Heroku Deployments

jim stafford

Fall 2022 v2020-07-30: Built: 2022-12-07 06:17 EST

Table of Contents

Chapter 1. Introduction

To date we have been worrying about the internals of our applications, how to configure them, test them, interface with them, and how to secure them. We need others to begin seeing our progress as we continue to fill in the details to make our applications useful.

In this lecture — we will address deployment to a cloud provider. We will take a hands-on look at deploying to Heroku — a cloud platform provider that makes deploying Spring Boot and Docker-based applications part of their key business model without getting into more complex hosting frameworks.

> ⚠ After over 10 years of availability, Heroku has announced that their free deployments will terminate Nov 28, 2022. Obviously, this impacts the specific deployment aspects provided in this lecture. However, it does not impact the notion of what is deployable to alternate platforms when identified.

1.1. Goals

You will learn:

- to deploy an application under development to an cloud provider to make it accessible to Internet users
- to deploy incremental and iterative application changes

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. create a new Heroku application with a choice of names
2. deploy a Spring Boot application to Heroku using the Heroku Maven Plugin
3. interact with your developed application on the Internet
4. make incremental and iterative changes

Chapter 2. Heroku Background

According to their website, Heroku is a cloud provider that provides the ability to "build, deliver, monitor, and scale apps". They provide a fast way to go from "idea to URL" by bypassing company managed infrastructure. [1]

There are many cloud providers but not many are in our sweet spot of offering a platform for Spring Boot and Docker applications without the complexity of bare metal OS or a Kubernetes cluster. They also offer these basic deployments for no initial cost for non-commercial applications — such as proof of concepts and personal projects that stay within a 512MB memory limit.

There is a lot to Heroku that will not be covered here. However, this lecture will provide a good covering of how to achieve successful deployment of a Spring Boot application. In a follow-on lecture we will come back to Heroku to deploy Docker images and see the advantages it doing so. The following lists a few resources on the Heroku web site

- Working with Spring Boot
- Developing with Docker

[1] *"What is Heroku"*, Heroku Web Site, July 2020

Chapter 3. Setup Heroku

You will need to setup an account with Heroku in order to use their cloud deployment environment. This is a free account and stays free until we purposely choose otherwise. If we exceed free constraints — our deployment simply will not run. There will be no surprise bill.

- visit the Heroku Web Site
- select [Sign Up For Free]
- create a free account and complete the activation
  - I would suggest skipping 2-factor authentication for simplicity for class use. You can always activate it later.
  - Salesforce bought Heroku and now has some extra terms to agree to
- install the command line interface (CLI) for your platform. It will be necessary to work at the shell level quite a bit
  - refer to the Heroku CLI reference as necessary

Chapter 4. Heroku Login

Once we have an account and the CLI installed — we need to login using the CLI. This will redirect us to the browser where we can complete the login.

Heroku Command Line Login

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/f944d777-93c7-40af-
b772-0a1c5629c609
Logging in... done
Logged in as ...
```

Chapter 5. Create Heroku App

At this point you are ready to perform a one-time (per deployment app) process that will reserve an app-name for you on herokuapp.com. When working with Heroku — think of app-name as a deployment target with an Internet-accessible URL that shares the same name. For example, my app-name of `ejava-boot` is accessible using https://ejava-boot.herokuapp.com. I can deploy one of many Spring Boot applications to that app-name (one at a time). I can also deploy the same Spring Boot application to multiple Heroku app-names (e.g., integration and production)

Let jim have ejava :)

Please use you own naming constructs. I am kind of fond of the `ejava-` naming prefix.

Example Create Heroku App

```
$ heroku create [app-name] ①

Creating ⬢ [app-name]... done
https://app-name.herokuapp.com/ | https://git.heroku.com/app-name.git
```

① if app-name not supplied, a random app-name will be generated

Heroku also uses Git repositories for deployment

Heroku creates a Git repository for the app-name that can also be leveraged as a deployment interface. I will not be covering that option.

You can create more than one heroku app and the app can be renamed with the following `apps:rename` command.

Example Rename Heroku App

```
$ heroku apps:rename --app oldname newname
```

Visit the Heroku apps page to locate technical details related to your apps.

Heroku will try to determine the resources required for the application when it is deployed the first time. Sometimes we have to give it details (e.g., provision DB)

Chapter 6. Create Spring Boot Application

For this demonstration, I have created a simple Spring Boot web application (docker-hello-example) that will be part of a series of lectures this topic area. Don’t worry about the "Docker" naming for now. We will be limiting the discussion relative to this application to only the Spring Boot portions during this lecture.

6.1. Example Source Tree

The following structure shows the simplicity of the web application.

Example Spring Boot Web Application Source Tree

