example 8-13 shows the components and calls necessary to bootstrap the GORM infrastructure in our application.

Example 8-13. The module for setting up GORM

```
package app

import com.google.inject.AbstractModule
import com.google.inject.Provides
import com.google.inject.Singleton
import grails.orm.bootstrap.HibernateDatastoreSpringInitializer
import org.h2.Driver
import org.springframework.context.support.GenericApplicationContext
import org.springframework.jdbc.datasource.DriverManagerDataSource

class GormModule extends AbstractModule {  ❶
  @Override
  protected void configure() {
      ❷
  }

  @Provides
  @Singleton
  GenericApplicationContext genericApplicationContext() {
    new GenericApplicationContext()  ❸
  }

  @Provides
  @Singleton
  DriverManagerDataSource dataSource(GenericApplicationContext appCtx) {  ❹
    def dataSource =
        new DriverManagerDataSource("jdbc:h2:mem:grailsDb1;DB_CLOSE_DELAY=-1",
            'sa', '')  ❺
```

```
    dataSource.driverClassName = Driver.name ❻
    appCtx.beanFactory.registerSingleton 'dataSource', dataSource ❼
    dataSource
  }

  @Provides
  @Singleton
  HibernateDatastoreSpringInitializer initializer(DriverManagerDataSource ds, ❽
                                        GenericApplicationContext appCtx) {
    def datastoreInitializer = new HibernateDatastoreSpringInitializer(Person) ❾
    datastoreInitializer.configureForBeanDefinitionRegistry(appCtx) ❿
    appCtx.refresh() ⓫
    datastoreInitializer
  }
}
```

❶ The `GormModule` is built no different from any other Guice module that our application would use.

❷ In this case, we do not need to do anything in the `configure` block, as we will use micro-providers to get access to the different components during their initialization phases.

❸ As noted earlier, GORM utilizes the Spring Framework behind the scenes for a variety of tasks, including transaction management. We can construct a simplistic `GenericApplicationContext`, as we only need a limited container to make GORM work in a standalone fashion.

❹ When constructing the `DriverManagerDataSource`, we will need to register it in the Spring application context, so using the Guice micro-provider mechanism, we can inject it as part of the provider method signature.

❺ Construct the `dataSource` that GORM will use. This is utilizing the H2 in-memory connect string, similar to that demonstrated in the prior section.

❻ Here, we set the `driverClassName` on the `dataSource` to the `org.h2.Driver` class name, also similar to what was demonstrated in the prior section.

❼ Now we register the `dataSource` as a singleton in Spring application context.

❽ Note that here we request the `DriverManagerDataSource`, but do not explicitly need it for initializing the `HibernateDatastoreSpringInitializer`. We do this to ensure that all the datasource is properly registered in the Spring application context.

⑨ We can use the `HibernateDatastoreSpringInitializer` to initialize the `Person` class for use with GORM. If you have a robust domain model, multiple domain classes can be supplied here, but all of the domain classes that you intend to use will need to be initialized through this mechanism.

⑩ Here, we configure the `datastoreInitializer` using the Spring application context.

⑪ Finally, we need to initalize the Spring application context using the `refresh()` call.

This might seem complex and complicated, but if you follow through the flow of initialization and preparation, only three components are truly necessary to get GORM working in our application.

With the GORM fixtures bootstrapped and in place, we can expand out our *ratpack.groovy* file to provide an endpoint to inspect the data in the `PERSON` database table. Example 8-14 shows the addition of a `get` handler.

Example 8-14. The get handler listing Person database records

```
import app.GormModule
import app.Person
import grails.orm.bootstrap.HibernateDatastoreSpringInitializer
import ratpack.exec.Blocking
import ratpack.service.Service
import ratpack.service.StartEvent

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module GormModule

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        e.getRegistry().get(HibernateDatastoreSpringInitializer)
        Blocking.exec {
          Person.withNewSession {
            new Person(name: "Luke Daley").save()
            new Person(name: "Rob Fletch").save()
            new Person(name: "Dan Woods").save()
          }
        }
      }
    }
  }
}
```

```
handlers {
  get {  ❶
    Blocking.get {  ❷
      Person.withNewSession {  ❸
        Person.list().collect { p ->  ❹
          [id: p.id, version: p.version, name: p.name]  ❺
        }
      }
    } then { personList ->
      render(json(personList))
    }
  }
}
```

❶ The `get` handler, bound to the default route.

❷ We wrap the database calls in `Blocking.get(..)` to ensure they are bound to a blocking thread.

❸ Again, we use the `withNewSession` to ensure a database session is bound to the blocking thread.

❹ The call to `Person.list()` executes the equivalent of a `SELECT * FROM PERSON`. As noted in the introduction to this section, when we work with GORM domain objects, we need to make sure they are no longer bound to a Hibernate session before moving them out of the blocking and then into the computation thread. To accomplish this, we can map the resulting properties to a data transfer object (in this case, a `Map`).

❺ We can render the resulting data as JSON, just as you would any other data.

If we start this application and navigate a browser to *http://localhost:5050*, we will see the resulting JSON data shown in Example 8-15.

Example 8-15. JSON representation of Person objects

```
[
  {
    "id": 1,
    "version": 0,
    "name": "Luke Daley"
  },
  {
    "id": 2,
    "version": 0,
    "name": "Rob Fletch"
  },
```

```
  {
    "id": 3,
    "version": 0,
    "name": "Dan Woods"
  }
]
```

We have successfully built our first data-driven web application with Ratpack and GORM! To add data to the database, we can expose a POST handler at /create, which pulls the name property from provided URL-encoded form data. Example 8-16 shows the addition of the post handler.

Example 8-16. Adding the post handler to create and save Person records

```groovy
import app.GormModule
import app.Person
import grails.orm.bootstrap.HibernateDatastoreSpringInitializer
import ratpack.service.Service
import ratpack.service.StartEvent
import ratpack.form.Form
import ratpack.exec.Blocking
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module GormModule

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        e.getRegistry().get(HibernateDatastoreSpringInitializer)
        Blocking.exec {
          Person.withNewSession {
            new Person(name: "Luke Daley").save()
            new Person(name: "Rob Fletch").save()
            new Person(name: "Dan Woods").save()
          }
        }
      }
    }
  }
  handlers {
    get {
      Blocking.get {
        Person.withNewSession {
          Person.list().collect { p ->
            [id: p.id, version: p.version, name: p.name]
          }
        }
      } then { personList ->
        render(json(personList))
```

```
        }
      }
    post("create") {
      parse(Form).then { f -> ❶
        def name = f.name
        if (name) { ❷
          Blocking.get { ❸
            Person.withNewSession { ❹
              new Person(name: name).save() ❺
            }
          } onError { t -> ❻
            response.status(400)
            render(json([success: false, error: t.message]))
          } then {
            render(json([success: true, error: null])) ❼
          }
        } else { ❽
          response.status(400)
          render(json([success: false, error: "name not provided"]))
        }
      }
    }
  }
}
```

❶ As we have seen previously, here we parse the request data using the Form type, from which we can access the name property in the following line.

❷ Check to make sure that a name value was actually sent.

❸ If it was, then move the call to the database to a blocking thread.

❹ Again, use the withNewSession GORM mechanism to create a new Hibernate session on the blocking thread.

❺ Create the new Person object and save it.

❻ Capture any errors here and respond to the client accordingly.

❼ If there were no errors, then render back a success message.

❽ If we did not receive a name value with the request, then send back a friendly error message indicating the error.

This is all that is necessary to demonstrate how to use GORM in Ratpack to build robust, data-driven web applications! That said, we can do a little more work and make this an even simpler process.

We can build on the fact that we are not utilizing the `@Entity` annotation and corresponding AST transformation to apply the rich domain model capabilities to the `Person` class to simplify the process of working with GORM domain objects. GORM has done an excellent job of allowing developers to extend its capabilities, and we can leverage that fact to overload the `withNewSession` method to automatically schedule database calls to the `blocking` thread pool. Consider the trait shown in Example 8-17 to understand the customizations we can provide GORM.

Example 8-17. The GormEntity trait adapted for Ratpack

```
package app

import groovy.transform.CompileStatic
import org.grails.datastore.gorm.GormEnhancer
import org.grails.datastore.gorm.GormEntity
import org.grails.datastore.gorm.GormStaticApi
import ratpack.exec.Blocking
import ratpack.exec.Promise

@CompileStatic
trait RatpackGormEntity<D> extends GormEntity<D> {
  private static GormStaticApi<D> internalStaticApi ❶

  static GormStaticApi<D> currentGormStaticApi() {
    if (internalStaticApi == null) {
      internalStaticApi = (GormStaticApi<D>) GormEnhancer.findStaticApi(this) ❷
    }
    internalStaticApi
  }

  static <V> Promise<V> withNewSession(Closure callable) { ❸
    Blocking.get { ❹
      (V) currentGormStaticApi().withNewSession(callable) ❺
    }
  }
}
```

❶ The most important thing that we need to keep track of is the `GormStaticApi`. This provides the domain model with the static methods that allow you to easily work with databases through the rich domain model fixtures.

❷ We capture the `GormStaticApi`, which we will delegate the `withNewSession` call to from within our blocking thread.

❸ This is where we overload the `withNewSession` method.

❹ And we ensure that any calls to the database are scheduled to a blocking thread.

❺ This call to the `GormStaticApi` is what will actually perform the database calls that are wrapped within the `Closure` supplied to `withNewSession`.

With the `RatpackGormEntity` trait in place, we can implement it from the `Person` domain object, as shown in Example 8-18.

Example 8-18. Person with RatpackGormEntity

```groovy
package app

class Person implements RatpackGormEntity<Person> {
  Long id
  Long version
  String name

  static constraints = {
    name blank: false
  }
}
```

As you can see, the change to `Person` is trivial, but we now get the benefit of seamlessly integrating into Ratpack's execution model. Given that `Person` now implements `RatpackGormEntity`, we can change our application to remove the unnecessary `Blocking` calls. Example 8-19 demonstrates calling `Person.withNewSession` directly now.

Example 8-19. Removing the unnecessary blocking calls

```groovy
import app.GormModule
import app.Person
import grails.orm.bootstrap.HibernateDatastoreSpringInitializer
import ratpack.form.Form
import ratpack.service.Service
import ratpack.service.StartEvent

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module GormModule

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        e.getRegistry().get(HibernateDatastoreSpringInitializer)
        Person.withNewSession {
          new Person(name: "Luke Daley").save()
          new Person(name: "Rob Fletch").save()
          new Person(name: "Dan Woods").save()
```

```
          } operation() then()
        }
      }
    }
    handlers {
      get {
        Person.withNewSession {
          Person.list().collect { p ->
            [id: p.id, version: p.version, name: p.name]
          }
        } then { personList ->
          render(json(personList))
        }
      }
      post("create") {
        parse(Form).then { f ->
          def name = f.name
          if (name) {
            Person.withNewSession {
              new Person(name: name).save()
            } onError { t ->
              response.status(400)
              render(json([success: false, error: t.message]))
            } then { person ->
              render(json([success: true, error: null]))
            }
          } else {
            response.status(400)
            render(json([success: false, error: "name not provided"]))
          }
        }
      }
    }
  }
}
```

Because of GORM's ability to extend its out-of-the-box capabilities to fit our use case, we can easily imagine more robust integrations that also integrate with the execution model. These integrations can serve as a welcome fixture for newcomers to Ratpack who are familiar with building domain objects in Grails, and want to leverage GORM's capabilities.

GORM has a lot of capabilities and flexibilities that have not been demonstrated in this section. It is left as an exercise to the reader to review the extensive documentation provided by the Grails project.

Designing Data-Driven Service APIs in Ratpack

Regardless of the method you choose to access databases within your data-driven web applications, a good strategy to follow is to decouple your data access layer from your handler logic. In that regard, it is advisable to create a service tier that manages the

interactions your application will have with your data. From within your handler logic, you can employ services to get access to the data you wish to serve.

The tactic of decoupling your handler logic from your data access logic not only creates a conceptual boundary, or "architecture slice", within your application, but it can also allow you to change out the data access implementation you have chosen as your requirements evolve. To understand this better, consider that you are building a simple data-driven application that only needs to create and provide the Person model object that we discussed in the prior section. Your initial implementation may choose to use the Groovy SQL mechanism described at the beginning of this chapter. As your requirements evolve, you may find that you need to persist and serve additional models that are referenced by the Person class. To facilitate this, you may elect to move to GORM to garner its support for modeling collection relationships.

Were you to build the data access implementation directly into your handler logic, a migration from Groovy SQL to GORM would certainly be nontrivial. It may, in fact, require rearchitecting the entire application. Instead, if you were to build your data access into your application's service layer, you would only need to modify the service implementation to change the underlying persistence mechanism.

If we again consider the Person model object, but instead have moved the storage and retrieval of this type into a service, we can then choose to utilize the service within our handler to get access to the Person types. The interface in Example 8-20 demonstrates how we can model the service class to suit our needs.

Example 8-20. The PersonService interface

```groovy
package app

import ratpack.exec.Operation
import ratpack.exec.Promise

interface PersonService {
  /**
   * List all the people in the database
   * @return a promise to a list of {@link Person} models
   */
  Promise<List<Person>> list()

  /**
   * Saves the provided {@link Person} model
   * @param person
   */
  Operation save(Person person)
}
```

Arguably the most important aspect to understand about this is the service's contracts will specify `Promise` and `Operation` return types throughout. Using these types allows us to decide not only the underlying implementation type, but also whether the implementation will use asynchronous or synchronous APIs behind the scenes. Example 8-21 demonstrates a concrete implementation using Groovy SQL.

Example 8-21. The PersonService implementation with Groovy Sql

```groovy
package app

import com.google.inject.Inject
import groovy.sql.Sql
import ratpack.exec.Blocking
import ratpack.exec.Operation
import ratpack.exec.Promise

class GroovySqlPersonService implements PersonService {
  private final Sql sql

  @Inject
  GroovySqlPersonService(Sql sql) {
    this.sql = sql
  }

  @Override
  Promise<List<Person>> list() {
    Blocking.get {
      sql.rows("select * from test").collect {
        def id = (long)it["id"]
        def name = it["name"]
        new Person(id: id, name: name)
      }
    }
  }

  @Override
  Operation save(Person person) {
    Blocking.get {
      sql.execute "INSERT INTO TEST (NAME) VALUES($person.name)"
    }.operation()
  }
}
```

Now the application code will need to be modified to bind the `PersonService` to the `GroovySqlPersonService` concrete type, and the handler logic will need to be changed to call the service class instead of using the `Sql` type directly. The updated application logic in Example 8-22 demonstrates these changes.

Example 8-22. Updated ratpack.groovy using the PersonService

```groovy
import app.GroovySqlPersonService
import app.Person
import app.PersonService
import groovy.sql.Sql
import org.h2.jdbcx.JdbcDataSource
import ratpack.form.Form
import ratpack.groovy.sql.SqlModule
import ratpack.service.Service
import ratpack.service.StartEvent

import javax.sql.DataSource

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module SqlModule
    bindInstance DataSource, new JdbcDataSource(
      URL: "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
      user: "sa",
      password: ""
    )

    bind PersonService, GroovySqlPersonService     ❶

    bindInstance new Service() {
      void onStart(StartEvent e) throws Exception {
        // ... bootstrapping logic from beginning of the chapter ...
      }
    }
  }
  handlers {
    get { PersonService personService ->           ❷
      personService.list().then { personList ->     ❸
        render(json(personList))
      }
    }
    post("create") { PersonService personService -> ❹
      parse(Form).then { f ->
        def name = f.name
        if (name) {
          def person = new Person(name: name)       ❺
          personService.save(person).onError { t ->  ❻
            render(json([success: false, error: t.message]))
          } then {
            render(json([success: true]))
          }
        } else {
          response.status(400)
```

```
        render(json([success: false, error: "name is required"]))
      }
    }
  }
}
```

❶ In the `bindings` block, we specify to `bind` the `PersonService` interface to the `GroovySqlPersonService` concrete implementation.

❷ Within the handler closure, we can inject the `PersonService` interface, which will resolve the `GroovySqlPersonService` concrete type.

❸ Within the handler logic, we simply interact with the `list()` method on the `PersonService`. The data access is completely abstracted from the handler now.

❹ Same as with the `get` handler, we inject the `PersonService`.

❺ A change here is that we create a `Person` type for passing to the `PersonService`.

❻ We call the `save()` method on the `PersonService`, which returns an operation (a `Promise` with `void` return type).

With the data access layer now cleanly moved into the `GroovySqlPersonService`, the handler logic can focus on what it is most appropriately designed for: handling the request. Any of the data access concerns are now isolated to the `PersonService` implementation. In the future, if we decide to migrate to using GORM instead of Groovy SQL, we simply need to create the appropriate `PersonService` implementation, and trade out the binding in the `bindings` section of the application. No update to the handler logic is necessary.

Chapter Summary

In this chapter, you have been exposed to the various ways in which Ratpack can facilitate data access for building robust, data-driven web applications. We have explored the database integration provided by Groovy SQL and Ratpack's `SqlModule`; we have discussed connection pooling and how to accomplish that using HikariCP; we have explored integrating with GORM; and we have discussed a clean architecture pattern between handlers and data access.

With the understanding provided up until this point in the book, you are now fully equipped to build high-performance, data-driven web applications that leverage a vast majority of the features provided by Ratpack. You are well through your journey to being a Ratpack power developer!

Ratpack and Spring Boot

Earlier in the book, we discussed Ratpack's concept of registries and how registries are used to store and retrieve components that are used throughout the framework and your application. As you well know by now, registries are the mechanism that allow you to build Ratpack applications with no dependency injection (DI) with Guice and Guice modules. It is also through the `Registry` paradigm that we can accommodate an application architecture that employs multiple DI frameworks to allow for the most flexibility and use of best-in-breed tooling.

All of this is to say, a unique an important feature of Ratpack's is its ability to interoperate with one or more DI providers in isolation, allowing you to leverage the strengths of whatever component providing framework you choose, without the risk of those frameworks overlapping one another. From an application development perspective, if you continue to use the `Registry` as your mechanism for retrieving components, the underlying backing of that registry will not drastically change how your code looks, and will allow you to choose to bootstrap components in your application as is best suited to your requirements.

This is important to application development because there are many libraries available that ship with integrations specifically for Guice or Spring. With other web frameworks that do not provide you an abstraction on DI, your application needs to be structured around the DI provider that is already available, or you need to make a choice to architect your application around a single DI provider. With Ratpack, you are given the opportunity to leverage many DI providers to suit your needs. This also means that Ratpack's core features, which are provided as Guice modules, can continue to be leveraged even in a code base that wishes to bind application components through Spring.

Similar to Guice's integration with Ratpack, Spring Boot integration is achieved through a `Registry` implementation that has backing in a Spring application context.

Because registries can be layered, the components bound to Guice are able to be provided directly alongside those that were bound within a Spring application context. Indeed, given the `Registry` paradigm's flexibility, components that were bound in Guice are able to be overridden from a Spring-backed registry.

Spring is perhaps most well known for its popularity as a web framework, which provides application developers with a structure for building web applications around the model-view-controller design pattern. The underlying structure of Spring-based web applications relies on the Servlet API, which integrates with a web application container like Tomcat or Jetty. Modern Spring web applications that are built on Spring Boot can be packaged into standalone deployments that do not require a persistent web application container, though they still rely on it (this is usually provided as an embedded fixture of the application).

As we have already discussed, Ratpack is built on Netty's networking library, not servlets, and it does not require any web application container. To that end, servlet-based web applications cannot be affixed over top of Ratpack's HTTP layer as a stand-in replacement for request dispatching. Web components that are provided from a Spring web application context will not find a suitable home in a Ratpack application. `Controller` and `Filter` types are not supported as integrated components in a Ratpack application's HTTP request processing flow. Furthermore, other Java EE adaptations in Spring, such as rendering JSP views, will not be supported in Ratpack.

However, Spring is much more than just a web framework. Indeed, the web facets to Spring's arsenal of modules is but one higher-level layer to a robust and comprehensive underlying infrastructure. Modern Spring applications built using Spring Boot's convention-over-configuration engine can employ the vast majority of the Spring ecosystem of integrations without the need for ever integrating the servlet-based web layer.

Spring Boot is the basis upon which Ratpack and Spring integrate to give you the opportunity to leverage Ratpack's high-performance web layer with Spring's mature infrastructure and vast community. Developers who are accustomed to working with Spring can enjoy a familiar programming model, while garnering all the benefits that Ratpack provides. This includes all of the recent advancements that have taken place in Spring Boot to make the process of building and defining a Spring application structure so easy. Ratpack and Spring Boot's goals of providing a friendly development experience and producing lightweight deployables are so aligned that the integration of the two frameworks is a perfect fit for developers coming from a Spring background.

The differences between the two frameworks in developing an HTTP request processing flow will need to be considered. From a strictly architectural design perspective of web applications, regardless of the underlying web framework, you would reserve some sanity for yourself to have a clear delineation between the logic that was

responsible for parsing data from a request and the logic that was responsible for doing something with that data. To say that more concretely, the logic at the edge of your web application—in Ratpack, a `Handler`; in Spring, a `Controller`—should only have enough logic to get the data necessary out of the incoming request and delegate that data off to a service class to do some processing. The service class, in turn, would return some data that the request-taking logic would send back with the response.

Consider a scenario where you have a web application that provides a RESTful HTTP API for working with `Product` model objects. For the sake of simplicity, let's say that the API is only responsible for getting a `Product` by a specific `id` or returning a list of all `Product` types. The code in Example 9-1 shows what a traditional interface for this service might look like.

Example 9-1. ProductService interface

```
interface ProductService {
  Product get(Long id)
  List<Product> list()
}
```

Let's keep it simple and say that the `ProductService` is coming from legacy code or a client library and that it is non-async and blocks on both the `get` and `list` calls. By now, the code demonstrated in Example 9-2 should be familiar to you.

Example 9-2. Ratpack application with Product REST API

```
ratpack {
  bindings {
    bind(ProductService, BlockingProductService) ❶
  }
  handlers {
    prefix("product") {
      get(":id") { ProductService productService -> ❷
        def id = pathTokens.asLong("id")
        Blocking.get {
          productService.get(id)
        }.then { product ->
          if (product) {
            render(json(product))
          } else {
            response.status(404)
            render(json([status: "not_found"]))
          }
        }
      }
    }

    get { ProductService productService -> ❸
      Blocking.get {
```

```
            productService.list()
        }.then { products ->
            render(json(products))
        }
    }
  }
}
}
```

❶ We provide the binding of the `ProductService` interface to the assumed concrete implementation, `BlockingProductService`.

❷ This is the request handler for getting a `Product` by ID.

❸ This is the request handler for getting a list of `Product` types from the `Product Service`.

As you can see in the example, the handler logic for both getting `Product` types by ID and listing all products are focused strictly on dealing with the properties of the request and response lifecycle. All the heavy lifting of looking up products is done in our application's service layer through the `ProductService`.

Now, let's consider how the preceding Ratpack code translates to a Spring Boot application that is using a `Controller` to serve the `Product` REST API, as is shown in Example 9-3.

Example 9-3. Spring Boot application with Product REST API

```
@RestController
@RequestMapping("/product")
class ProductController { ❶

  private final ProductService productService

  @Autowired
  ProductController(ProductService productService) { ❷
    this.productService = productService
  }

  @RequestMapping(method = RequestMethod.GET)
  Product get(@PathVariable Long id, HttpServletResponse response) { ❸
    def product = this.productService.get(id)
    if (product) {
      return product
    }
    response.status = 404
    return null
  }
```

```
  @RequestMapping(method = RequestMethod.GET)
  List<Product> list() { ❹
    this.productService.list()
  }
}
```

❶ The ProductController is annotated with @RestController, which informs
   Spring that the resulting objects should be marshalled to their content-
   appropriate type. Also using the @RequestMapping annotation, we bind the con-
   troller to the /product endpoint.

❷ Here, we autowire the ProductService using constructor injection (which is
   favored over field injection).

❸ This is where we define the endpoint for getting a specific Product by its ID.

❹ This is the endpoint for listing all products.

This Spring Boot web application still needs a bit more work. Let's assume, for the
sake of brevity, that the application is a standalone Spring Boot (i.e., defines a runna-
ble Main class) and that the Spring dependencies are contained within the Main class.
The code for the Main class is shown in Example 9-4.

Example 9-4. Main class for Spring Boot Product API application

```
@SpringBootApplication
class Main { ❶

  @Bean
  ProductService productService() { ❷
    new BlockingProductService()
  }

  static void main(args) {
    SpringApplication.run(Main, args) ❸
  }
}
```

❶ We annotate the Main class with @SpringBootApplication, which among many
   other things, will component scan the classpath, allowing the ProductControl
   ler to participate in the web application.

❷ This is where we bind the ProductService. We can bind it in the Spring applica-
   tion context using the @Bean annotation on the method.

❸ This is the starting point to the Spring Boot application.

With all the pieces in place, we have a good parallel between Ratpack and Spring Boot implementations of the `Product` REST API. These examples illustrate how in both implementations the logic that is responsible for managing the request-response life-cycle is focused solely on that task, and leaves the actual work of getting the `Product` types to the `ProductService`.

