Building microservices with Spring Boot – part 2

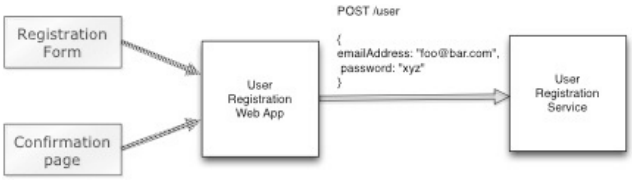
Posted on May 5, 2014

In part 1 of this series, we looked at how the Microservice architecture pattern decomposes an application into a set of services. We described how Spring Boot simplifies the development of RESTFul web service within a microservice architecture. In this article we will look at developing a microservice that's a traditional (i.e. HTML generating) Spring MVC-based web application. You will learn how Spring Boot simplifies the development of this kind of web application. You can find the example code on github.

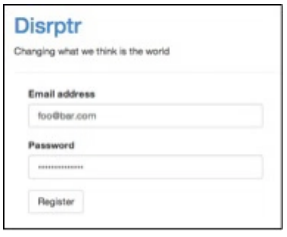
Building the registration UI

We could implement an application's server side-presentation tier as a single monolithic web application containing all of the Spring MVC controllers, views, etc. However, for large, complex applications we will very likely encounter the same kinds of problems with the monolithic architecture that were discussed in part 1. Consequently, it makes sense to apply the Microservice architecture pattern to the presentation tier and decompose it into a collection of web applications. Each web application implements the UI for one or more related stories or use cases. This enables the developers working on a particular part of the UI to independently develop and deploy their code.

In this article, we are going to implement the UI for the user registration example we discussed in part 1. The following diagram shows how the web application fits into the overall architecture.



The UI is implemented by a Spring-MVC based web application that invokes the RESTful API provided by the backend registration service. The UI consists of a registration page and a registration confirmation page, which are shown below:



The registration page consists of a form for entering the user's email address and desired password. Clicking the "register" button POSTs to server, which responds by either redirecting the browser to the confirr

page or by redisplaying the form with error messages.

The registration page uses the JQuery validation plugin for browser-side validation a
Bootstrap CSS for layout. Let's now look at the Spring MVC components that handle t
generate HTML.

Using Spring MVC

As you might expect, the Spring MVC-based presentation tier has a UserRegistration
the HTTP requests, and a RegistrationRequest command class or form-backing object. The properties of the RegistrationRequest class, which correspond to fields of the registration form, have validation annotations that ensure that the email address appears to be valid and that the password meets the minimum length requirement.

```
class RegistrationRequest {

  @BeanProperty
  @Email
  @NotNull
  var emailAddress: String = _

  @BeanProperty
  @NotNull
  @Size(min = 8, max = 30)
  var password: String = _

}
```

Here is the Scala source code for the controller.

```
@Controller
class UserRegistrationController
         @Autowired()(restTemplate: RestTemplate) {

  @Value("${user_registration_url}")
  var userRegistrationUrl : String = _

  @RequestMapping(value = Array("/register.html"),
                  method = Array(RequestMethod.GET))
  def beginRegister = "register"

  @RequestMapping(value = Array("/register.html"),
                  method = Array(RequestMethod.POST))
  def register(@Valid() @ModelAttribute("registration")
                   request: RegistrationRequest,
               bindingResult: BindingResult,
               redirectAttributes: RedirectAttributes): String = {
    if (bindingResult.getErrorCount != 0)
      return "register"

    val response = try
      restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(request.getEmailAddress,
                                   request.getPassword),
        classOf[RegistrationBackendResponse])
    catch {
      case e: HttpClientErrorException
                if e.getStatusCode == HttpStatus.CONFLICT =>
```

```
            bindingResult.rejectValue("emailAddress",
                        "duplicate.email.address",
                        "Email address already registered")
            return "register"
        }

        response.getStatusCode match {
          case HttpStatus.OK =>
            redirectAttributes.addAttribute(
                  "emailAddress", request.getEmailAddress)
            "redirect:registrationconfirmation.html"
        }
      }

      @RequestMapping(value = Array("/registrationcomplete.html"),
                      method = Array(RequestMethod.GET))
      def registrationComplete(@RequestParam emailAddress: String,
                      model: Model) = {
           model.addAttribute("emailAddress", emailAddress)
           "registrationconfirmation"
      }
    }
```

The controller has three methods that handle HTTP requests. The HTTP GET requests are handled by the beginRegister() method, which displays the registration page, and the registrationComplete() method, which redirects to the registration confirmation page.