```
docker-hello-example/
|-- pom.xml
`-- src/main/java/info.ejava.examples.svc.docker
        |       `-- hello
        |           |-- DockerHelloExampleApp.java
        |           `-- controllers
        |               |-- ExceptionAdvice.java
        |               `-- HelloController.java
        `-- resources
            `-- application.properties
```

6.1.1. HelloController

The supplied controller is a familiar "hello" example, with optional authentication. The GET method will return a hello to the name supplied in the `name` query parameter. If authenticated, the controller will also issue the caller’s associated username.

HelloController

```
@RestController
public class HelloController {
    @GetMapping(path="/api/hello",
            produces = {MediaType.TEXT_PLAIN_VALUE})
    public String hello(
            @RequestParam("name")String name,
            @AuthenticationPrincipal UserDetails user) {
        String username = user==null ? null : user.getUsername();
        String greeting = "hello, " + name;
        return username==null ? greeting : greeting + " (from " + username + ")";
    }
}
```

6.2. Starting Example

We can start the web application using the Spring Boot plugin `run` goal.

Starting Example Spring Boot Web Application

```
$ mvn spring-boot:run
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (2.7.0)
...
Tomcat started on port(s): 8080 (http) with context path ''
Started DockerHelloExampleApp in 1.972 seconds (JVM running for 2.401)
```

6.3. Client Access

Once started, we can access the `HelloController` running on localhost and the assigned `8080` port.

Accessing Local Spring Boot Web Application

```
$ curl http://localhost:8080/api/hello?name=jim
hello, jim
```

Security is enabled, so we can also access the same endpoint with credentials and get authentification feedback.

Accessing Local Spring Boot Web Application with Credentials

```
$ curl http://localhost:8080/api/hello?name=jim -u "user:password"
hello, jim (from user)
```

6.4. Local Unit Integration Test

The example also includes a set of unit integration tests that perform the same sort of functionality that we demonstrated with curl a moment ago.

Local Unit Integration Test

```
docker-hello-example/
`-- src/test/java/info/ejava/examples/svc
        |      `-- docker
        |          `-- hello
        |              |-- ClientTestConfiguration.java
        |              `-- HelloLocalNTest.java
        `-- resources
            `-- application-test.properties
```

Local Unit Integration Test Results

```
$ mvn clean test
10:12:54.692 main  INFO     i.e.e.svc.docker.hello.HelloLocalNTest#init:38
baseUrl=http://localhost:51319
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.079 s - in
info.ejava.examples.svc.docker.hello.HelloLocalNTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 7. Maven Heroku Deployment

When ready for application deployment, Heroku provides two primary styles of deployment with for a normal Maven application:

- git repository
- Maven plugin

The git repository requires that your deployment follow a pre-defined structure from the root — which is not flexible enough for a class demonstration tree with nested application modules. If you go that route, it may also require a separate Procfile to address startup.

The Heroku Maven plugin encapsulates everything we need to define our application startup and has no restriction on root repository structure.

7.1. Spring Boot Maven Plugin

The `heroku-maven-plugin` will deploy our Spring Boot executable JAR. We, of course, need to make sure our `heroku-maven-plugin` and `spring-boot-maven-plugin` configurations are consistent.

The `ejava-build-parent` defines a classifier value, which gets used to separate the Spring Boot executable JAR from the standard Java library JAR.

Base Project Maven Properties

```
<properties>
    <spring-boot.classifier>bootexec</spring-boot.classifier>
</properties>
```

spring-boot-maven-plugin pluginManagement Definition

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>${springboot.version}</version>
    <configuration>
        <classifier>${spring-boot.classifier}</classifier> ①
    </configuration>
    <executions>
      <execution>
        <id>package</id>
        <phase>package</phase>
        <goals>
          <goal>repackage</goal>
        </goals>
      </execution>
    </executions>
</plugin>
```

① used in naming the built Spring Boot executable JAR

7.1.1. Child Project Spring Boot Maven Plugin Declaration

The child module declares the `spring-boot-maven-plugin`, picking up the pre-configured `repackage` goal.

spring-boot-maven-plugin plugin declaration

```
<plugin> <!-- builds a Spring Boot Executable JAR -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

The Spring Boot executable JAR has the `bootexec` classifier name appended to the version.

```
$ mvn clean package
...
target/
|...
|-- [ 31M]  docker-hello-example-6.0.1-SNAPSHOT-bootexec.jar ②
|-- [9.7K]  docker-hello-example-6.0.1-SNAPSHOT.jar ①
```

① standard Java library JAR

② Spring Boot Executable JAR to be deployed to Heroku

7.2. Heroku Maven Plugin

The following snippets show an example use of the Heroku Maven Plugin used in this example. Documentation details are available on GitHub. It has been parameterized to be able to work with most applications and is defined in the pluginDependencies section of the `ejava-build-parent` parent pom.xml.