If you follow this pattern of implementing request handling logic separate from the heavy lifting of gathering data for a request, then you are well on your way to employing Spring Boot in your Ratpack applications. By the end of this chapter, you will understand how to take the `@Bean` definitions from a Spring Boot application and make them accessible to your `Handler` types in Ratpack.

Adding Spring Boot to Your Ratpack Project

Like most of Ratpack's framework-level features, the Spring Boot integration is provided as an optional module that you can incorporate into your project's Gradle build script. The build script in Example 9-5 demonstrates a Ratpack Groovy project with Spring Boot support.

Example 9-5. Gradle build script with Ratpack Spring Boot dependency

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency("spring-boot") ❶
}
```

❶ We use the `ratpack.dependency(..)` faculty, as we would any other framework dependency.

With the Spring Boot dependency in place, we can create a Spring Boot configuration class, which is where we will define the bean definitions and enable Spring Boot's convention-over-configuration mechanisms to take over the bootstrapping of com-

ponents. Example 9-6 demonstrates how simple it is to build Ratpack applications with Spring Boot.

Example 9-6. Spring Boot configuration class

```
package app

import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication ❶
class AppSpringConfig {
}
```

❶ As noted in the introductory section to the chapter, `@SpringBootApplication` does a lot of work to detect application-level components and make them available in the Spring application context.

This alone is enough to give you a plethora of functionality from Spring. When the `@SpringBootApplication` annotation is provided, Spring Boot's *autoconfiguration* engine is activated, and the classpath will be scanned for certain packages and types, for which, if found, Spring will create the appropriate bean definitions that you will be able to use throughout your Ratpack application.

If we build on the example from earlier in the chapter and say that we want our Ratpack application to provide a RESTful API for a `Product` domain, then we can clearly outline the true strengths that Spring Boot brings. To begin implementing the `ProductService` shown earlier, we will need some access to a database. The discussion in the previous chapter could serve as one solution to this problem. However, given that we are now working in the Spring ecosystem, our application can make use of a powerful tool that is now at our disposal: Spring Data.

Spring Data provides a mature and comprehensive platform for building data access objects and working with data-driven domain models. Through Spring Data, and thus in our Ratpack application, we can even leverage the Java Persistence API, which should be familiar to seasoned Java developers. To begin going down this path, we will need to add some additional dependencies from Spring to our project. The updated Gradle build script in Example 9-7 applies a so-called "starter" dependency for Spring Data's JPA integration.

Example 9-7. Gradle build script with Spring data starter

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
```

```
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency("spring-boot")
  compile 'org.springframework.boot:spring-boot-starter-data-jpa:1.2.4.RELEASE' ❶
  compile 'com.h2database:h2:1.4.187' ❷
}
```

❶ Here we add the Spring Data JPA starter dependency, which provides the dependencies necessary for Spring Boot to bootstrap the Spring Data JPA components.

❷ We will also add the H2 embedded database driver, which Spring Boot will recognize and automatically configure a datasource for Spring Data.

Note that we will use the H2 embedded database driver simply for demonstration's sake. The Spring Data JPA project page has comprehensive documentation on using drivers for other databases.

The next step will be to create our Product domain class as a JPA entity. We can use standard JPA annotations to define the necessary properties of the model object. Example 9-8 shows the domain model we will be working with.

Example 9-8. Product JPA entity

```
package app

import javax.persistence.Column
import javax.persistence.Entity
import javax.persistence.GeneratedValue
import javax.persistence.Id

@Entity
class Product {
  @Id
  @GeneratedValue
  Long id

  @Column(nullable = false)
  String name

  @Column(nullable = false)
  String description
```

```
  @Column(nullable = false, precision = 7, scale = 2)
  BigDecimal price
}
```

When Spring Boot's autoconfiguration engine detects the JPA libraries on the class-path with the H2 embedded database, it will automatically create the database schema on our behalf. It will also automatically enable JPA repository types within Spring Data, which will allow us to succinctly create a data access object for the `Product` database table. The sample code in Example 9-9 demonstrates how much power and simplicity Spring Boot can bring to your project.

Example 9-9. Spring Data ProductRepository

```
package app

import org.springframework.data.repository.CrudRepository
import org.springframework.stereotype.Repository

@Repository ❶
interface ProductRepository extends CrudRepository<Product, Long> { ❷
}
```

❶ We annotate the interface with `@Repository`, which instructs Spring Data that it should utilize this interface as a data access object.

❷ The interface extends the `CrudRepository` interface, which provides all of the necessary create-read-update-delete methods that you could ever need.

This is literally all that is needed for Spring Boot to create a data access layer and bootstrap the schema on our embedded database.

Creating a Spring Boot–Backed Registry

Now that we have built all of the pieces necessary for Spring Boot to do the heavy lifting, we must incorporate the component bindings into our Ratpack application. This is made simple through a factory method provided as part of the `ratpack-spring-boot` module that was already incorporated as a project dependency earlier in the chapter.

It was outlined earlier in the book in the discussion on registries, but at this point it is especially important to understand that there are several different layers of registries in Ratpack, thus giving several different ways to incorporate your Spring Boot config-uration into your Ratpack application. Given a Ratpack Groovy application that uti-lizes the *ratpack.groovy* script to define the application structure, you will need to define the Spring Boot–backed registry as part of your handler chain. To accomplish

this, you will first need to create the registry from the `AppSpringConfig` class shown in the prior section. The *ratpack.groovy* script shown in Example 9-10 demonstrates how to do this.

Example 9-10. Ratpack Groovy script with Spring Boot registry

```groovy
import app.AppSpringConfig
import static ratpack.groovy.Groovy.ratpack
import static ratpack.spring.Spring.spring
import static ratpack.jackson.Jackson.json
import ratpack.exec.Blocking

def springRegistry = spring(AppSpringConfig) ❶

ratpack {
  handlers {
    register(springRegistry) ❷

    prefix("product") {
      get(":id") { ProductRepository productRepository -> ❸
        def id = pathTokens.asLong("id")
        Blocking.get {
          productRepository.findOne(id)
        }.then { product ->
          if (product) {
            render(json(product))
          } else {
            response.status(404)
            render(json([status: "not_found"]))
          }
        }
      }
      get { ProductRepository productRepository ->
        Blocking.get {
          productRepository.findAll()
        }.then { products ->
          render(json(products))
        }
      }
    }
  }
}
```

❶ Using the `ratpack.spring.Spring.spring` method, we can use the `AppSpring Config` class to get a handle on a `Registry` implementation that provides components from the Spring application context.

❷   Here, we use the handler chain's `reigster` method to apply the `springRegistry` to the rest of the chain. When using the *ratpack.groovy* script, this is the best way to accomplish Spring Boot integration.

❸   With the Spring Boot–backed registry applied, we can inject the `ProductReposi tory` and make use of it from within our handlers.

Although our API only surfaces methods for getting and listing `Product` types, the `ProductRepository` is very capable of creating, updating, and deleting `Product` entires from the database. To better demonstrate this, if we wanted to bootstrap some data before our application starts up, we can do so just prior to the application defini-tion, still by leveraging the Spring Boot–backed registry. The code snippet in Example 9-11 demonstrates inserting two entries prior to the application starting.

Example 9-11. Bootstrapping product data

```groovy
def springRegistry = spring(AppSpringConfig)
def repo = springRegistry.get(ProductRepository) ❶
def product1 = new Product( ❷
  name: "Learning Ratpack",
  description: "The Best Book on Ratpack so far",
  price: 42.99
)
def product2 = new Product(
  name: "Programming Grails",
  description: "Top 10 Book on Grails",
  price: 38.99
)
repo.save([product1, product2]) ❸

ratpack {
  // ... rest of app ...
}
```

❶   We use the `springRegistry` to extract the `ProductRepository`.

❷   Create a couple of `Product` objects to be stored.

❸   Call the `ProductRepository#save` method to save them.

If you add these changes in, start the application, and navigate to *http://localhost: 5050/product*, you will see the JSON rendering of the product definitions we have put in place, exactly as expected. You can test further by navigating to *http://localhost: 5050/product/1*, which will provide you with the JSON rendering of the `product1` entry.

This strategy makes your Spring Boot bindings available to your application through a *context registry*. For most use cases, this is fine. However, if you build your Ratpack application from a `Main` class, you can leverage Spring Boot as a *user registry*, which has some added benefits. Most specifically, components that you bind in your Spring Boot configuration will be able to participate in the application startup event.

Consider the `Main` class shown in Example 9-12.

Example 9-12. Ratpack Main class with Spring Boot

```groovy
package app

import ratpack.server.RatpackServer
import ratpack.exec.Blocking

import static ratpack.spring.Spring.spring

class Main {
  static void main(args) {
    RatpackServer.start { spec -> spec
      .registry(spring(AppSpringConfig)) ❶
      .handlers { chain ->
        chain.prefix("product") { pchain ->
          pchain.get(":id") { ctx ->
            ProductRepository productRepository = ctx.get(ProductRepository) ❷
            def id = pathTokens.asLong("id")
            Blocking.get {
              productRepository.findOne(id)
            }.then { product ->
              if (product) {
                render(json(product))
              } else {
                response.status(404)
                render(json([status: "not_found"]))
              }
            }
          }
          pchain.get { ctx ->
            ProductRepository productRepository = ctx.get(ProductRepository) ❸
            Blocking.get {
              productRepository.findAll()
            }.then { products ->
              render(json(products))
            }
          }
        }
      }
    }
  }
}
```

❶ You can see here that we are using the `Spring.spring(..)` call to build the Spring Boot–backed registry and apply it to the `registry(..)` method on the application definition. This places the registry as a *user registry*.

❷ Using this manner of constructing the application, we need to change our code to pull the `ProductRepository` from the `Context` object that is supplied to our `Handler` closure.

❸ Similarly, when listing all objects, we can pull the `ProductRepository` from the `Context`.

Note that in this example we are not doing any explicit bootstrapping of data. To do this, let's change things a little bit, and drive the bootstraping data from configuration that we pull using Ratpack's configuration mechanisms. If we create a `ProductBoot strapConfig` class, as shown in Example 9-13, and inform Ratpack that it should bind its configuration sources to it, then we can begin to use data-driven techniques to initialize our data set. The modified application definition is shown in Example 9-15.

Example 9-13. ProductBootstrapConfig class

```
package app

class ProductBootstrapConfig {
        List<Product> products  ❶
}
```

❶ This is all that we need on this class for now. We can use the model directly, as it is essentially just a Java (Groovy) bean.

We will map to this class from the YAML configuration file shown in Example 9-14.

Example 9-14. Bootstrap YAML config file

```
product:
  products:
    - name: Learning Ratpack
      description: The Best Book on Ratpack so far
      price: 42.99
    - name: Programming Grails
      description: Top 10 Book on Grails
      price: 38.99
```

Now, we can modify the application's definition to incorporate the configuration file and map it to the `ProductBootstrapConfig`.

Example 9-15. Main class using Config to provide bootstrapping data

```
RatpackServer.start { spec -> spec
  .serverConfig { b -> b ❶
    .yaml("bootstrap.yml") ❷
    .sysProps()
    .env()
    .require("/product", ProductBootstrapConfig) ❸
    .baseDir(BaseDir.find())
    .build()
  }
  .registry(Spring.spring(AppSpringConfig))
  .handlers { chain ->
    // handlers
  }
}
```

❶ We configure the application's `ServerConfig` here in the definition.

❷ And reference the *bootstrap.yml* file. Reminder that this file exists within the base directory of the project. For the purposes of the main class demonstration, the later call to `baseDir(BaseDir.find())` will scan the classpath for the `.ratpack` marker file to know where to get *bootstrap.yml* from.

❸ Then, we map the config's data from the top-level `product:` key to the `Product BootstrapConfig` class, which will become accessible in the registry.

Next, we can modify the `AppSpringConfig` class to provide Ratpack with a `Service` implementation that can participate in the application startup event, as shown in Example 9-16. Remember that this is possible because we are providing the Spring Boot–backed registry as a *user registry*.

Example 9-16. Service instance in AppSpringConfig

```
package app

import ratpack.service.Service
import ratpack.service.StartEvent
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.context.annotation.Bean

@SpringBootApplication
class AppSpringConfig {

  @Bean
  Service bootstrapDb(ProductRepository repo) { ❶
    new Service() {
      void onStart(StartEvent e) throws Exception {
        def registry = e.registry ❷
```

```
        def bootstrapConfig = registry.get(ProductBootstrapConfig) ❸
        repo.save(bootstrapConfig.products) ❹
      }
    }
  }
}
```

❶   Here, we create a bean definition for our `Service` class, and we use method injection to inform Spring that we want access to the `ProductRepository` from here.

❷   The `StartEvent` that is provided to the `Service#onStart` method is the joined server and user registries, so we can access any component that has been included as part of the application definition.

❸   Because we used the `require(..)` call on the `serverConfig` block of our application definition to map the product config, we can access it here.

❹   Finally, we can save the `Product` definitions that were mapped as part of the configuration binding.

This demonstration of interoperating Ratpack and Spring Boot mechanisms to provide a data-driven bootstrapping sequence shows how well integrated the two frameworks can be.

API Design with Ratpack and Spring

It is important when we are designing APIs with Ratpack and Spring to remember that we have some powerful faculties at our disposal for making asynchronous APIs from those that are not inherently async. If you recall from earlier in the book, Ratpack's execution model provides deterministic execution of asynchronous calls, and using the `Blocking` mechanism can adapt synchronous APIs to asynchronous functionality.

The `ProductRepository` that we have built from Spring Data is powerful, easy to use, and easy to build, but it is not asynchronous and is certainly not nonblocking. It would be a design flaw to simply accept that our handler logic had to wrap the calls to the repository in `Blocking`, as that makes inferences about how the service should be operating. It would be more beneficial for us to design another service that uses the `ProductRepository`, but allows our handlers to work with `Promise` types that abstract away the synchronous nature of the underlying repository. That way, we can ensure that our handler code remains clean and unaware of the rest of the code's implementation detail. Also, in the future, if we find an asynchronous backing that is more suitable than the `CrudRepository` provided by Spring Data, we simply change out the backing implementation with no need to change our code.

To begin, let's create a `ProductService` interface that will act as our service interface for the `Product` data. The code in Example 9-17 shows the contracts we will use.

Example 9-17. ProductService interface

```
package app

import ratpack.exec.Promise

interface ProductService {

  Promise<Product> get(Long id) ❶

  Promise<List<Product>> list() ❷
}
```

❶ We define the contract for the method that will be used to get an individual `Product` by ID. Note that we are using the `Promise` return type now.

❷ Similarly, we create a `list()` method that returns a `Promise` for the `List` of `Product` types.

Next, we must create a concrete implementation that we bind to the `ProductService`. Because we will be using our Spring Data repository, we can use the implementation demonstrated in Example 9-18.

Example 9-18. Spring Data ProductService implementation

```
package app

import org.springframework.stereotype.Service
import org.springframework.beans.factory.annotation.Autowired
import ratpack.exec.Promise
import ratpack.exec.Blocking

@Service ❶
class SpringDataProductService implements ProductService {

  private final ProductRepository repo

  @Autowired
  SpringDataProductService(ProductRepository repo) { ❷
    this.repo = repo
  }

  @Override
  Promise<Product> get(Long id) {
    Blocking.get { ❸
      repo.findOne(id)
```

```
    }
  }

  @Override
  Promise<List<Product>> list() {
    Blocking.get { ❹
      repo.findAll()
    }
  }
}
```

❶ We will use the `@Service` Spring stereotype annotation to define the function of this service. Because our `AppSpringConfig` class is doing component scanning, we do not need to explicitly add this bean definition—it will be detected and bound automatically.

❷ The `@Autowired` can be used on the constructor to inject the `ProductRepository` into our service class.

❸ We use `Blocking` here to inform Ratpack's execution model about where to execute the code segment. Our handler logic will stay clean and be agnostic of the fact that our underlying implementation is synchronous.

❹ `Blocking` operates here in the same way as described in the previous instance.

Given that we now have a concrete `ProductService` implementation that will be automatically incorporated into the Spring application context, we can change our handler logic to use this class instead of the `ProductRepository` directly.

Other Notes on API Design with Ratpack and Spring

Our application handlers are still probably doing too much, considering they are making opinions about *how* the `Product` domains should be serialized back to callers. That is to say, in the current example's implementations, we only support rendering back as JSON. That is probably fine for a simple use case, but a more reasonable and robust implementation would be to provide a `Renderer` for the `Product` classes.

`Renderer` types can also be made available to Ratpack through the Spring Boot registry, and furthermore can be picked up automatically by Spring's component scanning, when annotated properly. The code in Example 9-19 shows the `ProductRenderer` class that will make our handler logic simpler.

Example 9-19. ProductRenderer Spring component

```
package app
```

```
import org.springframework.stereotype.Component
import ratpack.handling.Context
import ratpack.render.Renderer
import ratpack.groovy.Groovy

import static ratpack.jackson.Jackson.json

@Component ❶
class ProductRenderer implements Renderer<Product> {

  @Override
  Class<Product> getType() {
    Product
  }

  @Override
  void render(Context ctx, Product product) throws Exception {
    ctx.byContent { spec -> spec ❷
      .json {
        ctx.render(json(product))
      }
      .xml {
        // provide XML rendering
      }
      .html {
        ctx.render(Groovy.groovyTemplate([product: product], "product.html"))
      }
    }
  }
}
```

❶ We can annotate the `ProductRenderer` with the `@Component` stereotype, and Spring will automatically pick it up and make it available to Ratpack through its registry.

❷ In the `render` method, we can use the `Context#byContent` mechanism to render the provided `Product` according to the type the caller is capable of receiving.

Given this, our handler logic can be simplified. If we build on the `ProductService` example from earlier in this section, we can see what our updated handler logic would look like. The updated Ratpack Groovy script is shown in Example 9-20.

Example 9-20. Ratpack Groovy script with ProductService and Renderer

```
import app.AppSpringConfig
import ratpack.groovy.template.TextTemplateModule
import static ratpack.groovy.Groovy.ratpack
import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.spring.Spring.spring
import static ratpack.jackson.Jackson.json
```

```
def springRegistry = spring(AppSpringConfig)

ratpack {
  bindings {
    module TextTemplateModule
  }
  handlers {
    register(springRegistry)

    prefix("product") {
      get(":id") { ProductService productService ->
        def id = pathTokens.asLong("id")
        render productService.get(id) ❶
      }
      get { ProductService productService ->
        productService.list().then { products ->
          byContent { ❷
            json {
              render(json(products))
            }
            xml {
              // render XML of products
            }
            html {
              render groovyTemplate([products: products], "products.html")
            }
          }
        }
      }
    }
  }
}
```

❶   You can see here the logic for rendering a `Product` back to clients is drastically simplified. We make use of the fact that Ratpack can render `Promise` types directly and ascertain the appropriate `Renderer` implementation.

❷   Note that due to type erasure in the JVM, we cannot provide a `Renderer` for generic `List` types, so our handler still has to do some of the rendering logic directly.

The most important takeaway from this section is that we should always be cognizant of *how* the data is being accessed by the underlying implementation, and take measures to ensure that we are simplifying our handler logic, and letting our service tier do the heavy lifting. When we follow this architectural paradigm within Ratpack, we can more heavily leverage the Spring Boot integration throughout our application.

Known Limitations

It is a known limitation that Guice and Spring do not cleanly interoperate with one another. Many different approaches have been taken to help bridge component bindings between Guice and Spring, but there is still not a single definitive way to interconnect the two frameworks. With Ratpack, that is OK, because we can access component bindings through the provided registries. But Ratpack does not take any additional measures to ensure that Guice bindings are available in the Spring application context, or vice versa. That is to say, you should not expect that framework-level components bound from Guice modules would be available as autowire-candidates in your Spring application context. When you need access to component bindings that extend beyond Guice and Spring, however, the Ratpack registry can be used to accommodate.

For example, a `ratpack.service.Service` implementation gets access to the user registry at application start time via the `onStart` method, as shown earlier. From within this block, you can access components that were bound in the Guice server registry from within your Spring components, and vice versa.

When you are using both Spring Boot and Guice bound components, it is advisable to make use of the registry to get access to components.

Chapter Summary

This chapter has provided a simple introduction to incorporating Spring Boot into your Ratpack projects. A small demonstration of integrating with the Spring ecosystem has been shown through the examples that leverage Spring Data, but the world of Spring is vast, with much to offer. Many of the other available Spring projects, including Spring Cloud, can be brought into your Ratpack projects to help you leverage some of the most sophisticated technologies available on the JVM.

Reactive Programming in Ratpack

No conversation on Ratpack would be complete without a discussion on its reactive and functional programming capabilities. Indeed, at its core, Ratpack employs reactive programming strategies to provide the framework-level features for which you are already familiar. Indirectly, you have already been exposed to some of its reactive programming paradigms. This chapter will serve to further round out your understanding of how Ratpack utilizes reactive and functional programming, and how you can leverage the mechanisms it provides in your own applications.

This chapter does not provide exhaustive coverage of all reactive programming paradigms, for this subject is gross and expansive in its own right. Instead, we will cover the capabilities offered to you by Ratpack and its host of framework features that serve as the underpinnings for building robust reactive systems. Generally speaking, most Ratpack applications can be built in entirety without the developer ever having to think about the fact that they are writing reactive code. This is one of the many strong suits to Ratpack's easy-to-understand API; however, when the need arises for more complex reactive implementations, the capabilities are there, and Ratpack will support you through your development process.

Overview of Reactive Programming

The conversation of reactive programming has become a hot topic in the JVM space within just the last few years. Other languages have long since had the concept of reactive programming, and some of those languages—namely those in the .NET family—have even formalized the data structures and interfaces upon which reactive systems are built. It has not been until relatively recently, however, that those concepts have been implemented and brought to the Java ecosystem.

Functional and reactive programming often go hand in hand with one another, wherein a reactive system is supported by functional programming interfaces. This is evident in the reactive infrastructure provided by Ratpack, as most of the interfaces for performing reactive operations rely on functional programming interfaces and interfaces with single abstract methods. These interfaces provide the footprint upon which complex reactive processing pipelines are built, and as we explore the facilities in place to accommodate reactive programming in Ratpack, you will begin to better understand how the framework makes it as easy as possible to build reactive applications.

Before we can delve into the fixtures that Ratpack provides for reactive programming, we must first answer the most fundamental question in this conversation: what is reactive programming? The answer to this question comes in many forms, but the most definitive definition could perhaps be explained as follows: reactive programming is a technique by which a processing function requests data from a producing function, a single element at a time. Multiple elements are delivered through a pipeline, wherein one or many functions can affect the flow and form of data.

A reactive processing pipeline can be thought of in its simplest form as a two-part system that is comprised of a *producer* and a *subscriber*. The reactive part of the process is initiated when the subscriber requests data from the producer. Data is said to be *streamed* to the subscriber from the producer until the producer has no more data to provide, at which time the processing pipeline is terminated.

The system of streaming data from producer to subscriber allows the subscriber to remain focused on the processing of a single element of data, and is amenable to a functional programming style. Following functional programming paradigms, a subscription, given an input, will always produce the same output. That is to say, if the same data goes into the subscribing function, it will always produce the same result.

Working in the reactive programming style means that opinions about how the data is processed can easily be imposed upon the functions of a pipeline, without any explicit involvement from user code. Given that you have a firm understanding of Ratpack's execution model, you can likely understand now how the functions in a reactive pipeline are able to be adapted into Ratpack's processing system to ensure determinstic execution. Indeed, the execution model in and of itself is built upon the same technique of reactive pipelining that your application code uses.

Another aspect to reactive programming is that the various functions comprising a processing pipeline are able to be composed in a way that allows for robust strategies of data processing, including transformation, composition, collections, and map/reduce strategies. If you consider the diagram in Figure 10-1, you can see how the subscription at the bottom of the chain initiates the processing flow, thereby requesting a data element from its upstream producer, which is a transforming function that is both a producer and subscriber.
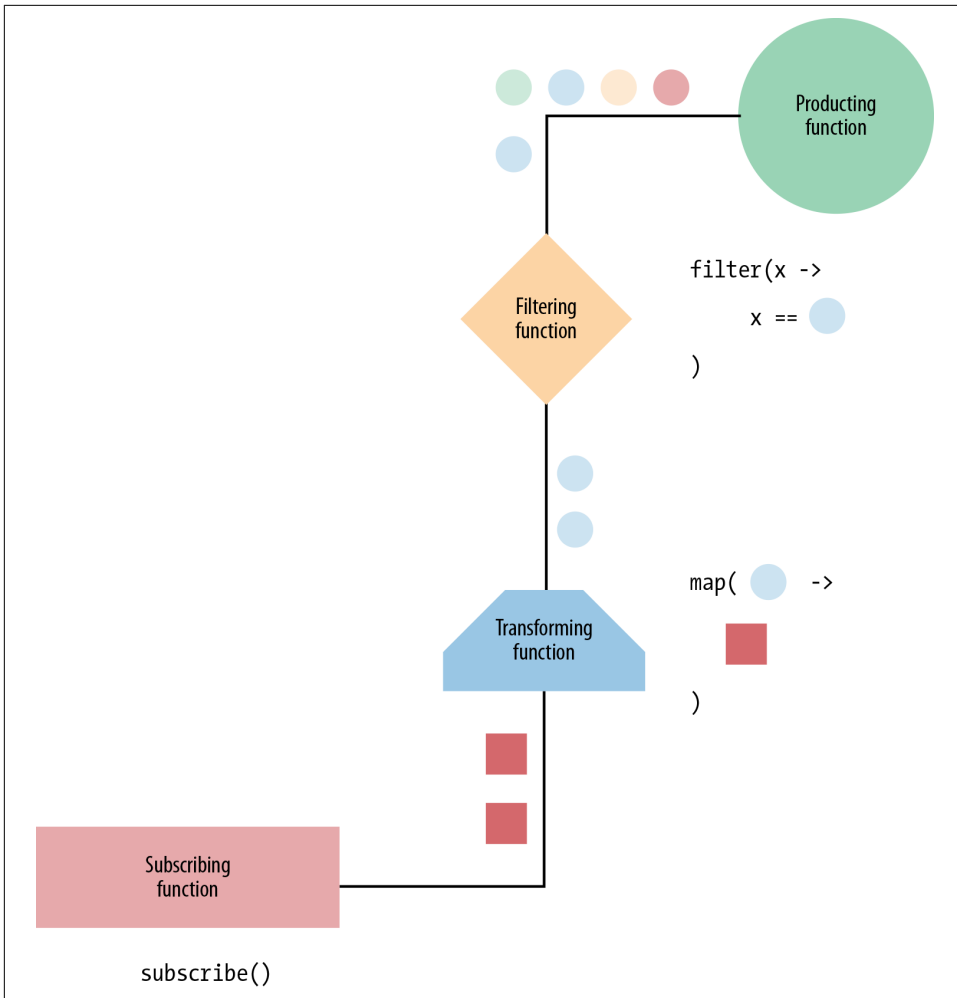
Figure 10-1. Reactive programming diagram

As you can see from the processing flow, the subscriber at the bottom of the pipeline initiates the processing flow, thereby requesting data from its producing function, which is shown here as the *transforming function*. The transforming function then requests a data element from its upstream producer, the *filtering function*. A request is then made to the *producing function*, which is the source of the data stream, to begin emitting elements.

