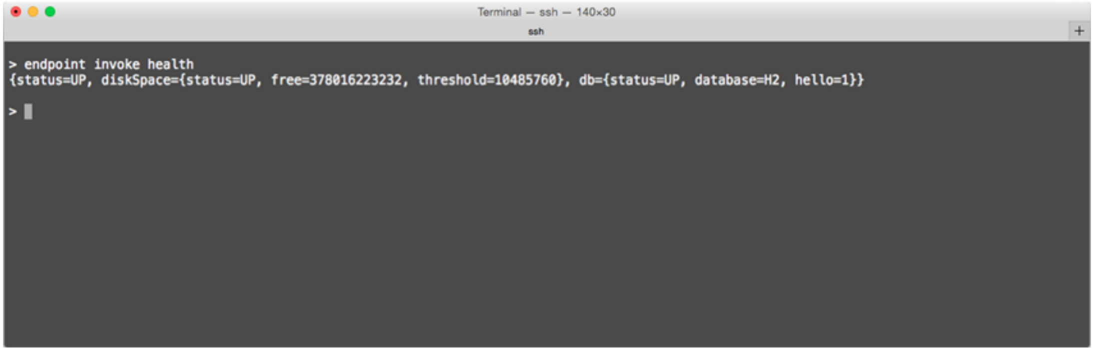

Figure 7.5 Invoking the health endpoint

a pipe and pretty-prints it. And you can always cut and paste it into a tool of your choosing for further review or formatting.

7.3 Monitoring your application with JMX

In addition to the endpoints and the remote shell, the Actuator also exposes its endpoints as MBeans to be viewed and managed through JMX (Java Management Extensions). JMX is an attractive option for managing your Spring Boot application, especially if you're already using JMX to manage other MBeans in your applications.

All of the Actuator's endpoints are exposed under the org.springframework.boot domain. For example, suppose you want to view the request mappings for your application. Figure 7.6 shows the request mapping endpoint as viewed in JConsole.

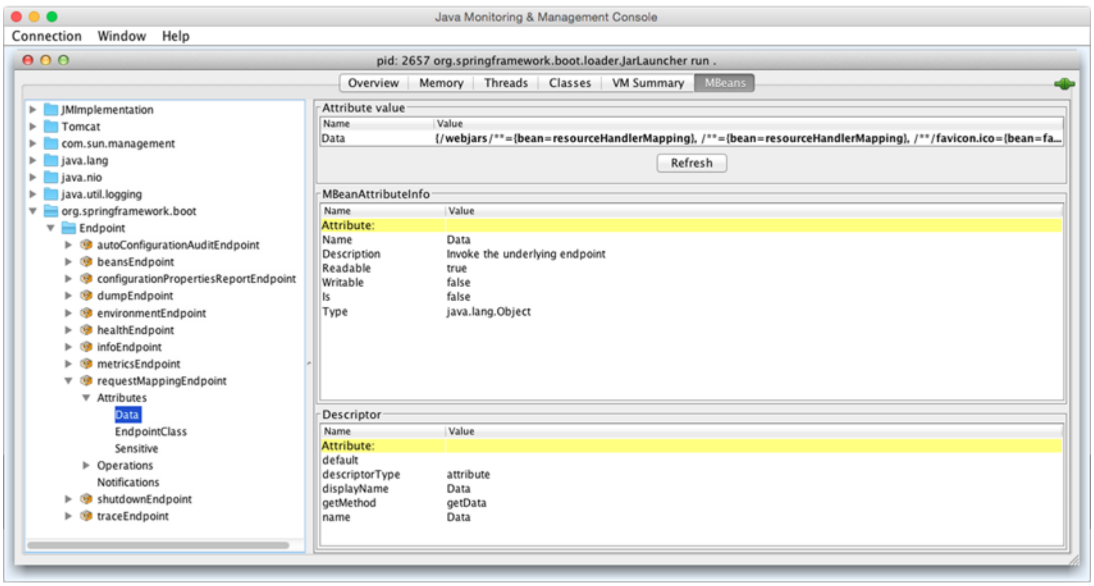

Figure 7.6 Request mapping endpoint as viewed in JConsole

As you can see, the request mapping endpoint is found under `requestMappingEndpoint`, which is under `Endpoint` in the org.springframework.boot domain. The `Data` attribute contains the JSON reported by the endpoint.

As with any MBean, the endpoint MBeans have operations that you can invoke. Most of the endpoint MBeans only have accessor operations that return the value of one of their attributes. But the shutdown endpoint offers a slightly more interesting (and destructive!) operation, as shown in figure 7.7

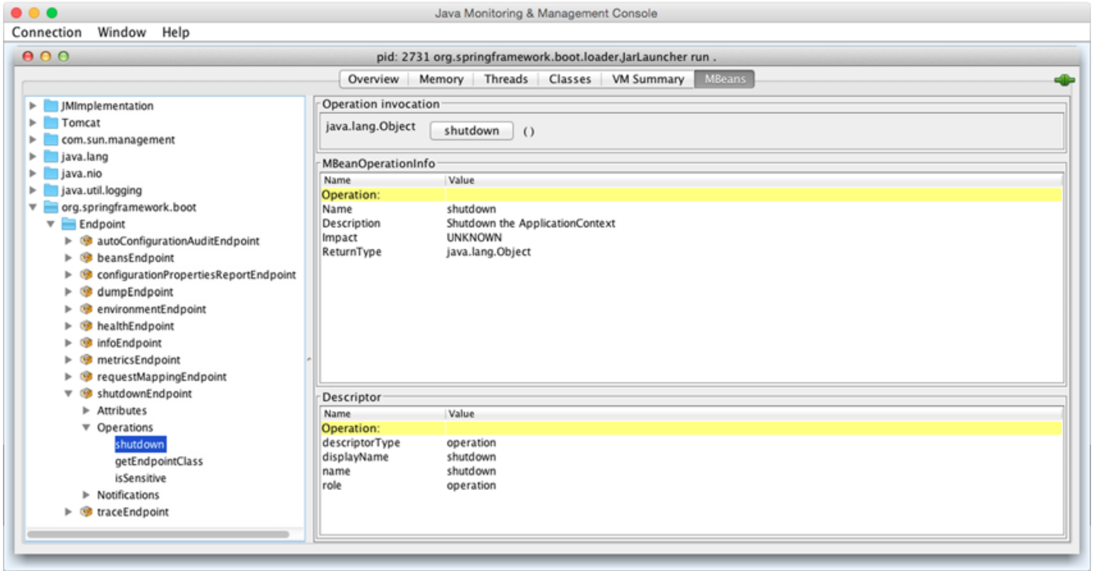

Figure 7.7 Shutdown button invokes the endpoint.

If you ever need to shut down your application (or just like living dangerously), the shutdown endpoint is there for you. As shown in figure 7.7, it's waiting for you to click the "shutdown" button to invoke the endpoint. Be careful, though—there's no turning back or "Are you sure?" prompt. The very next thing you'll see is shown in figure 7.8.

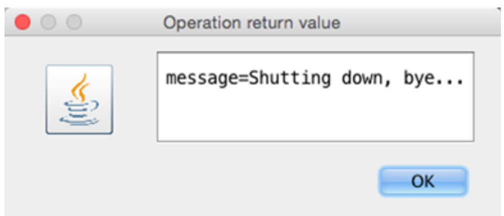

Figure 7.8 Application immediately shuts down.

After that, your application will have been shut down. And because it's dead, there's no way it could possibly expose another MBean operation for restarting it. You'll have to restart it yourself, the same way you started it in the first place.

7.4 Customizing the Actuator

Although the Actuator offers a great deal of insight into the inner workings of a running Spring Boot application, it may not be a perfect fit for your needs. Maybe you don't need everything it offers and want to disable some of it. Or maybe you need to extend it with metrics custom-suited to your application.

As it turns out, the Actuator can be customized in several ways, including the following:

- Renaming endpoints
- Enabling and disabling endpoints
- Defining custom metrics and gauges
- Creating a custom repository for storing trace data
- Plugging in custom health indicators

We're going to see how to customize the Actuator, bending it to meet our needs. We'll start with one of the simplest customizations: renaming the Actuator's endpoints.

7.4.1 Changing endpoint IDs

Each of the Actuator endpoints has an ID that's used to determine that endpoint's path. For example, the `/beans` endpoint has `beans` as its default ID.

If an endpoint's path is determined by its ID, then it stands to reason that you can change an endpoint's path by changing its ID. All you need to do is set a property whose name is `endpoints.`*`endpoint-id`*`.id`.

To demonstrate how this works, consider the `/shutdown` endpoint. It responds to `POST` requests sent to `/shutdown`. Suppose, however, that you'd rather have it handle `POST` requests sent to `/kill`. The following YAML shows how you might assign a new ID, and therefore a new path, to the `/shutdown` endpoint:

```
endpoints:
  shutdown:
    id: kill
```

There are a couple of reasons you might want to rename an endpoint and change its path. The most obvious is that you might simply want to name the endpoints to match the terminology used by your team. But you might also think that renaming an endpoint will hide it from anyone who might be familiar with the default names, thus creating a sense of security by obscurity.

Unfortunately, renaming an endpoint doesn't really secure it. At best, it will only slow down a hacker looking to gain access to an endpoint. We'll look at how you can secure Actuator endpoints in section 7.5. For now, let's see how to completely disable any (or all) endpoints that you don't want anyone to have access to.

7.4.2 Enabling and disabling endpoints

Although all of the Actuator endpoints are useful, you may not want or need all of them. By default, all of the endpoints (except for `/shutdown`) are enabled. We've already seen how to enable the `/shutdown` endpoint by setting `endpoints.shutdown.enabled` to `true` (in section 7.1.1). In the same way, you can disable any of the other endpoints by setting `endpoints._endpoint-id.enabled` to `false`.

For example, suppose you want to disable the `/metrics` endpoint. All you need to do is set the `endpoints.metrics.enabled` property to `false`. In application.yml, that would look like this:

```
endpoints:
  metrics:
    enabled: false
```

If you find that you only want to leave one or two of the endpoints enabled, it might be easier to disable them all and then opt in to the ones you want to enable. For example, consider the following snippet from application.yml:

```
endpoints:
  enabled: false
  metrics:
    enabled: true
```

As shown here, all of the Actuator's endpoints are disabled by setting `endpoints.enabled` to `false`. Then the `/metrics` endpoint is re-enabled by setting `endpoints.metrics.enabled` to `true`.

7.4.3 Adding custom metrics and gauges

In section 7.1.2, you saw how to use the `/metrics` endpoint to fetch information about the internal metrics of a running application, including memory, garbage collection, and thread metrics. Although these are certainly useful and informative metrics, you may want to define custom metrics to capture information specific to your application.

Suppose, for instance, that we want a metric that reports how many times a user has saved a book to their reading list. The easiest way to capture this number is to increment a counter every time the `addToReadingList()` method is called on `ReadingListController`. A counter is simple enough to implement, but how would you expose the running total along with the other metrics exposed by the `/metrics` endpoint?

Let's also suppose that we want to capture a timestamp for the last time a book was saved. We could easily capture that by calling `System.currentTimeMillis()`, but how could we report that time in the `/metrics` endpoint?

As it turns out, the auto-configuration that enables the Actuator also creates an instance of `CounterService` and registers it as a bean in the Spring application context. `CounterService` is an interface that defines three methods for incrementing, decrementing, or resetting a named metric, as shown here:

```
package org.springframework.boot.actuate.metrics;

public interface CounterService {
  void increment(String metricName);
  void decrement(String metricName);
  void reset(String metricName);
}
```

Actuator auto-configuration will also configure a bean of type `GaugeService`, an interface similar to `CounterService` that lets you record a single value to a named gauge metric. `GaugeService` looks like this:

```
package org.springframework.boot.actuate.metrics;

public interface GaugeService {
  void submit(String metricName, double value);
}
```

We don't need to implement either of these interfaces; Spring Boot already provides implementations for them both. All we must do is inject the `CounterService` and `GaugeService` instances into any other bean where they're needed, and call the methods to update whichever metrics we want.

