I think things.

Sometimes I write about them.

Spring-Boot

Moving to Spring Boot

The Spring framework has been the de facto standard framework to build a Java application on for some time now. Providing an IoC container and performing dependency injection was just the start. Since it's initial release in 2002, Spring has expanded and matured, providing developers with familiar, patterns-based abstractions for common components throughout an application's layers. As Spring grew, the configuration became more and more unwieldy and the framework became known as one that involved a fair amount of effort to set up and get going, and even the most trivial of projects came with a fair amount of boilerplate configuration. There was not an easy place to start. Maven filled this gap in the early days, pushing the community toward the concepts of convention over configuration and dependency management, and through the use of project archetypes, but the same problem eventually cropped up – repetitive, difficult-to-manage configuration.

With the first release of Rails in 2005, the developer community saw what was possible in terms of a developer-friendly framework that all but eliminated the perceived shortcomings of frameworks like Spring. Frameworks like Rails came to be known as Rapid Application Development (RAD) frameworks. These frameworks shared many of the same characteristics – a well-defined convention, opinionated default configurations, scaffolding tools used to quickly create pre-configured components. In 2009, the Spring developers responded to the trend of RAD frameworks with the release of Spring Roo. Spring Roo was never billed as an attempt to replace Spring, only enhance it by eliminating the shortcomings of vanilla Spring. Spring Roo provided a well-defined convention and scaffolding tools, but was driven by AspectJ and relied on a significant amount of code generation to eliminate boilerplate code. This led to difficulty troubleshooting configuration problems, and a steeper learning curve for developers new to Java and Spring.

Enter Spring Boot…
In 2014, the Spring development team released a next-generation take on Spring named Spring Boot. Spring Boot provides many of the same RAD-like features of frameworks like Rails, and goes a step further than Roo by eliminating cumbersome XML-based configuration and the mystery of generated

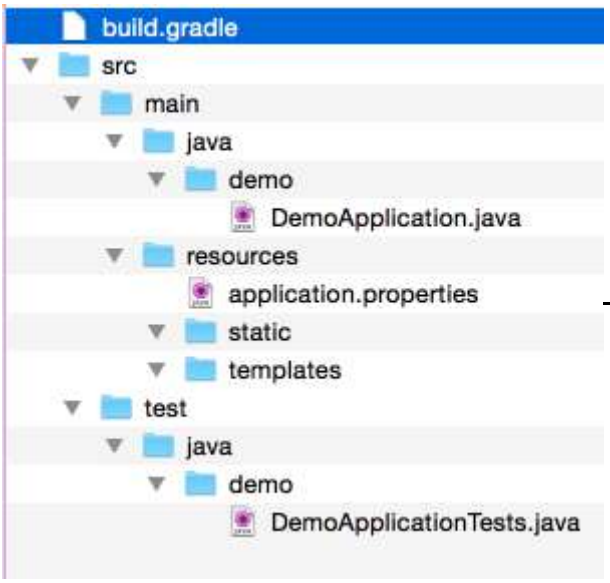
code. This is accomplished through the use of auto-configuration classes. Each Spring Boot module is packaged with a default configuration – the boilerplate code developers used to be responsible for creating. These auto-configuration classes provide the opinionated configuration familiar to users of other RAD frameworks, and that follows the basic best practices familiar to users of traditional Spring. In the next few sections, we'll get a new project up and running from scratch and see auto-configuration in action.

Creating A New Project

A newer feature of the Spring Boot project is the Spring Initializr, a website (http://start.spring.io (http://start.spring.io)) that allows a developer to choose a starting point for their application with a 1 page form the concludes with a 'Generate Project' button and a download of the shell project. Below, you can find the steps I used to configured a basic project:

(https://justinrodenbostel.files.wordpress.com/2015/02/screen-shot-2015-02-15-at-4-12-48-pm.png)

These choices produced the following project structure:



(https://justinrodenbostel.files.wordpress.com/2015/02/screen-shot-2015-02-15-at-3-08-51-pm.png)

This project can be built and run, but without at least one controller we'll have no page to display other than a default error page. Let's create a controller and test out the app.