In this example, let's say the producing function is getting its data elements from a database, say by running a `SELECT * FROM TABLE` query. Each of the rows returned from that query is then streamed to the downstream subscriber. Again, following our example, let's say the filtering function in the pipeline is responsible for ensuring that

only rows that satisfy a predicate condition are streamed down the pipeline. Those rows that make it through the filtering function are then sent to the transforming function, which would be responsible for taking the row data and turning it into a model object, for example. Finally, the subscriber that initiated the processing flow is given the fully composed model object and sends it back to a consumer of our web application.

This example serves strictly to exemplify the flow of data in a reactive pipeline. Indeed, the possibilities for subscriber and producer implementations is definitively endless. That said, there are many common processing use cases that systems like those employed by Ratpack surface for the everyday needs of development. The following list outlines some of those use cases and describes their function:

map
    Transforms a data element from one structure to another.

filter
    Checks a predicate condition against the data element to see if it should be sent downstream for processing.

flatMap
    Provides the means to map the result of a `Promise` into the stream.

wiretap
    Data from the stream will be sent to this "listening" function prior to being sent to the subscriber. This method is useful for tracing the flow of data through a stream.

As we delve into Ratpack's support for reactive programming, you will find that there are various levels of reactive programming strategies available to you. From the most simplistic, `Promise.then(..)` processing, to more robust reactive data streaming, and even further into Ratpack's integration with the comprehensive RxJava reactive programming library, all the faculties you may ever need for building robust reactive systems are at your fingertips.

Promise as a Reactive Data Structure

As we tour the showcase of reactive programming capabilities offered by Ratpack, we first encounter an old friend with whom we are by now quite familiar, the `Promise` type. Until this point, it may not have been clear that a `Promise` in Ratpack is actually a reactive programming data structure, but as we look at some of its more advanced usages, you will begin to understand how well it fits the very definition of reactive programming. The only deviation you will find in the `Promise` processing flow from how other reactive processing systems work is that a `Promise` is guaranteed to only ever emit a single element into its stream. This is important to understand, because it

means that functions in the reactive pipeline need to be prepared to work with what-ever data is emitted from the `Promise`. This could be a single element of data or a col-lection of data, depending on what the promise-returning function delivers to the subscription.

If we analyze a `Promise` type's usage in strict reactive terminology, we can begin to understand how the different parts of working with a `Promise` fit into the reactive programming mold. As you are by now well familiar with building APIs that return `Promise`, let's revisit a familiar example from before of a data access object that pulls some data from our application's database and renders it back in JSON format. The code in Example 10-1 shows our `BlockingDatabaseService`.

Example 10-1. BlockingDatabaseService implementation

```groovy
package app

import javax.inject.Inject
import groovy.sql.Sql
import ratpack.exec.Promise
import ratpack.exec.Blocking

class BlockingDatabaseService {

  private final Sql sql

  @Inject
  BlockingDatabaseService(Sql sql) {
    this.sql = sql
  }

  Promise<User> getUser(String username) {  ❶
    Blocking.get {
      def row = sql.firstRow("SELECT * FROM USER WHERE USERNAME = $username")
      if (row) {
        new User(
          username: row["username"],
          email: row["email"],
          password: row["password"]
        )
      } else {
        null
      }
    }
  }

  Promise<List<User>> getUsers() {  ❷
    Blocking.get {
      sql.rows("SELECT * FROM USER").collect { row ->
        new User(
          username: row["username"],
```

```
            email: row["email"],
            password: row["password"]
          )
      }
    }
  }
}
```

❶ The `getUser` method returns a `Promise` for a single `User` object.

❷ The `getUsers` method returns a `Promise` for a `List` of `User` objects.

As you can see from this implementation, the `getUser` method is responsible for returning a single `User` object, while the `getUsers` method returns an entire `List` of `User` objects. The `Promise` here represents the *producer* in reactive programming terms. However, it is important to understand that no processing will take place here until there is a subscription that requests the data. That is where the code in our handler logic comes into play. Consider the corresponding Ratpack application shown in Example 10-2, which utilizes the `BlockingDatabaseService` to serve endpoints for accessing user data.

Example 10-2. Ratpack Groovy application

```
import ratpack.groovy.sql.SqlModule
import ratpack.hikari.HikariModule
import app.BlockingDatabaseService

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module(SqlModule)
    module(HikariModule) { c ->
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'
      c.username = 'sa'
      c.password = ''
    }

    bind(BlockingDatabaseService)
  }
  handlers {
    prefix("user") {
      get(":username") { BlockingDatabaseService db ->
        db.getUser(pathTokens.username).then { user ->  ❶
          render json(user)
        }
      }
```

```
      get { BlockingDatabaseService db ->
        db.getUsers().then { users -> ❷
          render json(users)
        }
      }
    }
  }
}
```

❶ The call to `BlockingDatabaseService#getUser` emits a single item, and within our handler we are prepared to deal with that.

❷ The call to `getUsers`, however, emits a list of users, and in our `get` handler here that is exactly what we expect.

The second part to understanding the reactive nature of `Promise` types is understanding that when your code makes the call to `Promise#then`, you are actually attaching a subscription to the `Promise`. The `Promise` is the *producer* and the block of code that you supply to the `then(..)` method is the function representing your subscription.

Transforming Data with Promises

Taking your understanding of `Promise` types as reactive data structures to the next level, there are many associated methods on `Promise` that can help you build comprehensive reactive processing pipelines. For example, given the `BlockingDatabaseSer` `vice` code that we are demonstrating now, it may not be desirable for our application's RESTful API to render back the password hash that gets populated into the `User` object's `password` property. To accommodate this, we will need to *transform* the data that is being produced, and we can do so as part of a reactive processing pipeline in our handler. The updated handler chain in Example 10-3 shows how we use the `Promise#map` method to transform our `User` objects into `Map` types that are stripped of any property besides the `username` and `email`.

Example 10-3. Using map to transform Promise data

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.groovy.sql.SqlModule
import ratpack.hikari.HikariModule
import app.BlockingDatabaseService
import ratpack.jackson.Jackson.json

ratpack {
  bindings {
    module(SqlModule)
    module(HikariModule) { c ->
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'
```

```
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'
      c.username = 'sa'
      c.password = ''
    }

    bind(BlockingDatabaseService)
  }
  handlers {
    prefix("user") {
      get(":username") { BlockingDatabaseService db ->
        db.getUser(pathTokens.username).map { u ->
          [username: u.username, email: u.email]
        }.then { map ->
          render json(map)
        }
      }
      get { BlockingDatabaseService db ->
       db.getUsers().map { users ->
          users.collect { u ->
            [username: u.username, email: u.email]
          }
       }.then { maps ->
          render json(maps)
       }
      }
    }
  }
}
```

As you can see now, the `Promise` types returned from `getUser` and `getUsers` have
attached a transforming function into the processing stream via the `map` method,
which is responsible for putting the data into the appropriate form before the `then`
subscription sends the data back to the client.

Filtering Data with Promises

Because `Promise` types are guaranteed to emit but a single element to their stream,
filtering the stream does not necessarily apply in strictly reactive programming terms.
However, Ratpack provides a mechanism off of `Promise` for you to reactively inspect
the item and branch processing according to a predicate function. This operates in
practice similar to filtering, except that it is slightly more simplistic and representative
of if-then-else style processing in a reactive way.

Building on the prior example of the handler logic that we had set up for getting a
single user, we can attach the `route` function to the processing pipeline to inspect the
returned user object and make a decision about what goes back to the client. If you
consider the handler logic depicted in Example 10-4, you will find a practical demon-
stration of using `route` with a `Promise` type.

Example 10-4. Routing promises

```
get(":username") { BlockingDatabaseService db ->
  db.getUser(pathTokens.username).map { user ->
    user ? [username: user.username, email: user.email] : null ❶
  }.route { u -> u == null } { u -> ❷
    response.status(404)
    render json([error: "not found"])
  }.then { user -> ❸
    render json(user)
  }
}
```

❶ Because `Promise` types are guaranteed to always emit a single item, they will also emit null values. Here, we apply a null value check against the user, and map it accordingly or pass-through the null value otherwise.

❷ Using the `route` method, we supply two parameters: a predicate function and an action function. If the predicate function returns a true response, the action function is invoked.

❸ The `then` subscription to the `Promise` is invoked if the `route`'s predicate function was not truthful. Note that the `then(..)` subscription is still required to invoke the processing stream.

As you can see in this example, when the user object returned is `null`, we set the appropriate response status and send back a "not found" error message to the client. This is a perfect use case for "filtering" data to some other processor according to its state or properties.

For the purposes of completeness, the prior example can be simplified using a short-hand method on `Promise` that allows us to route according to whether the value emitted is `null` or not. Adding the `onNull` method into the processing pipeline, we can reduce the complexity of the code, as shown in Example 10-5.

Example 10-5. Routing promises with onNull

```
get(":username") { BlockingDatabaseService db ->
  db.getUser(pathTokens.username).map { user ->
    user ? [username: user.username, email: user.email] : null
  }.onNull {
    response.status(404)
    render json([error: "not found"])
  }.then { user ->
    render json(user)
  }
}
```

More complex reactive programming techniques will need to be employed to filter single elements from a collection of elements. That is to say, for methods that return a `Promise` of type `List`, where you wish to filter one or more objects out of that list, you will want to utilize some of the more advanced reactive programming mechanisms provided by Ratpack.

Composing Data with Promises

As you no doubt have found by now, comprehensively working with data in a robust web application will mean pulling information from many (possibly disparate) sources. At this point you have already been exposed to the `Promise` type's `flatMap` method, which provides the ability to transform the arity of a `Promise` stream to another type, based on the result of another `Promise` type. To fully exemplify this capability, let's consider a scenario where we have a web application that first makes a call to a database to get a user model object, then makes a subsequent call to a remote web service to get the user's profile. Properties of the two resulting objects are used to compose a comprehensive model object that we serve back from our application's handler. If we consider the service interfaces shown in Example 10-6 and Example 10-7, we can see the APIs we will need to work with.

Example 10-6. The Promise based UserService interface

```
package app

import ratpack.exec.Promise

interface UserService {

  Promise<User> getUser(String username)
}
```

Example 10-7. The Promise based UserProfileService interface

```
package app

import ratpack.exec.Promise

interface UserProfileService {

  Promise<UserProfile> getUserProfile(String username)
}
```

Notice that while the `getUserProfile` method on the `UserProfileService` retrieves a `UserProfile` by username, we likely will not want to make this call until we have succesfully retrieved the `User` object from the `UserService`. From our application's RESTful API, we want to provide an endpoint that allows aspects of the `User` object as

well as the `UserProfile` to be provided strictly by username. To accommodate this, we will need to first get the `User` object, then subsequently retrieve the `UserProfile` object, before responding to the request.

Using the `Promise#flatMap` method, we can easily stream these two calls into a final comprehensive model. A simple implementation for this might look like the one shown in Example 10-8.

Example 10-8. Composing promises

```
import app.*

import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

ratpack {
  bindings {
    bind(UserService, DefaultUserService)
    bind(UserProfileService, DefaultUserProfileService)
  }
  handlers {
    prefix("user") {
      get(":username") { UserService userService,
                         UserProfileService profileService ->
        userService.getUser(pathTokens.username).flatMap { user -> ❶
          profileService.getUserProfile(pathTokens.username)
            .map { userProfile -> ❷
              [
                username: user.username,
                email: user.email,
                jobTitle: userProfile.jobTitle,
                phoneNumber: userProfile.phoneNumber
              ]
            }
        }.then { map -> ❸
          render json(map)
        }
      }
    }
  }
}
```

❶ Here, we make the call to get the user, and using `flatMap`, we can make the subsequent call to the `UserProfileService`.

❷ We make the call to get the user's profile, and then use the `map` method on `Promise` to transform the `User` and `UserProfile` objects into a single model.

❸ Finally, we use the `then` method to subscribe to the call, which invokes the full processing flow and returns us the resulting, composed model for our client.

When working with two `Promise` type calls, where the second call is not dependent upon data from the first, we can simply use the composition process by building a `Pair` type object. This type holds the resulting items from two `Promise` results in a single object for use within the `then` subscription. Using this capability, we can streamline the processing flow considerably, as shown in Example 10-9.

Example 10-9. Composing data with Pair

```
get(":username") { UserService userService, UserProfileService userProfileService ->
  def username = pathTokens.username
  userService.getUser(username)
            .right(userProfileService.getUserProfile(username))
            .map { pair ->
              def user = pair.left
              def userProfile = pair.right
              [
                username: user.username,
                email: user.email,
                jobTitle: userProfile.jobTitle,
                phoneNumber: userProfile.phoneNumber
              ]
            }
            .then { map ->
              render json(map)
            }
}
```

Use of `Promise#right` or `Promise#left` will place the result of the provided `Promise` onto the `right` or `left` property of the `Pair` object accordingly. As you can see in the example, the processing flow is well streamlined and the resulting `User` and `UserProfile` objects are able to be transformed with the `Promise#map` method, before the model is delivered to the `then` subscription for rendering back to the client.

Reactive Streams

The growing popularity in reactive programming libraries for the JVM has recently prompted the Java reactive programming community to come together and agree upon a standard by which reactive programming paradigms are implemented. The drive for standardization in this space has resulted in the creation of the *Reactive Streams Manifesto*, which clearly outlines the problems that reactive programming on the JVM intends to solve, the data structures and interfaces to be used in solving those problems, and an in-depth specification for which implementors of a reactive programming libraries can conform to provide maximum interoperability for appli-

cations that intend to use reactive programming. Perhaps more importantly than all of that, the manifesto defines a nomenclature and language that can be used when talking about reactive programming on the JVM.

Using the standardized interfaces provided by reactive streams also means that frameworks such as Ratpack can utilize their own implementations for the various problem domains. Applications built on those frameworks then simply program to the interface specification and need not be concerned with the intricacies of the underlying implementation. From a practical perspective, this approach to reactive programming is akin to Java application development that utilizes Java Enterprise Edition (EE) APIs. That is to say, similar to how Java EE applications inherit the implementation according to their enterprise application container, so too do applications that employ reactive streams programming interfaces inherit the opinions of the compliant library they choose.

Ratpack provides an implementation of the reactive streams specification as part of its core library. It comes completely free with every Ratpack application, and provides more robust reactive data structures and interfaces for building and working with reactive pipelines in your application. Furthermore, comprehensive fixtures are available when working with reactive streams data structures to easily map a `Publisher` type to and from a `Promise` type.

The major difference between a reactive streams `Publisher` type and a `Promise` type is that a `Publisher` will produce a *stream* of data, whereby many elements of the same type may be *published* to a downstream `Subscriber`. As discussed earlier in the chapter, a `Promise` is guaranteed to only ever emit a single item to its subscriber, but when we are working with the reactive streams types in Ratpack, we can build a truly reactive processing pipeline that definitively fits the definition of reactive programming.

Working with reactive streams in Ratpack is designed to be as simple as possible. All of the fixtures for creating a `Publisher` type are provided through convenient and concise methods off of the `ratpack.stream.Streams` call. To better understand working with reactive streams in Ratpack, it will help to look at an example: So, let's consider a shortened version of the `UserService` example from the prior section, and build a reactive pipeline in our handler that filters all `User` objects that start with `nore ply` and maps the `User` data to a new object structure for rendering back to our caller. The updated `UserService` interface is depicted in Example 10-10.

Example 10-10. UserService reactive streams implementation

```
package app

import org.reactivestreams.Publisher

interface UserService {
```

```
  Publisher<User> getUsers() ❶
}
```

❶ Notice that now our `getUsers` method returns an `org.reactivestreams.Pub
  lisher` type, which is the standardized interface for reactive publisher types.
  Even though the `getUsers` method will return multiple `User` objects, we do not
  need to specify that there is a collection type here, as we did with `Promise`,
  because the subscribers to the publisher will be working with each `User` element
  on an individual level.

Given this interface, let's take a quick look at what a database-backed implementation
might look like. Consider the `DatabaseUserService` shown in Example 10-11, which
utilizes Groovy SQL and Ratpack's `Blocking` mechanism to query the database and
create the corresponding `Publisher` type.

Example 10-11. DatabaseUserService reactive streams implementation

```
package app

import javax.inject.Inject
import groovy.sql.Sql
import org.reactivestreams.Publisher
import ratpack.exec.Blocking

class DatabaseUserService implements UserService {

  private final Sql sql

  @Inject
  DatabaseUserService(Sql sql) {
    this.sql = sql
  }

  Publisher<User> getUsers() {
    Blocking.get { ❶
      sql.rows "SELECT * FROM USER"
    }
    .publish() ❷
  }
}
```

❶ As noted, we can still use the `Blocking` mechanism (or any other `Promise`-based
  API) to ensure our processing is scheduled to the blocking thread pool.

❷ When we are using Groovy, we can simply call `Promise#publish()` to transform
  a `Promise` into a `Publisher`.

In Groovy, we can leverage the fact that the `ratpack-core` module exports Groovy extensions that allow us to work with the static methods on the `Streams` class as though they are instance-level methods on `Promise`. This allows us to transform a `Promise` to a `Publisher` simply by calling `Promise#publish`. In a Java environment, we would need to explicitly wrap the `Promise` type in a call to `Streams.pub lish(Promise)`. Either way, the interoperability between `Promise` and `Publisher` types could not be easier.

With our `DatabaseUserService` in place, let's take a look at what the Ratpack application and handler chain look like for our RESTful API. The application code provided in Example 10-12 shows how we can go about implementing our filtering and mapping logic against the stream of `User` objects coming from the `getUsers Publisher`.

Example 10-12. Ratpack Groovy application with publisher filtering and mapping

```
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

import ratpack.groovy.sql.SqlModule
import ratpack.hikari.HikariModule
import app.*

ratpack {
  bindings {
    module(SqlModule)
    module(HikariModule) { c ->
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'
      c.username = 'sa'
      c.password = ''
    }

    bind(UserService, DatabaseUserService)
  }
  handlers {
    get("user") { UserService userService ->
      userService.getUsers()
        .filter { user ->               ❶
          !user.email.startsWith("noreply")
        }
        .map { user ->                  ❷
          [
              username: user.username,
              email: user.email
          ]
        }
        .bindExec()                     ❸
        .toList()                       ❹
        .then { maps ->                 ❺
```

```
            render json(maps)
        }
    }
  }
}
```

❶ We start the reactive pipeline by filtering out any `User` objects with an email that starts with `"noreply"`.

❷ Given the `User` object satisfies the filtering condition, we transform it to the `Map` type we expect.

❸ Here, we call `bindExec`, which is a helper method provided by Ratpack to ensure the `Subscription` types are properly bound to the execution. Note that in this case, this call is not completely necessary, since the subscription does no asynchronous processing, but it is generally a good idea to get in habit of adding this call.

❹ We can use the `toList()` method on `Publisher` to collect the results into a `List` type before passing them to the downstream subscriber.

❺ Finally, we subscribe to the pipeline, which initiates the data processing, and we get back a list of filtered, transformed models to send back to the client.

Publishers and bindExec

A simple note on the implementation here is that reactive streams `Subscriber` types do not automatically fit into Ratpack's execution model. We need to ensure that when we are using reactive streams that all the processing is bound to the execution model. When using Groovy, we can again leverage the Groovy extensions provided to use the `Streams#bindExec` static method as though it were an instance-level method on `Publisher`. In Java, we would need to wrap the `Publisher` in a call to `Streams.bindExec(Publisher)`.

If we do not call `bindExec` against a `Publisher` type, then the `Subscriber` may overrun the current execution, and the deterministic processing model guaranteed by Ratpack will not be satisfied. When using `Publisher` types in your application, it is critically important that you ensure that `bindExec` is called as well.

RxJava

RxJava is arguably the most popular of the reactive programming libraries available on the JVM. Its implementation of reactive programming is adapted directly from the *Reactive Extensions* (Rx) project for .NET languages. Similar to reactive streams,

RxJava reactive pipelines are streams of data where `Observable` types emit a single item at a time to `Subscriber` types. The library is battle proven with a vast ecosystem, and provides comprehensive implementations for reactive programming techniques and strategies. It is recommended that you utilize Ratpack's integration with RxJava when building applications of sufficient reactive programming complexity.

Like Ratpack, RxJava has an execution model, and external libraries are given the opportunity to integrate their own execution strategy—known as a *scheduler*—through the use of a plugin system. This system is how Ratpack integrates its execution model with RxJava `Observable` types. When the `RxRatpack` integration is initialized, `Observable` types can be transparently mapped to `Promise` types and vice versa. Ratpack's integration with RxJava gives application developers the best of both worlds: a powerful deterministic execution model in Ratpack, and a rich reactive pipelining and processing system in RxJava.

To begin incorporating RxJava into your application, you need to include the `ratpack-rx` framework dependency. This can be included in your Gradle build script using the `compile ratpack.dependency("rx")` call in the `dependencies` block, in the same manner used for Ratpack's other framework dependencies. Ratpack's RxJava dependency is dissimilar from Ratpack's other framework features in that it does not require the incorporation of a Guice module into your project to use it. Instead, we simply need to inform RxJava that it should use Ratpack's own `RxJavaObservableExecutionHook` implementation hook for scheduling the execution of `Observable` types.

The initialization of Ratpack's RxJava features is statically invoked, and need only be called once per running instance of a JVM. This means that if you are building your Ratpack application using the Groovy script approach, you can simply call the `RxRatpack.initialize()` method at any point prior to the use of an `Observable` in your application. This is most often satisfied by wiring a `ratpack.server.Service` instance responsible only for initializing the `RxRatpack` system. The application code provided in Example 10-13 shows this common approach. With the `RxRatpack` system initialized, `Observable` types can be safely used throughout your application.

Example 10-13. Initializing the RxRatpack system

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.rx.RxRatpack
import ratpack.server.StartEvent

ratpack {
  bindings {
    bindInstance new Service() {
      @Override
      void onStart(StartEven) {
        RxRatpack.initialize() ❶
```

```
      }
    }
  }

  handlers {
    // ... application handlers here ...
  }
}
```

❶ We call `RxRatpack.initialize()` to ensure `Observable` types are scheduled into the Ratpack execution.

As noted, `Observable` types can be seamlessly mapped to `Promise` types, and vice versa, but it should be noted that when doing so, the number of elements emitted from the `Observable` stream should be roughly known, so the proper type can be mapped. As `Promise` types only ever emit a single value, while `Observable` types emit a stream of elements, when mapping an `Observable` that emits multiple elements to a `Promise`, the `Promise` will need to collect those elements and emit them as a combined list to subscribers. If you know that an `Observable` will only emit a single element, you can choose to map using the *promise single* strategy, which will only capture a single element from the `Observable`. If you know the `Observable` will emit more than one element, you can choose the default mapping, which translates to a `Promise<List<T>>` type.

The RxRatpack class provides static methods to assist with mapping `Observable` types to `Promise` types. When under regular Java conditions, you can simply call the `RxRatpack.promise(Observable)` method to get a `Promise`. Likewise, you can call `RxRatpack.promiseSingle(Observable)` to map a single element `Observable`. To demonstrate this, let's start by looking at the `RxJavaUserService` interface, which specifies `getUser` and `getUsers` methods, both of which return `Observable` types. Note that because RxJava returns a stream of data, the `getUsers` method does not specify that a `List` will be returned. As with a reactive streams `Publisher` type, whether a single element or multiple elements are emitted, the signature does not change.