The POST of the registration form is handled by the register() method. Spring MVC binds the form fields to method's RegistrationRequest command parameter. The @Valid annotation on that parameter triggers validation of the command as defined by the validation annotations on the RegistrationRequest class. If form validation fails then the controller redisplays the registration form with one or more error messages.

Otherwise, if form validation succeeds, the controller makes a registration request to the web service that we built in part 1 of this series. The URL for the registration backend is injected into the controller using the @Value annotation on the userRegistrationUrl field.

If registration is successful (as indicated by an HTTP status of 200), the register() method then redirects to the confirmation page. Otherwise, if the call to the registration web service fails then the controller redisplays the registration form with one or more error messages.

In addition to the controller, the web application has views that generate HTML for the registration and confirmation pages. We could implement these views using JSPs. However, since Java 8 has an excellent JavaScript engine and today's cool kids are using JavaScript-based templating frameworks this web application implements the views using DustJS. In a later article, we will describe the implementation of these DustJS views including how DustJS is integrated with Spring MVC.

Spring Boot + WebJars = easy delivery of JavaScript and CSS files

The views generate HTML pages that use Bootstrap CSS and JQuery Validation JavaScript. Those pages also use application-specific CSS as well as some JavaScript that configures JQuery Validation to validate the registration form. With Spring Boot it's remarkably easy to configure the web application to serve the necessary JavaScript and CSS. First, we will look at how to serve static assets for Bootstrap and the JQuery

Validation plugin. After that we will look at how to serve application-specific JS and CSS files.

There are a couple of different ways to serve up CSS and JavaScript for Bootstrap and JQuery validation. One option is for the HTML to reference a content delivery network (CDN). While a CDN is great for production, it's an obstacle for offline development. Another option is to manually download those files and configure Spring MVC to serve them as static content. While this isn't too difficult, it typically means downloading multiple files – including transitive dependencies – from multiple sites, which is quite tedious.

A newer and much more convenient approach is to use WebJars. WebJars are client-side libraries (e.g. CSS and JavaScript) packaged as JAR files and published to a Maven repository. To use a particular client-side library you need to add the corresponding WebJar as a Gradle/Maven project dependency and configure Spring MVC to serve it as static content.

In order to use Bootstrap CSS and the JQuery validation plugin, the pom.xml for the registration web application has the following dependencies:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.1.1</version>
</dependency>

<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery-validation</artifactId>
    <version>1.11.1</version>
</dependency>
```

But that's all we need to do since a very nice feature of Spring Boot is that detects these WebJars on the classpath and, by default, serves their contents under the path /webjars/**. The HTML pages can reference the CSS and JavaScript files as follows:

```
<html>
…
  <link rel="stylesheet" href="webjars/bootstrap/3.1.1/css/bootstrap.css">
…
  <script src="webjars/jquery/2.1.0/jquery.js"></script>
  <script src="webjars/jquery-validation/1.11.1/jquery.validate.js"></script>
…
```

Notice that jquery.js is available as static content even though it was not explicitly defined as a webjar dependency. That's because the JQuery WebJar is a dependency of the JQuery validation WebJar and so Maven treats it as a transitive dependency.

Spring Boot also makes it easy to serve application-specific CSS and JavaScript files. It configures Spring MVC to treat a /static directory on the classpath as containing static content. Consequently, we can simply put the application-specific CSS and JavaScript files in the src/main/resources/static directory and the HTML pages can access them as follows:

```
    <link rel="stylesheet" href="styles/main.css">
…
    <script src="js/registration.js"> </script>
```

The CSS styles/main.css file is served from src/main/resources/static/styles/main.css and the js/registration.js file is served from src/main/resources/static/js/registration.js

Let's now look at the other Spring Boot-related parts of the application: the Maven pom.xml and a Java configuration class.

Maven pom.xml

This application is built using Maven because the Gradle Scala plugin currently doesn't work with Java 8. The following listing shows the Spring Boot related parts of the Maven pom.xml:

```
<project>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.0.1.RELEASE</version>
</parent>

<dependencies>

   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   …
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
   …
</dependencies>