Create a file in the root package of the project – in my case it's /src/main/java/com/spr/demo/SampleController:

```
package demo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * Created by justin on 2/15/15.
 */
@Controller
public class SampleController {
    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
    }
}
```

Next, start the server from the root of your project using the pre-configured Gradle script provided to us by Spring Initializr.

```
Justins-MacBook-Pro:demo justin$ gradle bootRun
```

You can see in the console output that the app started an already-configured Tomcat server running on port 8080. If we browse to http://localhost:8080 (http://localhost:8080) in a browser, we should see the following:

(https://justinrodenbostel.files.wordpress.com/2015/02/screen-shot-2015-02-15-at-3-42-49-pm.png)

That's a full-fledged Spring app with a wired controller and a base configuration. Counting every line of code, that's only 30 lines of code!

Adding A Template

You may remember that we chose a templating library (Thymeleaf) as part of our initial configuration on the Spring Initializr page. Let's add a page template to our example to show how simple it is to set that up as well. To do this, we'll have to create the template itself and change our controller slightly. In the earlier screenshot of our project, you'll see we have a 'templates' directory in our 'src/main/resources' directory. Create a file there named 'hello.html':

/src/main/resources/hello.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
<head lang="en">
    <meta charset="UTF-8" />
    <title>HELLO</title>
</head>
<body>
<p th:text="${message}"></p>
</body>
</html>
```

You can see we've added a placeholder for a string named 'message' that we'll supply from our controller.

Next, let's update our controller to populate the 'message' element:

/src/main/java/demo/SampleController.java:

```
1  package demo;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6
7  /**
8   * Created by justin on 2/15/15.
9   */
10 @Controller
11 public class SampleController {
12
13     @RequestMapping("/")
14     public String index(Model model) {
15         model.addAttribute("message", "HELLO WORLD!");
16         return "hello";
17     }
18 }
```

Now, when we run our app, we should see a different result – one that builds a page using the template we just created:

(https://justinrodenbostel.files.wordpress.com/2015/02/screen-shot-2015-02-15-at-4-02-52-pm.png)

Who's Behind The Curtain?
You can see we've created a full Spring app with a basic configuration, configured a controller, and started using a template engine to render our pages. If there is no generated code driving this, where is the configuration coming from?

If we take a closer look at the console output from the server starting you can see several references to a class named 'org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration'. This is where the magic is happening. Let's take a look at the source on Github (https://github.com/spring-projects/spring-boot/blob/master/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java (https://github.com/spring-projects/spring-boot/blob/master/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java)). Browsing through this source, we can see references to familiar bean configurations, the paths to pre-configured property file locations, the base configuration to enable Spring MVC, and much more.

Continued Reading
The Spring Boot docs contain great resources for getting started. The 'Getting Started' section provides a nice foundation for those new to the tool. http://docs.spring.io/spring-boot/docs/current-

SNAPSHOT/reference/htmlsingle/#getting-started (http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started)

While the Spring Boot docs contain a lot of valuable information, several of the examples fall a bit short of what we commonly see at clients. Last year, I wrote a 5 part series that expands on the Getting Started guides provided by Spring. https://justinrodenbostel.com/2014/04/08/beyond-the-examples/ (https://justinrodenbostel.com/2014/04/08/beyond-the-examples/). This explores some common problems that Spring easily solves: nested form binding, security integration, internationalization, and more.

Posted in Java, Spring and tagged Java, SPR, Spring, Spring-Boot on February 16, 2015 by Justin. Leave a comment

Part 5a: Additional Credential Security – Spring Data JPA + Jasypt

In this short addendum to my earlier series on Spring Boot, I'll be covering a fairly trivial problem whose solution I had a somewhat hard time finding an example of. Like normal, I hope this will help someone out who is just getting started.

On my current project, we're working with a security-focused consulting firm named Jemurai (http://jemurai.com (http://jemurai.com)). Before I go further I wanted to mention that I learned about this tool and many, many other things security-related from one of their people (@mkonda). A great learning experience, and a pleasure to work with. Check them out.

Back to my current project. We had a goal to encrypt sensitive data at rest in properties files. One of the things you'll notice if you've been following through Part 5 is that the database credentials are stored, in plain text, in properties files. In this installment, we'll be encrypting the password in that credential using Jasypt (http://jasypt.org), an encryption tool for Java.

As usual, for this installment, I've created a copy of the code from Part 5 and created a new project called Part 5 With Jasypt. It's committed to Github (https://github.com/jrodenbostel/beyond-the-examples), ready for cloning.

Download and Configure
Add the jasypt dependency to your build.gradle file:

/build.gradle:

```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile('org.springframework.boot:spring-boot-starter-security')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE')
    compile('org.jasypt:jasypt-spring31:1.9.2') //<—here it is.
    runtime('mysql:mysql-connector-java:5.1.6')

    testCompile('junit:junit')
}
```

Next, download the jasypt distribution from http://jaspyt.org: http://sourceforge.net/projects/jasypt/files/jasypt/jasypt%201.9.2/ (http://jaspyt.org: http://sourceforge.net/projects/jasypt/files/jasypt/jasypt%201.9.2/). Since we'll be encrypting the database credentials that are currently stored in a properties file, follow the instructions on the jasypt site outlining using their CLI tools (http://www.jasypt.org/encrypting-configuration.htm (http://www.jasypt.org/encrypting-configuration.htm)l) to encrypt your credential. I used the command below to perform this task. The task was run from the '/bin' directory in the jasypt distribution. The param named "input" should be string you wish to encrypt, and the password param is the decryption key used to decode your password as it's being ingested by Spring during app startup.

```
Justins-MacBook-Pro:bin justin$ ./encrypt.sh input="password" password=testtest
```

…and the output should look something similar to this:

```
----ENVIRONMENT-----------------

Runtime: Oracle Corporation Java HotSpot(TM) 64-Bit Server VM 24.51-b03



----ARGUMENTS-------------------

input: password
password: testtest



----OUTPUT----------------------

xpPrNtXz+SQmTYB0WQrc+2T8ZTubofox
```

Updating Properties

If you read the jasypt manual, you've probably already updated your properties file with the newly encrypted value. In order to decrypt it, we need to give jasypt a directive – an indicator that is used by jasypt to determine which values need to be decrypted while they're being ingested. I've updated my application.properties file to include my newly encrypted password and the encryption directive.

```
spring.datasource.url=jdbc:mysql://localhost:3306/beyond-the-examples
spring.datasource.username=root
spring.datasource.password=ENC(xpPrNtXz+SQmTYB0WQrc+2T8ZTubofox)
spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.jpa.hibernate.dialect= org.hibernate.dialect.MySQLInnoDBDialect
spring.jpa.generate-ddl=false
```

Updating Spring Config

Last, you'll need to update your Spring config to use Jasypt utilities to decrypt the password while ingesting the necessary properties prior to constructing our datasource.

```
@Value("${spring.datasource.driverClassName}")
private String databaseDriverClassName;

@Value("${spring.datasource.url}")
private String datasourceUrl;

@Value("${spring.datasource.username}")
private String databaseUsername;

private String databasePassword;

@Bean
public DataSource datasource() throws IOException {
    org.apache.tomcat.jdbc.pool.DataSource ds = new org.apache.tomcat.jdbc.poo
    ds.setDriverClassName(databaseDriverClassName);
    ds.setUrl(datasourceUrl);
    ds.setUsername(databaseUsername);
    ds.setPassword(getSecurePassword());