Example 10-14. The UserService with RxJava

```
package app;

import rx.Observable;

public interface RxJavaUserService {
  public Observable<User> getUser(String username);
  public Observable<User> getUsers();
}
```

Next, let's look at what a Java Ratpack application might look like that utilizes the RxJavaUserService. The code for this is provided in Example 10-15.

Example 10-15. Initializing RxJava in a main class

```java
package app;

import ratpack.server.RatpackServer;
import ratpack.rx.RxRatpack;
import rx.Observable;

import static ratpack.jackson.Jackson.json;

public class Main {

  public static void main(String[] args) throws Exception {
    RxRatpack.initialize(); ❶

    RatpackServer.start(spec -> spec
      .registryOf(r -> r
        .add(RxJavaUserService.class, new DefaultRxJavaUserService()) ❷
      )
      .handlers(chain -> chain
        .prefix("user", pchain -> pchain
          .get(":username", ctx -> {
            RxJavaUserService userService = ctx.get(RxJavaUserService.class);
            String username = ctx.getPathTokens().get("username");
            Observable<User> userObs = userService.getUser(username); ❸
            RxRatpack.promiseSingle(userObs).then(user -> ❹
              ctx.render(json(user))
            );
          })
          .get(ctx -> {
            RxJavaUserService userService = ctx.get(RxJavaUserService.class);
            Observable<User> usersObs = userService.getUsers(); ❺
            RxRatpack.promise(usersObs).then(users -> ❻
              ctx.render(json(users))
            );
          })
        )
      )
    );
  }
}
```

❶ Here, we initialize the RxRatpack system.

❷ We bind some default implementation of the RxJavaUserService in our application.

❸ Within our handler logic, we call the `RxJavaUserService#getUser` method, which returns an `Observable` for us.

❹ Using the `RxRatpack#promiseSingle` method, we map the `Observable` to a `Promise` type and subscribe to it, just as we normally would.

❺ Here, we make the call to get all users, which we know will result in more than one element being emitted to the stream.

❻ Thus, we use the `RxRatpack#promise` method, which will transform the `Observable<User>` stream to a `Promise<List<User>>`, wherein all `User` objects will be collected and emitted as a single value from the `Promise`.

If you are building a Ratpack Groovy application, the `ratpack-rx` dependency gives you some extra support. As part of this framework module, Ratpack also ships Groovy extensions for the `RxRatpack` class, which allows the static methods therein to be treated as instance-level methods. Consider the same application, this time depicted under Groovy conditions, as shown in Example 10-16.

Example 10-16. RxJava Groovy application

```groovy
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

import ratpack.service.Service
import ratpack.service.StartEvent
import ratpack.rx.RxRatpack
import app.RxJavaUserService
import app.DefaultRxJavaUserService
import rx.Observable

ratpack {
  bindings {
    bind(RxJavaUserService, DefaultRxJavaUserService)

    bindInstance new Service() {
      @Override
      void onStart(StartEvent e) throws Exception {
        RxRatpack.initialize()
      }
    }
  }
  handlers {
    prefix("user") {
      get(":username") { RxJavaUserService userService ->
        userService.getUser(pathTokens.username)
          .promiseSingle() ❶
          .then { user ->
```

```
        render json(user)
      }
    }
    get { RxJavaUserService userService ->
      userService.getUsers()
        .promise() ❷
        .then { users ->
          render json(users)
        }
    }
  }
 }
}
```

❶　Here, we can call `promiseSingle()` directly off of the `Observable` type because of the Groovy extensions Ratpack provides.

❷　Similarly, we can call `promise()` to get a `List<User>` from the mapped `Promise` type.

As you can see, with the Groovy extensions for RxJava, there is no need to explicitly call `RxRatpack`. This greatly simplifies the semantics of working with `Observable` types when building Groovy Ratpack and RxJava web applications.

Similarly, Ratpack `Promise` types can be seamlessly mapped to `Observable` types. When mapping from `Promise` to `Observable`, it is important that you know what the `Promise` will return, so that you can determine the appropriate method to use. If it returns a collection of objects, you should use `RxRatpack#observeEach`; if it returns a single object, you should use `RxRatpack#observe`. Consider again the `UserService` interface, which is designed to return `Promise` types for the `getUser` and `getUsers` methods. The Java main class provided in Example 10-17 demonstrates mapping the resulting `Promise` types to `Observable` within our handler methods.

Example 10-17. Mapping Promise to Observable

```
package app;

import ratpack.exec.Promise;
import ratpack.rx.RxRatpack;
import ratpack.server.RatpackServer;
import java.util.List;

import static ratpack.jackson.Jackson.json;

public class Main {

  public static void main(String[] args) throws Exception{
    RxRatpack.initialize();
```

```
        RatpackServer.start(spec -> spec
          .registryOf(r -> r
            .add(UserService.class, new DefaultUserService()))
          )
          .handlers(chain -> chain
            .prefix("user", pchain -> pchain
              .get(":username", ctx -> {
                UserService userService = ctx.get(UserService.class);
                String username = ctx.getPathTokens().get("username");
                Promise<User> userPromise = userService.getUser(username);
                RxRatpack.observe(userPromise).subscribe(user ->   ❶
                  ctx.render(json(user))
                );
              })
              .get(ctx -> {
                UserService userService = ctx.get(UserService.class);
                Promise<List<User>> usersPromise = userService.getUsers();
                RxRatpack.observeEach(usersPromise).toList().subscribe(users ->  ❷
                  ctx.render(json(users))
                );
              })
            )
          )
        );
      }
    }
```

❶   We use the RxRatpack#observe method to map a single object Promise type to
    an Observable, then we *subscribe* to the Observable to get the User object.

❷   We can map the Promise<List<User>> type to Observable<User> using the
    RxRatpack#observeEach.

In the latter case, if we wanted to perform more in-depth reactive processing that
incorporated more of RxJava's reactive programming functions, we could easily inline
them following the call to observeEach, because the User objects will be streamed to
the pipeline. Note that in either scenario, no processing of the upstream Promise
takes place until the Observable#subscribe method is called.

Like before, this code can be greatly ameliorated by using Groovy, which will apply
the RxRatpack Groovy extensions that allow us to use instance-level methods to map
Promise types to Observable types. Consider the same application, this time rewrit-
ten in Groovy, as shown in Example 10-18.

Example 10-18. Mapping Promise to Observable (Groovy style)

```
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json
```

```
import app.UserService
import app.DefaultUserService
import ratpack.rx.RxRatpack
import ratpack.server.Service
import ratpack.server.StartEvent

ratpack {
  bindings {
    bind(UserService, DefaultUserService)

    bindInstance new Service() {
      @Override
      void onStart(StartEvent e) {
        RxRatpack.initialize()
      }
    }
  }
  handlers {
    prefix("user") {
      get(":username") { UserService userService ->
        userService.getUser(pathTokens.username)
          .observe() ❶
          .subscribe { user ->
            render json(user)
          }
      }
      get { UserService userService ->
        userService.getUsers()
          .observeEach() ❷
          .toList()
          .subscribe { users ->
            render json(users)
          }
      }
    }
  }
}
```

❶ Here, we can treat the `RxRatpack#observe` method as an instance-level method
    on `Promise`.

❷ Similarly, the `RxRatpack#observeEach` method becomes instance-level when
    using Groovy.

The capabilities offered by Ratpack's RxJava integration provide a solid foundation
upon which to build powerful and robust reactive systems. When building Ratpack
RxJava applications with Groovy, you also get the added benefit of having the meth-
ods for translating `Observable` to `Promise` and vice versa as instance-level methods
that are concise and easy to use.

Parallel Processing Using RxJava

The scheduler that Ratpack provides to RxJava can be used for more than simply scheduling `Observable` executions onto Ratpack's execution model. Using RxJava's constructs for imposing parallelism onto a reactive pipeline, we can utilize Ratpack's execution model to perform reactive parallel processing across the computation threads available to your application. This may read as a somewhat complex process, but the usage is simple, and the benefit provided by parallel processing of data may help improve your application's performance.

When working with an `Observable` that we know will emit multiple items that we want to be processed in parallel, we can utilize the `RxRatpack#forkEach` mechanism, which creates a new execution for each data element in the stream. These executions still conform to Ratpack's guaranteed and deterministic execution model, so you can safely operate under those constraints when building your parallel reactive pipeline.

Think back to the example provided in the previous chapter: we wanted to provide a RESTful API for getting `User` and `UserProfile` data from the respective `UserService` and `UserProfileService` interfaces. This time, however, instead of providing an endpoint for retrieving this data for just a single user, we want to provide it for all users known to our `UserService`. This may turn out to be many users, so processing this data sequentially will not fit our needs. Instead, we can parallelize the composition of data as the `User` objects are streamed from the `UserService`.

Let's begin by revisiting the `UserService` and `UserProfile` interfaces to now return `Observable` types instead of `Promise` types, as shown in Example 10-19.

Example 10-19. UserService interface with Observable signatures

```
package app

import rx.Observable

interface UserService {

  Observable<User> getUsers()
}
```

For demonstration purposes, let's assume this time the `UserProfileService` requires that we provide it the user's `id`. The interface contract is depicted in Example 10-20.

Example 10-20. UserProfileService interface with Observable signatures

```
package app

import rx.Observable
```

```
interface UserProfileService {

  Observable<UserProfile> getUserProfile(Long id)
}
```

With these interfaces in place, and a presumable default implementation for each, we can build an application around the User stream where we parallelize the calls to the UserProfileService as part of a reactive processing pipeline. The Groovy application in Example 10-21 shows what the implementation for this technique looks like.

Example 10-21. Parallel processing with the UserService

```
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json

import ratpack.rx.RxRatpack
import ratpack.server.Service
import ratpack.server.StartEvent
import ratpack.func.Pair
import app.UserService
import app.UserProfileService
import app.DefaultUserService
import app.DefaultUserProfileService

ratpack {
  bindings {
    bind(UserService, DefaultUserService)
    bind(UserProfileService, DefaultUserProfileService)

    bindInstance new Service() {
      @Override
      void onStart(StartEvent e) {
        RxRatpack.initialize()
      }
    }
  }
  handlers {
    get("user") { UserService userService, UserProfileService userProfileService ->
      userService.getUsers()
        .compose(RxRatpack.&forkEach) ❶
        .flatMap { user ->
          userProfileService.getUserProfile(user.id).map { userProfile ->
            new Pair(user, userProfile) ❷
          }
        }
        .map { pair -> ❸
            def user = pair.left
            def userProfile = pair.right
            [
              username: user.username,
```

```
                email: user.email,
                jobTitle: userProfile.jobTitle,
                phoneNumber: userProfile.phoneNumber
            ]
          }
        .promise() ❹
        .then { maps ->
          render json(maps)
        }
      }
    }
}
```

❶ The `Observable#compose` allows us to provide a strategy for parallelizing the
    stream coming out of the `userService.getUsers()` call.

❷ We can continue to utilize Ratpack's functional programming interfaces to help
    work with our reactive pipeline. Here, we leverage the `Pair` type, as before, to
    combine the `User` and `UserProfile` objects into a single object for processing
    downstream.

❸ Here, we use `map` as a transformation function to compose the `User` and `UserPro
    file` objects into a single combined object.

❹ We can simplify the process of collecting the `Observable` stream into a `List` by
    translating it to a `Promise` type (which, remember, inherently maps the resulting
    stream to a `List`).

Using this strategy for composing asynchronous data in a parallel stream allows you
to leverage the full capabilities of your system when composing data that derives from
disparate sources. Ratpack's integration with RxJava allows you to continue to lever-
age the deterministic execution model while parallelizing your reactive pipeline.

Further Reading on RxJava

RxJava is a comprehensive library for working with reactive programming para-
digms. As such, much of the conversation on RxJava is outside the scope of this text.
It is left as an exercise to the reader to explore the extensive documentation to better
understand how reactive programming works with RxJava. Documentation on all
reactive operators in the RxJava arsenal can be found on the project's GitHub page.

Chapter Summary

This chapter has exposed you to the many ways that Ratpack utilizes and provides
reactive programming capabilities. At its core, Ratpack is first and foremost a reactive

programming web framework, and the understanding that you take away from this chapter places you in a position to write powerful reactive web applications. From the simplest usage of `Promise` type streams, to the reactive streams support, all the way through to how Ratpack integrates seamlessly with RxJava, there is not a reactive programming solution that is not available to you with Ratpack. As your experience with Ratpack continues to evolve, you will find that its Ratpack programming fixtures set you in a good position to succeed in working with data streams and functional programming strategies.

Sessions and Security

It is generally advisable that web applications remain as stateless as possible in order to facilitate scalability. However, there are times when it is important for data to be persistent throughout the scope of an HTTP session. Particularly when you are building applications that have user-based authentication and authorization requirements, the need for session-scoped data becomes apparent. Sessions, session data storage, and cookies are all mature aspects in Ratpack's infrastructure. The use of sessions is provided to you as an optional framework dependency, while cookies are able to be used without the need for additional libraries.

This chapter will give you demonstrations and context for building applications that rely on HTTP sessions and cookies. The knowledge you carry forward from here will prove valuable as you progress through the chapters that follow, particularly the conversation on how Ratpack applications implement security features. Your exposure to the concepts outlined in this chapter will serve as the necessary foundation for your comprehensive understanding of how Ratpack works.

Integrating Session Support

Adding support for HTTP sessions to your Ratpack application is no different than adding any other optional framework dependency to your project. The `ratpack-session` dependency provides the module necessary to make the session constructs available to your code base. The Gradle build script in Example 11-1 demonstrates adding the session module to your Ratpack project.

Example 11-1. Gradle build script with Ratpack session dependency

```
buildscript {
  repositories {
```

```
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency('session') ❶
}
```

❶   Here, we add the `ratpack-session` framework dependency to our project.

With the dependency in place, we can incorporate the `SessionModule` into our `rat pack.groovy` script, as shown in Example 11-2.

Example 11-2. Ratpack Groovy script with SessionModule

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule

ratpack {
  bindings {
    module(SessionModule) ❶
  }
  handlers {
    // ... application handlers ...
  }
}
```

❶   We use the `BindingsSpec#module` method to add the `ratpack.session.Session Module`, just as we would any other framework module.

Alternatively, given a Java-based Ratpack application, the skeleton for a main class application might look like the one shown in Example 11-3.

Example 11-3. Ratpack Java main class with SessionModule

```
package app;

import ratpack.session.SessionModule;
import ratpack.server.RatpackServer;
import ratpack.guice.Guice;
```

```
public class Main {

  public static void main(String[] args) {
    RatpackServer.start(spec -> spec
      .registry(Guice.registry(b -> b
        .module(SessionModule) ❶
      ))
      .handlers(chain -> chain
        // ... application handlers ...
      )
    );
  }
}
```

❶ As you already know how to build a Guice-backed registry, you can see the simi-
   larities to the previous Groovy example.

The `SessionModule` provides a `Session` object to the context registry, which can be
utilized from within any handler. Through the `Session` object, we can store and
retrieve data for the duration of the user's session. For example, the Ratpack Groovy
script in Example 11-4 shows an `all` handler at the top of the chain that retrieves a
request count and increments it with each request, and a `get` handler downstream
that renders the request count back to the client.

Example 11-4. Ratpack Groovy script using Session

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import ratpack.session.Session

ratpack {
  bindings {
    module(SessionModule) ❶
  }
  handlers {
    all { Session session -> ❷
      session.get("req-count").map { o -> ❸
        o.orElse(0)
      }.flatMap { count -> ❹
        session.set("req-count", count+1).promise() ❺
      }.then {
        next() ❻
      }
    }

    get { Session session ->
      session.get("req-count").then { o -> ❼
        render o.get().toString() ❽
      }
  }
```

```
      }
   }
}
```

❶ Bind the `SessionModule`, as shown earlier.

❷ Pull the `Session` object from the context registry.

❸ `Session#get` returns a `Promise<Optional<T>>`, which means that loads possibly empty data asynchronously, so here we will also use the `Promise#map` method to map the value from its `Optional` type to an `Integer` we can work with.

❹ We need to `flatMap` the call to set the new value on the session, as all session interactions are asynchronous, and thus use `Promise` types.

❺ Here, we increment the count and set it on the session. Note that `Session#set` is an `Operation` type, so we must map it to a `Promise` type using `promise()` to be compatible with `flatMap`.

❻ Here, we delegate processing downstream.

❼ Similarly, we extract the `req-count` key, which now will be set here.

❽ And finally, we render it back to a client.

That's it! Now you have a concise and simple example of storing and retrieving session-based data with Ratpack! If you run the application and open a browser to *http://localhost:5050*, you will see the number 1 on the initial page load. Refresh the page a few times and you will see the number incrementing with each request.

Persisting Objects

Trivial examples of storing and retrieving numbers in the session will not get you too far in a real-world scenario. Luckily, Ratpack allows you to store and retrieve any object that is `Serializable` into the session. Say, for example, that we wanted to implement some functionality that would track the views of particular pages in our application and keep track of a count. In a real-world scenario, this might help to provide better navigation from the user interface or otherwise discern what type of data a user is looking at to give a better user experience.

To start with, we can build a simple class, `ViewTracker`, which will act as our interface for registering views and incrementing the counts. This class will need to have some serializable storage mechanism (i.e., a `Map`) and a mechanism for safely incrementing

and retrieving the view counts. The code in Example 11-5 demonstrates a possible implementation.

Example 11-5. The ViewTracker session object

```groovy
package app

import com.google.common.collect.Maps
import groovy.transform.Immutable

class ViewTracker implements Serializable { ❶
  private Map<String, View> views = Maps.newConcurrentMap() ❷

  void increment(String uri) { ❸
    def count = 1
    if (views.containsKey(uri)) {
      count = views.get(uri).count+1 ❹
    }
    views[uri] = new View(uri, count) ❺
  }

  List<View> list() { ❻
    views.values() as List
  }

  @Immutable
  static class View implements Serializable {
    String uri
    int count
  }
}
```

❶ Here, we ensure that `ViewTracker` implements `Serializable` so that the Java Object Serialization system that undersits the `Session` persistence is capable of writing and reading the object.

❷ This will be our storage of `uri` to `View` types that we will work with.

❸ Here, we provide the mechanism for incrementing a view count.

❹ If the view has already been registered, safely retrieve the count and increment it.

❺ Store the new reference data for the URI.

❻ Here, we provide a way to simply list the values in the storage.

With this in place, we can modify the example from earlier in the chapter to demonstrate storing the `ViewTracker` in the session and retrieving it for use. The Ratpack Groovy script in Example 11-6 shows an enhancement on the prior demonstration.

Example 11-6. Ratpack Groovy with ViewTracker

```
import static ratpack.groovy.Groovy.ratpack
import static ratpack.jackson.Jackson.json
import ratpack.session.SessionModule
import ratpack.session.Session
import app.ViewTracker

ratpack {
  bindings {
    module(SessionModule)
  }
  handlers {
    all { Session session ->
      session.get("view-tracker").flatMap { o ->        ❶
        def tracker = o.orElse(new ViewTracker())       ❷
        tracker.increment(request.uri)                  ❸
        session.set("view-tracker", tracker).promise()  ❹
      }.then {
        next()
      }
    }

    all { Session session ->
      session.get("view-tracker").then { o ->
        def tracker = o.get()
        render json(tracker.list())                     ❺
      }
    }
  }
}
```

❶ In this upstream `all` handler, we get the object stored against the `view-tracker` key. We use `flatMap` here, as we will save the `ViewTracker` before proceeding down the chain.

❷ If the `ViewTracker` exists, then we get that value back; if not, we create a new one.

❸ Here, we utilize the `ViewTracker#increment` method to increment the count for the `request.uri` value.

❹ Before proceeding to the rest of the chain, we save the modified `ViewTracker` in the session.

❺ For demonstration's sake, we can render the `ViewTracker#list` results as JSON.

Running this code and opening a web browser to *http://localhost:5050* will show you that you have viewed the **/** URI once. If you change the location in your browser to append the */foo* route, you will see that a new entry is added to the list. Play around with this with a few different URIs and you can begin to see how this can become a powerful utility for managing a client's interactions with your service across multiple page loads.

Configuring the SessionModule

Like many of Ratpack's framework modules, the `SessionModule` is a `Configurable Module`, which allows you to override the module's default configuration values to suit your application's needs. Like most Java web applications, Ratpack's session support stores a cookie on the client that references the session ID. When configuring the `SessionModule`, you are also able to specify configuration for the session ID cookie, including the domain it is stored under, its max age, or the path under which the session ID cookie should be retrieved from the client. The example application in Example 11-7 shows the possible values that you can configure and how to go about doing so.

Example 11-7. Configuring SessionModule

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import java.time.Duration

ratpack {

  bindings {
    module(SessionModule) {
      expires(Duration.ofDays(7)) ❶
      domain("https://my-domain.com") ❷
      path("/") ❸
      idName("JSESSIONID") ❹
      httpOnly(true) ❺
      secure(false) ❻
    }
  }
  handlers {
    // ... application handlers here ...
  }
}
```

❶ Setting the `expires` value to a `java.time.Duration` value allows you to specify how long a user's session should remain valid.

❷ You can set the `domain` for which the session ID cookie should be valid. This can be useful when used in conjunction with additional application configuration for ensuring that multitenant deployments do not share sessions across domains.

❸ Using the `path` setting, you can configure under what request paths the session ID cookie should be retrieved. If only a portion of your application requires sessions, this can be a useful setting to configure.

❹ The default key for session cookie is `JSESSIONID`, which is similar to other Java web applications. You can override the key for the cookie using the `idName` configuration method if you wish to customize this value.

❺ By default, session cookies are allowed to be transmitted over unencrypted HTTP. If you wish to override this configuration, you can do so by setting `httpOnly` to `false`.

❻ If you wish to have session cookies only transmitted over encrypted HTTP (HTTPS), you can configure the `secure` setting to `true` (the default is `false`).

By default, sessions are stored in memory, which provides an excellent footprint for getting started with building session support into your application. However, it is important to understand that these sessions will ultimately be volatile and will not survive a restart of the application. Additionally, they will not scale across many instances of your application. This may be fine if you will only ever have a single instance of your application, but for scalability you will want to utilize Ratpack's client-side session storage or its support for building distributed sessions.

Client-Side Sessions

Ratpack's client-side session support allows your application to persist the session in a cookie that lives on the client. This cookie can optionally be encrypted to ensure the security of your users' session data. This feature of Ratpack can be used to build scalability into your application's deployment. When a client connects to your Ratpack application, its session cookie is transmitted and can be serialized and deserialized across many instances of your deployment.

Your application can employ client-side session support by using the `ClientSideSessionModule`. This is also a configurable module, giving you the ability to customize much of its operation. It is used in conjunction with the `SessionModule` shown in the prior section, so you can continue to customize configuration for the *session ID cookie* while making use of the client-side *session cookie*.

Support for client-side sessions is provided as part of the `ratpack-session` module, and none of the code in your handlers for working with the `Session` object needs to

be changed to implement this feature. The code provided in Example 11-8 details incorporating the `ClientSideSessionModule` into your Ratpack application.

Example 11-8. Configuring ClientSideSessionModule

```groovy
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import ratpack.session.clientside.ClientSideSessionModule
import java.time.Duration

ratpack {

  bindings {
    module(SessionModule) { c ->
      // SessionModule customizations go here
    }
    module(ClientSideSessionModule) { c ->
      c.sessionCookieName = "ratpack_session" ❶
      c.secretToken = Math.floor(System.currentTimeMillis() / 10000) ❷
      c.secretKey = '!c$mB&aGkL112345' ❸
      c.macAlgorithm = "HmacSHA1" ❹
      c.cipherAlgorithm = "AES/CBC/PKCS5Padding" ❺
      c.maxSessionCookieSize = 1932 ❻
      c.maxInactivityInterval = Duration.ofHours(24) ❼
    }
  }
}
```

❶ The `sessionCookieName` property allows you to customize the key under which the session cookie is stored with the client. This value defaults to `ratpack_ses sion`.

❷ The `secretToken` property is the value used to sign the serialized session. This value defaults to a time-based value unless otherwise specified. Signing the serialized session with this value prevents tampering.

❸ You can specify a value to `secretKey` to encrypt the client-side session cookie. If no value is specified here, then by default the session cookie will not be encrypted.

❹ If the session cookie is to be encrypted, you can override the MAC.[1] This value defaults to `HmacSHA1`, which uses the SHA-1 cryptographic hashing function to

---

1 The "message authentication code" algorithm is used in conjunction with the `secretKey` to produce a key that validates that it was your application that produced the session data.

generate the MAC. The value specified here must be one of the values specified by the `javax.crypto.Mac` class.

❺ You can also override the cipher algorithm that is employed to perform the encryption of the session cookie. This value defaults to `AES/CBC/PKCS5Padding`, which provides 128-bit encryption of the session cookie. Overrides of this value must be one of the values supported by the `javax.crypto.Cipher` class.

❻ The `maxSessionCookieSize` value can be used to specify the maximum size of the client-side session cookie. Any value within the range of 1024 and 4096 are valid for this property, and the default size is specified as 1932.

❼ The last value, `maxInactivityInterval`, is pretty straightforward. It specifies the duration under which a session should remain valid.

A caveat to using the client-side session cookie support is that the size of the session must be fairly small. As noted in the breakdown, the maximum size of the session cookie is 4KB, so if you need to store large objects in the session, then you will want to use a mechanism other than cookies for doing so. Luckily, Ratpack provides support for building distributed sessions, which facilitates scalability and does not depend on client-side storage for retrieving the session object.

Because it is generally inadvisable to hardcode values like the `secretKey` property on the `ClientSideSessionModule` configuration, you can utilize your application configuration, driven through the `serverConfig` to populate sensitive values in the module's configuration. For example, the code in Example 11-9 shows a setup that pulls configuration from a YAML file, then from environment variables, and finally from system properties. Then, within the `bindings` block, we can extract values from the provided application configuration for use in customizing the `ClientSideSession Module`.

Example 11-9. Application configuration for the ClientSideSessionModule

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import ratpack.session.clientside.ClientSideSessionModule
import ratpack.session.clientside.ClientSideSessionConfig

ratpack {
  serverConfig {
    yaml(Paths.get("/etc/config.yml"))
    env()
    sysProps()
  }
  bindings {
    module(SessionModule)
```

```
    moduleConfig(ClientSideSessionModule,
      serverConfig.get("/session", ClientSideSessionConfig)) ❶
  }
  handlers {
    // ... application handlers here ...
  }
}
```

❶ Here, we are using the `serverConfig.get(path, class)` mechanism to extract
   the `ClientSideSessionConfig`. We also use the `moduleConfig` method to define
   that we are explicitly providing the configuration to the module.