Base Project Maven Properties

```
<properties>
    <java.source.version>17</java.source.version>
    <java.target.version>17</java.target.version>
    <heroku-maven-plugin.version>3.0.4</heroku-maven-plugin.version>
</properties>
```

heroku-maven-plugin pluginManagement Definition

```
<plugin>
    <groupId>com.heroku.sdk</groupId>
    <artifactId>heroku-maven-plugin</artifactId>
    <version>${heroku-maven-plugin.version}</version>
    <configuration>
```

```
        <jdkVersion>${java.target.version}</jdkVersion>
        <includeTarget>false</includeTarget> ①
        <includes> ②
            <include>target/${project.build.finalName}-${spring-
boot.classifier}.jar</include>
        </includes>
        <processTypes> ③
            <web>java $JAVA_OPTS -jar target/${project.build.finalName}-${spring-
boot.classifier}.jar --server.port=$PORT $JAR_OPTS
            </web>
        </processTypes>
    </configuration>
</plugin>
```

① don't deploy entire contents of target directory

② identify specific artifacts to deploy; Spring Boot executable JAR — accounting for classifier

③ takes on role of `Procfile`; supplying the launch command

You will see mention of the `$PORT` parameter in the Heroku Profile documentation. This is a value we need to set our server port to when deployed. We can easily do that with the `--server.port` property.

`$JAR_OPTS` is an example of being able to define other properties to be expanded — even though we don't have a reason at this time. Any variables in the command line can be supplied/overridden with the configVars element. For example, we could use that property to set the Spring profile(s).

configVars example

```
<configVars>
  <JAR_OPTS>--spring.profiles.active=authorities,authorization</JAR_OPTS>
</configVars>
```

7.2.1. Child Project Heroku Maven Plugin Declaration

The child module declares the `heroku-maven-plugin`, picking up the pre-configured plugin.

```
<plugin>
    <groupId>com.heroku.sdk</groupId>
    <artifactId>heroku-maven-plugin</artifactId>
</plugin>
```

7.3. Deployment appName

The deployment will require an app-name. Heroku recommends creating a profile for each of the deployment environments (e.g., development, integration, and production) and supplying the appName in those profiles. However, I am showing just a single deployment — so I set the appName separately through a property in my settings.xml.

Example appName Setting

```
<properties>
  <heroku.appName>ejava-boot</heroku.appName> ①
</properties>
```

① the Heroku Maven Plugin can have its `appName` set using a Maven property or element.

7.4. Example settings.xml Profile

The following shows an example of setting our `heroku.appName` Maven property using `$HOME/.m2/settings.xml`. The upper `profiles` portion is used to define the profile. The lower `activeProfiles` portion is used to statically declare the profile to always be active.

Example $HOME/.m2/settings.xml Profile

```
<?xml version="1.0"?>
<settings xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">

    <profiles>
        <profile> ①
            <id>ejava</id>
            <properties>
                <heroku.appName>ejava-boot</heroku.appName>
            </properties>
        </profile>
    </profiles>

    <activeProfiles> ②
      <activeProfile>ejava</activeProfile>
    </activeProfiles>
</settings>
```

① defines a group of of Maven properties to be activated with a Maven profile

② profiles can be statically defined to be activated

The alternative to `activeProfiles` is to use either an activation in the pom.xml or on the command line.

Example Command Line Profile Activation

```
$ mvn (command) -Pejava
```

7.5. Using Profiles

If we went the profile route, it could look something like the following with `dev` being unique per developer and `stage` having a more consistent name across the team.

Example Use of Profiles to Set Property

```
<profiles>
    <profile>
        <id>dev</id>
        <properties> ①
          <heroku.appName>${my.dev.name}</heroku.appName>
        </properties>
    </profile>
    <profile>
        <id>stage</id>
        <properties> ②
          <heroku.appName>our-stage-name</heroku.appName>
        </properties>
    </profile>
</profiles>
```

① variable expansion based on individual `settings.xml` values when `-Pdev` profile set

② well-known-name for staging environment when `-Pstage` profile set

7.6. Maven Heroku Deploy Goal

The following shows the example output for the `heroku:deploy` Maven goal.