For the metrics we want, we'll need to inject the `CounterService` and `GaugeService` beans into `ReadingListController` and call their methods from the `addToReadingList()` method. Listing 7.9 shows the necessary changes to `ReadingListController`.

Listing 7.9 Using injected gauge and counter services

```
@Controller
@RequestMapping("/")
@ConfigurationProperties("amazon")
public class ReadingListController {

  ...

  private CounterService counterService;

  @Autowired
  public ReadingListController(
      ReadingListRepository readingListRepository,
      AmazonProperties amazonProperties,
      CounterService counterService,
      GaugeService gaugeService) {                  <-- Inject the counter and gauge services
    this.readingListRepository = readingListRepository;
    this.amazonProperties = amazonProperties;
    this.counterService = counterService;
    this.gaugeService = gaugeService;
```

```
    }

    ...

    @RequestMapping(method=RequestMethod.POST)
    public String addToReadingList(Reader reader, Book book) {
      book.setReader(reader);
      readingListRepository.save(book);

      counterService.increment("books.saved");

      gaugeService.submit(
              "books.last.saved", System.currentTimeMillis());
      return "redirect:/";
    }

}
```

Increment "books.saved"

Record "books.last.saved"

This change to `ReadingListController` uses autowiring to inject the `CounterService` and `GaugeService` beans via the controller's constructor, which then stores them in instance variables. Then, each time that the `addToReadingList()` method handles a request, it will call `counterService.increment("books.saved")` and `gaugeService.submit("books.last.saved")` to adjust our custom metrics.

Although `CounterService` and `GaugeService` are simple to use, there are some metrics that are hard to capture by incrementing a counter or recording a gauge value. For those cases, we can implement the `PublicMetrics` interface and provide as many custom metrics as we want. The `PublicMetrics` interface defines a single `metrics()` method that returns a collection of `Metric` objects:

```
package org.springframework.boot.actuate.endpoint;

public interface PublicMetrics {
  Collection<Metric<?>> metrics();
}
```

To put `PublicMetrics` to work, suppose that we want to be able to report some metrics from the Spring application context. The time when the application context was started and the number of beans and bean definitions might be interesting metrics to include. And, just for grins, let's also report the number of beans that are annotated as `@Controller`. Listing 7.10 shows the implementation of `PublicMetrics` that will do the job.

Listing 7.10 Publishing custom metrics

```
package readinglist;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.endpoint.PublicMetrics;
import org.springframework.boot.actuate.metrics.Metric;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;

@Component
public class ApplicationContextMetrics implements PublicMetrics {

  private ApplicationContext context;

  @Autowired
  public ApplicationContextMetrics(ApplicationContext context) {
    this.context = context;
  }

  @Override
  public Collection<Metric<?>> metrics() {
    List<Metric<?>> metrics = new ArrayList<Metric<?>>();
    metrics.add(new Metric<Long>("spring.context.startup-date",
        context.getStartupDate()));

    metrics.add(new Metric<Integer>("spring.beans.definitions",
        context.getBeanDefinitionCount()));

    metrics.add(new Metric<Integer>("spring.beans",
        context.getBeanNamesForType(Object.class).length));

    metrics.add(new Metric<Integer>("spring.controllers",
        context.getBeanNamesForAnnotation(Controller.class).length));

    return metrics;
  }

}
```

Record startup date

Record bean definition count

Record bean count

Record controller bean count

The `metrics()` method will be called by the Actuator to get any custom metrics that `ApplicationContextMetrics` provides. It makes a handful of calls to methods on the injected `ApplicationContext` to fetch the numbers we want to report as metrics. For each one, it creates an instance of `Metric`, specifying the metric's name and the value, and adds the `Metric` to the list to be returned.

As a consequence of creating `ApplicationContextMetrics` as well as using `CounterService` and `GaugeService` in `ReadingListController`, we get the following entries in the response from the `/metrics` endpoint:

```
{
  ...
  spring.context.startup-date: 1429398980443,
  spring.beans.definitions: 261,
  spring.beans: 272,
  spring.controllers: 2,
  books.count: 1,
  gauge.books.save.time: 1429399793260,
  ...
}
```

Of course, the actual values for these metrics will vary, depending on how many books you've added and the times when you started the application and last saved a book. In case you're wondering, `spring.controllers` is 2 because it's counting `ReadingListController` as well as the Spring Boot–provided `BasicErrorController`.

7.4.4 Creating a custom trace repository

By default, the traces reported by the `/trace` endpoint are stored in an in-memory repository that's capped at 100 entries. Once it's full, it starts rolling off older trace entries to make room for new ones. This is fine for development purposes, but in a production application the higher traffic may result in traces being discarded before you ever get a chance to see them.

One way to remedy that problem is to declare your own `InMemoryTraceRepository` bean and set its capacity to some value higher than 100. The following configuration class should increase the capacity to 1000 entries:

```
package readinglist;
import org.springframework.boot.actuate.trace.InMemoryTraceRepository;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ActuatorConfig {

  @Bean
  public InMemoryTraceRepository traceRepository() {
    InMemoryTraceRepository traceRepo = new InMemoryTraceRepository();
    traceRepo.setCapacity(1000);
    return traceRepo;
  }

}
```

Although a tenfold increase in the repository's capacity should keep a few of those trace entries around a bit longer, a sufficiently busy application might still discard traces before you get a chance to review them. And because this is an in-memory trace repository, we should be careful about increasing the capacity too much, as it will have an impact on our application's memory footprint.

Alternatively, we could store those trace entries elsewhere—somewhere that's not consuming memory and that will be more permanent. All we need to do is implement Spring Boot's `TraceRepository` interface:

```
package org.springframework.boot.actuate.trace;
import java.util.List;
import java.util.Map;

public interface TraceRepository {
  List<Trace> findAll();
  void add(Map<String, Object> traceInfo);
}
```

As you can see, `TraceRepository` only requires that we implement two methods: one that finds all stored `Trace` objects and another that saves a `Trace` given a `Map` containing trace information.

For demonstration purposes, perhaps we could create an instance of `TraceRepository` that stores trace entries in a MongoDB database. Listing 7.11 shows such an implementation of `TraceRepository`.

Listing 7.11 Saving trace data to Mongo

```
package readinglist;
import java.util.Date;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.trace.Trace;
import org.springframework.boot.actuate.trace.TraceRepository;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.stereotype.Service;

@Service
public class MongoTraceRepository implements TraceRepository {

  private MongoOperations mongoOps;

  @Autowired
  public MongoTraceRepository(MongoOperations mongoOps) {   <-- Inject MongoOperations
    this.mongoOps = mongoOps;
  }

  @Override
  public List<Trace> findAll() {
    return mongoOps.findAll(Trace.class);   <-- Fetch all trace entries
  }

  @Override
  public void add(Map<String, Object> traceInfo) {
    mongoOps.save(new Trace(new Date(), traceInfo));   <-- Save a trace entry
  }

}
```

The `findAll()` method is straightforward enough, asking the injected `MongoOperations` to find all `Trace` objects. The `add()` method is only slightly more interesting, instantiating a `Trace` object given the current date/time and the `Map` of trace info before saving it via `MongoOperations.save()`. The only question you might have is where `MongoOperations` comes from.

In order for `MongoTraceRepository` to work, we're going to need to make sure that we have a `MongoOperations` bean in the Spring application context. Thanks to Spring Boot starters and auto-configuration, that's simply a matter of adding the MongoDB starter as a dependency. The Gradle dependency you need is as follows:

```
compile("org.springframework.boot:spring-boot-starter-data-mongodb")
```

If your project is built with Maven, this is the dependency you'll need:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

By adding this starter, Spring Data MongoDB and supporting libraries will be added to the application's classpath. And because those are in the classpath, Spring Boot will auto-configure the beans necessary to support working with a MongoDB database, including a `MongoOperations` bean. The only other thing you'll need to do is be sure that there's a MongoDB server running for `MongoOperations` to talk to.

7.4.5 Plugging in custom health indicators

As we've seen, the Actuator comes with a nice set of out-of-the-box health indicators for common needs such as reporting the health of a database or message broker that the application is using. But what if your application interacts with some system for which there's no health indicator?

Because our application includes links to Amazon for books in the reading list, it might be interesting to report whether or not Amazon is reachable. Sure, it's not likely that Amazon will go down, but stranger things have happened. So let's create a health indicator that reports whether Amazon is available. Listing 7.12 shows a `HealthIndicator` implementation that should do the job.

Listing 7.12 Defining a custom Amazon health indicator

```
package readinglist;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class AmazonHealth implements HealthIndicator {

  @Override
  public Health health() {

    try {
      RestTemplate rest = new RestTemplate();
      rest.getForObject("http://www.amazon.com", String.class);     Send request to Amazon
      return Health.up().build();
    } catch (Exception e) {
      return Health.down().build();     Report "down" health
    }
  }

}
```

The AmazonHealth class isn't terribly fancy. The health() method simply uses Spring's RestTemplate to perform a GET request to Amazon's home page. If it works, it returns a Health object indicating that Amazon is "UP". On the other hand, if an exception is thrown while requesting Amazon's home page, then health() returns a Health object indicating that Amazon is "DOWN".

The following excerpt from the /health endpoint's response shows what you might see if Amazon is unreachable:

```
{
    "amazonHealth": {
        "status": "DOWN"
    },
    ...
}
```

You wouldn't believe how long I had to wait for Amazon to crash so that I could get that result![1]

If you'd like to add additional information to the health record beyond a simple status, you can do so by calling withDetail() on the Health builder. For example, to add the exception's message as a reason field in the health record, the catch block could be changed to return a Health object created like this:

```
return Health.down().withDetail("reason", e.getMessage()).build();
```

As a result of this change, the health record might look like this when the request to Amazon fails:

```
"amazonHealth": {
    "reason": "I/O error on GET request for
               \"http://www.amazon.com\":www.amazon.com;
               nested exception is java.net.UnknownHostException:
               www.amazon.com",
    "status": "DOWN"
},
```

You can add as many additional details as you want by calling withDetail() for each additional field you want included in the health record.

7.5 Securing Actuator endpoints

We've seen that many of the Actuator endpoints expose information that may be considered sensitive. And some, such as the /shutdown endpoint, are dangerous and can be used to bring your application down. Therefore, it's very important to be able to secure these endpoints so that they're only available to authorized clients.

As it turns out, the Actuator endpoints can be secured the same way as any other URL path: with Spring Security. In a Spring Boot application, this means adding the

[1] Not really. I just disconnected my computer from the network. No network, no Amazon.

Security starter as a build dependency and letting security auto-configuration take care of locking down the application, including the Actuator endpoints.

In chapter 3, we saw how the default security auto-configuration results in all URL paths being secured, requiring HTTP Basic authentication where the username is "user" and the password is randomly generated at startup and written to the log file. This was not how we wanted to secure the application, and it's likely not how you want to secure the Actuator either.

We've already added some custom security configuration to restrict the root URL path (/) to only authenticated users with READER access. To lock down Actuator endpoints, we'll need to make a few changes to the `configure()` method in `SecurityConfig.java`.

Suppose, for instance, that we want to lock down the `/shutdown` endpoint, requiring that the user have ADMIN access. Listing 7.13 shows the changes required in the `configure()` method.