Given this application code, the */etc/config.yml* file may be defined like the file shown
in Example 11-10.

Example 11-10. ClientSideSessionModule config file

```
session:
  sessionCookieName: "my_app_session"
  secretKey: "!c$mB&aGkL112345"
```

With Ratpack's configuration model, the configuration values can easily be overrid-
den through environment variables or system properties to accommodate multiple
runtime environments.

Distributed Sessions

Distributed sessions are a valuable Ratpack feature that comes into play when you
need scalability and client-side session cookies are not going to work for you. For
this, Ratpack provides the ability to persist sessions to Redis as a backend storage
medium. Redis is a great choice for session storage, as it can be clustered and replica-
ted, and it can scale independently of your application. Furthermore, for applications
that run in cloud- or platform-based deployments, Redis is a widely available storage
solution, making the effort to get a distributed session infrastructure in place rela-
tively low. For those applications that run in an on-premises infrastructure, the leg-
work needed to get Redis up and running is still quite trivial.

Support for Redis-backed sessions comes from the optional `ratpack-session-redis`
module, which, like the `ClientSideSessionModule` shown in the last section, is used
in conjunction with the `SessionModule`. Indeed, the `SessionModule` is a hard
dependency, and the `RedisSessionModule` *must* be incorporated into your applica-
tion *after* the `SessionModule` has been specified.

The `RedisSessionModule` allows you to specify a host, port, and password for your
Redis connection. The `password` field is strictly optional, and by default Redis does
not configure with backend authentication. It is important that you *never* allow public

access to the Redis instance that stores your application's sessions. If you choose to employ the Ratpack Redis session support, it is also a good idea to build a security model between your application instances and your Redis instance. This can be accomplished through firewall rules that only allow access to the Redis instance from your application.

You can safely evolve your application's code from using in-memory sessions, to client-side sessions, and finally out to Redis-backed sessions without having to change the way your handlers get access to the session data. Nothing more is required of incorporating Redis-backed sessions than applying the `RedisSessionModule` within your `bindings` block, as shown in Example 11-11.

Example 11-11. Configuring RedisSessionModule

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import ratpack.session.store.RedisSessionModule

ratpack {

  bindings {
    module(SessionModule) ❶
    module(RedisSessionModule) { c ->
      c.host = "127.0.0.1" ❷
      c.port = 6379 ❸
      c.password = "..." ❹
    }
  }
  handlers {
    // ... application handlers here ...
  }
}
```

❶   Again, note that the `SessionModule` *must* come before the `RedisSessionModule`.

❷   The `host` property specifies the hostname or IP address to your Redis instance. This defaults to `localhost`.

❸   The `port` to which Reids is bound on the remote system. This defaults to `6379`, which is Redis' default binding port.

❹   The optional `password` field for the Redis instance. This defaults to null.

Depending on your application's requirements, you will likely have the need to use different Redis instances according to the runtime environment (development versus production, for example) or the tenancy of the deployment (customer1 versus customer2). Like what was shown in the prior section, we can drive the `RedisSession`

`Module` configuration through our application's configuration, by pulling the values from configuration. The code in Example 11-12 demonstrates this capability.

Example 11-12. Configuring RedisSessionModule with application configuration

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.session.SessionModule
import ratpack.session.store.RedisSessionModule

ratpack {
  serverConfig {
    yaml("config.yml")
    env()
    sysProps()
  }
  bindings {
    module(SessionModule)
    moduleConfig(RedisSessionModule,
      serverConfig.get("/session", RedisSessionModule.Config))
  }
  handlers {
    // ... application handlers here ...
  }
}
```

Working with Cookies

Cookies are applicable to a broader conversation than just HTTP sessions. There are many reasons why an application would wish to set and retrieve cookies from users. For example, your application may wish to pass a cookie on to a client that stores some view layer configuration or preferences. Similarly, you may find the need to set a cookie that tracks the last several pages that a user has visited in order to provide an enhanced user experience. Whatever your application's requirement, working with cookies in Ratpack involves little more than interfacing with the `request` and `response` objects.

To begin the demonstration of working with cookies, consider the scenario where you wish to dynamically render view-related decisions based on a configuration provided by a cookie. To illustrate this, consider that your application's main landing screen (Figure 11-1) provides users with the ability to reorder the position of objects in its grid. By default, the view may want to provide a standard left alignment of view objects, with an explicit request realing them to center or right columns.

Figure 11-1. Default main landing view

The desired behavior in this view is that when a user chooses to realign the view object, we want the new position to be remembered so that subsequent visits to the application align the view as they desire. For example, if the user clicks the "Move →" button, we want to both move the object to the middle column (Figure 11-2) and ensure that when the user comes back in the future the object appears in the middle. This is also a good strategy for managing simple preferences that may extend beyond the time window of a session.
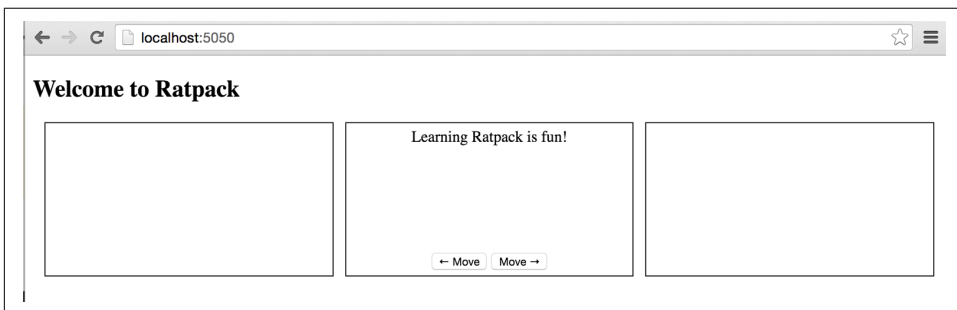
Figure 11-2. Center-aligned main landing view

The Groovy text template for this view is shown in Example 11-13.

Example 11-13. Groovy text template (index.html)

```
<!DOCTYPE html>
<html>
<head>
<title>Learning Ratpack</title>
<style> ❶
.row {
  width: 100%;
}
.row > .col {
  display: inline-block;
```

```
    float: left;
    text-align: center;
    padding-top: 5px;
    position: relative;
}
.col.col-3 {
    width: calc(33.333% - 16px);
    margin-left: 12px;
    border: 1px solid #000;
    min-height: 150px;
}
.controls {
    position: absolute;
    bottom: 5px;
    width: 100%;
}
.controls > form {
    display: inline-block;
}
</style>
</head>
<body>
<h2>Welcome to Ratpack</h2>
<div class="row">
  <% 3.times { n -> %> ❷
    <div class="col col-3">
    <% if (model.position == n) { %> ❸
      Learning Ratpack is fun!
      <div class="controls">
        <% if (n > 0) { %>
        <form action="/updatePosition" method="post"> ❹
          <input type="hidden" name="next_pos" value="${model.position-1}">
          <input type="submit" value="&larr; Move">
        </form>
        <% } %>
        <% if (n < 2) { %>
        <form action="/updatePosition" method="post">
          <input type="hidden" name="next_pos" value="${model.position+1}">
          <input type="submit" value="Move &rarr;">
        </form>
        <% } %>
      </div>
    <% } %>
    </div>
  <% } %>
</div>
</body>
</html>
```

❶ The template adds some basic styles that make the grid and bottom control but-
tons render properly.

❷ Here, we use the Groovy `Number#times` call to loop three times to fill in the grid. The index number of the iteration is provided to the closure, and represented here as the `n` variable.

❸ Within the loop, we check that the provided model's `position` value is equal to the current loop index, and if it is, we fill in the view content.

❹ The control buttons are wrapped in a `form` that submits the next desired position (`next_pos`) to our application.

Given this view, we know that we need handlers for both serving the landing page and capturing the next desired position from the form POSTs. To accommodate this, we can realize an application structure like the one shown in Example 11-14.

Example 11-14. Working with cookies application

```
import ratpack.groovy.template.TextTemplateModule
import ratpack.form.Form

import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module TextTemplateModule ❶
  }
  handlers {
    post("updatePosition") {
      parse(Form).then { data -> ❷
        response.cookie("ratpack-view-position", data.next_pos) ❸
        redirect "/"
      }
    }
    get {
      def position = request.oneCookie("ratpack-view-position")?.toInteger() ?: 0 ❹
      render groovyTemplate([position: position], "index.html") //
    }
  }
}
```

❶ Because we are rendering Groovy text templates as dynamic content, we need the `TextTemplateModule` applied to our application.

❷ Within the `updatePosition` POST handler, we parse the posted form here.

❸ Next, we use the `cookie` method on the `response` object to set the `ratpack-view-position` cookie to the value provided by the `next_pos` value from the form.

❹ Within the `get` handler, we start processing by pulling the `ratpack-view-position` cookie from the `request object` and translating it into an integer. Cookie values will always be string types, and you will recall that the view logic was built to work with a `Number` type. The `oneCookie` method extracts a specific cookie from the request. If there is no cookie of that name available, then the value will be null, so here we simply default to zero.

If you run this application and navigate a browser to *http://localhost:5050*, you will be met with a landing screen like the one shown in Figure 11-1. Choosing the "Move →" button, you will notice that the page POSTs to the `/updatePosition` endpoint and reloads. Subsequent reloads will keep the content in the position that you specified. You can use your browser's development tools to inspect the client-side resources and validate that the cookie was properly set. Figure 11-3 shows an example of the cookie value in Chrome's *Resources* view.
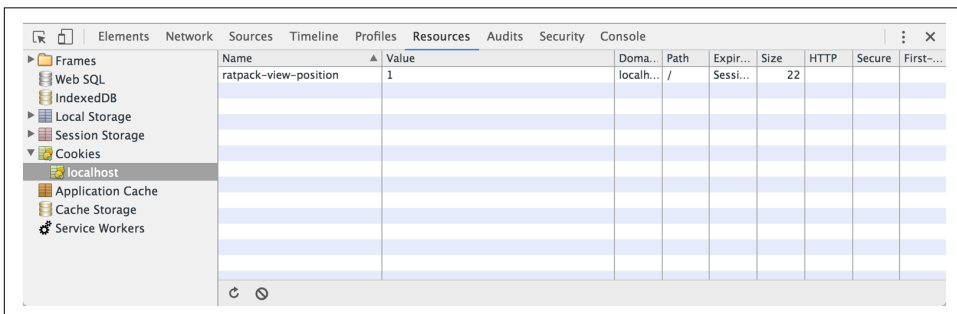
Figure 11-3. Cookie in Chrome Resources

Tuning Cookies

Cookies are set with clients using unopinionated defaults. For example, no expiry, domain, secure flag, or HTTP-only flags are set when a cookie is created. Configuration of these values can be done so through the `Cookie` object that is returned when the `cookie` method is called on the `response` object. If we change the `updatePosition` POST handler slightly, we can demonstrate setting these properties on the transmitted cookie. The updated application code in Example 11-15 shows the tuning of cookie values.

Example 11-15. Tuning cookie values

```
import ratpack.groovy.template.TextTemplateModule
import ratpack.form.Form

import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module TextTemplateModule
  }
  handlers {
    post("updatePosition") {
      parse(Form).then { data ->
        def cookie = response.cookie("ratpack-view-position", data.next_pos)
        cookie.maxAge = 365 * 24 * 60 * 60 ❶
        cookie.domain = "localhost" ❷
        cookie.httpOnly = true ❸
        cookie.secure = false ❹
        redirect "/"
      }
    }
    get {
      def position = request.oneCookie("ratpack-view-position")?.toInteger() ?: 0
      render groovyTemplate([position: position], "index.html")
    }
  }
}
```

❶ It may not be desirable for your user experience for your application to respect cookies after a period of time has elapsed since their creation. Setting the maxAge property to a number of *seconds* allows you to tune how long a browser should respect a cookie. Here, we set the value to one year.

❷ We can also set the domain on the cookie to ensure that cookies are appropriately associated with the domain name for your application.

❸ Here, we set the httpOnly flag on the cookie. This indicates whether JavaScript code should be able to access the cookie once it is stored by the browser. A value of true here indicates that the cookie is only valid for request-response lifecycles, and cannot be accessed by the view's JavaScript. The default value for httpOnly is false.

❹ The secure property indicates whether cookies should be transmitted over unencrypted HTTP. By default this is set to false, but if you wish to only transmit cookies when serving your application over HTTPS, then set this value to true. It

is generally a bad practice to store sensitive data in cookies, but if your requirements demand it, then ensure you are properly setting this value.

If you run this application again and change the content position within the view's grid, you will see that the cookie in your browser has been updated with the properties we specified. Figure 11-4 shows the Chrome console's Resources tab again, this time indicating that the cookie values have been properly set in our handler.
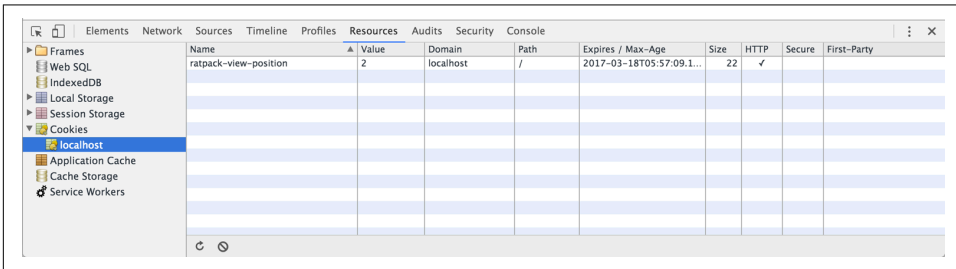
Figure 11-4. Updated cookie values

Expiring Cookies

When working with cookies, you will undoubtedly have the need to explicitly remove a cookie from a client. This is accomplished by expiring the cookie from within a handler. For the purposes of our demonstration, consider that we want to provide the means for the view configuration to be set back to its defaults. We can achieve this by providing a Reset View button that expires the `ratpack-view-position` cookie, and redirects back to the landing page.

To illustrate expiring cookies, let's consider a change to the *index.html* template, which adds a button for resetting the view. Example 11-16 shows this addition.

Example 11-16. Reset View button added

```
<!DOCTYPE html>
<html>
<head>
<title>Learning Ratpack</title>
<style>
.row {
  width: 100%;
}
.row > .col {
  display: inline-block;
  float: left;
  text-align: center;
  padding-top: 5px;
  position: relative;
}
```

```
.col.col-3 {
  width: calc(33.333% - 16px);
  margin-left: 12px;
  border: 1px solid #000;
  min-height: 150px;
}
.controls {
  position: absolute;
  bottom: 5px;
  width: 100%;
}
.controls > form {
  display: inline-block;
}
</style>
</head>
<body>
<h2>Welcome to Ratpack</h2>
<form action="/resetView" method="post" style="margin-bottom: 10px"> ❶
  <input type="submit" value="Reset View">
</form>
<div class="row">
  <% 3.times { n -> %>
    <div class="col col-3">
    <% if (model.position == n) { %>
      Learning Ratpack is fun!
      <div class="controls">
        <% if (n > 0) { %>
        <form action="/updatePosition" method="post">
          <input type="hidden" name="next_pos" value="${model.position-1}">
          <input type="submit" value="&larr; Move">
        </form>
        <% } %>
        <% if (n < 2) { %>
        <form action="/updatePosition" method="post">
          <input type="hidden" name="next_pos" value="${model.position+1}">
          <input type="submit" value="Move &rarr;">
        </form>
        <% } %>
      </div>
    <% } %>
    </div>
  <% } %>
</div>
</body>
</html>
```

❶ Underneath the landing page's welcome message, we create a form that simply POSTs to /resetView.

Next, we must update the application code to provide a `post` handler for the `reset View` endpoint. Example 11-17 shows this change.

Example 11-17. Adding the resetView handler

```
import ratpack.groovy.template.TextTemplateModule
import ratpack.form.Form

import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module TextTemplateModule
  }
  handlers {
    post("resetView") { ❶
      response.expireCookie("ratpack-view-position") ❷
      redirect "/" ❸
    }
    post("updatePosition") {
      parse(Form).then { data ->
        def cookie = response.cookie("ratpack-view-position", data.next_pos)
        cookie.maxAge = 365 * 24 * 60 * 60
        cookie.domain = "localhost"
        cookie.httpOnly = true
        cookie.secure = false
        redirect "/"
      }
    }
    get {
      def position = request.oneCookie("ratpack-view-position")?.toInteger() ?: 0
      render groovyTemplate([position: position], "index.html")
    }
  }
}
```

❶ Here, we add a POST handler for the `/resetView` endpoint to which the Reset View button will submit.

❷ It takes no more than calling the `expireCookie` method on `response` with the name of the cookie to have it be removed from the client.

❸ We redirect to the landing page, which will now render back to the default view.

If we run this application, we will see that the landing page now presents the Reset View button (Figure 11-5).
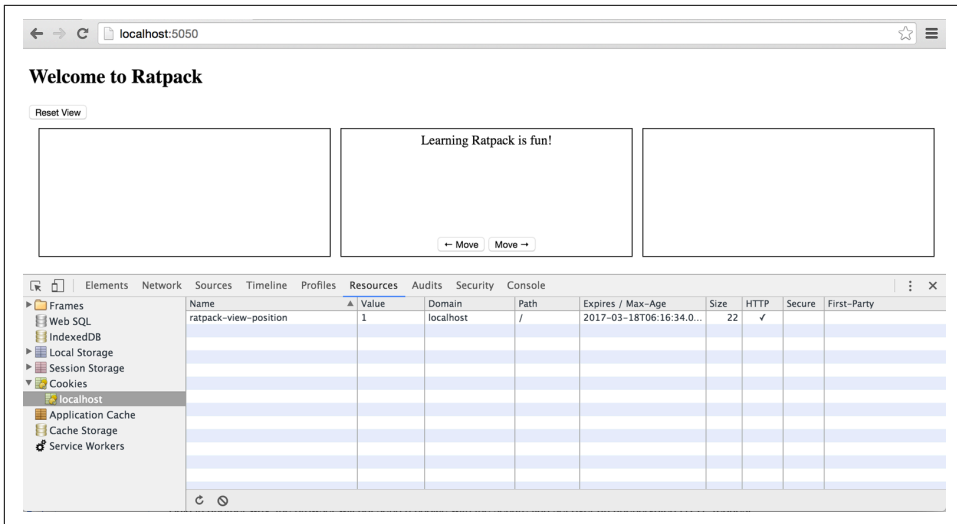
Figure 11-5. Reset View button

Clicking the Reset View button will perform the POST, and when the landing page is reloaded, we will see that the cookie has indeed been expelled from our browser and the landing page preferences have been reset (Figure 11-6).
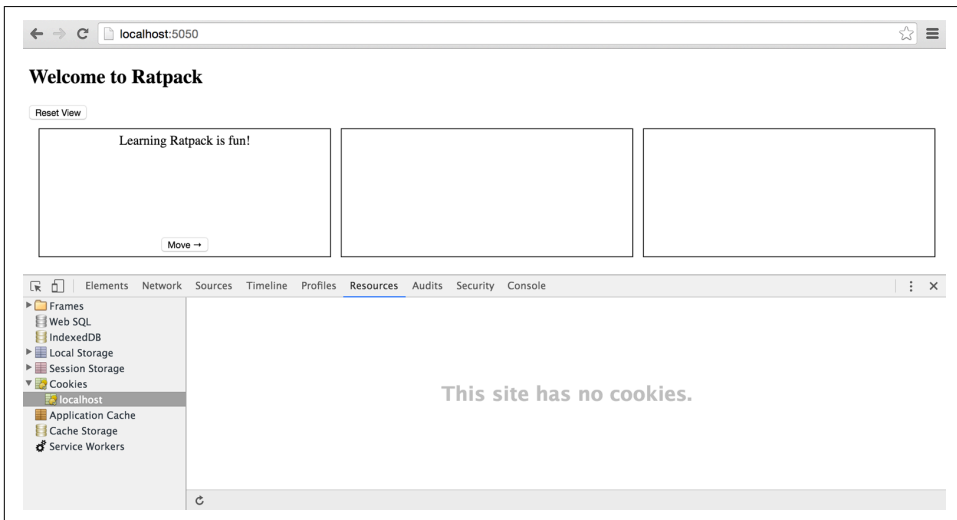
Figure 11-6. Cookie removed and view reset

From here, you can go about customizing the content position and you will find that the cookie again is set when you do. Being able to expire cookies through such a seamless and easy-to-use mechanism further exemplifies Ratpack's nature as a frame-

work capable of building robust applications that are able to focus on user experience instead of the nuances of the underlying framework.

Chapter Summary

The concepts and integrations covered in this chapter will service you as a fundamental understanding when building user-centric applications. As your journey in learning Ratpack continues, the discussions from this chapter will prove invaluable. In the next chapter, we will discuss introducing security into your application, and your understanding of Ratpack's HTTP session support will be necessary for this conversation. Furthermore, the knowledge and experience that you now have for working with cookies allows you to build powerful, modern applications that provide a top-quality user experience.

Application Security

Application security is a feature given immense attention in Ratpack. The conversation of security takes many forms; from access control to encryption of communications and user data, no modern production-grade application would be complete without the ability to ensure security for its users and data. We have already seen some of the levels of security Ratpack provides, with the ability to encrypt user session data, but there are many more core and optional security features available from the framework. As with the implementation of many other Ratpack features, the goal with these features is to make it as easy as possible to get security into your application.

SSL Support

A core feature of Ratpack is the ability to provide applications with the means to support secure communications with clients. It is highly advisable that any application that supports user sessions or authentication leverage secure communications. Following this practice will ensure that your users' data is secure along the wire, and that no potential eavesdropper can hijack sensitive data as it flows from application to user.

The capability to secure communications in Ratpack comes in the form of supporting SSL HTTP channels in your application. In traditional JVM web applications (i.e., those that are servlet-based), integrating SSL support can be a complex configuration detail, and is often applied secondarily or after the fact. In Ratpack, however, supporting SSL need be little more than a one-liner in your application's definition.

To begin incorporating SSL support into your application, you must first have an SSL certificate to use. We can begin testing the secure Ratpack application by generating a self-signed certificate for us to use. To do this, we can leverage the `keytool` utility

provided by the Java Development Kit (JDK) to generate the self-signed certificate and store it into a Java KeyStore (JKS) file for our application to use. Example 12-1 shows the appropriate shell command to use to build our development JKS file, as well as the output you should see.

Example 12-1. Using keytool to generate a JKS file

```
$ keytool -genkey -alias ratpack \ ❶
  -keyalg RSA -keystore /etc/server.jks \ ❷
  -keypass changeit -storepass changeit \ ❸
  -validity 365 -keysize 2048 ❹
What is your first and last name?
  [Unknown]:  Dan Woods
What is the name of your organizational unit?
  [Unknown]:  Ratpack Web Framework
What is the name of your organization?
  [Unknown]:  Development
What is the name of your City or Locality?
  [Unknown]:  Everywhere, Earth
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=Dan Woods, OU=Ratpack Web Framework, O=Development, L="Everywhere, Earth", ↵
ST=Unknown, C=Unknown correct?
  [no]:  yes
$
```

❶ This part of the command denotes the fact that we want `keytool` to generate a key for our JKS file. This will generate the public and private key pairs that will be used to encrypt the communication to clients. Also, we give the key the `alias` of `ratpack` so that if we need to inspect the JKS file later, it will be clear that this was our key.

❷ These inputs denote that we want the key to use RSA encryption and we want the certificate stored in the */etc/server.jks* file. The actual path to the JKS file does not matter, so long as the application can access it. Choose the proper location according to your system's requirements.

❸ Here, we specify the passwords for our certificate and JKS file. We will provide these values to our application specification to unlock the JKS file for use. Note that `changeit` is definitely a bad choice for a password, and you should make this value something that cannot be easily discovered.

❹   Finally, we set the validity period and key size for our certificate. For a self-signing certificate, 365 days is generally a safe choice; 2048 bit key size is also a good default.

With the demonstration JKS file in place, we can instruct our Ratpack application that we want to provide secure communications. To do this, we leverage the `ssl` method on the `ServerConfigBuilder` from the `serverConfig` block of our application definition. The code in Example 12-2 demonstrates employing our newly created JKS file.