Example Maven heroku:deploy Goal

```
$ mvn heroku:deploy
...
[INFO] --- heroku-maven-plugin:3.0.3:deploy (default-cli) @ docker-hello-example ---
[INFO] -----> Packaging application...
[INFO]        - including: target/docker-hello-example-6.0.1-SNAPSHOT-SNAPSHOT.jar
[INFO]        - including: pom.xml
[INFO] -----> Creating build...
[INFO]        - file: /var/folders/zm/cskr47zn0yjd0zwkn870y5sc0000gn/T/heroku-
deploy10792228069435401014source-blob.tgz
[INFO]        - size: 22MB
[INFO] -----> Uploading build...
[INFO]        - success
[INFO] -----> Deploying...
[INFO] remote:
[INFO] remote: -----> heroku-maven-plugin app detected
[INFO] remote: -----> Installing JDK 11... done
[INFO] remote: -----> Discovering process types
[INFO] remote:        Procfile declares types -> web
```

```
[INFO] remote:
[INFO] remote: -----> Compressing...
[INFO] remote:        Done: 81.6M
[INFO] remote: -----> Launching...
[INFO] remote:        Released v3
[INFO] remote:        https://ejava-boot.herokuapp.com/ deployed to Heroku
[INFO] remote:
[INFO] -----> Done
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  35.516 s
```

7.7. Tail Logs

We can gain some insight into the application health by tailing the logs.

```
$ heroku logs --app ejava-boot --tail
Starting process with command `--server.port\=\$\{PORT:-8080\}`
...
Tomcat started on port(s): 54644 (http) with context path ''
Started DockerHelloExampleApp in 9.194 seconds (JVM running for 9.964)
```

7.8. Access Site

We can access the deployed application at this point using HTTPS.

Access Deployed Application on Heroku

```
$ curl -v https://ejava-boot.herokuapp.com/api/hello?name=jim
hello, jim
```

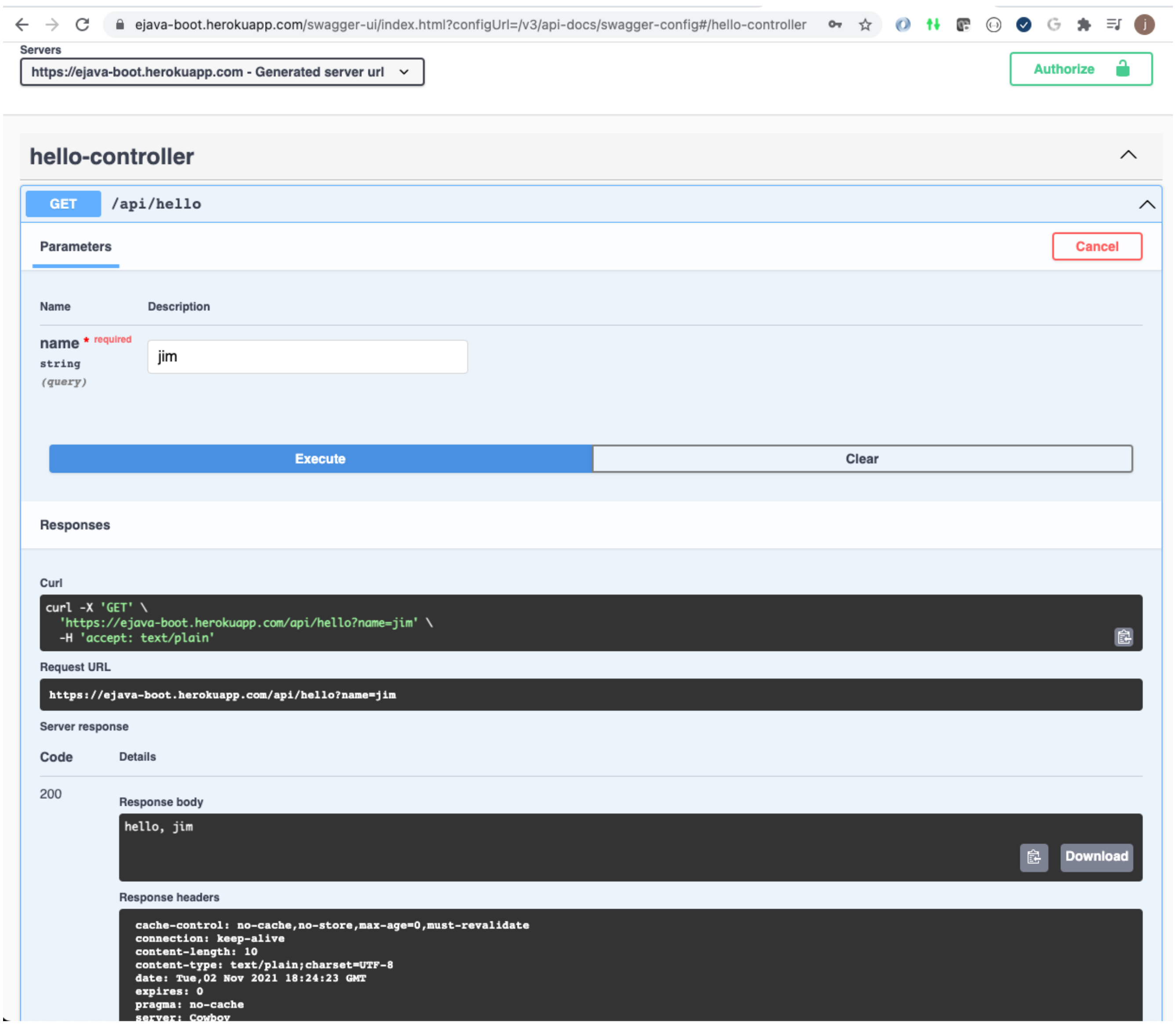
Notice that we deployed an HTTP application and must access the site using HTTPS. Heroku is providing the TLS termination without any additional work on our part.