    return ds;
}

private String getSecurePassword() throws IOException {
    StandardPBEStringEncryptor encryptor = new StandardPBEStringEncryptor();
    encryptor.setPassword(System.getProperty("blogpost.jasypt.key"));
    Properties props = new EncryptableProperties(encryptor);
    props.load(this.getClass().getClassLoader().getResourceAsStream("applicati
    return props.getProperty("datasource.password");
}
```

Here you can see I've changed the reference to my "databasePassword" property – it is no longer being populated by the @Value annotation. You can also see that I've replaced the value passed to the mutator of the password property of the datasource bean with a reference to a helper method that retrieves our password.

In that method ("getSecurePassword"), there are a few things going on. We're retrieving the password the encryptor will use to decrypt the value stored in our properties file. Note that it's the same value we used on the CLI during encoding. Also note that I'm assuming the value used for this is stored in a system property. Anywhere you can read it from will work, although environment variables/system properties seem to be the best place from what I've heard anecdotally and read in the Jasypt docs. There are some tricks to passing JVM options to the bootRun task. See here (http://stackoverflow.com/questions/23367507/how-to-pass-system-property-to-gradle-task) and here (https://github.com/spring-projects/spring-boot/issues/406) for examples. You can see we're using an extension of java.util.Properties called 'EncryptableProperties'. This is a Jasypt class that understands how to react when reading a property value enclosed by the directive we talked about above. After that, we're using these properties like we would use any other java.util.Properties class.

Testing

To test our changes, simply execute the app using gradle bootRun. If we did it property (and remembered to specify our jasypt password somewhere), we should notice no user-facing changes – just the same app we had before, but now with encrypted properties. Note that I specified the value of my SystemProperty in the gradle script.

Conclusion
I hope you found this entry a useful continuation of the Jasypt and Spring.io documentation. Check back soon for another installment!

Posted in Spring and tagged @mkonda, Encryption, Jasypt, jemurai, Redpoint, Spring, Spring Data, Spring Security, Spring-Boot, Tutorial on June 6, 2014 by Justin. 13 Comments

Part 5: Integrating Spring Security with Spring Boot Web

Spring Boot provides utilities for quick and easy setup of Spring Security via auto-configuration and Java-based configuration. The getting started guide is quick and easy leads through configuring an in-memory AuthenticationManager in just minutes. Going beyond these examples, this installation will quickly review the getting started guide provided at Spring.io (http://spring.io), and conclude with the configuration of a datasource-backed AuthenticationManager that uses Spring Data JPA, and the MySQL database platform.

As usual, for this installment, I've created a copy of the code from Part 4 and created a new project called Part 5. It's committed to Github (https://github.com/jrodenbostel/beyond-the-examples), ready for cloning.

Updating Dependencies
To install Spring Security, we first need to update our gradle script to include a dependency on spring-boot-starter-security. Update build.gradle to include the following dependency as seen below.

/build.gradle:

```
1  dependencies {
2      compile("org.springframework.boot:spring-boot-starter-web")
3      compile("org.springframework.boot:spring-boot-starter-security")
4      compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
5
6      testCompile("junit:junit")
7  }
```

Following that, executing a build should pull in our new dependencies.

Creating The Security Configuration
Continuing to lift code from the Spring.io (http://spring.io) docs for review, below you'll find the example of the base Java security configuration. We'll review the important bits after the jump. We'll create this in the same directory as our other configuration files:

/src/main/java/com.rodenbostel.sample/SecurityConfiguration.java:

```java
package com.rodenbostel.sample;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.
import org.springframework.security.config.annotation.method.configuration.Ena
import org.springframework.security.config.annotation.web.builders.HttpSecurit
import org.springframework.security.config.annotation.web.configuration.WebSec
import org.springframework.security.config.annotation.web.servlet.configuratic
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;

@Configuration
@EnableWebMvcSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests().anyRequest().authenticated();
        http
                .formLogin().failureUrl("/login?error")
                .defaultSuccessUrl("/")
                .loginPage("/login")
                .permitAll()
                .and()
                .logout().logoutRequestMatcher(new AntPathRequestMatcher("/log
                .permitAll();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Excepti
        auth.inMemoryAuthentication().withUser("user").password("password").ro
    }
}
```

As usual, the @Configuration annotation lets Spring know that this file contains configuration information. The next two annotations (@EnableWebMvcSecurity and @EnableGlobalMethodSecurity(prePostEnabled=true)) setup the automatically-configured portions of our security scheme, provided by Spring-Boot. EnableWebMvcSecurity basically pulls in the default SpringSecurity/SpringMVC integration. It's an extension of the WebMvcConfigurerAdapter (http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurerAdapter.html?is-external=true), and adds methods for handling and generating CSRF tokens and resolving the logged in user, and configures default AuthenticationManagers and Pre/Post object authorization implementations. The @EnableGlobalMethodSecurity sets up processors for authorization advice that can be added around methods and classes. This authorization advice lets a developer write Spring EL that inspects input parameters and return types.

Our SecurityConfiguration class also extends WebSecurityConfigurerAdapter (http://docs.spring.io/spring-security/site/docs/3.2.0.RC2/apidocs/org/springframework/security/config/annotation/web/configuration

/WebSecurityConfigurerAdapter.html). In Spring/Spring Boot, Configurer Adapters are classes that construct default bean configurations and contain empty methods which are meant to be overridden. Overriding these methods allow a developer to customize the Web Security Configuration during startup. Typically, the default configurations are constructed, and immediately following, the empty methods are called. If you've overridden an empty method, you're able to inject custom behavior into the default configuration during the startup of the container.

In our case, the two coded parts of our SecurityConfiguration class (two methods named "configure") are examples of these empty methods meant to be overridden. During container startup, after the HttpSecurity object's default configuration is specified, our overridden method is called. Here we are able to customize the default configuration by specifying which requests to authorize, and how to route various security-related requests: default success URL, error routing, where to send logouts, etc. Also during container startup, after the AuthenticationManagerBuilder is configured, our configure method is called, and in this case we're altering the default configuration, giving instructions to the AuthenticationManagerBuilder to build an in-memory AuthenticationManager with a default user credential and role.

You'll notice in this configuration we've specified several URL paths that do not exist. There's no login page or controller, and no way for a user to interact with the security configuration when the app is started up. Next, we'll need to construct and wire in a login page to complete our beginning configuration.

Building The Login Page

The login page in the Spring.io (http://spring.io) sample is very straightforward. Just a simple form with an input for username and password. Let's build that and review a few key parts.

/src/main/resources/templates/login.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3 (ht
<head>
    <title>Spring Security Example</title>
</head>
<body>
<div th:if="${param.error}">
    Invalid username and password.
</div>
<div th:if="${param.logout}">
    You have been logged out.
</div>
<form th:action="@{/login}" method="post">
    <div><label> User Name : <input type="text" name="username"/> </label></di
    <div><label> Password: <input type="password" name="password"/> </label></
    <div><input type="submit" value="Sign In"/></div>
</form>
</body>
</html>
```