Example 12-2. Groovy Ratpack file with SSL

```
import static ratpack.groovy.Groovy.ratpack
import ratpack.ssl.SSLContexts
import java.nio.file.Paths

ratpack {
  serverConfig {
    ssl SSLContexts.sslContext(Paths.get("/etc/server.jks"), "changeit") ❶
  }
  handlers {
    all {
      render "Hello, SSL World!"
    }
  }
}
```

❶   We use the `ssl` method with the assistance of `ratpack.ssl.SSLContexts` to build a `javax.net.ssl.SSLContext` object. This is a helper method provided by Ratpack to make the process of setting up SSL as simple as possible. The first argument supplied to the `sslContext` method is a `Path` object referencing our JKS file. The second argument is the password that we specified when building the JKS file.

When you run this Ratpack application, you will notice that the startup log message has changed slightly. This time, you should see "Ratpack started for https://localhost: 5050." Note that the server note recognizes that you are providing secure communications and has provided you with an "https" URL to access your application.

If you open a browser and navigate to *https://localhost:5050*, you will undoubtedly see a message indicating that the certificate could not be verified. This is expected, as we have self-signed the certificate in our JKS file. For demonstration's sake, if you instruct your browser to accept the certificate, you will see the "Hello, SSL World!" message prominently disabled over a secure communication channel.

When you are ready to serve your application beyond local development, you will need to obtain an SSL certificate that has been signed by a trusted certificate authority

(CA). There are several services available to provide issuance of SSL certificates that are signed by trusted certificate authorities (CA). Most of them will be offered for a small price, but new services like Let's Encrypt offer valid SSL certificates at no cost.

Obtaining an SSL certificate is a two-part process that starts with you creating a certificate signing request (CSR). We can again use the keytool command to generate the CSR, which your CA of choice will ask for when producing your SSL certificate. Example 12-3 shows the command to run to generate a CSR file.

Example 12-3. Using keytool to generate a CSR file

```
$ keytool -certreq \ ❶
  -keyalg RSA -alias ratpack \ ❷
  -file csr.csr \ ❸
  -keystore /etc/server.jks \ ❹
  -storepass changeit ❺
```

❶ Using the certreq flag, we specify to keytool that we intend to generate a CSR.

❷ Here, we ensure that the key algorithm is RSA and that we reference the proper alias from Example 12-1, when we created the JKS file.

❸ This is the file where the CSR will be output.

❹ The keystore flag references the path to where our JKS file lives.

❺ And for convenience, here we specify the password to the store. Were this not specified as an argument to keytool, you would be prompted for the password.

Successfully running this command results in no output, but a *csr.csr* file should be generated. Example 12-4 shows what an example output might look like.

Example 12-4. Sample output (csr.scr file)

```
$ cat csr.csr
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIDADCCAegCAQAwgYoxEDAOBgNVBAYTB1Vua25vd24xEDAOBgNVBAgTB1Vua25vd24xGjAYBgNV
BAcTEUV2ZXJ5d2hlcmUsIEVhcnRoMRQwEgYDVQQKEwtEZXZlbG9wbWVudDEeMBwGA1UECxMVUmF0
cGFjayBXZWIgRnJhbWV3b3JrMRIwEAYDVQQDEwlEYW4gWwgV29vZHMwggEiMA0GCSqGSIb3DQEBAQUA
A4IBDwAwggEKAoIBAQCO9smGiM++8hKIV0s4q3bRao6w7IpzmIQvWL2LSlAp8yOIhNZTqL3exhlL
xpIDT4Q0C4qTwmqx3oE8oj4Kx2WLn+sAix+6Z83ZcMEBueOu3PsqfuM64fC73m8OM06lNmvi9kiF
07LVprDOvtx+HxaRsb7MzbuC32bpHF82dqOOPSRVyLA0ssOmOkEYtFIvKZBcLoPYqBQYWv32HR3P
KSZlfKWnjuv2k0KAIXw3ukp+nILZTSs/kv359aanm4XZiOiLEo+KeS1jUEF5ctIs/SGTf1jBv70U
npvU/S2/N0hc9NRBNy3MIn4+sKzUaXcLYlAh0/WpJyLfMkDQgAwmhN5nAgMBAAGgMDAuBgkqhkiG
9w0BCQ4xITAfMB0GA1UdDgQWBBS+syvlZfYKg3As+lTL7nzUqySmcTANBgkqhkiG9w0BAQsFAAOC
AQEAdGfsJfgJ9Ym+YLen0vG5JZtqHyV1GcfazvySIWRiy//a/zH5hPrZS+fLQVrHfx2B6DsMr+MQ
XaJMlsldIDX9YZMkjl6i+TSSE4zi6eWpMClvu4DNQjcLn9ymkJVGb3QGjNRrgpPb4LCu7ErtYL7L
```

zq4LzYFw+pudO7pA1i5dNDD/6CiOn+70UGEeued3dLOo+TXR+ZBS1GcFAV4aSLL2SrkxXQzd1SYg
B1WO3ok961i7O5IYsXM8ImCwQ7iry6QY6jJZyK8PDsg0dOA4pO8fkFW3rJqYQgeMin6hg02GSIdK
roClngHNz9r0f5x2/QxT3npvYuYBu0I+f+WaBUqoxw==
-----END NEW CERTIFICATE REQUEST-----

After supplying the signing request to your CA, you will be given three certificate files: your CA's *root* certificate; their *intermediate* certificate; and finally your SSL certificate. The next step will be to import the three certificates. The console commands to do so, again using `keytool`, are shown in Example 12-5.

Example 12-5. Importing certificates

```
$ keytool -import -alias root \ ❶
 -keystore /etc/server.jks \ ❷
 -storepass changeit \ ❸
 -trustcacerts -file root.crt ❹

$ keytool -import -alias intermed \ ❺
  -keystore /etc/server.jks \
  -storepass changeit \
  -trustcacerts -file intermediate.crt ❻

$ keytool -import -alias ratpack \ ❼
  -keystore /etc/server.jks \
  -storepass changeit \
  -trustcacerts -file ratpack.crt ❽
```

❶ We begin by importing the CA's root certificate. Here, we use the `import` command to specify that we are importing a certificate to the keystore file. Be sure to use the `root` alias here so that the root certificate is properly imported.

❷ Like before, we reference the location to our JKS file.

❸ Also like before, here we specify the JKS password. If you do not specify this, you will be prompted for a password on the command line.

❹ We specify the `trustcacerts` flag to indicate that we want `keytool` to trust the other root CA certificates for which your distribution is already aware. Finally, we use the `file` flag to specify the location to the provided root certificate file. This will often be named something in accodance with your CA's naming scheme, so `root.crt` should be replaced with what you were provided.

❺ Next, we need to import the CA's intermediate certificate. We again use the `import` command, and we ensure that the `alias` flag specifies `intermed`.

❻ Like before, we use the `trustcacerts` flag, and use the `file` flag to specify the location of the CA's provided intermediate certificate.

❼ Finally, we issue the command to import the signed SSL certificate for your application. We again use `import`, but this time we specify `ratpack` for the `alias` flag, as this is what we used when we created the JKS before.

❽ The `file` flag here now references the certificate that is for your application.

With the certificates imported into your JKS file, you can now use the JKS when deploying your application to your SSL secured domain. You will need to ensure that your deployment environment is equipped for handling SSL at the application layer, and that during startup your JKS file is available as outlined before.

Ratpack's SSL support makes it easy to incorporate secure communication channels within your application. Using the knowledge that you have gained from this section, you are now empowered with the knowledge to ensure that you are building applications that have your users' security at the forefront of their experience.

Basic Authentication

As noted earlier, authenticating and authorizing users can come in many forms. To that extent, Ratpack integrates with Pac4j, which is a sort-of Swiss Army knife for authentication and authorization in Java applications. The optional framework dependency `ratpack-pac4j` provides users with easy-to-use `Handler` implementations that can be applied to the handler chain to enforce authentication and authorization.

One option for applications that need to protect some resource in their application, but do not require an invested user experience, is basic authentication. Use of basic authentication can also be valuable for services that communicate point-to-point (though OAuth is definitely the preferred mechanism for this). Whatever your need for basic authentication, Ratpack supports it. It is important to note that use of authentication and authorization requires a user session and thus the `SessionModule` must also be applied, as covered previously in this chapter.

To begin with setting up basic authentication you will need to include the `ratpack-pac4j` dependency in your application's dependencies. Like other framework modules, the Ratpack Gradle plugin provides a simple means for integrating this dependency. The Gradle build script in Example 12-6 shows how to incorporate Pac4j into your project. One important caveat to note here is that Pac4j, like Ratpack, exists as a set of libraries with optional dependencies and few opinions about how you integrate authentication and authorization into your application. In that respect, the two projects are very well aligned. The `ratpack-pac4j` framework dependency provides

integration with the core Pac4j infrastructure, but you must also include the specific Pac4j library that you wish to build authentication upon. In Example 12-6 you will notice that we also need to include the `org.pac4j:pac4j-http` dependency, which provides the integrations necessary for building basic authentication.

Example 12-6. Gradle build script with Pac4j

```groovy
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'idea'
apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency("session")     ❶
  compile ratpack.dependency("pac4j")        ❷
  compile "org.pac4j:pac4j-http:1.8.6"       ❸
  compile "commons-codec:commons-codec:1.10" ❹

}
```

❶ As noted, we need to include the `ratpack-session` dependency in our project. As we will demonstrate, `ratpack-pac4j` does not have a module itself that needs to be applied, but instead relies on the user's session to store and retrieve authentication.

❷ Here, we apply the `ratpack-pac4j` dependency to the project.

❸ In addition, we apply the `pac4j-http` dependency, which allows us to build in basic authentication.

❹ Finally, `pac4j-http` basic authentication support depends on `commons-codec`, but it is not transitively included in the dependency, so we must add it here explicitly.

Pac4j has a well-defined structure for how its authentication flow works. The flow begins by defining a *client*, which is the abstract concept that enables authentication of the user. The Pac4j client implementation retrieves the provided credentials for a

given request and validates them against its given *authenticator*. If the credentials are validated from the authenticator, then the client's *profile creator* creates a "profile" for the authenticated user and stores it in the HTTP session. The profile holds all the details about the authenticated user, which can be used to make downstream decisions about how the application interacts with the particular user.

Ratpack's integration with Pac4j makes it easy to tie together the different parts necessary for authentication. Given the Gradle build script from earlier, with the project dependencies in place, we can demonstrate basic authentication with Pac4j through the simple, though robust, code depicted in Example 12-7.

Example 12-7. Basic authentication Ratpack application

```
import org.pac4j.http.client.indirect.IndirectBasicAuthClient
import org.pac4j.http.credentials.authenticator.test.
        SimpleTestUsernamePasswordAuthenticator
import org.pac4j.http.profile.creator.AuthenticatorProfileCreator
import ratpack.pac4j.RatpackPac4j
import ratpack.session.SessionModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module SessionModule ❶
  }
  handlers {
    all( ❷
      RatpackPac4j.authenticator( ❸
        new IndirectBasicAuthClient( ❹
          new SimpleTestUsernamePasswordAuthenticator(), ❺
          AuthenticatorProfileCreator.INSTANCE ❻
        )
      )
    )
    get("auth") { ❼
      RatpackPac4j.login(context, IndirectBasicAuthClient).then { ❽
        redirect "/" ❾
      }
    }
    get { ❿
      RatpackPac4j.userProfile(context) ⓫
        .route { o -> o.present } { o -> render "Hello, ${o.get().id}!" } ⓬
        .then  { render "Not Authenticated!" } ⓭
    }
    get("logout") { ⓮
      RatpackPac4j.logout(context).then { ⓯
        redirect "/" ⓰
      }
    }
  }
```

```
    }
}
```

❶ As noted, we must have HTTP sessions available to work with authentication. Here, we apply the `SessionModule`, though your application may leverage any of the session capabilities outlined previously.

❷ The mechanism that performs authentication should be applied as an `all` handler, so that different HTTP verbs do not gain unintended access to protected resources. The helper methods from `RatpackPac4j` provide `Handler` implementations, so note here that we do not construct a handler ourselves, as we would by providing a `Closure`. Instead, we simply make a method call to `all`.

❸ `RatpackPac4j#authenticator` is the mechanism by which we will supply the Pac4j client. The handler provided from here will incorporate the provided Pac4j client into the context registry for use downstream when we wish to actually initiate the login/authentication sequence.

❹ In this example, we make use of the `IndirectBasicAuthClient`, which is provided to our project by the `pac4j-http` dependency.

❺ The `IndirectBasicAuthClient` takes a `UsernamePasswordAuthenticator` implementation to perform the authentication. This gives us a great deal of flexibility in determining the exact manner in which we authenticate a client. For the purposes of demonstration, we will use the `SimpleTestUsernamePasswordAuthenticator`, though it should be noted that this is strictly for demonstrative purposes and is not something you want to bring into a real-world application.

❻ Most appropriate for basic authentication is the `AuthenticatorProfileCreator`, which creates a limited profile with the username data stored. This is provided by `pac4j-http` and can be used to get access to the authenticated username, as demonstrated further down in the application.

❼ Here, we provide an endpoint that initiates user authentication.

❽ The `RatpackPac4j#login` method gets the client implementation that was placed in the context registry earlier and initiates the authentication sequence.

❾ Upon successful authentication, the request is redirected to the `/` endpoint, which represents the application's protected resource in this demonstration.

❿ The handler defined here is the application's protected resource, and the behavior for interacting with the client is determined by whether they have successfully authenticated or not.

⓫ We use the `RatpackPac4j#userProfile` method to extract the profile that was created by the client. The profile is retrieved from the HTTP session, so subsequent requests do not require re-authentication to gain access to protected resources.

⓬ The presence of a user profile indicates successful authentication, so here we can use the `Promise#route` method to route the response according to whether there is a user profile present in the session. The `RatpackPac4j#userProfile` method returns an `Optional<T>`, so the predicate condition checks whether the `Optional` type has a value or not. If it does, then we render a "Hello, <username>!" message back to the client.

⓭ If there is no user profile in the session, then we render an "unauthenticated" response via the `then` method.

⓮ This provides the endpoint for a user to log out when they are finished with their session.

⓯ The `RatpackPac4j#logout` method invalidates the current session's request.

⓰ After logout, we redirect the user back to `/`.

If you run this application and open your browser to *http://localhost:5050*, you will be met with a simple, "Not Authenticated!" message. This demonstrates how we have protected the `/` endpoint from unauthenticated requests. Redirecting your browser location to *http://localhost:5050/auth*, you will find that you are met with a basic authentication dialog, where you are asked for a username and password. The demonstration here is using the `SimpleTestUsernamePasswordAuthenticator`, which simply checks that the username matches the password. As noted in the application listing's callouts, you should not use this authenticator for anything more than demonstration's sake. If you type a matching username and password into your browser's authentication prompt, you will be redirected to the `/` endpoint, and this time you will be met with the "Hello, <username>!" message.

If you next direct your browser to your application's `/logout` endpoint, you will find that you are then redirected back to `/` and met again with the "Not Authenticated!" message. Your authentication has been invalidated.

Custom UsernamePasswordAuthenticator

As noted, the `SimpleTestUsernamePasswordAuthenticator` will not get you far for providing real authentication for your application. There are no out-of-the-box opinions on how your application solves this problem, however it is easy to implement your own `UsernamePasswordAuthenticator` to match your requirements.

For example, say that you wish to provide a static configuration of usernames and their corresponding password hash to authenticate against. We can leverage Ratpack's configuration mechanism to map those values to a model object that we use within our application's custom `UsernamePasswordAuthenticator`. To get started, we need only implement the `validate` method from the `UsernamePasswordAuthenticator` interface. The code in Example 12-8 demonstrates what this implementation might look like.

Example 12-8. Custom map-based MapUsernamePasswordAuthenticator

```
package app

import org.pac4j.core.exception.CredentialsException
import org.pac4j.core.profile.CommonProfile
import org.pac4j.http.credentials.UsernamePasswordCredentials
import org.pac4j.http.credentials.authenticator.UsernamePasswordAuthenticator
import org.pac4j.http.profile.HttpProfile

import javax.crypto.SecretKeyFactory
import javax.crypto.spec.PBEKeySpec

class MapUsernamePasswordAuthenticator implements UsernamePasswordAuthenticator { ❶

  private static final int ITERATIONS = 1000 ❷
  private static final int KEY_LENGTH = 192 ❸

  Map<String, String> userMap

  public MapUsernamePasswordAuthenticator(Map<String, String> userMap) { ❹
    this.userMap = userMap
  }

  @Override
  void validate(UsernamePasswordCredentials credentials) { ❺
    def passHash = userMap.get(credentials.username) ❻

    if (!passHash || passHash != hashPassword(credentials.password,
        credentials.username)) { ❼
      throwsException("Invalid username or password.") ❽
    }
```

```
    credentials.userProfile = new HttpProfile(id: credentials.username) ➒
  }

  protected void throwsException(final String message) {
    throw new CredentialsException(message);
  }

  public static String hashPassword(String password, String salt) { ➓
    char[] passwordChars = password.toCharArray();
    byte[] saltBytes = salt.getBytes();

    PBEKeySpec spec = new PBEKeySpec(passwordChars, saltBytes, ITERATIONS,
        KEY_LENGTH); ⓫
    SecretKeyFactory key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512"); ⓬
    byte[] hashedPassword = key.generateSecret(spec).getEncoded(); ⓭
    return String.format("%x", new BigInteger(hashedPassword)); ⓮
  }
}
```

❶ As noted, we need to implement the org.pac4j.http.credentials.authentica
tor.UsernamePasswordAuthenticator interface.

❷ Here, we define a static value for how many iterations are used when building the
key for the password hash (more on this to come).

❸ We also define the key length to be used.

❹ Here, we provide the map of username-to-password-hash so that we can retrieve
the appropriate hash by the supplied username.

❺ The validate method is what will be called by the Pac4j infrastructure, which
provides us with a UsernamePasswordCredentials object to get access to the
supplied username and password.

❻ From the provided username-to-password-hash map, we retrieve the password
hash for the given user.

❼ We can provide a sanity check to ensure the user is known, and then validate that
the configured hash matches the hash specified by the configuration.

❽ If there is no user or if the hashes do not match, then we throw a CredentialsEx
ception, which informs Pac4j that authentication has failed.

❾ Here, we have created a UserProfile object in the form of a simple HttpProfile
instance, for which we set the id field to the authenticated user. We set the profile

on the `UsernamePasswordCredentials` object so that it is accessible for subsequent requests.

❿ The `hashPassword` method is the main utility in our authenticator. It takes the provided password and a salt. In this case, we can use the username as the salt for the password, though you may find benefit in using a pre-configured salt provided by your application's configuration.

⓫ From the password and username, using the previously defined `ITERATIONS` and `KEY_LENGTH` count, we construct the key specification that will be used to generate the password hash (which will be validated against the configured value from earlier).

⓬ You can specify whatever digest algorithm you desire. Here we specify the `PBKDF2WithHmacSHA512` algorithm, which uses PBKDF2 with HMAC-SHA-512 as the pseudorandom function. (As a note, this is a good approach for password security, though there are caveats to the level of rigidity of this security mechanism. For example, too low an iteration count could leave your passwords more-easily brute-forced. Play with the `ITERATIONS` value to ensure you are providing the appropriate level of security. For this, 1000 count is a good starting place).

⓭ Here, we generate and capture the encoded value, which is a byte array of our password hash.

⓮ Finally, we format and return the hashed password for validation.

Using the same logic that we employ to validate the user and their password, we can fashion a simple standalone Groovy script to generate the password hash that we store in our application configuration. The script in Example 12-9 demonstrates the relevant code necessary for this.

Example 12-9. Standalone Groovy script for password hash generation

```groovy
import javax.crypto.SecretKeyFactory
import javax.crypto.spec.PBEKeySpec

ITERATIONS = 1000
KEY_LENGTH = 192

println hashPassword("mypassword", "username")

String hashPassword(String password, String salt) {
  char[] passwordChars = password.toCharArray();
  byte[] saltBytes = salt.getBytes();

  PBEKeySpec spec = new PBEKeySpec(passwordChars, saltBytes, ITERATIONS, KEY_LENGTH);
```

```
    SecretKeyFactory key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512");
    byte[] hashedPassword = key.generateSecret(spec).getEncoded();
    return String.format("%x", new BigInteger(hashedPassword));
}
```

Note that we extract the `hashPassword` function and match the `ITERATIONS` and `KEY_LENGTH` values to those in our `MapUsernamePasswordAuthenticator` class. We can run this script using the `groovy` command-line utility to generate the necessary hash.

From here, we need to build a configuration file to store the password hash. By now, you have a firm understanding of how Ratpack's configuration mechanism works, and likely already have a configuration file set up for your application. We can add to it a simple object entry with our username and generated hash. The listing in Example 12-10 shows what a YAML configuration file entry would look like with our username and password hash map included.

Example 12-10. Config YAML with password hash map

```
security:
  basic:
    userPassMap:
      username: -19ab61bf5cff843597ed69e28d59baf77e2e1f6078c48718  ❶
```

❶   Note that here you will want to replace `username` with your user's username, and the corresponding hash value with that generated from the above script.

To incorporate the security configuration into our application, we should have a Java/Groovy bean that represents the directives we have defined. Consider the `Security Config` class depicted in Example 12-11, which we will use in our application to map the YAML config to an object structure we can work with in our application.

Example 12-11. Security configuration for custom authenticator

```
package app

class SecurityConfig {

  BasicAuthConfig basic  ❶

  static class BasicAuthConfig {  ❷
    Map<String, String> userPassMap = [:]  ❸
  }
}
```

❶   We embed the structure of the `security.basic` directive in a nested object type, `BasicAuthConfig`.

❷ The `BasicAuthConfig` can be defined as its own standalone class, but for demonstration (and simplicity), we will define it here as a static inner class.

❸ The `userPassMap` will hold the mappings of *username* to *password* as defined in the `security.basic.userPassMap` directive of our config file. This `Map` will be what we work with when validating authentication.

Next, we must read our application's configuration file into our application, and map it to the `SecurityConfig` class for use within our authentication flow. Following that, we now give the `MapUsernamePasswordAuthenticator` to the Pac4j `BasicAuthClient` so that it knows how to authenticate the incoming user. To accomplish this, we need only change our use of the `BasicAuthClient` Pac4j client shown earlier, and provide it with our new authenticator. The code provided in Example 12-12 shows how to accomplish this.

Example 12-12. Basic authentication with MapUsernamePasswordAuthenticator

```
import app.MapUsernamePasswordAuthenticator
import app.SecurityConfig
import org.pac4j.http.client.indirect.IndirectBasicAuthClient
import org.pac4j.http.profile.creator.AuthenticatorProfileCreator
import ratpack.pac4j.RatpackPac4j
import ratpack.session.SessionModule

import static ratpack.groovy.Groovy.ratpack

ratpack {
  serverConfig { ❶
    yaml("config.yml")
    sysProps()
    env()
    require("/security", SecurityConfig)
  }
  bindings {
    module SessionModule
  }
  handlers {
    all(
        RatpackPac4j.authenticator(
            new IndirectBasicAuthClient(
                new MapUsernamePasswordAuthenticator(
                    registry.get(SecurityConfig).basic.userPassMap), ❷
                AuthenticatorProfileCreator.INSTANCE
            )
        )
    )
    get("auth") {
      RatpackPac4j.login(context, IndirectBasicAuthClient).then {
        redirect "/"
```

```
          }
        }
      get {
        RatpackPac4j.userProfile(context)
            .route { o -> o.present } { o -> render "Hello, ${o.get().id}!" }
            .then  { render "Not Authenticated!" }
      }
      get("logout") {
        RatpackPac4j.logout(context).then {
          redirect "/"
        }
      }
    }
  }
}
```

❶ Within the `serverConfig` block of our application definition (which you should
   be familiar with now), we specify that we want to consume the *app.yml* YAML
   configuration file, and allow configuration overrides via Java System Properties
   or appropriately named environment variables. Finally, we map the structure of
   the `security` configuration from our YAML file to the `SecurityConfig` class. As
   you well know by now, the `require(..)` syntax maps the configuration and
   places the resulting object in the registry, for use within our application handlers.

❷ Instead of using the `SimpleTestUsernamePasswordAuthenticator` here, we now
   supply our `MapUsernamePasswordAuthenticator`. Note that the argument to the
   constructor is the `userPassMap` from our `SecurityConfig` class, so at this point,
   we can get the `SecurityConfig` object (which has our configuration values map-
   ped to it) from the registry, and provide the `` ` ` `` to the `MapUsernamePasswordAu`
   `thenticator`.

The structure of the project at this point looks like the tree shown in Example 12-13.
This is noted for the reader to understand that for the purposes of this demonstra-
tion, the *app.yml* must be within the project's *base directory* (in this case, *src/ratpack*).
If you wish to load configuration from a filesystem path outside of the base directory,
you will need to provide the `ServerConfigBuilder#yaml` method with the
`java.nio.file.Path` that points to the location of your configuration file.