Listing 7.13 Securing the `/shutdown` endpoint

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/").access("hasRole('READER')")
      .antMatchers("/shutdown").access("hasRole('ADMIN')")    <- Require ADMIN access
      .antMatchers("/**").permitAll()
    .and()
    .formLogin()
      .loginPage("/login")
      .failureUrl("/login?error=true");
}
```

Now the only way to access the `/shutdown` endpoint is to authenticate as a user with ADMIN access.

The custom `UserDetailsService` we created in chapter 3, however, is coded to only apply READER access to users it looks up via the `ReaderRepository`. Therefore, you may want to create a smarter `UserDetailsService` implementation that is able to apply ADMIN access to some users. Optionally, you can configure an additional authentication implementation, such as the in-memory authentication shown in listing 7.14.

Listing 7.14 Adding an in-memory admin authentication user

```
@Override
protected void configure(
            AuthenticationManagerBuilder auth) throws Exception {
  auth
    .userDetailsService(new UserDetailsService() {    <- Reader authentication
      @Override
      public UserDetails loadUserByUsername(String username)
```

```
            throws UsernameNotFoundException {
          UserDetails user = readerRepository.findOne(username);
          if (user != null) {
            return user;
          }
          throw new UsernameNotFoundException(
                        "User '" + username + "' not found.");
        }
      })
      .and()
      .inMemoryAuthentication()
        .withUser("admin").password("s3cr3t")
                          .roles("ADMIN", "READER");          <-- Admin authentication
}
```

With the in-memory authentication added, you can authenticate with "admin" as the username and "s3cr3t" as the password and be granted both ADMIN and READER access.

Now the `/shutdown` endpoint is locked down for everyone except users with ADMIN access. But what about the Actuator's other endpoints? Assuming you want to lock them down with ADMIN access as for `/shutdown`, you can list each of them in the call to `antMatchers()`. For example, to lock down `/metrics` and `/configprops` as well as `/shutdown`, call `antMatchers()` like this:

```
.antMatchers("/shutdown", "/metrics", "/configprops")
                          .access("hasRole('ADMIN')")
```

Although this approach will work, it's only suitable if you want to secure a small subset of the Actuator endpoints. It becomes unwieldy if you use it to lock down all of the Actuator's endpoints.

Rather than explicitly list all of the Actuator endpoints when calling `antMatchers()`, it's much easier to use wildcards to match them all with a simple Ant-style expression. This is challenging, however, because there's not a lot in common between the endpoint paths. And we can't apply ADMIN access to "/**" because then everything except for the root path (/) would require ADMIN access.

Instead, consider setting the endpoint's context path by setting the `management.context-path` property. By default, this property is empty, which is why all of the Actuator's endpoint paths are relative to the root path. But the following entry in application.yaml will prefix them all with /mgmt.

```
management:
  context-path: /mgmt
```

Optionally, you can set it in application.properties like this:

```
management.context-path=/mgmt
```

With `management.context-path` set to /mgmt, all Actuator endpoints will be relative to the /mgmt path. For example, the `/metrics` endpoint will be at /mgmt/metrics.

With this new path, we now have a common prefix to work with when assigning ADMIN access restriction to the Actuator endpoints:

```
.antMatchers("/mgmt/**").access("hasRole('ADMIN')")
```

Now all requests beginning with /mgmt, which includes all Actuator endpoints, will require an authenticated user who has been granted ADMIN access.

7.6 Summary

It can be difficult to know for sure what's going on inside a running application. Spring Boot's Actuator opens a portal into the inner workings of a Spring Boot application, exposing components, metrics, and gauges to help understand what makes the application tick.

In this chapter, we started by looking at the Actuator's web endpoints—REST endpoints that expose runtime details over HTTP. These include endpoints for viewing all of the beans in the Spring application context, auto-configuration decisions, Spring MVC mappings, thread activity, application health, and various metrics, gauges, and counters.

In addition to web endpoints, the Actuator also offers two alternative ways to dig into the information it provides. The remote shell offers a way to securely shell into the application itself and issue commands that expose much of the same data as the Actuator's endpoints. Meanwhile, all of the Actuator's endpoints are exposed as MBeans that can be monitored and managed by a JMX client.

Next, we took a look at how to customize the Actuator. We saw how to change Actuator endpoint paths by changing the endpoint IDs as well as how to enable and disable endpoints. We also plugged in a few custom metrics and created a custom trace repository to replace the default in-memory trace repository.

Finally, we looked at how to secure the Actuator's endpoints, restricting access to authorized users.

Coming up in the next chapter, we'll see how to take an application from the coding phase to production, looking at how Spring Boot helps when deploying an application to a variety of platforms, including traditional application servers and the cloud.

8 Deploying Spring Boot applications

This chapter covers

- Deploying WAR files
- Database migration
- Deploying to the cloud

Think of your favorite action movie. Now imagine going to see that movie in the theater and being taken on a thrilling audio-visual ride with high-speed chases, explosions, and battles, only to have it come to a sudden end just before the good guys take down the bad guys. Instead of seeing the movie's conflict resolved, the theater lights come on and everyone is ushered out the door.

Although the lead-up was exciting, it's the climax of the movie that's important. Without it, it's action for action's sake.

Now imagine developing applications and putting a lot of effort and creativity into solving the business problem, but then never deploying the application for others to use and enjoy. Sure, most applications we write don't involve car chases or explosions (at least I hope not), but there's a certain rush we get along the way. Of course, not every line of code we write is destined for production, but it'd be a big letdown if none of it ever was deployed.

Up to this point we've been focused on using features of Spring Boot that help us develop an application. There have been some exciting steps along the way. But it's all for nothing if we don't cross the finish line and deploy the application.

In this chapter we're going to step beyond developing applications with Spring Boot and look at how to deploy those applications. Although this may seem obvious for anyone who has ever deployed a Java-based application, there are some unique features of Spring Boot and related Spring projects we can draw on that make deploying Spring Boot applications unique.

In fact, unlike most Java web applications, which are typically deployed to an application server as WAR files, Spring Boot offers several deployment options. Before we look at how to deploy a Spring Boot application, let's consider all of the options and choose a few that suit our needs best.

8.1 Weighing deployment options

There are several ways to build and run Spring Boot applications. You've already seen a few of them:

- Running the application in the IDE (either Spring ToolSuite or IntelliJ IDEA)
- Running from the command line using the Maven `spring-boot:run` goal or Gradle `bootRun` task
- Using Maven or Gradle to produce an executable JAR file that can be run at the command line
- Using the Spring Boot CLI to run Groovy scripts at the command line
- Using the Spring Boot CLI to produce an executable JAR file that can be run at the command line

Any of these choices is suitable for running the application while you're still developing it. But what about when you're ready to deploy the application into a production or other non-development environment?

Although none of the choices listed seems fitting for deploying an application beyond development, the truth is that all but one of them is a valid choice. Running an application within the IDE is certainly ill-suited for a production deployment. Executable JAR files and the Spring Boot CLI, however, are still on the table and are great choices when deploying to a cloud environment.

That said, you're probably wondering how to deploy a Spring Boot application to a more traditional application server environment such as Tomcat, WebSphere, or WebLogic. In those cases, executable JAR files and Groovy source code won't work. For application server deployment, you'll need your application wrapped up in a WAR file.

As it turns out, Spring Boot applications can be packaged for deployment in several ways, as described in table 8.1.

Table 8.1 Spring Boot deployment choices

Deployment artifact	Produced by	Target environment
Raw Groovy source	Written by hand	Cloud Foundry and container deployment, such as with Docker
Executable JAR	Maven, Gradle, or Spring Boot CLI	Cloud environments, including Cloud Foundry and Heroku, as well as container deployment, such as with Docker
WAR	Maven or Gradle	Java application servers or cloud environments such as Cloud Foundry

As you can see in table 8.1, your target environment will need to be a factor in your choice. If you're deploying to a Tomcat server running in your own data center, then the choice of a WAR file has been made for you. On the other hand, if you'll be deploying to Cloud Foundry, you're welcome to choose any of the deployment options shown.

In this chapter, we're going to focus our attention on the following options:

- Deploying a WAR file to a Java application server
- Deploying an executable JAR file to Cloud Foundry
- Deploying an executable JAR file to Heroku (where the build is performed by Heroku)

As we explore these scenarios, we're also going to have to deal with the fact that we've been using an embedded H2 database as we've developed the application, and we'll look at ways to replace it with a production-ready database.

To get started, let's take a look at how we can build our reading-list application into a WAR file that can be deployed to a Java application server such as Tomcat, WebSphere, or WebLogic.

8.2 Deploying to an application server

Thus far, every time we've run the reading-list application, the web application has been served from a Tomcat server embedded in the application. Compared to a conventional Java web application, the tables were turned. The application has not been deployed in Tomcat; rather, Tomcat has been deployed in the application.

Thanks in large part to Spring Boot auto-configuration, we've not been required to create a web.xml file or servlet initializer class to declare Spring's `DispatcherServlet` for Spring MVC. But if we're going to deploy the application to a Java application server, we're going to need to build a WAR file. And so that the application server will know how to run the application, we'll also need to include a servlet initializer in that WAR file.

8.2.1 Building a WAR file

As it turns out, building a WAR file isn't that difficult. If you're using Gradle to build the application, you simply must apply the "war" plugin:

```
apply plugin: 'war'
```

Then, replace the existing `jar` configuration with the following `war` configuration in build.gradle:

```
war {
    baseName = 'readinglist'
    version = '0.0.1-SNAPSHOT'
}
```

The only difference between this `war` configuration and the previous `jar` configuration is the change of the letter *j* to *w*.

If you're using Maven to build the project, then it's even easier to get a WAR file. All you need to do is change the `<packaging>` element's value from `jar` to `war`.

```
<packaging>war</packaging>
```

Those are the only changes required to produce a WAR file. But that WAR file will be useless unless it includes a web.xml file or a servlet initializer to enable Spring MVC's `DispatcherServlet`.

Spring Boot can help here. It provides `SpringBootServletInitializer`, a special Spring Boot-aware implementation of Spring's `WebApplicationInitializer`. Aside from configuring Spring's `DispatcherServlet`, `SpringBootServletInitializer` also looks for any beans in the Spring application context that are of type `Filter`, `Servlet`, or `ServletContextInitializer` and binds them to the servlet container.

To use `SpringBootServletInitializer`, simply create a subclass and override the `configure()` method to specify the Spring configuration class. Listing 8.1 shows `ReadingListServletInitializer`, a subclass of `SpringBootServletInitializer` that we'll use for the reading-list application.

Listing 8.1 Extending `SpringBootServletInitializer` for the reading-list application

```
package readinglist;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class ReadingListServletInitializer
       extends SpringBootServletInitializer {

  @Override
  protected SpringApplicationBuilder configure(
                                    SpringApplicationBuilder builder) {
    return builder.sources(Application.class);
  }

}
```

Specify Spring configuration

As you can see, the `configure()` method is given a `SpringApplicationBuilder` as a parameter and returns it as a result. In between, it calls the `sources()` method to register any Spring configuration classes. In this case, it only registers the `Application` class, which, as you'll recall, served dual purpose as both a bootstrap class (with a `main()` method) and a Spring configuration class.

Even though the reading-list application has other Spring configuration classes, it's not necessary to register them all with the `sources()` method. The `Application` class is annotated with `@SpringBootApplication`, which implicitly enables component-scanning. Component-scanning will discover and pull in any other configuration classes that it finds.

Now we're ready to build the application. If you're using Gradle to build the project, simply invoke the `build` task:

```
$ gradle build
```

Assuming no problems, the build will produce a file named readinglist-0.0.1-SNAPSHOT.war in build/libs.

For a Maven-based build, use the `package` goal:

```
$ mvn package
```

After a successful Maven build, the WAR file will be found in the "target" directory.

All that's left is to deploy the application. The deployment procedure varies across application servers, so consult the documentation for your application server's specific deployment procedure.

For Tomcat, you can deploy an application by copying the WAR file into Tomcat's webapps directory. If Tomcat is running (or once it starts up if it isn't currently running), it will detect the presence of the WAR file, expand it, and install it.

Assuming that you didn't rename the WAR file before deploying it, the servlet context path will be the same as the base name of the WAR file, or /readinglist-0.0.1-SNAPSHOT in the case of the reading-list application. Point your browser at http://*server*:_port_/readinglist-0.0.1-SNAPSHOT to kick the tires on the app.

One other thing worth noting: even though we're building a WAR file, it may still be possible to run it without deploying to an application server. Assuming you don't remove the `main()` method from `Application`, the WAR file produced by the build can also be run as if it were an executable JAR file:

```
$ java -jar readinglist-0.0.1-SNAPSHOT.war
```

In effect, you get two deployment options out of a single deployment artifact!

At this point, the application should be up and running in Tomcat. But it's still using the embedded H2 database. An embedded database was handy while developing the application, but it's not a great choice in production. Let's see how to wire in a different data source when deploying to production.

8.2.2 Creating a production profile

Thanks to auto-configuration, we have a `DataSource` bean that references an embedded H2 database. More specifically, the `DataSource` bean is a data source pool, typically org.apache.tomcat.jdbc.pool.DataSource. Therefore, it may seem obvious that in order to use some database other than the embedded H2 database, we simply need to declare our own DataSource bean, overriding the auto-configured DataSource, to reference a production database of our choosing.

For example, suppose that we wanted to work with a PostgreSQL database running on localhost with the name “readingList”. The following @Bean method would declare our DataSource bean:

```
@Bean
@Profile("production")
public DataSource dataSource() {
  DataSource ds = new DataSource();
  ds.setDriverClassName("org.postgresql.Driver");
  ds.setUrl("jdbc:postgresql://localhost:5432/readinglist");
  ds.setUsername("habuma");
  ds.setPassword("password");
  return ds;
}
```

Here the DataSource type is Tomcat’s org.apache.tomcat.jdbc.pool.DataSource, not to be confused with javax.sql.DataSource, which it ultimately implements. The details required to connect to the database (including the JDBC driver class name, the database URL, and the database credentials) are given to the DataSource instance. With this bean declared, the default auto-configured DataSource bean will be passed over.

The key thing to notice about this @Bean method is that it is also annotated with @Profile to specify that it should only be created if the “production” profile is active. Because of this, we can still use the embedded H2 database while developing the application, but use the PostgreSQL database in production by activating the profile.

Although that should do the trick, there’s a better way to configure the database details without explicitly declaring our own DataSource bean. Rather than replace the auto-configured DataSource bean, we can configure it via properties in application.yml or application.properties. Table 8.2 lists all of the properties that are useful for configuring the DataSource bean.

Table 8.2 DataSource configuration properties

Property (prefixed with spring.datasource.)	Description
name	The name of the data source
initialize	Whether or not to populate using data.sql (default: true)
schema	The name of a schema (DDL) script resource
data	The name of a data (DML) script resource
sql-script-encoding	The character set for reading SQL scripts

Table 8.2 `DataSource` **configuration properties *(continued)***

Property (prefixed with `spring.datasource.`)	Description
`platform`	The platform to use when reading the schema resource (for example, "schema-{platform}.sql")
`continue-on-error`	Whether or not to continue if initialization fails (default: `false`)
`separator`	The separator in the SQL scripts (default: ;)
`driver-class-name`	The fully qualified class name of the JDBC driver (can often be automatically inferred from the URL)
`url`	The database URL
`username`	The database username
`password`	The database password
`jndi-name`	A JNDI name for looking up a datasource via JNDI
`max-active`	Maximum active connections (default: 100)
`max-idle`	Maximum idle connections (default: 8)
`min-idle`	Minimum idle connections (default: 8)
`initial-size`	The initial size of the connection pool (default: 10)
`validation-query`	A query to execute to verify the connection
`test-on-borrow`	Whether or not to test a connection as it's borrowed from the pool (default: `false`)
`test-on-return`	Whether or not to test a connection as it's returned to the pool (default: `false`)
`test-while-idle`	Whether or not to test a connection while it is idle (default: `false`)
`time-between-eviction-runs-millis`	How often (in milliseconds) to evict connections (default: `5000`)
`min-evictable-idle-time-millis`	The minimum time (in milliseconds) that a connection can be idle before being tested for eviction (default: `60000`)
`max-wait`	The maximum time (in milliseconds) that the pool will wait when no connections are available before failing (default: `30000`)
`jmx-enabled`	Whether or not the data source is managed by JMX (default: `false`)

Most of the properties in table 8.2 are for fine-tuning the connection pool. I'll leave it to you to tinker with those settings as you see fit. What we're interested in now, however, is setting a few properties that will point the `DataSource` bean at PostgreSQL instead of the embedded H2 database. Specifically, the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties are what we need.

As I'm writing this, I have a PostgreSQL database running locally, listening on port 5432, with a username and password of "habuma" and "password". Therefore, the following "production" profile in application.yml is what I used:

```
---
spring:
  profiles: production
  datasource:
    url: jdbc:postgresql://localhost:5432/readinglist
    username: habuma
    password: password
  jpa:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
```

Notice that this excerpt starts with `---` and the first property set is `spring.profiles`. This indicates that the properties that follow will only be applied if the "production" profile is active.

Next, the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties are set. Note that it's usually unnecessary to set the `spring.datasource.driver-class-name` property, as Spring Boot can infer it from the value of the `spring.datasource.url` property. I also had to set some JPA properties. The `spring.jpa.database-platform` property sets the underlying JPA engine to use Hibernate's PostgreSQL dialect.

To enable this profile, we'll need to set the `spring.profiles.active` property to "production". There are several ways to set this property, but the most convenient way is by setting a system environment variable on the machine running the application server. To enable the "production" profile before starting Tomcat, I exported the `SPRING_PROFILES_ACTIVE` environment variable like this:

```
$ export SPRING_PROFILES_ACTIVE=production
```

You probably noticed that `SPRING_PROFILES_ACTIVE` is different from `spring.profiles.active`. It's not possible to export an environment variable with periods in the name, so it was necessary to alter the name slightly. From Spring's point of view, the two names are equivalent.

We're almost ready to deploy the application to an application server and see it run. In fact, if you are feeling adventurous, go ahead and try it. You'll run into a small problem, however.

By default, Spring Boot configures Hibernate to create the schema automatically when using the embedded H2 database. More specifically, it sets Hibernate's `hibernate.hbm2ddl.auto` to `create-drop`, indicating that the schema should be created when Hibernate's `SessionFactory` is created and dropped when it is closed.

But it's set to do nothing if you're not using an embedded H2 database. This means that our application's tables won't exist and you'll see errors as it tries to query those nonexistent tables.

8.2.3 Enabling database migration

One option is to set the `hibernate.hbm2ddl.auto` property to `create`, `create-drop`, or `update` via Spring Boot's `spring.jpa.hibernate.ddl-auto` property. For instance, to set `hibernate.hbm2ddl.auto` to `create-drop` we could add the following lines to application.yml:

```
spring:
  jpa:
    hibernate:
      ddl-auto: create-drop
```

This, however, is not ideal for production, as the database schema would be wiped clean and rebuilt from scratch any time the application was restarted. It may be tempting to set it to `update`, but even that isn't recommended in production.

Alternatively, we could define the schema in schema.sql. This would work fine the first time, but every time we started the application thereafter, the initialization scripts would fail because the tables in question would already exist. This would force us to take special care in writing our initialization scripts to not attempt to repeat any work that has already been done.

A better choice is to use a database migration library. Database migration libraries work from a set of database scripts and keep careful track of the ones that have already been applied so that they won't be applied more than once. By including the scripts within each deployment of the application, the database is able to evolve in concert with the application.

Spring Boot includes auto-configuration support for two popular database migration libraries:

- Flyway (http://flywaydb.org)
- Liquibase (www.liquibase.org)

All you need to do to use either of these database migration libraries with Spring Boot is to include them as dependencies in the build and write the scripts. Let's see how they work, starting with Flyway.

DEFINING DATABASE MIGRATION WITH FLYWAY

Flyway is a very simple, open source database migration library that uses SQL for defining the migration scripts. The idea is that each script is given a version number, and Flyway will execute each of them in order to arrive at the desired state of the database. It also records the status of scripts it has executed so that it won't run them again.

For the reading-list application, we're starting with an empty database with no tables or data. Therefore, the script we'll need to get started will need to create the Reader and Book tables, including any foreign-key constraints and initial data. Listing 8.2 shows the Flyway script we'll need to go from an empty database to one that our application can use.

Listing 8.2 A database initialization script for Flyway

```
create table Reader (                       <- Create Reader table
  id serial primary key,
  username varchar(25) unique not null,
  password varchar(25) not null,
  fullname varchar(50) not null
);

create table Book (    <- Create Book table
  id serial primary key,
  author varchar(50) not null,
  description varchar(1000) not null,
  isbn varchar(10) not null,
  title varchar(250) not null,
  reader_username varchar(25) not null,
  foreign key (reader_username) references Reader(username)
);

create sequence hibernate_sequence;    <- Define a sequence

insert into Reader (username, password, fullname)    <- An initial Reader
            values ('craig', 'password', 'Craig Walls');
```

As you can see, the Flyway script is just SQL. What makes it work with Flyway is where it's placed in the classpath and how it's named. Flyway scripts follow a naming convention that includes the version number, as illustrated in figure 8.1.

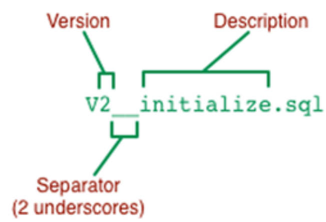

Figure 8.1 Flyway scripts are named with their version number.

All Flyway scripts have names that start with a capital V which precedes the script's version number. That's followed by two underscores and a description of the script. Because this is the first script in the migration, it will be version 1. The description given can be flexible and is primarily to provide some understanding of the script's purpose. Later, should we need to add a new table to the database or a new column to an existing table, we can create another script named with 2 in the version place.

Flyway scripts need to be placed in the path /db/migration relative to the application's classpath root. Therefore, this script needs to be placed in src/main/resources/db/migration within the project.

You'll also need to tell Hibernate to not attempt to create the tables by setting `spring.jpa.hibernate.ddl-auto` to `none`. The following lines in application.yml take care of that:

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
```

All that's left is to add Flyway as a dependency in the project build. Here's the dependency that's required for Gradle:

```
compile("org.flywaydb:flyway-core")
```

In a Maven build, the `<dependency>` is as follows:

```
<dependency>
  <groupId>org.flywayfb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

When the application is deployed and running, Spring Boot will detect Flyway in the classpath and auto-configure the beans necessary to enable it. Flyway will step through any scripts in /db/migration and apply them if they haven't already been applied. As each script is executed, an entry will be written to a table named schema_version. The next time the application starts, Flyway will see that those scripts have been recorded in schema_version and skip over them.

DEFINING DATABASE MIGRATION WITH LIQUIBASE

Flyway is simple to use, especially with help from Spring Boot auto-configuration. But defining migration scripts with SQL is a two-edged sword. Although it's easy and natural to work with SQL, you run the risk of defining a migration script that works with one database platform but not another.

Rather than be limited to platform-specific SQL, Liquibase supports several formats for writing migration scripts that are agnostic to the underlying platform. These include XML, YAML, and JSON. And, if you really want it, Liquibase also supports SQL scripts.

The first step to using Liquibase with Spring Boot is to add it as a dependency in your build. The Gradle dependency is as follows:

```
compile("org.liquibase:liquibase-core")
```

For Maven, here's the `<dependency>` you'll need:

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
</dependency>
```

Spring Boot auto-configuration takes it from there, wiring up the beans that support Liquibase. By default, those beans are wired to look for all of the migration scripts in a single file named db.changelog-master.yaml in /db/changelog (relative to the classpath root). The migration script in listing 8.3 includes instructions to initialize the database for the reading-list application.