Heroku Server Cert

```
* Server certificate:
*  subject: CN=*.herokuapp.com
*  start date: Jun  1 00:00:00 2021 GMT
*  expire date: Jun 30 23:59:59 2022 GMT
*  subjectAltName: host "ejava-boot.herokuapp.com" matched cert's "*.herokuapp.com"
*  issuer: C=US; O=Amazon; OU=Server CA 1B; CN=Amazon
*  SSL certificate verify ok.
```

7.9. Access Via Swagger

We can also access the site via swagger with a minor amount of configuration.


ejava-boot.herokuapp.com/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/hello-controller
Servers
https://ejava-boot.herokuapp.com - Generated server url
Authorize
hello-controller
GET
/api/hello
Parameters
Cancel
Name
Description
name * required
string
(query)
jim
Execute
Clear
Responses
Curl

```
curl -X 'GET' \
  'https://ejava-boot.herokuapp.com/api/hello?name=jim' \
  -H 'accept: text/plain'
```

Request URL

```
https://ejava-boot.herokuapp.com/api/hello?name=jim
```

Server response

Code	Details
200	Response body

```
hello, jim
```

Response headers

```
cache-control: no-cache,no-store,max-age=0,must-revalidate
connection: keep-alive
content-length: 10
content-type: text/plain;charset=UTF-8
date: Tue,02 Nov 2021 18:24:23 GMT
expires: 0
pragma: no-cache
server: Cowboy
```

Figure 1. Access Via Swagger

To configure Swagger to ignore the injected `@AuthenticationPrincipal` parameter — we need to annotate it as hidden, using a Swagger annotation.

Eliminate Injected Parameters

```
import io.swagger.v3.oas.annotations.Parameter;
...
public String hello(
        @RequestParam("name")String name,
        @Parameter(hidden = true) //for swagger
        @AuthenticationPrincipal UserDetails user) {
```

Chapter 8. Remote IT Test

We have seen many times that there are different levels of testing that include:

- unit tests (with Mocks)
- unit integration tests (horizontal and vertical; with Spring context)
- integration tests (heavyweight process; failsafe)

No one type of test is going to be the best in all cases. In this particular case we are going to assume that all necessary unit (core functionality) and unit integration (Spring context integration) tests have been completed and we want to evaluate our application in an environment that resembles production deployment.

8.1. JUnit IT Test Case

To demonstrate the remote test, I have created a single `HelloHerokuIT` JUnit test and customized the `@Configuration` to be able to be used to express remote server aspects.

8.1.1. Test Case Definition

The failsafe integration test case looks like most unit integration test cases by naming `@Configuration` class(es), active profiles, and following a file naming convention (`IT`). The `@Configuration` is used to define beans for the IT test to act as a client of the remote server. The `heroku` profile contains properties defining identity of remote server.

Test Case Definition

```
@SpringBootTest(classes=ClientTestConfiguration.class, ①
    webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"test","heroku"}) ③
public class HelloHerokuIT { ④
```

① `@Configuration` defines beans for client testing

② no server active within JUnit IT test JVM

③ activate property-specific profiles

④ failsafe test case class name ends with IT

8.1.2. Injected Components and Setup

This specific test injects 2 users (anonymous and authenticated), the username of the authenticated user, and the baseUrl of the remote application. The baseUrl is used to define a template for the specific call being executed.

Injected Components and Setup

```
@Autowired
private RestTemplate anonymousUser;
```

```
@Autowired
private RestTemplate authnUser;
@Autowired
private String authnUsername;
private UriComponentsBuilder helloUrl;

@BeforeEach
void init(@Autowired URI baseUrl) {
    log.info("baseUrl={}", baseUrl);
    helloUrl = UriComponentsBuilder.fromUri(baseUrl).path("api/hello")
            .queryParam("name","{name}"); ①
}
```

① `helloUrl` is a baseUrl + `/api/hello?name={name}` template

8.2. IT Properties

The integration test case pulls production properties from `src/main` and test properties from `src/test`.