Example 12-13. Basic authentication project structure

```
.
├── build.gradle
└── src
    ├── main
    │   └── groovy
    │       └── app
    │           ├── MapUsernamePasswordAuthenticator.groovy
    │           └── SecurityConfig.groovy
```

```
        └── ratpack
            ├── config.yml
            └── ratpack.groovy
```

5 directories, 5 files

If you start this application and open your browser to *http://localhost:5050*, you will notice that the unauthenticated behavior has not changed much since the last version of the code. We are still met with the "Not Authenticated!" screen when we land on the / endpoint without providing any authentication. If you navigate now to /auth, you will see the basic authentication prompt that you are familiar with from earlier. This time, if you specify the username and password combination from your configuration file, you will find that you are now validated according to your application's configuration!

As demonstrated in this section, basic authentication can serve as a good utility for implementing base-level security into your application. Ratpack's handler chain makes it a powerful option for securing your protected resource routes and easily defining the application behavior according to whether the user is authenticated or not. From a service-to-service interaction perspective, basic authentication may prove a suitable option for securing access to particular consumers.

Form-Based Authentication

Basic authentication is not a practical solution for applications that wish to provide a rich user experience. Given that, we will need to provide some way that users can authenticate to our application through an authentication flow that fits into the desired user experience. Ratpack's integration with Pac4j gives you the flexibility to design your authentication mechanism in the way that works best for your application. In this section, we will explore a means for building a data-backed, form-based authentication system.

Similar to the BasicAuthClient demonstrated in the prior section, for building form-based authentication we can utilize the FormClient and provide a UsernamePasswordAuthenticator, just as we did previously. The FormClient, which is provided as part of the same pac4j-http dependency, provides us the means to specify the URL of the login form that is to be used for authentication. When an unauthenticated client attempts to access a protected resource, the Pac4j infrastructure will redirect the request to the specified login form's URL, thus initiating the authentication flow. The login form need only POST username and password form parameters to the Pac4j authentication callbackUrl, which is provided to us by the FormClient. We can use Ratpack's advanced templating support to facilitate the construction of the login form.

To begin, we must first envision the application structure that will exemplify form-based authentication. For the purposes of demonstration, consider that you have a basic application landing page, which is represented by an *index.html* file; second to that, we have a *login.html*, which provides the view within which users will authenticate; finally, we have a *protectedIndex.html* file, which is rendered specifically for the authenticated user. If we again start with the `SimpleTestUsernamePasswordAuthenticator`, as we did in the prior section, then we can realize a project structure similar to the one shown in Example 12-14. When a user is authenticated, we want to show them the *protectedIndex.html* file when they visit the / route, while an unauthenticated user visiting / should get the *index.html* file.

Example 12-14. Initial project structure for form-based authentication

```
.
├── build.gradle
└── src
    └── ratpack
        ├── ratpack.groovy
        └── templates
            ├── index.html
            ├── login.html
            └── protectedIndex.html

3 directories, 5 files
```

Nothing need change in our *build.gradle* file from the demonstration in the prior section. Our *ratpack.groovy*, however, does need to change slightly to accommodate the HTML nature of what we are going to accomplish. Consider the code provided in Example 12-15, which fully implements form-based authentication in our application.

Example 12-15. Ratpack Groovy application with FormClient

```groovy
import org.pac4j.http.client.indirect.FormClient
import org.pac4j.http.credentials.authenticator.test.
        SimpleTestUsernamePasswordAuthenticator
import org.pac4j.http.profile.creator.AuthenticatorProfileCreator

import static ratpack.groovy.Groovy.ratpack
import static ratpack.groovy.Groovy.groovyTemplate
import static java.util.Collections.singletonMap

import ratpack.session.SessionModule
import ratpack.pac4j.RatpackPac4j
import ratpack.groovy.template.TextTemplateModule

ratpack {
  bindings {
```

```
    module SessionModule
    module TextTemplateModule ❶
  }
  handlers {
    def formClient = new FormClient( ❷
      "auth",
      new SimpleTestUsernamePasswordAuthenticator(),
      AuthenticatorProfileCreator.INSTANCE
    )

    all(RatpackPac4j.authenticator("auth", formClient)) ❸

    get("login") { ❹
      render(groovyTemplate(
              singletonMap("callbackUrl", formClient.loginUrl),
              "login.html"))
    }

    get("logout") { ❺
      RatpackPac4j.logout(context).then {
        redirect '/'
      }
    }

    get { ❻
      RatpackPac4j.userProfile(context)
        .route { o -> o.present } { o ->
          render(groovyTemplate([profile: o.get()], "protectedIndex.html"))
        }
        .then {
          render(groovyTemplate([:], "index.html"))
        }
    }
  }
}
```

❶ For the purposes of this web application, we will leverage Ratpack's Groovy Text
   Template support for rendering HTML pages, so we must bind the TextTemplate
   Module.

❷ Within the handler chain, we can construct the FormClient for use as part of the
   authentication flow. The first argument we supply is the authentication endpoint,
   defined here as auth; the second argument is the UsernamePasswordAuthentica
   tor. As noted, here we will use the SimpleTestUsernamePasswordAuthenticator
   again for the purposes of demonstration; the third and final argument we supply
   is the UsernameProfileCreator, so that we can capture the logged-in user's
   details in the protectedIndex.

❸ Here, we attach the `Pac4jAuthenticator`, which will provide the `/auth` endpoint for which our login form will supply the `username` and `password` parameters from our web interface.

❹ The `/login` endpoint will be the route within which we will render our application's login form. We supply to the *login.html* template the `callbackUrl`, which we retrieve from the `formClient.loginUrl` property. This will be the endpoint to which the login form POSTs the user's credentials for authentication.

❺ Our *protectedIndex.html* interface will provide not only a user-authenticated interface, but also the means for a user to log out. Here, we define the endpoint that a user will access to invalidate their authentication to our web application.

❻ Finally, we define the landing page interaction. When a user is authenticated and accesses the `/` endpoint, they will be rendered the *protectedIndex.html* template; when a user is unauthenticated, they will be rendered our application's unauthenticated *index.html* page.

Next, we should take a look at what these HTML template files look like. Let's start with our application's landing page (the *index.html* file in our project's *templates* directory). This is a fairly simple demonstration, but you can notice the *index.html* file has no model data associated with it. Example 12-16 shows what the code for our basic landing page will look like.

Example 12-16. Landing page HTML file

```html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to My Product!</title>
</head>
<body>

Welcome to My Product's landing page! To access your resources,
you will need to <a href="/login">Login</a>.

</body>
</html>
```

As you can see, the landing page is a fairly simple HTML page, but it does provide users with a link to the `/login` endpoint, which will give them the form within which they can provide their credentials. Let's take a look at the contents of the login page. Again, so as to not conflate the demonstration, the login page's contents are simple, as shown in Example 12-17.

Example 12-17. Login page HTML file

```html
<!DOCTYPE html>
<html>
<head>
<title>My Product Login Page</title>
</head>
<body>
<h1>Please Login</h1>
<form action="${model.callbackUrl}" method="POST"> ❶
  <span>Username: </span>
  <input type="text" name="username" placeholder="username"> ❷
  <br/>
  <span>Password: </span> <input type="password" name="password"> ❸
  <br/>
  <input type="hidden" name="client_name" value="FormClient"> ❹
  <input type="submit" value="Login"> ❺
</form>
</body>
</html>
```

❶ Here, we define the form and pull in from the template's model the `callbackUrl`, to which the form's parameters will be POSTed. This is the `/auth` value specified in our *ratpack.groovy* file.

❷ Here, we specify an input text field with the `username` parameter name.

❸ Next, we specify a password input field with the `password` parameter name.

❹ We must also inform the authenticator to which client it should be authenticating, so here we attach a hidden form input field with the `client_name` parameter name and the `FormClient` value. This will inform Pac4j to which client implementation it should authenticate the user.

❺ Finally, we provide a submit button for the user to initiate the authentication sequence.

Lastly, we should take a look at what the authenticated user will get when they successfully log in. This is housed in the *protectedIndex.html* template, and should provide some user-specific interface according to our application's requirements. Example 12-18 shows the code for a simple, user-specific landing page.

Example 12-18. Authenticated landing page HTML file

```html
<!DOCTYPE html>
<html>
<head>
<title>My Product &bull; ${model.profile.id}</title> ❶
```

```
</head>
<body>
<h1>Welcome ${model.profile.id}!</h1> ❷
<p>This is your protected product page!</p>
<p><a href="/logout">Logout</a></p> ❸
</body>
</html>
```

❶ We can utilize the profile supplied to the template's model to get access to the authenticated user's ID and render a custom title for their landing page (in this case, their username).

❷ We can also use the same method to render some user-specific value in the body (in this case, a simple welcome message).

❸ Finally, we provide a link to the logout facility so users can escape their authenticated interface.

With all this in place, if you fire up the application and navigate your browser to *http://localhost:5050*, you will find that you are welcomed with the unauthenticated user's landing page. From here, you can access the link to the login screen. Doing so navigates you to the `/login` endpoint and you are met with a form requesting the username and password credentials. Because we are using the `SimpleTestUsername PasswordAuthenticator`, if you provide a matching username and password combination and submit the form, you will find that you are redirected to our application's *protectedIndex.html* page, which has your user-specific details rendered in view.

You can use the same `MapUsernamePasswordAuthenticator` from the prior section to drive authentication for your application based off of a fixed list in your configuration, but that does not scale very well for applications that need authentication to be data driven. For this, however, we can still utilize the `MapUsernamePasswordAuthenti cator`, but hydrate the `userPassMap` with values from a database table instead of from static configuration.

Data-Driven Form Authentication

Building on the data-driven techniques that we learned earlier in the book and matching that with what we now know about building a custom `UsernamePasswordAu thenticator`, we can incorporate a data-driven approach to authenticating users in our application. To start with, we will want to modify the previously shown *build.gradle* file to also incorporate the `ratpack-hikari` dependency. We will use Ratpack's Hikari support to provide a connection pool for our authenticator. For the purposes of demonstration (and simplicity), we will again employ the H2-embedded database, and bootstrap it with some simple credential data. The *build.gradle* build script is shown in Example 12-19.

Example 12-19. Build script with database dependencies

```groovy
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}

dependencies {
  compile ratpack.dependency("session")
  compile ratpack.dependency("pac4j")
  compile ratpack.dependency("hikari")
  compile 'com.h2database:h2:1.4.190'
  compile "org.pac4j:pac4j-http:1.8.5"
}
```

With the necessary dependencies in place, let's take a look at what our *ratpack.groovy* file evolves into. The application in Example 12-20 demonstrates creating the `Data Source` object, building the user authentication table, and populating it with some data. We will take the Groovy SQL approach to working with our `DataSource`, as the simplicity works well for this use case.

Example 12-20. Groovy Ratpack application (DataSource setup)

```groovy
import app.DatabaseUsernamePasswordAuthenticator
import groovy.sql.Sql
import org.pac4j.http.client.indirect.FormClient
import org.pac4j.http.profile.UsernameProfileCreator
import org.pac4j.http.profile.creator.AuthenticatorProfileCreator
import ratpack.groovy.sql.SqlModule
import ratpack.groovy.template.TextTemplateModule
import ratpack.hikari.HikariModule
import ratpack.pac4j.RatpackPac4j
import ratpack.server.Service
import ratpack.server.StartEvent
import ratpack.session.SessionModule

import static java.util.Collections.singletonMap
import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
```

```
  bindings {
    module SessionModule
    module TextTemplateModule
    module SqlModule ❶
    module(HikariModule) { c -> ❷
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'
      c.username = 'sa'
      c.password = ''
    }

    bindInstance new Service() { ❸
      void onStart(StartEvent e) {
        Sql sql = e.registry.get(Sql)
        sql.execute "CREATE TABLE USER_AUTH(USER VARCHAR(255), PASS VARCHAR(255))"
        sql.execute "INSERT INTO USER_AUTH (USER, PASS) " +
          "VALUES('learningratpack', " +
            "'768122eeeebdafa3eb878f868b0e4e6a4944367aa635538f')" ❹
      }
    }
  }
  handlers {
    // ... no changes here yet ...
  }
}
```

❶   Remember that we need to apply the `SqlModule`, so that the `Sql` object will be properly constructed with our `DataSource`.

❷   We apply the `HikariModule` and configure it here to construct an H2 in-memory embedded database.

❸   As you're already familiar, we use a `ratpack.server.Service` instance to bootstrap the auth data into our database.

❹   Here, we will add a username/password combination of `learningratpack/``r4tp@CKrul3z!`. We use the same mechanism for hashing the password as from the prior section.

This is a good start for getting our application set up to pull authentication data from our database. The next step will be to implement a new `UsernamePasswordAuthenti``cator` for use with the `FormClient`. This time, instead of reading the authentication information from a `Map`, we will query the database for the supplied username and get the stored password hash. The implementation for the `DatabaseUsernamePasswordAu``thenticator` looks very similar to `MapUsernamePasswordAuthenticator`, save for the fact that this time we inject the `Sql` object, which will be our interface to the database. The code in Example 12-21 shows what this implementation looks like.

Example 12-21. Custom database-backed DatabaseUsernamePasswordAuthenticator

```groovy
package app

import com.google.inject.Inject
import groovy.sql.Sql
import org.pac4j.core.exception.CredentialsException
import org.pac4j.http.credentials.UsernamePasswordCredentials
import org.pac4j.http.credentials.authenticator.UsernamePasswordAuthenticator
import org.pac4j.http.profile.HttpProfile

import javax.crypto.SecretKeyFactory
import javax.crypto.spec.PBEKeySpec

class DatabaseUsernamePasswordAuthenticator
      implements UsernamePasswordAuthenticator {
  private static final int ITERATIONS = 1000
  private static final int KEY_LENGTH = 192

  private final Sql sql

  @Inject
  public DatabaseUsernamePasswordAuthenticator(Sql sql) {   ❶
    this.sql = sql
  }

  @Override
  void validate(UsernamePasswordCredentials credentials) {
    def userRow = sql.firstRow(
    "SELECT * FROM USER_AUTH WHERE USER = ${credentials.username}"   ❷
    )

    if (!userRow) {   ❸
      throwsException("Invalid username or password")
    }

    def passHash = userRow["PASS"]   ❹

    if (!passHash || passHash != hashPassword(credentials.password,
        credentials.username)) {
      throwsException("Invalid username or password.")
    }

    credentials.userProfile = new HttpProfile(id: credentials.username)
  }

  protected void throwsException(final String message) {
    throw new CredentialsException(message)
  }

  public static String hashPassword(String password, String salt) {
    char[] passwordChars = password.toCharArray();
```

```
    byte[] saltBytes = salt.getBytes();

    PBEKeySpec spec = new PBEKeySpec(passwordChars, saltBytes, ITERATIONS,
                                     KEY_LENGTH);
    SecretKeyFactory key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512");
    byte[] hashedPassword = key.generateSecret(spec).getEncoded();
    return String.format("%x", new BigInteger(hashedPassword));
  }
}
```

❶ As noted, we use the Guice `@Inject` annotation to get access to the Groovy SQL `Sql` object for querying the database in the `validate` method.

❷ From the credentials supplied by the user, we query the USER_AUTH database table for the row corresponding to the specified username. We can safely make this call here synchronously, because Ratpack assumes the authentication sequence will block, so the `UsernamePasswordAuthenticator` is pre-emptively scheduled to a blocking thread.

❸ If there is no row for the given username, then throw an exception, which will short-circuit the authentication sequence within Pac4j.

❹ Provided we have found the user's authentication row in the database, capture the PASS field, which is the hashed version of the user's password.

The next step is that we must now incorporate the `DatabaseUsernamePasswordAu thenticator` into the application's Guice-backed registry. We use the `bind` method in the `bindings` block to add our new component. Following that, we must slightly modify the handler chain to pull the `DatabaseUsernamePasswordAuthenticator` from the registry during the request cycle and supply it to the `FormClient`. The code in Example 12-22 shows the full application definition using our new `DatabaseUser namePasswordAuthenticator`.

Example 12-22. Ratpack Groovy application with
DatabaseUsernamePasswordAuthenticator

```
import app.DatabaseUsernamePasswordAuthenticator
import groovy.sql.Sql
import org.pac4j.http.client.indirect.FormClient
import org.pac4j.http.profile.UsernameProfileCreator
import org.pac4j.http.profile.creator.AuthenticatorProfileCreator
import ratpack.groovy.sql.SqlModule
import ratpack.groovy.template.TextTemplateModule
import ratpack.hikari.HikariModule
import ratpack.pac4j.RatpackPac4j
import ratpack.service.Service
import ratpack.service.StartEvent
```

```
import ratpack.session.SessionModule

import static java.util.Collections.singletonMap
import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack

ratpack {
  bindings {
    module SessionModule
    module TextTemplateModule
    module SqlModule
    module(HikariModule) { c ->
      c.dataSourceClassName = 'org.h2.jdbcx.JdbcDataSource'
      c.addDataSourceProperty 'URL', 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1'
      c.username = 'sa'
      c.password = ''
    }

    bindInstance new Service() {
      @Override
      void onStart(StartEvent e) throws Exception {
        Sql sql = e.registry.get(Sql)
        sql.execute "CREATE TABLE USER_AUTH(USER VARCHAR(255), PASS VARCHAR(255))"
        sql.execute "INSERT INTO USER_AUTH (USER, PASS) " +
          "VALUES('learningratpack', " +
            "'768122eeeebdafa3eb878f868b0e4e6a4944367aa635538f')"
      }
    }

    bind(DatabaseUsernamePasswordAuthenticator) ❶
  }
  handlers {
    def callbackUrl = "auth" ❷
    def formClient = new FormClient(
        callbackUrl,
        registry.get(DatabaseUsernamePasswordAuthenticator), ❸
        AuthenticatorProfileCreator.INSTANCE
    )
    all(RatpackPac4j.authenticator(callbackUrl, formClient))

    get("login") {
      render(groovyTemplate(singletonMap("callbackUrl", callbackUrl), "login.html"))
    }

    get("logout") {
      RatpackPac4j.logout(context).then {
        redirect '/'
      }
    }

    get {
      RatpackPac4j.userProfile(context)
```

```
      .route { o -> o.present } { o ->
        render(groovyTemplate([profile: o.get()], "protectedIndex.html"))
      }
      .then {
        render(groovyTemplate([:], "index.html"))
      }
    }
  }
}
```

❶ This line binds our `DatabaseUsernamePasswordAuthenticator` into the user registry.

❷ Here, we set the `callbackUrl`, which will specify the route that the HTML form should POST to, and we ensure it is bound to the `authenticator`.

❸ In the handler chain, we can access components from the user registry via the `registry.get(..)` call, as shown here.

Everything else in the application handler chain remains the same. With a little bootstrapping of data and a new mechanism for validating the supplied credentials, we have a fully data-driven authentication sequence! If you start the application and navigate to *http://localhost:5050*, you will again be met with the default landing page. Navigating to the login screen and supplying the previously referenced credential pair, you will find that you are directed to the fully authenticated page, and the whole sequence was entirely data driven from your database connection.

Based on what you know from earlier in the book, you can start to envision how this trivial application could grow into a powerful real-world user experience, complete with user registration. It is left as an exercise to the reader to implement the necessary endpoints for creating, modifying, and updating user accounts. With your robust knowledge of how Ratpack supports data-driven web applications, this will certainly be a painless task.

Additional Authentication Means

Basic and form-based authentication strategies are not the only game in town when it comes to authenticating users with Pac4j. Now that you have the necessary knowledge of how Ratpack and Pac4j integrate to provide authentication, it should be a seamless process to incorporate the other authentication clients that Pac4j provides. Simply trading out the `pac4j-http` dependency with the Pac4j dependency appropriate to your application's requirements, you will find new `Client` and `Authenticator` implementations that you can integrate into your application's authentication flow.

Some examples of the various authentication types supported by Pac4j and their corresponding dependency coordinates include:

- CAS - `org.pac4j:pac4j-cas`
- Google App Engine - `org.pac4j:pac4j-gae`
- JSON Web Tokens - `org.pac4j:pac4j-jwt`
- LDAP - `org.pac4j:pac4j-ldap`
- OAuth - `org.pac4j:pac4j-oauth`
- OpenID Connect (i.e., Google Apps) - `org.pac4j:pac4j-oidc`
- OpenID - `org.pac4j:pac4j-openid`
- SAML - `org.pac4j:pac4j-saml`

And many more continue to be added every day. Whatever your application's authentication requirements, you will no doubt find the necessary implementations available. As noted and demonstrated earlier, when your application's requirements do not fit exactly into the flow opined by Ratpack and Pac4j, the two frameworks get out of your way quickly to let you get the job done.

Chapter Summary

This chapter has covered a lot of ground in terms of introducing security into your application, but it should be noted that no single text can provide comprehensive coverage on the subject of web application security. Instead, this chapter will help you understand the means by which Ratpack provides security for your application. At this point, you should have an in-depth understanding of building high-performance, robust, and lean web applications with Ratpack.

Going to Production

Congratulations! By now, you have all the tools and strategies at your fingertips to build robust, reactive web applications with Ratpack, and your applications are fully capable and ready to be deployed to production. However, in closing we should discuss a few final considerations before doing so. As the final chapter to this book, we will briefly discuss some of the aspects that need to be considered for any production web application deployment, and the faculties provided by Ratpack for accommodating a production environment. We will cover how to bring insight to your applications in the form of metrics, as well as provide coverage of distribution techniques that support the continuous delivery and deployment tactics commonly employed in modern infrastructure architectures.

You should take away from this chapter a firm understanding of the remaining bits of application development with Ratpack that set your project up for long-term success.

Publishing Metrics

There is no disputing the fact that one of the most important aspects of running any web application in production is to provide external insight into what your code is doing. Facilitating this can take many forms, but a common and well-adopted approach is to publish metrics to a centralized metric server, according to your application's requirements. In doing so, you will provide a level of operationalization that ensures that your application remains stable when deployed to a production environment.

Many modern metric collectors provide the ability to build alert and notification strategies according to the metric data coming out of your application. Following this principle of application development will ensure that you know exactly what is happening with your application during deployment, so that you can more easily per-

form root-cause analysis when there is a problem, and become aware of problems that exist before your consumers notice them.

There are many strategies and libraries available for producing metrics from your application. Dropwizard Metrics is one such library that has wide adoption and allows for flexible implementations that can integrate with a variety of metric collectors. As such, Ratpack provides optional support for integrating with Dropwizard Metrics, and following suit with its other framework features, makes the process of generating various metric types as simple as possible.

To get started with incorporating Dropwizard Metrics into your application, you need to include the `ratpack-dropwizard-metrics` dependency as a project dependency. Similar to other framework modules, this dependency can be incorporated into the `dependencies` block of your Gradle build script using the `compile ratpack.dependency("dropwizard-metrics")` notation. With the necessary dependency in place, you will need to apply the `DropwizardMetricsModule` to your application via the `BindingsSpec`. The code in Example 13-1 shows the beginnings of an application with metrics support incorporated.

Example 13-1. Ratpack Groovy application with DropwizardMetricsModule

```
import static ratpack.groovy.Groovy.ratpack

import ratpack.dropwizard.metrics.DropwizardMetricsModule

ratpack {
  bindings {
    module(DropwizardMetricsModule)
  }
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

By default, the `DropwizardMetricsModule` prepends a `RequestTimingHandler` to your application's handler chain, which is responsible for capturing the amount of time spent in the request/response lifecycle of your handler chain. That is to say, with nothing more than applying the `DropwizardMetricsModule` to your application, you get timing metrics on how long it took your application to respond to a request. Additionally, out of the box you get counter metrics on handler response codes. Given that the module is prepending a `HandlerDecorator` to your handler chain, it is critically important that the module is applied before all other modules in your `bindings` block.

Default support is also provided that publishes a timer metric for how long your application spent in `Blocking` operations by tying into the execution model and observing when a `Promise` is scheduled to the blocking thread pool. Keeping track of this metric will give you an idea of how much time is spent working on I/O operations versus computation. This may give you the necessary insight to implement optimizations.

The metrics integration also supports the ability to capture JVM metrics coming out of your application. This is a simple flag to enable, and can be reported out alongside the rest of your application metrics. When running in production, having a view into what is happening in your application's JVM can yield insight into memory and CPU utilization, yield insight into areas of your application that may need improvement, and generally give you a good idea of how your application is performing operationally. Capturing JVM metrics is not enabled by default, but it is a simple configuration directive applied to the `DropwizardMetricsModule` configuration in your `bindings` block. The updated code in Example 13-2 shows how to turn on JVM metrics collection.

Example 13-2. Ratpack Groovy application with DropwizardMetricsModule

```
import static ratpack.groovy.Groovy.ratpack

import ratpack.dropwizard.metrics.DropwizardMetricsModule

ratpack {
  bindings {
    module(DropwizardMetricsModule) { c ->
      c.jvmMetrics true ❶
    }
  }
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

❶ Using the module configuration that you are already well familiar with, we set the `jvmMetrics` directive to `true` for the `DropwizardMetricsModule` configuration.

Enabling Reporting

In Dropwizard Metrics terms, a `Reporter` is a class that is responsible for sending published metrics to a collector for analysis. Ratpack supports all of the default reporters available from Dropwizard Metrics. These reporters include support for reporting to JMX, the standard output console, a CSV file, Slf4j, and Graphite. Rat-

pack further provides the ability to report and stream metrics over a WebSocket connection. It's possible to enable all of the reporting strategies simultaneously (you may also choose not to use any). The application code in Example 13-3 shows the evolved application code from earlier, this time with the JMX, console, and Slf4j reporters enabled.