Listing 8.3 A Liquibase script for initializing the reading-list database

```
databaseChangeLog:
  - changeSet:
      id: 1                          <- Changeset ID
      author: habuma
      changes:
        - createTable:
            tableName: reader        <- Create reader table
            columns:
              - column:
                  name: username
                  type: varchar(25)
                  constraints:
                    unique: true
                    nullable: false
              - column:
                  name: password
                  type: varchar(25)
                  constraints:
                    nullable: false
              - column:
                  name: fullname
                  type: varchar(50)
                  constraints:
                    nullable: false
        - createTable:
            tableName: book          <- Create book table
            columns:
              - column:
                  name: id
                  type: bigserial
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: author
                  type: varchar(50)
                  constraints:
                    nullable: false
              - column:
                  name: description
                  type: varchar(1000)
                  constraints:
                    nullable: false
              - column:
                  name: isbn
                  type: varchar(10)
                  constraints:
                    nullable: false
```

```
            - column:
                name: title
                type: varchar(250)
                constraints:
                  nullable: false
            - column:
                name: reader_username
                type: varchar(25)
                constraints:
                  nullable: false
                  references: reader(username)
                  foreignKeyName: fk_reader_username
      - createSequence:                                    <-- Define a
          sequenceName: hibernate_sequence                     sequence
      - insert:
         tableName: reader                    <-- Insert an initial
         columns:                                 reader
           - column:
              name: username
              value: craig
           - column:
              name: password
              value: password
           - column:
              name: fullname
              value: Craig Walls
```

As you can see, the YAML format is a bit more verbose than the equivalent Flyway SQL script. But it's still fairly clear as to its purpose and it isn't coupled to any specific database platform.

Unlike Flyway, which has multiple scripts, one for each change set, Liquibase changesets are all collected in the same file. Note the `id` property on the line following the `changeset` command. Future changes to the database can be included by adding a new `changeset` with a different `id`. Also note that the `id` property isn't necessarily numeric and may contain any text you'd like.

When the application starts up, Liquibase will read the changeset instructions in db.changelog-master.yaml, compare them with what it may have previously written to the databaseChangeLog table, and apply any changesets that have not yet been applied.

Although the example given here is expressed in YAML format, you're welcome to choose one of Liquibase's other supported formats, such as XML or JSON. Simply set the `liquibase.change-log` property (in application.properties or application.yml) to reflect the file you want Liquibase to load. For example, to use an XML changeset file, set `liquibase.change-log` like this:

```
liquibase:
  change-log: classpath:/db/changelog/db.changelog-master.xml
```

Spring Boot auto-configuration makes both Liquibase and Flyway a piece of cake to work with. But there's a lot more to what each of these database migration libraries can do beyond what we've seen here. I encourage you to refer to each project's documentation for more details.

We've seen how building Spring Boot applications for deployment into a conventional Java application server is largely a matter of creating a subclass of `SpringBootServletInitializer` and adjusting the build specification to produce a WAR file instead of a JAR file. But as we'll see next, Spring Boot applications are even easier to build for the cloud.

8.3 Pushing to the cloud

Server hardware can be expensive to purchase and maintain. Properly scaling servers to handle heavy loads can be tricky and even prohibitive for some organizations. These days, deploying applications to the cloud is a compelling and cost-effective alternative to running your own data center.

There are several cloud choices available, but those that offer a platform as a service (PaaS) are among the most compelling. PaaS offers a ready-made application deployment platform with several add-on services (such as databases and message brokers) to bind to your applications. In addition, as your application requires additional horsepower, cloud platforms make it easy to scale up (or down) your application on the fly by adding and removing instances.

Now that we've deployed the reading-list application to a traditional application server, we're going to try deploying it to the cloud. Specifically, we're going to deploy our application to two of the most popular PaaS platforms available: Cloud Foundry and Heroku.

8.3.1 Deploying to Cloud Foundry

Cloud Foundry is a PaaS platform from Pivotal, the same company that sponsors the Spring Framework and the other libraries in the Spring platform. One of the most compelling things about Cloud Foundry is that it is both open source and has several commercial distributions, giving you the choice of how and where you use Cloud Foundry. It can even be run inside the firewall in a corporate datacenter, offering a private cloud.

For the reading-list application, we're going to deploy to Pivotal Web Services (PWS), a public Cloud Foundry hosted by Pivotal at http://run.pivotal.io. If you want to work with PWS, you'll need to sign up for an account. PWS offers a 60-day free trial and doesn't even require you to give any credit card information during the trial.

Once you've signed up for PWS, you'll need to download and install the `cf` command-line tool from https://console.run.pivotal.io/tools. You'll use the `cf` tool to push applications to Cloud Foundry. But the first thing you'll use it for is to log into your PWS account:

```
$ cf login -a https://api.run.pivotal.io
API endpoint: https://api.run.pivotal.io

Email> {your email}

Password> {your password}
Authenticating...
OK
```

Now we're ready to take the reading-list application to the cloud. As it turns out, our reading-list project is already ready to be deployed to Cloud Foundry. All we need to do is use the `cf push` command:

```
$ cf push sbia-readinglist -p build/libs/readinglist.war
```

The first argument to `cf push` is the name given to the application in Cloud Foundry. Among other things, this name will be used as the subdomain that the application will be hosted at. In this case, the full URL for the application will be http://sbia-readinglist.cfapps.io. Therefore, it's important that the name you give the application be unique so that it doesn't collide with any other application deployed in Cloud Foundry (including those deployed by other Cloud Foundry users).

Because dreaming up a unique name may be tricky, the `cf push` command offers a `--random-route` option that will randomly produce a subdomain for you. Here's how to push the reading-list application so that a random route is generated:

```
$ cf push sbia-readinglist -p build/libs/readinglist.war --random-route
```

When using `--random-route`, the application name is still required, but two randomly chosen words will be appended to it to produce the subdomain. (When I tried it, the resulting subdomain was sbia-readinglist-gastroenterological-stethoscope.)

> **NOT JUST WAR FILES** Although we're going to deploy the reading-list application as a WAR file, Cloud Foundry will be happy to deploy Spring Boot applications in any form they come in, including executable JAR files and even uncompiled Groovy scripts run via the Spring Boot CLI.

Assuming everything goes well, the application should be deployed and ready to handle requests. Supposing that the subdomain is sbia-readinglist, you can point your browser at http://sbia-readinglist.cfapps.io to see it in action. You should be prompted with the login page. As you'll recall, the database migration script inserted a user named "craig" with a password of "password". Use those to log into the application.

Go ahead and play around with the application and add a few books to the reading list. Everything should work. But something still isn't quite right. If you were to restart the application (using the `cf restart` command) and then log back into the application, you'd see that your reading list is empty. Any book you've added before restarting the application will be gone.

The reason the data doesn't survive an application restart is because we're still using the embedded H2 database. You can verify this by requesting the Actuator's `/health` endpoint, which will reply with something like this:


```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "free": 834236510208,
    "threshold": 10485760
  },
  "db": {
    "status": "UP",
    "database": "H2",
    "hello": 1
  }
}
```


Notice the value of the `db.database` property. It confirms any suspicion we might have had that the database is an embedded H2 database. We need to fix that.

As it turns out, Cloud Foundry offers a few database options to choose from in the form of marketplace services, including MySQL and PostgreSQL. Because we already have the PostgreSQL JDBC driver in our project, we'll use the PostgreSQL service from the marketplace, which is named "elephantsql".

The elephantsql service comes with a handful of different plans to choose from, ranging from small development-sized databases to large industrial-strength production databases. You can get a list of all of the elephantsql plans with the `cf marketplace` command like this:

```
$ cf marketplace -s elephantsql
Getting service plan information for service elephantsql as craig@habuma.com...
OK

service plan   description          free or paid
turtle         Tiny Turtle          free
panda          Pretty Panda         paid
hippo          Happy Hippo          paid
elephant       Enormous Elephant    paid
```

As you can see, the more serious production-sized database plans require payment. You're welcome to choose one of those plans if you want, but for now I'll assume that you're choosing the free "turtle" plan.

To create an instance of the database service, you can use the `cf create-service` command, specifying the service name, the plan name, and an instance name:

```
$ cf create-service elephantsql turtle readinglistdb
Creating service readinglistdb in org habuma /
      space development as craig@habuma.com...
OK
```

Once the service has been created, we'll need to bind it to our application with the `cf bind-service` command:

```
$ cf bind-service sbia-readinglist readinglistdb
```

Binding a service to an application merely provides the application with details on how to connect to the service within an environment variable named `VCAP_SERVICES`. It doesn't change the application to actually use that service.

We *could* rewrite the reading-list application to read `VCAP_SERVICES` and use the information it provides to access the database service, but that's completely unnecessary. Instead, all we need to do is restage the application with the `cf restage` command:

```
$ cf restage sbia-readinglist
```

The `cf restage` command forces Cloud Foundry to redeploy the application and reevaluate the `VCAP_SERVICES` value. As it does, it will see that our application declares a `DataSource` bean in the Spring application context and replaces it with a `DataSource` that references the bound database service. As a consequence, our application will now be using the PostgreSQL service known as elephantsql rather than the embedded H2 database.

Go ahead and try it out now. Log into the application, add a few books to the reading list, and then restart the application. Your books should still be in your reading list after the restart. That's because they were persisted to the bound database service rather than to an embedded H2 database. Once again, the Actuator's `/health` endpoint will back up that claim:

```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "free": 834331525120,
    "threshold": 10485760
  },
  "db": {
    "status": "UP",
    "database": "PostgreSQL",
    "hello": 1
  }
}
```

Cloud Foundry is a great PaaS for Spring Boot application deployment. Its association with the Spring projects affords some synergy between the two. But it's not the only PaaS where Spring Boot applications can be deployed. Let's see what needs to be done to deploy the reading-list application to Heroku, another popular PaaS platform.

8.3.2 Deploying to Heroku

Heroku takes a unique approach to application deployment. Rather than deploy a completely built deployment artifact, Heroku arranges a Git repository for your application and builds and deploys the application for you every time you push it to the repository.

If you've not already done so, you'll want to initialize the project directory as a Git repository:

```
$ git init
```

This will enable the Heroku command-line tool to add the remote Heroku Git repository to the project automatically.

Now it's time to set up the application in Heroku using the Heroku command-line tool's `apps:create` command:

```
$ heroku apps:create sbia-readinglist
```

Here I've asked Heroku to name the application "sbia-readinglist". This name will be used as the name of the Git repository as well as the subdomain of the application at herokuapps.com. You'll want to be sure to pick a unique name, as there can't be more than one application with the same name. Alternatively, you can leave off the name and Heroku will generate a unique name for you (such as "fierce-river-8120" or "serene-anchorage-6223").

The `apps:create` command creates a remote Git repository at https://git.heroku.com/sbia-readinglist.git and adds a remote reference to the repository named "heroku" in the local project's Git configuration. That will enable us to push our project into Heroku using the `git` command.

The project has been set up in Heroku, but we're not quite ready to push it yet. Heroku asks that you provide a file named `Procfile` that tells Heroku how to run the application after it has been built. For our reading-list application, we need to tell Heroku to run the WAR file produced by the build as an executable JAR file using the `java` command.[1] Assuming that the application will be built with Gradle, the following one-line `Procfile` is what we'll need:

```
web: java -Dserver.port=$PORT -jar build/libs/readinglist.war
```

On the other hand, if you're using Maven to build the project, then the path to the JAR file will be slightly different. Instead of referencing the executable WAR file in build/libs, Heroku will need to find it in the target directory, as shown in the following `Procfile`:

```
web: java -Dserver.port=$PORT -jar target/readinglist.war
```

---

1 The project we're working with actually produces an executable WAR file, but as far as Heroku knows, it's no different than an executable JAR file.

In either case, you'll also need to set the `server.port` property as shown so that the embedded Tomcat server starts up on the port that Heroku assigns to the application (provided by the `$PORT` variable).

We're almost ready to push the application to Heroku, but there's a small change required in the Gradle build specification. When Heroku tries to build our application, it will do so by executing a task named `stage`. Therefore, we'll need to add a `stage` task to `build.gradle`:

```
task stage(dependsOn: ['build']) {
}
```

As you can see, this `stage` task doesn't do much. But it does depend on the `build` task. Therefore, the `build` task will be triggered when Heroku tries to build the application with the `stage` task, and the resulting JAR will be ready to run in the build/libs directory.