8.2.1. Application Properties

The application is using a single user with its username and password statically defined in `application.properties`. These values will be necessary to authenticate with the server — even when remote.

application.properties

```
spring.security.user.name=user
spring.security.user.password=password
```

Do Not Store Credentials in JAR

Do not store credentials in a resource file within the application. Resource files are generally checked into CM repositories and part of JARs published to artifact repositories. A resource file is used here to simpify the class example. A realistic solution would point the application at a protected directory or source of properties at runtime.

8.2.2. IT Test Properties

The `application-heroku.properties` file contains 3 non-default properties for the `ServerConfig`. `scheme` is hardcoded to `https`, but the `host` and `port` are defined with `${placeholder}` variables that will be filled in with Maven properties using the `maven-resources-plugin`.

- We do this for `host`, so that the `heroku.appName` can be pulled from an environment-specific properties
- We do this for `port`, to be certain that `server.http.port` is set within the `pom.xml` because the

ejava-build-parent configures failsafe to pass the value of that property as it.server.port.

application-heroku.properties

```
it.server.scheme=https ①
it.server.host=${heroku.appName}.herokuapp.com ②
it.server.port=${server.http.port} ③ ④
```

① using HTTPS protocol

② Maven resources plugin configured to filter value during compile

③ Maven filtered version of property used directly within IDE

④ runtime failsafe configuration will provide value override

8.2.3. Maven Property Filtering

Maven copies resource files from the source tree to the target tree by default using the `maven-resources-plugin`. This plugin supports file filtering when copying files from the `src/main` and `src/test` areas. This is so common, that the definition can be expressed outside the boundaries of the plugin. The snippet below shows the setup of filtering a single file from `src/test/resources` and uses elements `testResource/testResources`. Filtering a file from `src/main` (not used here) would use elements `resources/resource`.

The filtering is setup in two related definitions: what we are filtering (filtering=true) and everything else (filtering=false). If we accidentally leave out the filtering=false definition, then only the filtered files will get copied. We could have simply filtered everything but that can accidentally destroy binary files (like images and truststores) if they happen to be placed in that path. It is safer to be explicit about what must be filtered.

Maven Property Filtering

```
<build> ①
    <testResources> <!-- used to replace ${variables} in property files -->
        <testResource> ②
            <directory>src/test/resources</directory>
            <includes> <!-- replace ${heroku.appName} -->
                <include>application-heroku.properties</include>
            </includes>
            <filtering>true</filtering>
        </testResource>
        <testResource> ③
            <directory>src/test/resources</directory>
            <excludes>
                <exclude>application-heroku.properties</exclude>
            </excludes>
            <filtering>false</filtering>
        </testResource>
    </testResources>
    ....
```

① Maven/resources-maven-plugin configured here to filter a specific file in `src/test`

② `application-heroku.properties` will be filtered when copied

③ all other files will be copied but not filtered

Maven Resource Filtering can Harm Some Files

Maven resource filtering can damage binary files and naively constructed property files (that are meant to be evaluated at runtime versus build time). It is safer to enumerate what needs to be filtered than to blindly filter all resources.

8.2.4. Property Value Sources

The source for the Maven properties can come from many places. The example sets a default within the pom.xml. We expect the `heroku.appName` to be environment-specific, so if you deploy the example using Maven — you will need to add `-Dheroku.appName=your-app-name` to the command line or through your local `settings.xml` file.

```
<properties> ①
    <heroku.appName>ejava-boot</heroku.appName>
    <server.http.port>443</server.http.port>
</properties>
```

① default values - can be overridden by command and settings.xml values

8.2.5. Maven Process Resource Phases

The following snippet shows the two resource phases being executed. Our `testResources` are copied and filtered in the second phase.

Maven Process Resource Phases

```
$ mvn clean process-test-resources
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ docker-hello-example ---
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ docker-hello-
example ---
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ docker-hello-
example ---
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @
docker-hello-example ---

$ cat target/test-classes/application-heroku.properties
it.server.scheme=https
it.server.host=ejava-boot.herokuapp.com
it.server.port=443
```

The following snippet shows the results of the property filtering using a custom value for `heroku.appName`

Maven Property Override

```
$ mvn clean process-test-resources -Dheroku.appName=other-name ①
$ cat target/test-classes/application-heroku.properties
it.server.scheme=https
it.server.host=other-name.herokuapp.com ②
it.server.port=443
```

① custom Maven property supplied on command-line

② supplied value expanded during resource filtering

8.3. Configuration

The `@Configuration` class sets up 2 RestTemplate @Bean factories: anonymousUser and authnUser. Everything else is there to mostly to support the setup of the HTTPS connection. This same `@Configuration` is used for both the unit and failsafe integration tests. The `ServerConfig` is injected during the failsafe IT test (using `application-heroku.properties`) and instantiated locally during the unit integration test (using `@LocalPort` and default values).