Example 13-3. Enabling JMX, console, and Slf4j reporters

```groovy
import static ratpack.groovy.Groovy.ratpack

import ratpack.dropwizard.metrics.DropwizardMetricsModule

ratpack {
  bindings {
    module(DropwizardMetricsModule) { c ->
      c.jvmMetrics true
      c.jmx() ❶
      c.console() ❷
      c.slf4j() ❸
    }
  }
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

❶ Nothing more than a call to `jmx()` inside the module configuration is necessary to enable JMX reporting. When a JMX server management port is specified as part of your application's startup parameters, you will be able to attach to your process to get your application metrics.

❷ A simple call to `console()` reports metrics to your process's standard output.

❸ Applications using Slf4j for logging will see metrics output according to your logging configuration when this reporter is enabled via the `slf4j()` call.

Enabling the CSV and Graphite reporters requires some additional, albeit simple, configuration. For example, when the CSV reporter is enabled, a reporting directory must be specified. Periodically, the metrics collected from your application will be output to individual *.csv* files that you can work with in the ways that best suit your requirements. The code shown in Example 13-4 demonstrates setting up this reporter.

Example 13-4. Enabling CSV reporter

```
import static ratpack.groovy.Groovy.ratpack

import ratpack.dropwizard.metrics.DropwizardMetricsModule

ratpack {
  bindings {
    module(DropwizardMetricsModule) { c ->
      c.csv { csvConfig -> ❶
        def reportingDir = new File("metrics")
        reportingDir.mkdir()
        csvConfig.reportDirectory(reportingDir) ❷
      }
    }
  }
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

❶ Within the configuration we call the `csv` method, to which we provide an `Action` that supplies to us the `CsvConfig`.

❷ On the `CsvConfig`, we can provide the `reportDirectory` method with a `File` to the directory where we want metric CSV files written.

Graphite is a popular choice for publishing metrics, as it works as a standalone system for which your application will send metrics in real time. Within Graphite, you can get a comprehensive view of the metrics for your entire infrastructure. Its advanced graphing capabilities give you a digestible visual insight into what is happening across your deployment footprint. Note that publishing metrics to Graphite requires that an agent be running on your deployment server, like `collectd` (for more information, see the collectd Wiki.

The default configuration for reporting metrics to Graphite are good enough to get started with. When you need deeper customization, such as prefixing metrics or configuring the rate and duration units to be used with your metrics, you will need to supply those values to the `DropwizardMetricsModule` configuration as part of the `graphite` call. This process is similar to what we have just seen with implementing CSV reporting. The code in Example 13-5 shows how to apply these customizations.

Example 13-5. Enabling Graphite reporter

```
import static ratpack.groovy.Groovy.ratpack
```

```
import ratpack.dropwizard.metrics.DropwizardMetricsModule
import java.util.concurrent.TimeUnit

ratpack {
  bindings {
    module(DropwizardMetricsModule) { c ->
      c.graphite { graphiteConfig -> ❶
        graphiteConfig.prefix("myapp") ❷
          .rateUnit(TimeUnit.MILLISECONDS) ❸
          .durationUnit(TimeUnit.MILLISECONDS) ❹
      }
    }
  }
  handlers {
    get {
      render "Hello, World!"
    }
  }
}
```

❶ Within the configuration we call the `graphite` method, to which we provide an `Action` that supplies to us the `GraphiteConfig` for customization.

❷ This demonstrates setting a prefix on metrics that are reported to Graphite from our application.

❸ Here, we can configure the `TimeUnit` value that is to be used to convert rate units.

❹ Similarly, for metrics that supply durations, we can specify a `TimeUnit` value to ensure the metrics are converted properly before they are reported.

The various metric reporters provided out of the box by Ratpack give you an excellent footprint upon which you can gather valuable insight into your application's operational state. From here, you can provide further insight into your application's specific function by publishing custom metrics.

Publishing Custom Metrics

In addition to the default timer and counter metrics provided, you can easily surface custom metrics from your application. This can be useful for tracking any detail that is specific to your environment, project, or other general requirements. Using the provided `com.codahale.metrics.MetricRegistry`, you can publish any of the metrics types supported by the Dropwizard Metrics project. This includes support for complex gauges, counters, histograms, meters, and timers. This section will not serve as an exhaustive text on the various metric types or the methods of the `MetricRegistry`. Instead, it is left as an exercise to the reader to explore the Dropwizard Metrics

project's extensive user guide to better understand how and when to use the various metric types.

As a demonstration, however, consider the code in Example 13-6, which is an elaboration on the prior example. This example demonstrates incrementing a counter metric any time that a request hits a specific handler.

Example 13-6. Publishing custom metrics

```
import static ratpack.groovy.Groovy.ratpack

import com.codahale.metrics.MetricRegistry
import ratpack.dropwizard.metrics.DropwizardMetricsModule

ratpack {
  bindings {
    module(DropwizardMetricsModule) { c ->
      c.console()
    }
  }
  handlers {
    get("user") { MetricRegistry metricRegistry -> ❶
      metricRegistry.counter("myapp.user.hits").inc() ❷
      render "Hello, World!"
    }
  }
}
```

❶ Within our handler, we can inject the `MetricRegistry`, which is bound from the `DropwizardMetricsModule`.

❷ Here, we use the `counter` method off of the `MetricRegistry`, which gives us a `Counter` type that we can call `inc()` on to increment.

Application Health Checks

No conversation on going to production would be complete without discussing Ratpack's integrated means of providing health checks for your application. You can get started with building health checks into your application by implementing the `ratpack.health.HealthCheck` interface and adding the `ratpack.health.HealthCheckHandler` to your handler chain. Health checks are generally used to ensure that your application's service dependencies are accessible, and if they are not, report them as such. These dependencies may include databases or message queues, for example.

Especially in a production deployment, it is important that each instance of your application reports its health so that the deployment infrastructure knows whether your service is healthy or not. In cloud deployments, an unhealthy instance of your

application may cause it to be taken out of traffic rotation, thus ensuring that your users are never unnecessarily exposed to errors. Furthermore, it may allow your monitoring infrastructure to alert you to the fact that your application has entered an unhealthy state.

Let's build on the simplest demonstration of health checks, and simply drive the health check result based on a static configuration toggle. If we place a property holding class into the registry and allow a handler in the chain to manipulate a `healthy` boolean value on the class, we can demonstrate how the `HealthCheckHandler` reports the state of your application. Example 13-7 shows a simple application demonstrating this capability.

Example 13-7. Simple demonstration of the HealthCheckHandler

```
import ratpack.exec.Promise
import ratpack.health.HealthCheck
import ratpack.health.HealthCheckHandler

import static ratpack.groovy.Groovy.ratpack

class PropertyHolder { ❶
  boolean healthy = true
}

ratpack {
  bindings {
    bindInstance new PropertyHolder() ❷
    bindInstance HealthCheck.of("propertyHealth") { registry -> ❸
      def propertyHolder = registry.get(PropertyHolder)
      Promise.sync {
        propertyHolder.healthy ? ❹
            HealthCheck.Result.healthy() :
            HealthCheck.Result.unhealthy("the application is unhealthy")
      }
    }
  }
  handlers {
    get("toggleHealth") { PropertyHolder propertyHolder -> ❺
      propertyHolder.healthy = !propertyHolder.healthy
      response.send()
    }
    get("health/:name?", new HealthCheckHandler()) ❻
  }
}
```

❶ Here we define the `PropertyHolder` class that will hold the `healthy` value the health check will test against.

❷   We bind the `PropertyHolder` into the registry, so that it is accessible throughout our application.

❸   Here we use the `HealthCheck#of` factory to bind a new instance of a health check. Health checks are named, and in this case we call our check `property Health`. Health checks are given access to the registry for their processing, and must return a `Promise` of the `HealthCheck.Result` class.

❹   Inside of the `Promise#sync` facotry, we test the value of the `healthy` boolean on the `PropertyHolder` class. If it is true, we return a healthy value; if it is false, we return an unhealthy value with a message indicating the error.

❺   For the sake of demonstration, we can provide a simple endpoint for toggling the healthy state of the application. This will simply invert the boolean value of the `health` property on the `PropertyHolder` class.

❻   Finally, we must bind the `HealthCheckHandler`, and we do so to the `/health` endpoint. Note that the `name` path token is optionally bound; the `HealthCheck Handler` can test an individually named health check.

We can run this application and `curl` the `/health` endpoint to see its output. Example 13-8 shows the health check response.

Example 13-8. Output from the /health endpoint

```
$ curl -v localhost:5050/health
*    Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5050 (#0)
> GET /health HTTP/1.1
> Host: localhost:5050
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Cache-Control: no-cache, no-store, must-revalidate
< Pragma: no-cache
< Expires: 0
< content-type: application/octet-stream
< content-length: 24
< connection: keep-alive
<
* Connection #0 to host localhost left intact
propertyHealth : HEALTHY
```

As you can see, the application initializes with a healthy state, and that is reflected in the response details. Most notable is that the `HealthCheckHandler` returns a 200 sta-

tus code for healthy results. If we `curl` the `/toggleHealth` endpoint, then again check `/health`, we will see the health state change to *unhealthy*. Example 13-9 shows the output from these `curl` calls.

Example 13-9. Setting unhealthy state and checking /health endpoint

```
$ curl -v localhost:5050/toggleHealth
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5050 (#0)
> GET /toggleHealth HTTP/1.1
> Host: localhost:5050
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-length: 0
< connection: keep-alive
<
* Connection #0 to host localhost left intact

$ curl -v localhost:5050/health
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5050 (#0)
> GET /health HTTP/1.1
> Host: localhost:5050
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 503 Service Unavailable
< Cache-Control: no-cache, no-store, must-revalidate
< Pragma: no-cache
< Expires: 0
< content-type: application/octet-stream
< content-length: 57
< connection: keep-alive
<
* Connection #0 to host localhost left intact
propertyHealth : UNHEALTHY [the application is unhealthy]
```

As the output shows, after toggling the `health` property, our health checks now report as unhealthy. In the case of unhealthy results, the `HealthCheckHandler` returns a status code of 503. This is important because many application load balancers drive the decision as to whether an instance is healthy or not based on the status code. In these cases, generally anything besides a 2xx status code will cause the instance to be taken out of traffic rotation.

A more realistic scenario is one where your application health checks ensure its database connection is stable. In modern system architectures, however, it may also be true that your service has a dependency on another remote service. This is commonly

the case with microservice architectures. Let's illustrate this scenario in code by imaging health checks for both the database and a dependency on a remote service.

We will start by looking at the interface for our database. Example 13-10 shows the `UserService` interface, which we know makes calls to the database in its concrete implementation.

Example 13-10. The UserService database interface

```
package app.services

import app.model.User
import ratpack.exec.Promise
import ratpack.exec.Operation

interface UserService {

  Promise<User> getById(Long id)

  Promise<User> getByUsername(String username)

  Promise<List<User>> listUsers()

  Operation save(User user)

  Operation ping()
}
```

The most notable aspect to this service contract is the `ping` method. In its concrete implementation, this will perform a `SELECT 1;` call to the database (or similar vendor-specific query). It is important when building any production-grade application that the ability to query the database in this capacity exists.

Next, let's look at our remote service contract as shown in Example 13-11.

Example 13-11. The UserProfileService remote service interface

```
package app.services

import app.model.UserProfile
import ratpack.exec.Promise
import ratpack.exec.Operation

interface UserProfileService {

  Promise<UserProfile> getProfile(Long profileId)

  Operation ping()
}
```

Similar to the `UserService`, the `UserProfileService` has a `ping` method on it. This operation will also be leveraged by health checks to ensure that the remote service is accessible. This may be as simple as opening a connection to the remote endpoint, or as complex as actually querying the endpoint for some commonly known data. Your application's mileage will vary, but it is enough to say that your service contract should have this capability.

Given these service contracts, let's begin by implementing a health check against the `UserService`. Example 13-12 shows the code for this. Note that we now implement the health checks in their own classes, which implement the `ratpack.health.Health Check` interface.

Example 13-12. The health check for the UserService

```
package app.health

import app.services.UserService
import ratpack.exec.Promise
import ratpack.health.HealthCheck
import ratpack.registry.Registry

class UserServiceHealthCheck implements HealthCheck {
  final String name = "UserServiceCheck" ❶

  @Override
  Promise<HealthCheck.Result> check(Registry registry) throws Exception {
    def userService = registry.get(UserService)

    userService.ping().promise() ❷
      .map {
        HealthCheck.Result.healthy() ❸
      }
      .mapError { error ->
        HealthCheck.Result.unhealthy(error) ❹
      }
  }
}
```

❶ Here we set the `name` property on the health check. This field is required.

❷ We need to translate the `Operation` into a `Promise` so that we can return a `Promise<HealthCheck.Result>` from the `check` method.

❸ This is the happy path, where no errors have occurred. Here, we can safely return a healthy result.

❹ When catching errors, they can be passed simply to `unhealthy` if you wish only to have the mesage and stack trace filled in for the check diagnostic.

Next, we implement the health check for the `UserProfileService`. Let's say for the sake of argument that we know the service will throw a runtime `ConnectionError` when it is unable to communicate with the remote service. Furthermore, all validation errors will be instances of `RemoteServiceError`—that is to say, the connection to the remote system is able to be established, but the request from the `ping` was thought to be invalid. The former case we know to be fatal, since our service cannot communicate to the remote service, while the latter case may indicate a protocol problem between our two systems. In either event, we want the health check to be as comprehensive as possible, so we can discriminate on these exception types to provide a diagnostic message when the service ping has failed.

Example 13-13 shows our check for the `UserProfileService`.

Example 13-13. The health check for the UserProfileService

```
package app.health

import app.model.ConnectionError
import app.model.RemoteServiceError
import app.services.UserProfileService
import ratpack.exec.Promise
import ratpack.health.HealthCheck
import ratpack.registry.Registry

class UserProfileServiceHealthCheck implements HealthCheck {
  final String name = "UserProfileServiceCheck"  ❶

  @Override
  Promise<HealthCheck.Result> check(Registry registry)
      throws Exception {
    def userProfileService = registry.get(UserProfileService)

    userProfileService.ping().promise()
      .map {
        HealthCheck.Result.healthy()  ❷
      }
      .mapError { error ->
        def msg = "The was an error with the remote service"  ❸
        if (error instanceof ConnectionError) {
          msg = "unable to connect to remote service"  ❹
        } else if (error instanceof RemoteServiceError) {
          msg = "protocol failure when pinging remote service"  ❺
        }
        HealthCheck.Result.unhealthy(msg, error)  ❻
      }
```

```
        }
}
```

❶   Here we set the name for the health check.

❷   As before, this is the happy path, so we return the healthy result.

❸   When an error is captured, we set a default message here.

❹   Here we check if this was a ConnectionError, and if so change the message.

❺   If it was not a ConnectionError, then we check if it was a RemoteServiceError,
    and likewise set a helpful diagnostic message.

❻   Finally, we render back the unhealthy response.

By checking the exception type from the ping call, we can provide a helpful diagnos-
tic message when debugging what health check has caused the application to indicate
failure. This will allow you to more quickly deduce where the problem is, so you
know where to start looking for a solution.

We can test to make sure that our application is returning the proper responses for
our health checks by building a simple integration test that wires up the HealthCheck
Handler. The specification is shown in Example 13-14.

Example 13-14. The application health check integration specification

```groovy
package app

import app.health.UserProfileServiceHealthCheck
import app.health.UserServiceHealthCheck
import app.model.ConnectionError
import app.model.RemoteServiceError
import app.services.UserProfileService
import app.services.UserService
import ratpack.exec.Operation
import ratpack.groovy.test.embed.GroovyEmbeddedApp
import ratpack.health.HealthCheckHandler
import spock.lang.AutoCleanup
import spock.lang.Specification

class AppHealthIntegrationSpec extends Specification {

  def userProfileService = Mock(UserProfileService) ❶
  def userService = Mock(UserService) ❷
  def userServiceHealthCheck = new UserServiceHealthCheck() ❸
  def userProfileServiceHealthCheck = new UserProfileServiceHealthCheck() ❹
```

```
@AutoCleanup
@Delegate
GroovyEmbeddedApp app = GroovyEmbeddedApp.of { ❺
  registryOf { ❻
    add(userProfileService)
    add(userService)
    add(userServiceHealthCheck)
    add(userProfileServiceHealthCheck)
  }
  handlers {
    get("health", new HealthCheckHandler()) ❼
  }
}

void "should return healthy checks"() {
  when:
  def response = httpClient.get("health")

  then:
  1 * userProfileService.ping() >> Operation.noop()
  1 * userService.ping() >> Operation.noop()
  response.statusCode == 200
}

void "should fail when userService is offline"() {
  when:
  def response = httpClient.get("health")

  then:
  1 * userProfileService.ping() >> Operation.noop()
  1 * userService.ping() >> Operation.of { throw new RuntimeException("fail") }
  response.statusCode == 503
  response.body.text.contains("${userServiceHealthCheck.name} : UNHEALTHY")
  response.body.text.contains("${userProfileServiceHealthCheck.name} : HEALTHY")
}

void "should fail when userProfileService is offline"() {
  when:
  def response = httpClient.get("health")

  then:
  1 * userProfileService.ping() >> Operation.of {
    throw new RuntimeException("fail")
  }
  1 * userService.ping() >> Operation.noop()
  response.statusCode == 503
  response.body.text.contains("${userServiceHealthCheck.name} : HEALTHY")
  response.body.text.contains("${userProfileServiceHealthCheck.name} : UNHEALTHY")
}

void "should indicate proper failure type for userProfileService"() {
  when:
```

```
    def response = httpClient.get("health")

    then:
    1 * userService.ping() >> Operation.noop()
    1 * userProfileService.ping() >> Operation.of {
      throw error
    }
    response.body.text.
      contains("${userProfileServiceHealthCheck.name} : UNHEALTHY " +
        "[${message}]")

    where:
    error                   | message
    new ConnectionError()    | "unable to connect to remote service"
    new RemoteServiceError() | "protocol failure when pinging remote service"
    new RuntimeException()   | "The was an error with the remote service"
  }
}
```

❶ Here we create a mock of the `UserProfileService`. We do not need to use a real concrete implementation here, since we are simply validating the interactions.

❷ Similarly, here we create a mock of the `UserService`.

❸ We will use real instances of our health checks, so here we create an instance of the `UserServiceHealthCheck`.

❹ Likewise, we create an instance of the `UserProfileServiceHealthCheck`.

❺ Here, we use the `GroovyEmbeddedApp` test fixture to create an embedded application for our test. We use the `@Delegate` annotation to make our feature tests more concise. Also, we apply the `@AutoCleanup` annotation to ensure the embedded application is closed properly.

❻ Within our embedded application's registry, we add the `userProfileService`, the `userService`, and our health checks.

❼ The embedded application has only one handler: the `HealthCheckHandler`, which we attach to the `/health` endpoint.

Each of the subsequent feature tests are described testing their specific function. Between these feature tests, we are able to see how our application health checks will operate under varying conditions.

Health checks play a critical role in any production application, and Ratpack's acknowledgment of their importance further highlights its nature as a production-quality web framework. The ease of building health checks into your application

ensures that you will always be servicing your users, and know when and where errors occur.

Building Distributions

Another consideration before going to production is how you are going to distribute your application. Ratpack's ease of use and lightweight nature to make it a great fit for building microservices, and as such your project's distribution should be set up in a way that is amenable to working in that kind of environment. Generally speaking, modern application deployments for micro- and other lightweight services lend themselves nicely to the paradigm of immutable infrastructure.

Earlier in the book, we discussed strategies for packaging your application into runnable JAR files, and the means by which Ratpack's Gradle plugin integrates with the Gradle application plugin to provide standalone system distributions. When designing your project for a production deployment, you will need to be conscientious of the environment within which your application will run. To that effect, many major players in the microservice ecosystem have found benefit in providing application distributions in the form of operating system packages. Packaging your project distribution as an OS package is generally only applicable to those running in a Linux-based environment, and if that suits your deployment needs, you can find some benefit in the community tools that are available as such.

Projects such as Netflix's Nebula Plugins provide many integrations with the Gradle build system to support building production-grade deployments. Within this toolset, you will find the `gradle-ospackage-plugin`, which provides an easy-to-use DSL to your Gradle build script for creating RPM (for YUM-based Linux distributions) and DEB (for Debian-based Linux distributions) packages from your project. It is left as an exercise for the reader to explore the capabilities of these projects, but as a non-comprehensive demonstration, consider the Gradle build script depicted in Example 13-15. This build script shows the application of the `os-package` Gradle plugin, and the `ospackage` and `buildDeb` extensions, which specify what should go into the resulting artifact.

Example 13-15. Gradle build script with Nebula OS package plugin

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.3.3'
    classpath 'com.netflix.nebula:gradle-ospackage-plugin:2.2.6'
  }
}
```

```groovy
apply plugin: 'io.ratpack.ratpack-groovy'
apply plugin: 'os-package'

repositories {
  jcenter()
}

ospackage {
  packageName = "myapp" ❶
  release '3'
  into "/opt/myapp" ❷
  from "${project.buildDir}/install/${project.applicationName}" ❸
}

buildDeb {
  dependsOn installDist ❹
  //preInstall file('scripts/preInstall.sh') ❺
  //postInstall file('scripts/postInstall.sh')
}
```

❶   Within the `ospackage` extension, we can set the `packageName` property to a friendly name for our application. This will affect the name of the resulting artifact.

❷   The `ospackage` extension extends from Gradle's `CopySpec`, so here we are specifying where on the deployment system we want our project's distribution to be installed.

❸   Here we inform the plugin as to where our project's distribution files will be.

❹   Within the `buildDeb` block we specify that running this task depends on the `installDist` build task, which will build the distribution and put it into the location specified in the last step.

❺   This and the next line are commented out, but it is shown here to exemplify that you can specify shell scripts that should be run at pre- and post-installation phases of the installation of our project on the target system. Using this, you can place *preInstall.sh* and *postInstall.sh* scripts into a *scripts* directory in the root of your project and they will be packaged into your OS package.

If you use this build script and run the `./gradlew buildDeb` task, your project will be built, the distribution created, and a resulting OS package artifact generated. From there, you can publish the artifact to a repository for installation, or you can manually copy the artifact to a target server and install it.

The capability of OS packaging is outlined here to show you how your project can be integrated with other tools from the ecosystem to demonstrate the possibilities available to you with Ratpack and Gradle. Indeed, your mileage will certainly vary, and the requirements of your infrastructure will drive the manner in which you produce your production deployments.

Production Checklist

A final consideration in your journey toward production is building a simple checklist to ensure that your project is ready to be sent to production. As you ready your application for deployment, take a moment to think through on your project's requirements, and ensure that you have been diligent in getting your application ready for the wild. As you do, some of the following thoughts may serve as valuable reminders:

- Am I sure that none of my code will block a request-taking thread?
- Are my service APIs designed to use `Promise`, `Publisher`, or `Observable` types?
- Are the features I am deploying well tested?
- Are the configuration files and directives that will be needed for production known, set up, and available?
- Is my project's distribution verified on my target production server?
- Have I made sure that there are no secrets contained within my distribution?
- Does my application have proper metric publishing so that I have the insight I need following deployment?
- Are my users' passwords and sensitive data properly encrypted and secured?

You may find it valuable to add or subtract from this list as your software development experience with Ratpack grows, but these should serve as a good starting place for you. If you can comfortably walk away from these considerations, then it is clear that you are ready for your application to go to production!

Chapter Summary

Going to production is the final step of building web applications with Ratpack. Over the course of this book, you have journeyed from an introduction on the basic concepts of Ratpack, to an understanding of the motivation for the framework, to coverage of its support for testing and validating your code, all the way through to an indepth understanding and practice of its numerous framework features. By now, you can certainly be considered a Ratpack expert, and your applications are ready for prime-time production! Congratulations and happy Ratpacking!

Index

About the Author

Dan Woods is a full-stack developer who specializes in distributed and web architectures. He is a member of the Ratpack core team and contributes to many open source projects in the broader Java and Groovy ecosystem. Dan's professional emphasis is on building continuous delivery and cloud infrastructure automation tooling.

Colophon

The animals on the cover of *Learning Ratpack* are ring-tailed lemurs (*Lemur catta*). Like all lemurs, ring-tails are endemic to the island of Madagascar and favor forested areas where trees are plentiful. Most thickly forested areas in Madagascar have been cleared for livestock, however, so the ring-tailed lemur is listed as endangered due to habitat loss and falling wild population numbers.

Ring-tailed lemurs are highly social and live in female-dominated groups of 30 or more individuals. They are very vocal and use a wide range of calls to communicate. Males also use scent to mark territory and challenge each other—they participate in "stink fights" by covering their tails in scent and waving them at opponents.

As opportunistic omnivores, ring-tailed lemurs have a varied diet that includes fruits, leaves, flowers, bark, sap, spiders, small birds, and even chameleons. If available, the fruits and leaves of the tamarind tree are the most sought-after meal, but lemurs are willing to eat a wide variety of things depending on the season.

Actor and comedian John Cleese has a passion for lemurs and was the host of a 1998 BBC documentary called *In the Wild: Operation Lemur with John Cleese*. The show tracked the progress of lemurs being reintroduced back into the Betampona Reserve in Madagascar, a project that was partially funded by Cleese himself.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Cassell's *Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.