Most importantly, we have inputs with the names of "username" and "password". These are the Spring Security defaults. If you're routing a request to Spring Security to authenticate, these are the parameters on the request that it will be looking for. Next, you'll also notice that there are Thymeleaf conditionals

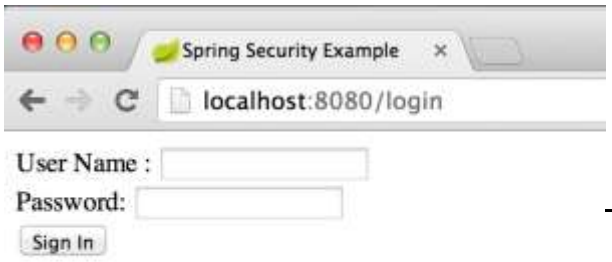
(th:if) for displaying logout and error messages if they are present in the response parameters during rendering. You'll also notice the path to this page is "/login", and the action on this form routes back to "/login" – but we don't have those registered anywhere…

Registering the Login Action
The path our login form is posting to is the default used by Spring Security. This is where what used to be called the "j_spring_security_check" servlet is listening for requests to authenticate. The request path (where we're retrieving the login form by issuing a GET to /login) is normally mapped to a controller, but in this case, since we're using automatically configured features of Spring Boot, we need to specify this mapping in our application configuration. Add the code below to your application configuration. You may notice the use of another @Override method – another hook where we can add logic to customize our application…

/src/main/java/com.rodenbostel.sample.Application.java:

```
1  @Override
2  public void addViewControllers(ViewControllerRegistry registry) {
3      registry.addViewController("/login").setViewName("login");
4  }
```

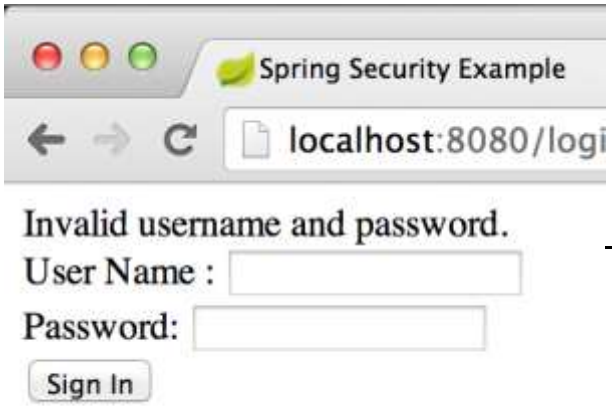
Log In!
Start your server, and try to access the app again. For me, that's simply visiting http://localhost:8080 (http://localhost:8080).



(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-30-at-4-59-05-pm.png)

I'm immediately challenged.