ClientTestConfiguration

```
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties //used to set it.server properties
@EnableAutoConfiguration
public class ClientTestConfiguration {
```

8.3.1. Authentication

Credentials (from `application.properties`) are injected into the `@Configuration` class using `@Value` injection. The username for the credentials is made available as a `@Bean` to evaluate test results.

Authentication

```
@Value("${spring.security.user.name}") ①
private String username;
@Value("${spring.security.user.password}")
private String password;

@Bean
public String authnUsername() { return username; } ②
```

① default values coming from `application.properties`

② username exposed only to support evaluating authentication results

8.3.2. Server Configuration (Client Properties)

The remote server configuration is derived from properties available at runtime and scoped under the "it.server" prefix. The definitions within the `ServerConfig` instance can be used to form the baseUrl for the remote server.

ServerConfig

```
@Bean
@ConfigurationProperties(prefix = "it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}

//use for IT tests
@Bean ①
public URI baseUrl(ServerConfig serverConfig) {
    URI baseUrl = serverConfig.build().getBaseUrl();
    return baseUrl;
}
```

① baseUrl resolves to `https://ejava-boot.herokuapp.com:443`

8.3.3. anonymousUser

An injectable RestTemplate is exposed with no credentials as "anonymousUser". As with most of our tests, the `BufferingClientHttpRequestFactory` has been added to support multiple reads required by the `RestTemplateLoggingFilter` (which provides debug logging). The `ClientHttpRequestFactory` was made injectable to support HTTP/HTTPS connections.

anonymousUser

```
@Bean
public RestTemplate anonymousUser(RestTemplateBuilder builder,
                                  ClientHttpRequestFactory requestFactory) { ①
    return builder.requestFactory(
                    //used to read the streams twice ③
                    ()->new BufferingClientHttpRequestFactory(requestFactory))
            .interceptors(new RestTemplateLoggingFilter()) ②
            .build();
}
```

① `requestFactory` will determine whether HTTP or HTTPS connection created

② `RestTemplateLoggingFilter` provides HTTP debug statements

③ `BufferingClientHttpRequestFactory` caches responses, allowing it to be read multiple times

8.3.4. authnUser

An injectable RestTemplate is exposed with valid credentials as "authnUser". This is identical to `anonymousUser` except credentials are provided through a `BasicAuthenticationInterceptor`.

authnUser

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                              ClientHttpRequestFactory requestFactory) {
    return builder.requestFactory(
                    //used to read the streams twice
                    ()->new BufferingClientHttpRequestFactory(requestFactory))
            .interceptors(
                    new BasicAuthenticationInterceptor(username, password), ①
                    new RestTemplateLoggingFilter())
            .build();
}
```

① valid credentials added

8.3.5. ClientHttpRequestFactory

The builder requires a `requestFactory` and we have already shown that it will be wrapped in a `BufferingClientHttpRequestFactory` to support debug logging. However, the core communications is implemented by the `org.apache.http.client.HttpClient` class.

ClientHttpRequestFactory

```
import org.apache.http.client.HttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import javax.net.ssl.SSLContext;
...
@Bean
public ClientHttpRequestFactory httpsRequestFactory(
        ServerConfig serverConfig, ①
        SSLContext sslContext) { ②
    HttpClient httpsClient = HttpClientBuilder.create()
            .setSSLContext(serverConfig.isHttps() ? sslContext : null)
            .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

① `ServerConfig` provided to determine whether HTTP or HTTPS required

② `SSLContext` provided for when HTTPS is required

8.3.6. SSL Context

The SSLContext is provided by the `org.apache.http.ssl.SSLContextBuilder` class. In this particular instance, we expect the deployment environment to use commercial, trusted certs. This will eliminate the need to load a custom truststore.

```
import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
```

```
...
@Bean
public SSLContext sslContext(ServerConfig serverConfig)  {
    try {
        URL trustStoreUrl = null;
        //using trusted certs, no need for customized truststore
        //...
        SSLContextBuilder builder = SSLContextBuilder.create()
                .setProtocol("TLSv1.2");
        if (trustStoreUrl!=null) {
            builder.loadTrustMaterial(trustStoreUrl, serverConfig
.getTrustStorePassword());
        }
        return builder.build();
    } catch (Exception ex) {
        throw new IllegalStateException("unable to establish SSL context", ex);
    }
}
```

8.4. JUnit IT Test

The following shows two sanity tests for our deployed application. They both use a base URL of `https://ejava-boot.herokuapp.com/api/hello?name={name}` and supply the request-specific `name` property through the `UriComponentsBuilder.build(args)` method.