You may also need to inform Heroku of the Java version we're building the application with so that it runs the application with the appropriate version of Java. The easiest way to do that is to create a file named system.properties at the root of the project that sets a `java.runtime.version` property:

```
java.runtime.version=1.7
```

Now we're ready to push the project into Heroku. As I said before, this is just a matter of pushing the code into the remote Git repository that Heroku set up for us:

```
$ git commit -am "Initial commit"
$ git push heroku master
```

After the code is pushed into Heroku, Heroku will build it using either Maven or Gradle (depending on which kind of build file it finds) and then run it using the instructions in `Procfile`. Once it's ready, you should be able to try it out by pointing your browser at http://{app name}.herokuapp.com, where "{app name}" is the name given to the application when you used `apps:create`. For example, I named the application "sbia-readinglist" when I deployed it, so the application's URL is http://sbia-readinglist.herokuapps.com.

Feel free to poke about in the application as much as you'd like. But then go take a look at the `/health` endpoint. The `db.database` property should tell you that the application is using the embedded H2 database. We should change that to use a PostgreSQL service instead.

We can create and bind to a PostgreSQL service using the Heroku command-line tool's `addons:add` command like this:

```
$ heroku addons:add heroku-postgresql:hobby-dev
```

Here we're asking for the addon service named `heroku-postgresql`, which is the PostgreSQL service offered by Heroku. We're also asking for the `hobby-dev` plan for that service, which is the free plan.

Now the PostgreSQL service is created and bound to our application, and Heroku will automatically restart the application to ensure that binding. But even so, if we were to go look at the /health endpoint, we'd see that the application is still using the embedded H2 database. That's because the auto-configuration for H2 is still in play, and there's nothing to tell Spring Boot to use PostgreSQL instead.

One option is to set the spring.datasource.* properties like we did when deploying to an application server. The information we'd need can be found on the database service's dashboard, which can be opened with the addons:open command:

```
$ heroku addons:open waking-carefully-3728
```

In this case, the name of the database instance is "waking-carefully-3728". This command will open a dashboard page in your web browser that includes all of the necessary connection information, including the hostname, database name, and credentials—everything we'd need to set the spring.datasource.* properties.

But there's an easier way. Rather than look up that information for ourselves and set those properties, why can't Spring look them up for us? In fact, that's what the Spring Cloud Connectors project does. It works with both Cloud Foundry and Heroku to look up any services bound to an application and automatically configure the application to use those services.

We just need to add Spring Cloud Connectors as a dependency in the build. For a Gradle build, add the following to build.gradle:

```
compile(
      "org.springframework.boot:spring-boot-starter-cloud-connectors")
```

If you're using Maven, the following <dependency> will add Spring Cloud Connectors to the build:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cloud-connectors</artifactId>
</dependency>
```

Spring Cloud Connectors will only work if the "cloud" profile is active. To activate the "cloud" profile in Heroku, use the config:set command:

```
$ heroku config:set SPRING_PROFILES_ACTIVE="cloud"
```

Now that the Spring Cloud Connectors dependency is in the build and the "cloud" profile is active, we're ready to push the application again:

```
$ git commit -am "Add cloud connector"
$ git push heroku master
```

After the application starts up, sign in to the application and view the /health endpoint. It should indicate that the application is connected to a PostgreSQL database:

```
"db": {
  "status": "UP",
  "database": "PostgreSQL",
  "hello": 1
}
```

Now our application is deployed in the cloud, ready to take requests from the world!

8.4 Summary

There are several options for deploying Spring Boot applications, including traditional application servers and PaaS options in the cloud. In this chapter, we looked at a few of those options, deploying the reading-list application as a WAR file to Tomcat and in the cloud to both Cloud Foundry and Heroku.

Spring Boot applications are often given a build specification that produces an executable JAR file. But we've seen how to tweak the build and write a `SpringBootServletInitializer` implementation to produce a WAR file suitable for deployment to an application server.

We then took a first step toward deploying our application to Cloud Foundry. Cloud Foundry is flexible enough to accept Spring Boot applications in any form, including executable JAR files, traditional WAR files, or even raw Spring Boot CLI Groovy scripts. We also saw how Cloud Foundry is able to automatically swap out our embedded data source bean with one that references a database service bound to the application.

Finally we saw how although Heroku doesn't offer automatic swapping of data source beans like Cloud Foundry, by adding the Spring Cloud Connectors library to our deployment we can achieve the same effect, enabling a bound database service instead of an embedded database.

Along the way, we also looked at how to enable database migration tools such as Flyway and Liquibase in Spring Boot. We used database migration to initialize our database on the first deployment and now are ready to evolve our database as needed on future deployments.

appendix A
Spring Boot Developer Tools

Spring Boot 1.3 introduced a new set of developer tools that make it even easier to work with Spring Boot at development time. Among its many capabilities are

- *Automatic restart*—Restarts a running application when files are changed in the classpath
- *LiveReload support*—Changes to resources trigger a browser refresh automatically
- *Remote development*—Supports automatic restart and LiveReload when deployed remotely
- *Development property defaults*—Provides sensible development defaults for some configuration properties

Spring Boot's developer tools come in the form of a library that can be added to a project as a dependency. If you're using Gradle to build your project, the development tools can be added with the following line in your build.gradle file:

```
compile "org.springframework.boot:spring-boot-devtools"
```

Or it can be added as a `<dependency>` in a Maven POM:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

The developer tools will be disabled when your application is running from a fully packaged JAR or WAR file, so it's unnecessary to remove this dependency before building a production deployment.

Automatic restart

With the developer tools active, any changes to files on the classpath will trigger an application restart. To make the restart as fast as possible, classes that won't change (such as those in third-party JAR files) will be loaded into a base classloader, whereas application code that is being worked on will be loaded into a separate restart classloader. When changes are detected, only the restart classloader is restarted.

There are some classpath resources that don't require an application restart when they change. View templates, such as Thymeleaf templates, can be edited on the fly without restarting the application. Static resources in /static or /public likewise don't require an application restart, so Spring Boot developer tools exclude the following paths from restart consideration: /META-INF/resources, /resources, /static, /public, /templates.

The default set of restart path exclusions can be overridden by setting the `spring.devtools.restart.exclude` property. For example, to only exclude /static and /templates, set `spring.devtools.restart.exclude` like this:

```
spring:
  devtools:
    restart:
      exclude: /static/**,/templates/**
```

On the other hand, if you'd rather disable automatic restart completely, you can set `spring.devtools.restart.enabled` to `false`:

```
spring:
  devtools:
    restart:
      enabled: false
```

Another option is to set a trigger file that must be changed in order for the restart to take place. For example, suppose you don't want a restart to happen unless a change is made to a file named .trigger. All you must do is set the `spring.devtools.restart.trigger-file` property like this:

```
spring:
  devtools:
    restart:
      trigger-file: .trigger
```

A trigger file is useful if your IDE continuously compiles changed files. Without a trigger file, every change would trigger a restart. With a trigger file, you can be sure that a restart doesn't happen unless you want it to (by making a change to the trigger file).

LiveReload

One of the most common rituals of web application development involves the following steps:

1. Make a change to rendered content (such as images, stylesheets, templates).
2. Click Refresh in the browser to see the results of the change.
3. Repeat starting at step 1.

Although it's not an arduous process, it would be nice if you could see the results of a change immediately, without clicking Refresh.

Spring Boot's developer tools integrate with LiveReload (http://livereload.com) to eliminate the Refresh step. When the developer tools are active, Spring Boot will start an embedded LiveReload server that can trigger a browser refresh whenever a resource is changed. All you need to do is install the LiveReload plugin into your web browser.

If you'd like to disable the embedded LiveReload server, you can do so by setting `spring.devtools.livereload.enabled` to `false`:

```
spring:
  devtools:
    livereload:
      enabled: false
```

Remote development

The automatic restart and LiveReload features of the developer tools are also optionally available when running the applications remotely (such as when deployed on a server or in a cloud environment). In addition, Spring Boot's developer tools enable remote debugging of Spring Boot applications.

In a typical deployment, you won't want the remote development feature enabled, as it will hinder performance. But in special cases, such as when developing an application that's deployed in a non-production environment set aside for development purposes, these tools can come in handy. This is especially useful if your application uses a cloud service that isn't available in your local development environment.

You must opt in to remote development by setting a remote secret:

```
spring:
  devtools:
    remote:
      secret: myappsecret
```

By setting this property, a server component is enabled in the running application to support remote development. This server will listen for requests asking it to accept incoming changes and will either restart the application or trigger a browser refresh.

In order to put this remote server to use, you'll need to run the remote development tools client locally. The remote client comes in the form of a class whose fully qualified name is `org.springframework.boot.devtools.RemoteSpringApplication`. It's designed to run in your IDE with an argument telling it where your remote application is deployed.

For example, suppose you're running the reading-list application remotely, deployed on Cloud Foundry at https://readinglist.cfapps.io. If you're using Eclipse or Spring ToolSuite, you can start the remote client with the following steps:

1. Select the Run > Run Configurations menu item.
2. Create a new Java Application launch configuration.
3. Select the Reading List project in the Project field (either by typing the project name or clicking the Browse button and finding it). See figure A.1.
4. Enter `org.springframework.boot.devtools.RemoteSpringApplication` into the Main Class field. See figure A.1.
5. On the Arguments tab, enter `https://readinglist.cfapps.io` into the Program Arguments field. See figure A.2.

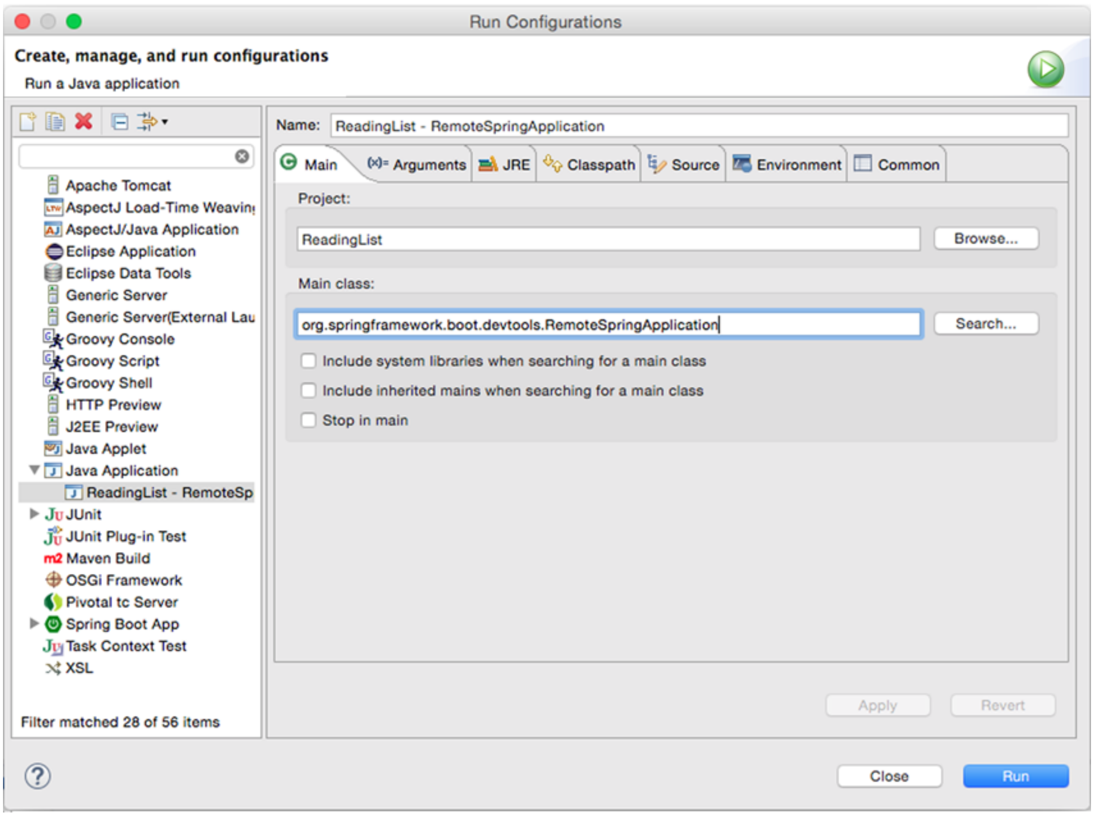

Figure A.1 `RemoteSpringApplication` **is the remote developer tools client.**

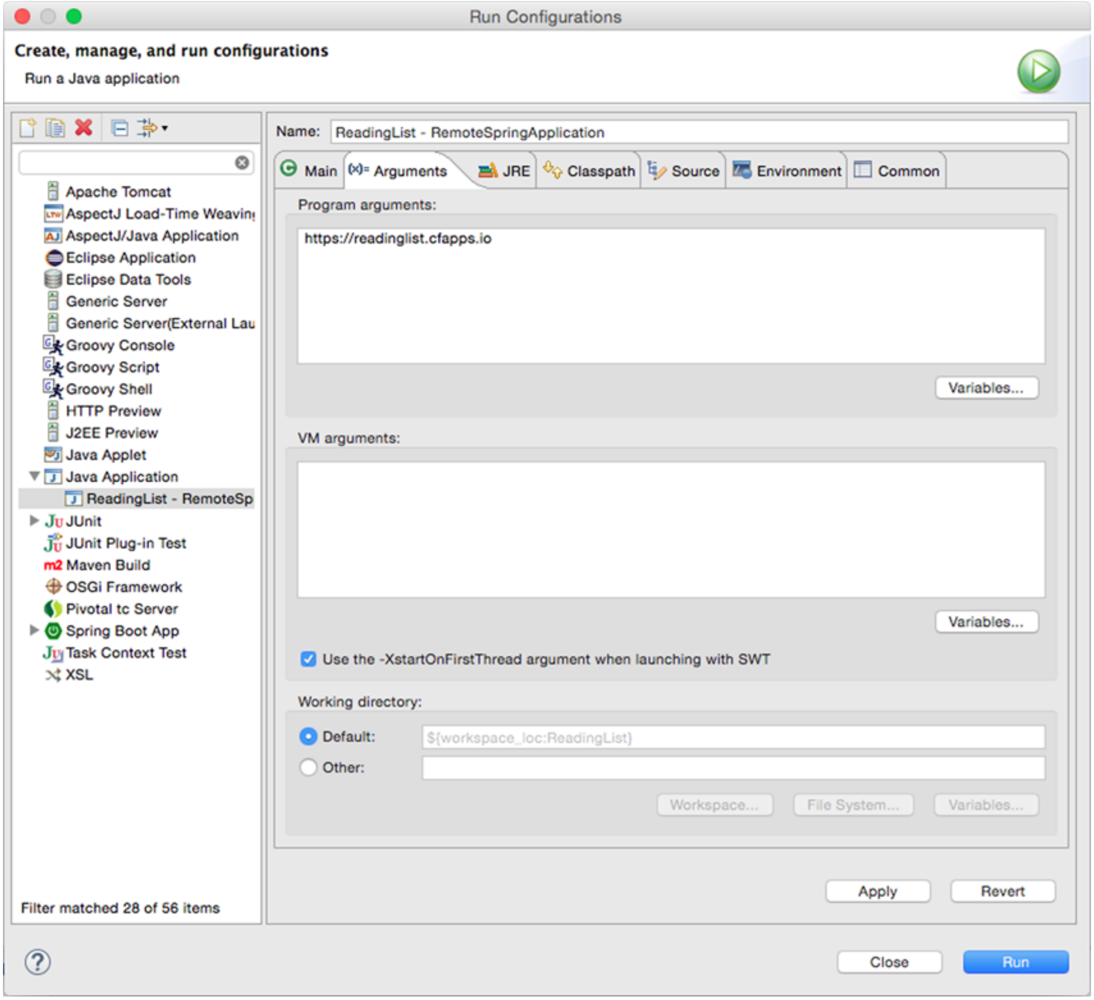

Figure A.2 **`RemoteSpringApplication` takes the remote app's URL as an argument.**

Once the client has started, you can start making changes to the application in your IDE. As changes are detected, they'll be pushed to the remote server and applied. If changes are made to a rendered web resource (such as a stylesheet or JavaScript), they'll also trigger a browser refresh using LiveReload.

The remote client will also enable tunneling of remote debug traffic over HTTP so that you can debug a remotely deployed application in your IDE. All you must do is ensure that the remote application has remote debugging enabled. This can usually be done by configuring `JAVA_OPTS`.

For example, if your application is deployed to Cloud Foundry, you can set `JAVA_OPTS` in your application's manifest.yml file like this:

```
---
  env:
    JAVA_OPTS: "-Xdebug -Xrunjdwp:server=y,transport=dt_socket,suspend=n"
```

Once the remote application is started and a connection is established with the local debug server, you should be able to set breakpoints and step through the code of the remote application much as if it were local (albeit a bit slower due to network latency).

Development property defaults

There are some configuration properties that are usually set at development time, but never in a production setting. View template caching, for instance, is best disabled during development so that you can see the results of any changes you make immediately. But in production, view template caching should be left enabled for better performance.

By default, Spring Boot will enable caching for any of the supported view template options (Thymeleaf, Freemarker, Velocity, Mustache, and Groovy templates). But if Spring Boot's developer tools are in play, that caching will be disabled.

Essentially what this means is that when the developer tools are active, the following properties are set to `false`:

- spring.thymeleaf.cache
- spring.freemarker.cache
- spring.velocity.cache
- spring.mustache.cache
- spring.groovy.template.cache

This saves you from having to disable them (likely in a development-profiled configuration) for development time.

Globally configuring developer tools

As you work with the developer tools, you'll probably find that you regularly use the same settings across multiple projects. For instance, if you use a restart trigger file, you're likely to name the trigger file consistently across projects. Rather than repeat developer tool configuration in each project, it may be more convenient to configure the developer tools globally.

To do this, create a file named .spring-boot-devtools.properties in your home directory. (Note that the name starts with a period.) In that file, set whatever developer tool properties you want to have applied across all of your projects.

For example, suppose that you want to set a trigger file named .trigger and disable LiveReload across all of your Spring Boot projects. To do that, you can create a .spring-boot-devtools.properties file with the following lines:

```
spring.devtools.restart.trigger-file=.trigger
spring.devtools.livereload.enabled=false
```

Then, should you want to override any of these properties, you can do so on a project-by-project basis by setting them in each project's application.properties or application.yml file.

appendix B
Spring Boot starters

Spring Boot starter dependencies greatly simplify the dependencies section of your project's build specification by aggregating commonly used dependencies under more coarse-grained dependencies. Your build will transitively resolve the dependencies that are declared in the starter dependency.

Not only do starter dependencies keep the dependencies section of the build smaller, they are typically organized by the type of functionality they bring to an application. For example, rather than specify specific libraries required for validation (such as Hibernate Validator and Tomcat's embedded expression language), you can simply add the `spring-boot-starter-validation` starter as a dependency.

Table B.1 lists all of Spring Boot's starter dependencies along with the dependencies that they transitively declare.

Table B.1 Spring Boot starters

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter	■ org.springframework.boot:spring-boot ■ org.springframework.boot:spring-boot-autoconfigure ■ org.springframework.boot:spring-boot-starter-logging ■ org.springframework:spring-core *(excludes commons-logging:commons-logging)* ■ org.yaml:snakeyaml
spring-boot-starter-actuator	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-actuator
spring-boot-starter-amqp	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-messaging ■ org.springframework.amqp:spring-rabbit

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-aop	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-aop ■ org.aspectj:aspectjrt ■ org.aspectj:aspectjweaver
spring-boot-starter-artemis	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-jms ■ org.apache.activemq:artemis-jms-client
spring-boot-starter-batch	■ org.springframework.boot:spring-boot-starter ■ org.hsqldb:hsqldb ■ org.springframework:spring-jdbc ■ org.springframework.batch:spring-batch-core
spring-boot-starter-cache	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-context ■ org.springframework:spring-context-support
spring-boot-starter-cloud-connectors	■ org.springframework.boot:spring-boot-starter ■ org.springframework.cloud:spring-cloud-spring-service-connector ■ org.springframework.cloud:spring-cloud-cloudfoundry-connector ■ org.springframework.cloud:spring-cloud-heroku-connector ■ org.springframework.cloud:spring-cloud-localconfig-connector
spring-boot-starter-data-elasticsearch	■ org.springframework.boot:spring-boot-starter ■ org.springframework.data:spring-data-elasticsearch
spring-boot-starter-data-gemfire	■ org.springframework.boot:spring-boot-starter ■ com.gemstone.gemfire:gemfire *(excludes commons-logging:commons-logging)* ■ org.springframework.data:spring-data-gemfire
spring-boot-starter-data-jpa	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-aop ■ org.springframework.boot:spring-boot-starter-jdbc ■ org.hibernate:hibernate-entitymanager *(excludes org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec)* ■ javax.transaction:javax.transaction-api ■ org.springframework.data:spring-data-jpa ■ org.springframework:spring-aspects
spring-boot-starter-data-mongodb	■ org.springframework.boot:spring-boot-starter ■ org.mongodb:mongo-java-driver ■ org.springframework.data:spring-data-mongodb

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-data-rest	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ com.fasterxml.jackson.core:jackson-annotations ■ com.fasterxml.jackson.core:jackson-databind ■ org.springframework.data:spring-data-rest-webmvc
spring-boot-starter-data-solr	■ org.springframework.boot:spring-boot-starter ■ org.apache.solr:solr-solrj *(excludes log4j:log4j)* ■ org.springframework.data:spring-data-solr ■ org.apache.httpcomponents:httpmime
spring-boot-starter-freemarker	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.freemarker:freemarker ■ org.springframework:spring-context-support
spring-boot-starter-groovy-templates	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.codehaus.groovy:groovy-templates
spring-boot-starter-hateoas	■ org.springframework.boot:spring-boot-starter-web ■ org.springframework.hateoas:spring-hateoas ■ org.springframework.plugin:spring-plugin-core
spring-boot-starter-hornetq	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-jms ■ org.hornetq:hornetq-jms-client
spring-boot-starter-integration	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-aop ■ org.springframework.integration:spring-integration-core ■ org.springframework.integration:spring-integration-file ■ org.springframework.integration:spring-integration-http ■ org.springframework.integration:spring-integration-ip ■ org.springframework.integration:spring-integration-stream
spring-boot-starter-jdbc	■ org.springframework.boot:spring-boot-starter ■ org.apache.tomcat:tomcat-jdbc ■ org.springframework:spring-jdbc

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-jersey	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-tomcat ■ org.springframework.boot:spring-boot-starter-validation ■ com.fasterxml.jackson.core:jackson-databind ■ org.springframework:spring-web ■ org.glassfish.jersey.core:jersey-server ■ org.glassfish.jersey.containers:jersey-container-servlet-core ■ org.glassfish.jersey.containers:jersey-container-servlet ■ org.glassfish.jersey.ext:jersey-bean-validation *(excludes javax.el:javax.el-api, org.glassfish.web:javax.el)* ■ org.glassfish.jersey.ext:jersey-spring3 ■ org.glassfish.jersey.media:jersey-media-json-jackson
spring-boot-starter-jetty	■ org.eclipse.jetty:jetty-servlets ■ org.eclipse.jetty:jetty-webapp ■ org.eclipse.jetty.websocket:websocket-server ■ org.eclipse.jetty.websocket:javax-websocket-server-impl
spring-boot-starter-jooq	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-jdbc ■ org.springframework:spring-tx ■ org.jooq:jooq
spring-boot-starter-jta-atomikos	■ org.springframework.boot:spring-boot-starter ■ com.atomikos:transactions-jms ■ com.atomikos:transactions-jta *(excludes org.apache.geronimo.specs:geronimo-jta_1.0.1B_spec)* ■ com.atomikos:transactions-jdbc ■ javax.transaction:javax.transaction-api
spring-boot-starter-jta-bitronix	■ org.springframework.boot:spring-boot-starter ■ javax.jms:jms-api ■ javax.transaction:javax.transaction-api ■ org.codehaus.btm:btm *(excludes javax.transaction:jta)*
spring-boot-starter-log4j	■ org.slf4j:jcl-over-slf4j ■ org.slf4j:jul-to-slf4j ■ org.slf4j:slf4j-log4j12 ■ log4j:log4j

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-log4j2	■ org.apache.logging.log4j:log4j-slf4j-impl ■ org.apache.logging.log4j:log4j-api ■ org.apache.logging.log4j:log4j-core ■ org.slf4j:jcl-over-slf4j ■ org.slf4j:jul-to-slf4j
spring-boot-starter-logging	■ ch.qos.logback:logback-classic ■ org.slf4j:jcl-over-slf4j ■ org.slf4j:jul-to-slf4j ■ org.slf4j:log4j-over-slf4j
spring-boot-starter-mail	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-context ■ org.springframework:spring-context-support ■ com.sun.mail:javax.mail
spring-boot-starter-mobile	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework.mobile:spring-mobile-device
spring-boot-starter-mustache	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ com.samskivert:jmustache
spring-boot-starter-redis	■ org.springframework.boot:spring-boot-starter ■ org.springframework.data:spring-data-redis ■ redis.clients:jedis
spring-boot-starter-remote-shell	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-actuator ■ org.crashub:crash.cli ■ org.crashub:crash.connectors.ssh *(excludes org.codehaus.groovy:groovy-all)* ■ org.crashub:crash.connectors.telnet *(excludes javax.servlet:servlet-api, log4j :log4j, commons-logging:commons-logging)* ■ org.crashub:crash.embed.spring *(excludes org.springframework:spring-web, org.codehaus.groovy:groovy-all)* ■ org.crashub:crash.plugins.cron *(excludes org.codehaus.groovy:groovy-all)* ■ org.crashub:crash.plugins.mail *(excludes org.codehaus.groovy:groovy-all)* ■ org.crashub:crash.shell *(excludes org.codehaus.groovy:groovy-all)* ■ org.codehaus.groovy:groovy

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-security	■ org.springframework.boot:spring-boot-starter ■ org.springframework:spring-aop ■ org.springframework.security:spring-security-config ■ org.springframework.security:spring-security-web
spring-boot-starter-social-facebook	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework.social:spring-social-config ■ org.springframework.social:spring-social-core ■ org.springframework.social:spring-social-web ■ org.springframework.social:spring-social-facebook
spring-boot-starter-social-linkedin	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework.social:spring-social-config ■ org.springframework.social:spring-social-core ■ org.springframework.social:spring-social-web ■ org.springframework.social:spring-social-linkedin
spring-boot-starter-social-twitter	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework.social:spring-social-config ■ org.springframework.social:spring-social-core ■ org.springframework.social:spring-social-web ■ org.springframework.social:spring-social-twitter
spring-boot-starter-test	■ junit:junit ■ org.mockito:mockito-core ■ org.hamcrest:hamcrest-core ■ org.hamcrest:hamcrest-library ■ org.springframework:spring-core *(excludes commons-logging:commons-logging)* ■ org.springframework:spring-test
spring-boot-starter-thymeleaf	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.thymeleaf:thymeleaf-spring4 ■ nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect
spring-boot-starter-tomcat	■ org.apache.tomcat.embed:tomcat-embed-core ■ org.apache.tomcat.embed:tomcat-embed-el ■ org.apache.tomcat.embed:tomcat-embed-logging-juli ■ org.apache.tomcat.embed:tomcat-embed-websocket

Table B.1 Spring Boot starters *(continued)*

Starter *(Group ID: org.springframework.boot)*	Transitively depends on
spring-boot-starter-undertow	■ io.undertow:undertow-core ■ io.undertow:undertow-servlet *(excludes org.jboss.spec.javax.servlet:jboss-servlet-api_3.1_spec)* ■ io.undertow:undertow-websockets-jsr ■ javax.servlet:javax.servlet-api ■ org.glassfish:javax.el
spring-boot-starter-validation	■ org.springframework.boot:spring-boot-starter ■ org.apache.tomcat.embed:tomcat-embed-el ■ org.hibernate:hibernate-validator
spring-boot-starter-velocity	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ commons-beanutils:commons-beanutils ■ commons-collections:commons-collections ■ commons-digester:commons-digester ■ org.apache.velocity:velocity ■ org.apache.velocity:velocity-tools ■ org.springframework:spring-context-support
spring-boot-starter-web	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-tomcat ■ org.springframework.boot:spring-boot-starter-validation ■ com.fasterxml.jackson.core:jackson-databind ■ org.springframework:spring-web ■ org.springframework:spring-webmvc
spring-boot-starter-websocket	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework:spring-messaging ■ org.springframework:spring-websocket
spring-boot-starter-ws	■ org.springframework.boot:spring-boot-starter ■ org.springframework.boot:spring-boot-starter-web ■ org.springframework:spring-jms ■ org.springframework:spring-oxm ■ org.springframework.ws:spring-ws-core ■ org.springframework.ws:spring-ws-support

appendix C Configuration properties

Although Spring Boot handles a lot of the grunt work when it comes to configuring the components in your application, you may want to fine-tune some of those components. That's where configuration properties come in handy.

Chapter 3 describes the `@ConfigurationProperties` annotation and how it can be used to expose properties that you can configure external to application code. Just as you can use `@ConfigurationProperties` in components that you create, many of Spring Boot's auto-configured components are also annotated with `@ConfigurationProperties`, making it possible to configure them via any supported property source.

For example, to specify the port that an embedded Tomcat or Jetty server should listen for requests on, you can set the `server.port` property. This can be set as a property in application.properties, in application.yml, in an operating system environment variable, or any of the other options listed in section 3.2.

This appendix lists all of the configuration properties offered by Spring Boot components. Note that the applicability of these properties is dependent upon the component being declared as a bean in the Spring application context (most likely by way of auto-configuration). Setting a property for an inactive component will have no effect.

- `flyway.baseline-description`

  The description to tag an existing schema with when executing baseline.
- `flyway.baseline-on-migrate`

  Whether to automatically call baseline when migrate is executed against a non-empty schema with no metadata table. (Default value: `false`)
- `flyway.baseline-version`

  Sets the version to tag an existing schema with when executing baseline. (Default value: `1`)

- `flyway.check-location`
  Check that migration scripts location exists. (Default value: `false`)
- `flyway.clean-on-validation-error`
  Whether to automatically call `clean` or not when a validation error occurs. (Default value: `false`)
- `flyway.enabled`
  Enable flyway. (Default value: `true`)
- `flyway.encoding`
  Sets the SQL migration encoding. (Default value: `UTF-8`)
- `flyway.ignore-failed-future-migration`
  Whether to ignore failed future migrations when reading the metadata table. (Default value: `false`)
- `flyway.init-sqls`
  SQL statements to execute to initialize a connection immediately after obtaining it.
- `flyway.locations`
  Locations of migrations scripts. (Default value: `db/migration`)
- `flyway.out-of-order`
  Whether or not "out of order" migrations are allowed. (Default value: `false`)
- `flyway.password`
  Login password of the database to migrate.
- `flyway.placeholder-prefix`
  Sets the prefix of every placeholder. (Default value: `${`)
- `flyway.placeholder-replacement`
  Whether placeholders should be replaced. (Default value: `true`)
- `flyway.placeholder-suffix`
  Sets the prefix of every placeholder. (Default value: `${`)
- `flyway.placeholders.[placeholder name]`
  Sets a placeholder value.
- `flyway.schemas`
  A case-sensitive list of schemes managed by Flyway. Defaults to the default schema of the connection.
- `flyway.sql-migration-prefix`
  The filename prefix for SQL migrations. (Default value: `V`)
- `flyway.sql-migration-separator`
  The filename separator for SQL migrations. (Default value: __)

- `flyway.sql-migration-suffix`
  The filename suffix for SQL migrations. (Default value: `.sql`)
- `flyway.table`
  The name of the schema metadata table to be used by Flyway. (Default value: `schema_version`)
- `flyway.target`
  The target version up to which Flyway should consider migrations. (Defaults to the latest version)
- `flyway.url`
  JDBC URL of the database to migrate. If not set, the primary configured data source is used.
- `flyway.user`
  Login user of the database to migrate.
- `flyway.validate-on-migrate`
  Whether to automatically validate when running migrate. (Default value: `true`)
- `liquibase.change-log`
  Change log configuration path. (Default value: `classpath:/db/changelog/db.changelog-master.yaml`)
- `liquibase.check-change-log-location`
  Check that the change log location exists. (Default value: `true`)
- `liquibase.contexts`
  Comma-separated list of runtime contexts to use.
- `liquibase.default-schema`
  Default database schema.
- `liquibase.drop-first`
  Drop the database schema first. (Default value: `false`)
- `liquibase.enabled`
  Enable Liquibase support. (Default value: `true`)
- `liquibase.password`
  Login password of the database to migrate.
- `liquibase.url`
  JDBC URL of the database to migrate. If not set, the primary configured data source is used.
- `liquibase.user`
  Login user of the database to migrate.

- `multipart.enabled`
  Enable support of multi-part uploads. (Default value: `true`)
- `multipart.file-size-threshold`
  Threshold after which files will be written to disk. Values can use the suffixes “MB” or “KB” to indicate a megabyte or kilobyte size. (Default value: `0`)
- `multipart.location`
  Intermediate location of uploaded files.
- `multipart.max-file-size`
  Max file size. Values can use the suffixes “MB” or “KB” to indicate a megabyte or kilobyte size. (Default value: `1MB`)
- `multipart.max-request-size`
  Max request size. Values can use the suffixes “MB” or “KB” to indicate a megabyte or kilobyte size. (Default value: `10MB`)
- `security.basic.authorize-mode`
  Security authorize mode to apply.
- `security.basic.enabled`
  Enable basic authentication. (Default value: `true`)
- `security.basic.path`
  Comma-separated list of paths to secure. (Default value: `[/**]`)
- `security.basic.realm`
  HTTP basic realm name. (Default value: `Spring`)
- `security.enable-csrf`
  Enable cross-site request forgery support. (Default value: `false`)
- `security.filter-order`
  Security filter chain order. (Default value: `0`)
- `security.headers.cache`
  Enable cache control HTTP headers. (Default value: `false`)
- `security.headers.content-type`
  Enable `X-Content-Type-Options` header. (Default value: `false`)
- `security.headers.frame`
  Enable `X-Frame-Options` header. (Default value: `false`)
- `security.headers.hsts`
  HTTP Strict Transport Security (HSTS) mode (`none`, `domain`, `all`).
- `security.headers.xss`
  Enable cross-site scripting (XSS) protection. (Default value: `false`)

- `security.ignored`
  Comma-separated list of paths to exclude from the default secured paths.
- `security.oauth2.client.access-token-uri`
  The URI used to fetch an access token.
- `security.oauth2.client.access-token-validity-seconds`
  How long an access token is to be valid before expiring.
- `security.oauth2.client.additional-information.[key]`
  Set additional information that token granters would like to add to the token.
- `security.oauth2.client.authentication-scheme`
  The method for transmitting the bearer token. One of `form`, `header`, `none`, or `query`. (Default value: `header`)
- `security.oauth2.client.authorities`
  The authorities to be granted to an authenticated client.
- `security.oauth2.client.authorized-grant-types`
  The grant types allowed to the client.
- `security.oauth2.client.auto-approve-scopes`
  The scope to automatically approve for a client.
- `security.oauth2.client.client-authentication-scheme`
  The method for transmitting authentication credentials when authenticating the client. One of `form`, `header`, `none`, or `query`. (Default value: `header`)
- `security.oauth2.client.client-id`
  OAuth2 client ID.
- `security.oauth2.client.client-secret`
  OAuth2 client secret. A random secret is generated by default.
- `security.oauth2.client.grant-type`
  The grant type for obtaining an access token for this resource.
- `security.oauth2.client.id`
  The application's client ID.
- `security.oauth2.client.pre-established-redirect-uri`
  The redirect URI that has been pre-established with the server. If present, the redirect URI will be omitted from the user authorization request because the server doesn't need to know it.
- `security.oauth2.client.refresh-token-validity-seconds`
  How long a refresh token will be valid before expiring.
- `security.oauth2.client.registered-redirect-uri`
  Comma-separated list of redirect URIs registered for the client.