<build>
        <plugins>
          <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
                   <version>1.0.1.RELEASE</version>
               </plugin>
          …
        </plugins>
    …
</build>
```

The pom specifies that its parent pom is spring-boot-starter-parent, which is a special Spring Boot starter that defines use Maven defaults.

The pom.xml defines a dependency on the Spring Boot spring-boot-starter-web starter artifact, which ensures that we get a consistent set of dependencies to build this web application including various Spring framework and embedded Tomcat artifacts. The spring-boot-starter-web artifact also enables the Spring Boot's auto-configuration.

In addition, the pom.xml defines a dependency on the Spring Boot spring-boot-starter-test starter artifact. This artifact pulls in artifacts for automated tests including the Spring framework's spring-test artifact and

some Spring Boot specific test classes.

The pom.xml also specifies the Spring Boot maven plugin, which is responsible for building the executable jar.

Simple Spring Boot based configuration

Since the application uses Spring Boot, it needs remarkably little configuration. There isn't even a web.xml. Instead, there is a single configuration class:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
class UserRegistrationConfiguration {

  @Bean
  @Primary
  def scalaObjectMapper() = new ScalaObjectMapper

  @Bean
  def dustViewResolver = {
    val resolver = new DustViewResolver
    resolver.setPrefix("/WEB-INF/views/")
    resolver.setSuffix(".dust")
    resolver
  }

  @Bean
  def restTemplate = {
    val restTemplate = new RestTemplate()
    restTemplate.getMessageConverters foreach {
      case mc: MappingJackson2HttpMessageConverter =>
        mc.setObjectMapper(scalaObjectMapper())
      case _ =>
    }
    restTemplate
  }

}
```

This configuration class defines three beans:

- DustViewResolver – configures Spring MVC to use DustJS views
- scalaObjectMapper – A Jackson JSON ObjectMapper that registers the DefaultScalaModule, which provides support for Scala objects. It's used by the RestTemplate
- RestTemplate – used by the controller to make requests to the registration backend. It's configured to use the ScalaObjectMapper to serialize/deserialize JSON requests to/from JSON.

UserRegistrationMain class

This class defines the main() method that runs the application. It's a one liner that calls the SpringApplication.run() method passing in the configuration class and the args parameter to main().

```
object UserRegistrationMain  {

  def main(args: Array[String]) : Unit =
```

```
        SpringApplication.run(
              classOf[UserRegistrationConfiguration], args :_ *)

}
```

The SpringApplication class is provided by Spring Boot. It's run() method creates and starts the web container that runs the application.

Putting it all together

We can build the application by running the 'mvn package' command:

```
$ mvn package
[INFO] Scanning for projects...
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ spring-boot-user-registration-webapp ---
[INFO] Building jar: /Users/cer/src/microservices-examples/spring-boot-webapp/target/spring-boot-user-
…
INFO] --- spring-boot-maven-plugin:1.0.1.RELEASE:repackage (default) @ spring-boot-user-registration-w
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

We can then run the application using the java –jar command:

```
$ java -jar target/spring-boot-user-registration-webapp-1.0-SNAPSHOT.jar
…
2014-04-25 16:38:36.599 INFO 29547 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat start
2014-04-25 16:38:36.602 INFO 29547 --- [ main] n.c.m.r.main.UserRegistrationMain$ : Started UserRegist
```

Summary

By applying the Microservice architecture pattern to the presentation tier we decompose what would otherwise be a large monolithic web application into many smaller web applications. Each web application implements the UI for one or more related stories or use cases. The example web application described in this article implements a user registration UI and registers users by making REST requests to a backend service. Because it's a standalone service, the developers working on the registration UI can develop new features, fix bugs and deploy the changes independently of the rest of the system.

Spring Boot significantly simplifies the development and deployment of web applications. Because it provides Convention-over-Configuration in the form of auto-configuration, web applications need very little explicit configuration. There is no web.xml. Spring Boot configures Spring MVC to serve any WebJars on the classpath as static content. It also configures Spring MVC to treat a /static directory on the classpath as static content. As a result, you can focus on developing your web application.

Spring Boot also simplifies how you deploy your web application. The Spring maven (and Gradle) plugin packages your web application as an executable jar file containing an embedded Tomcat (or Jetty) server. You can run the jar file on any machine with Java installed – no need to install and configure an application server.

In later posts, we will look at other aspects of developing microservices with Spring Boot including automated testing and using DustJS in a Spring MVC application, as well as look at how Spring Boot simplifies monitoring and management.

This entry was posted in architecture, microservices. Bookmark the permalink.

16 Responses to *Building microservices with Spring Boot – part 2*

Pingback: *Building microservices with Spring Boot – part 1 | plain old objects*

cblin *says:*
July 30, 2014 at 1:53 pm

when will you continue the serie ?

Reply

ceracm *says:*
July 30, 2014 at 1:59 pm

Thanks for your interest. Since these blog posts and the QCON article, I've been focussed on development. I definitely need to get going with more posts in the series. Hopefully, soon!

Reply

cblin *says:*
August 4, 2014 at 2:16 am

Indeed, I am grateful for your articles since I find spring boot documentation quite obscure.

My particular interest is non HTTP microservices (i.e rabbitMQ RPC).

That is, instead of having each microservices that creates a HTTP server, each microservices binds to a work queue and publish the results to a topic.
Scalabiliy is then more predictible and redeployments are easier (the work queue will automatically redisaptch the request when you crash the microservices)

Have you used this kind of microservices ?

Reply

ceracm *says:*
August 4, 2014 at 10:26 am

Thanks. Glad you like the articles. I hope to write more shortly.

Yes, in my current application, I have Spring Boot-based message-driven microservices. It's a good architectural approach. They also use HTTP for Spring Boot-based health monitoring.

I've also implemented message-driven microservices before Spring Boot.

Kevin Mandeville *says:*
September 25, 2014 at 1:40 pm

I have a question regarding the Database in your microservices. I have a Spring Boot based microservice that uses Hibernate/JPA. I'm looking for a good way to manage the state of the database (for production. For testing, Hibernate works well). When I deploy my microservice, it should make sure the database exists, has all the necessary tables and columns creating them/updating them if necessary and install any setup data that is needed.
Should we use liquibase scripts? Flyway? Is there some other way I should be using?

Reply

ceracm *says:*
October 21, 2014 at 9:38 pm

Kevin,

Sorry for the delayed response.
I'd use a tool such as flyway to manage DB migration.
It's not too different than with an monolithic application.
I would try to decouple database migrations from code deployments.

Chris

Reply

James Martignoni *says:*
October 21, 2014 at 3:51 am

I was going to try out your example but see you dont have the code for the second part on github ? or did I miss it ?

Reply

ceracm *says:*
October 21, 2014 at 9:35 pm

James,

you are correct I need to publish the code. I will do that shortly,

Chris

Reply

ceracm *says:*
October 23, 2014 at 6:29 pm

I pushed the code to github: https://github.com/cer/microservices-examples/tree/master/spring-boot-webapp

Thanks for the reminder.

Reply

Pingback: *Deploying Spring Boot-based microservices with Docker | plain old objects*

Alberto *says:*
January 11, 2015 at 7:41 pm

That was a great post thanks.
I'd like to ask a question:
I love the idea of applying the Microservice architecture pattern to the presentation tier, but what is the best way of taking all the HTML responses and compose them to form a single page?
Thank you.

Reply

ceracm *says:*
January 13, 2015 at 10:50 am

Glad you like the post. I was assuming that that each frontend microservice (mini-webapps) is responsible for one or more pages: catalog pages, checkout pages – rather than fragments that are composed to form a page (ie. a portal like approach). You would use what I referred to as a content router (see http://www.slideshare.net/chris.e.richardson/developing-apps-with-a-microservice-architecture-svforum-microservices-meetup/51) to provide a single entry point to all of the mini webapps.

A related part of the architecture is the API Gateway that provides a single entry point to the RESTful services: http://www.slideshare.net/chris.e.richardson/developing-apps-with-a-microservice-architecture-svforum-microservices-meetup/44

I hope this helps.

Reply

Alberto *says:*
January 13, 2015 at 11:15 am

Thank you very much for your response. It helps.
I guess that in my original idea all the mini-webapps would return HTML, that could be a full page or a fragment and then, some service running in front of the apps will take care of the routing and composing all the pieces together.
I was thinking like that because some of the mini-webapps would return content that it's shared by other mini-webapps (like a site header and footer that you want to always display no matter where you are in the site).
But now I'm thinking that I can I can push the content composition to the mini-webapps. So for example, the 'Checkout UI' app would return the content of the checkout page + the content from the 'Header & Fotter' app. Maybe there's no need of having an external service that does all the composition.
Thank you again!

rajsg *says:*
June 20, 2016 at 8:08 pm

Thanks a lot for your wonderful post, which helped to understand the fundamentals and the core concepts about micro services. Do we have any java variant available for the given sample Scala code ?

Reply

ceracm *says:*
June 22, 2016 at 5:38 pm

No. This version only exists in Scala. There are other Java examples – http://eventuate.io/exampleapps.html

Reply