8.5. Simple Communications Test

When successful, the simple communications test will return a 200/OK with the text "hello, jim"

Simple Communications Test

```
@Test
void can_contact_server() {
    //given
    String name="jim";
    URI url = helloUrl.build(name);
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = anonymousUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    then(response.getBody()).isEqualTo("hello, " + name); ①
}
```

① "hello, jim"

8.6. Authentication Test

When successful, the authentication test will return a 200/OK with the text "hello, jim (from user)". The name for "user" will be the username injected from the `application.properties` file.

Authentication Test

```
@Test
void can_authenticate_with_server() {
    //given
    String name="jim";
    URI url = helloUrl.build(name);
    RequestEntity<Void> request = RequestEntity.get(url).build();
    //when
    ResponseEntity<String> response = authnUser.exchange(request, String.class);
    //then
    then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    then(response.getBody()).isEqualTo("hello, " +name+ " (from " +authnUsername+ ")"
);①
}
```

① "hello, jim (from user)"

8.7. Automation

The IT tests have been disabled to avoid attempts to automatically deploy the application in every build location. Automation can be enabled at two levels: test and deployment.

8.7.1. Enable IT Test

We can enable the IT tests alone by adding `-DskipITs=value`, where `value` is anything but `true`, `false`, or blank.

- skipITs (blank) and skipITs=true will cause failsafe to not run. This is a standard failsafe behavior.
- skipITs=false will cause the application to be re-deployed to Heroku. This is part of our custom pom.xml definition that will be shown in a moment.

Execute IT Test Only

```
$ mvn verify -DitOnly -DskipITs=not_true ① ② ③
...
GET https://ejava-boot.herokuapp.com:443/api/hello?name=jim, returned OK/200
hello, jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

① `verify` goal completes the IT test phases

② `itOnly` - defined by `ejava-build-parent` to disable surefire tests

③ `skipITs` - controls whether IT tests are performed

skipITs can save Time and Build-time Dependencies

Setting `skipITs=true` can save time and build-time dependencies when all that is desired it a resulting artifact produced by `mvn install`.

8.7.2. Enable Heroku Deployment

The pom also has conditionally added the `heroku:deploy` goal to the `pre-integration` phase if `skipITs=false` is explicitly set. This is helpful if changes have been made. However, know that a full upload and IT test execution is a significant amount of time to spend. Therefore, it is not the thing one would use in a rapid test, code, compile, test repeat scenario.

Enable Heroku Deployment

```
<profiles>
    <profile> <!-- deploys a Spring Boot Executable JAR -->
        <id>heroku-it-deploy</id>
        <activation>
            <property> ①
                <name>skipITs</name>
                <value>false</value>
            </property>
        </activation>
        <properties> ②
            <spring-boot.repackage.skip>false</spring-boot.repackage.skip>
        </properties>
        <build>
            <plugins>
                <plugin> ③
                    <groupId>com.heroku.sdk</groupId>
                    <artifactId>heroku-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <id>deploy</id>
                            <phase>pre-integration-test</phase>
                            <goals>
                                <goal>deploy</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

① only fire of `skipITs` has the value `false`

② make sure that JAR is a Spring Boot executable JAR

③ add deploy step in `pre-integration` phase

IT Test Results with Heroku Deploy

```
$ mvn verify -DitOnly -DskipITs=false
...
[INFO] --- spring-boot-maven-plugin:2.4.2:repackage (package) @ docker-hello-example
---
[INFO] Replacing main artifact with repackaged archive
[INFO] <<< heroku-maven-plugin:3.0.3:deploy (deploy) < package @ docker-hello-example
<<<
[INFO] --- heroku-maven-plugin:3.0.3:deploy (deploy) @ docker-hello-example ---
[INFO] jakarta.el-3.0.3.jar already exists in destination.
...
[INFO] -----> Done
[INFO] --- maven-failsafe-plugin:3.0.0-M5:integration-test (integration-test) @
docker-hello-example ---
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (verify) @ docker-hello-example ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 9. Summary

In this module we learned:

- to deploy an application under development to Heroku cloud provider to make it accessible to Internet users
  - using naked Spring Boot form
- to deploy incremental and iterative changes to the application
- how to interact with your developed application on the Internet