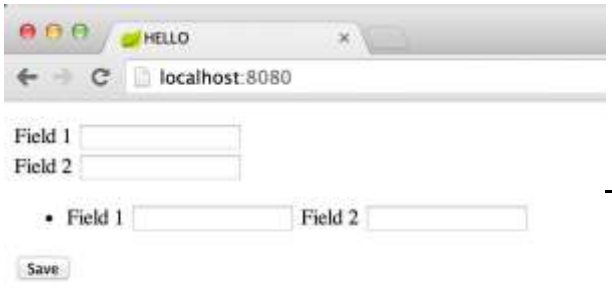
If I put in an invalid username or password, we should see an error:



(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-30-at-4-59-27-pm.png)

If I put in the correct credentials (id: user/password: password), we should be able to log in:

(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-30-at-5-00-33-pm.png)



(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-30-at-5-00-38-pm.png)

There's quite a bit missing here still – let's take this example a bit further – we'll wire in components that would make this configuration closer to production ready – an AuthenticationManager backed by JDBC, configurable password encoders, and a UserDetailsService implementation that we can use to manage users.

Beyond The Examples

To begin taking steps closer to this solution being production-ready, we first need to back our app with a database. I'll be using MySQL (http://www.mysql.com/). I'll assume you've got it installed and running (if you're on a mac, I'd use Homebrew (http://brew.sh/) to accomplish that.

First, we'll add the MySQL dependency to our gradle script:

/build.gradle:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.springframework.boot:spring-boot-starter-security")
    compile("org.springframework.boot:spring-boot-starter-data-jpa")
    compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")
    runtime('mysql:mysql-connector-java:5.1.6')

    testCompile("junit:junit")
}
```

Configuring A Datasource
I'll be calling my schema in MySQL "beyond-the-examples". I'll assume you've used the same name. Conveniently, Spring Boot Starter projects have an automatically configured property source path (http://spring.io/blog/2013/10/30/empowering-your-apps-with-spring-boot-s-property-support). This means that using a properties file for configuration data we'd like to externalize simply requires creating an "application.properties" file and putting it somewhere on the application's classpath. We'll create that file now, and add properties that we'll use to set up our datasource.

/src/main/resources/application.properties:

```
spring.datasource.url=jdbc:mysql://localhost:3306/beyond-the-examples
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.jpa.hibernate.dialect= org.hibernate.dialect.MySQLInnoDBDialect
spring.jpa.generate-ddl=false
```

You can see I'm using the default configuration for MySQL. I wouldn't recommend that for production.

Next, we'll build references to these properties in our application's configuration, so that we can use them to create a datasource bean that we can inject into our security configuration. Update the application configuration file to add these properties:

/src/main/java/com.rodenbostel.sample.Application.java:

```
@Value("${spring.datasource.driverClassName}")
private String databaseDriverClassName;

@Value("${spring.datasource.url}")
private String datasourceUrl;

@Value("${spring.datasource.username}")
private String databaseUsername;

@Value("${spring.datasource.password}")
private String databasePassword;
```

Next create a Datasource @Bean using these properties in the same file.

/src/main/java/com.rodenbostel.sample.Application.java:

```
@Bean
public DataSource datasource() {
    org.apache.tomcat.jdbc.pool.DataSource ds = new org.apache.tomcat.jdbc.poo
    ds.setDriverClassName(databaseDriverClassName);
    ds.setUrl(datasourceUrl);
    ds.setUsername(databaseUsername);
    ds.setPassword(databasePassword);

    return ds;
}
```

Now, we have a datasource configured that we can @Autowire into any of our Spring beans, configuration or otherwise.

Create the Spring Security Tables

The DDL from the Spring.io docs is for HSQLDB (http://docs.spring.io/spring-security/site/docs/3.0.x/reference/appendix-schema.html). It's syntax is not compliant with MySQL. Shout out to this guy (http://springinpractice.com/2010/07/06/spring-security-database-schemas-for-mysql (http://springinpractice.com/2010/07/06/spring-security-database-schemas-for-mysql)) for publishing the MySQL versions of the default Spring Security schema. If you're using MySQL like me, use the DDL from that blog to create a "users" table and an "authorities" table, then thank him. Since we'll be properly encoding our passwords, we may want to make that password column a bit wider. Here's what I ran:

```
create table users (    username varchar(50) not null primary key,    password
```

Building The New Configuration

To start using the new datasource in the security configuration, we first need to wire the datasource bean into our SecurityConfiguration class. Update your SecurityConfiguration file to instruct spring to @Autowire this bean:

/src/main/java/com.rodenbostel.sample.SecurityConfiguration.java:

```
@Autowired
private DataSource datasource;
```

Next, we're going to make a few significant changes to our AuthenticationManagerBuilder configuration to reference this datasource and a few other things, which I'll review after the code:

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Excepti
        JdbcUserDetailsManager userDetailsService = new JdbcUserDetailsManager
        userDetailsService.setDataSource(datasource);
        PasswordEncoder encoder = new BCryptPasswordEncoder();

        auth.userDetailsService(userDetailsService).passwordEncoder(encoder);
        auth.jdbcAuthentication().dataSource(datasource);

        if(!userDetailsService.userExists("user")) {
            List<GrantedAuthority> authorities = new ArrayList<GrantedAuthorit
            authorities.add(new SimpleGrantedAuthority("USER"));
            User userDetails = new User("user", encoder.encode("password"), au

            userDetailsService.createUser(userDetails);
        }
    }


Prior to this, our AuthenticationManagerBuilder was configured on a single lin


auth.inMemoryAuthentication()
```

to using:

```
auth.jdbcAuthentication().dataSource(datasource);
```

Assuming there are already users in the database, believe it or not, that's all we need to begin using the JDBC-backed AuthenticationManager. The requirement for creating new users and managing existing users is a foregone conclusion. In our case, we'd like to automatically configure a default user on app startup just like we were before. We can get a handle on the automatically configuration (by Spring Boot) UserDetailsService through our AuthenticationManagerBuilder at:

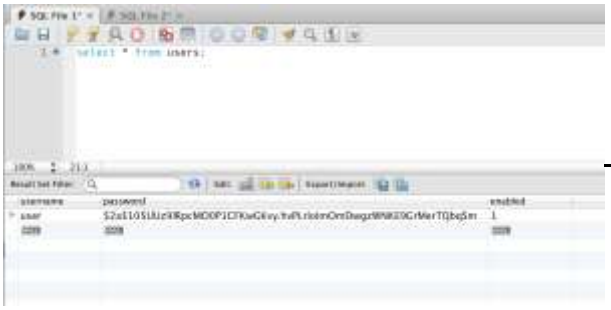
```
auth.getDefaultUserDetailsService();
```

…but that doesn't quite do everything we need. On the first line of our updated AuthenticationManagerBuilder configuration method, you can see we've created a new instance of one of the provide implementations of UserDetailsService provided by Spring. If you don't have a reason to customize how you manage users in your system, that is a perfectly suitable implementation, but there are things to consider. Please consult the API docs for more detail (http://docs.spring.io/spring-security/site/docs/3.2.4.RELEASE/apidocs/org/springframework/security/provisioning/JdbcUserDetailsManager.html (http://docs.spring.io/spring-security/site/docs/3.2.4.RELEASE/apidocs/org/springframework/security/provisioning/JdbcUserDetailsManager.html)). After creating the new reference to the JdbcUserDetailsManager, we need to set a reference to our datasource on it. Following that, we add our encoder for storing our passwords securely, and then we use the JdbcUserDetailsManager's built-in functionality to check to see if our test user exists, and create him if he doesn't.

Testing Again

Running the application should yield no change in behavior when compared with what we saw earlier. This is desired. What we will see that's different will be in our database. Startup the app using: "gradle

bootRun", and using your favorite database management tool, query the database to see our newly create user and their encoded password:

 (https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-30-at-5-44-38-pm.png)

Conclusion
I cobbled the information in this post from many sources - some I've remembered and have mentioned, and others I have not. I hope putting this information in a single post helps whoever stumbles upon it! That concludes this series of Spring Boot posts, but during the time I've been writing these, I've come up with two more topics to touch on, mostly surrounding further securing your app (http://www.jasypt.org/ (http://www.jasypt.org/)) and easier maintenance of your database tables (http://flywaydb.org/ (http://flywaydb.org/)). Check back soon!

Posted in Spring and tagged Redpoint, Spring, Spring MVC, Spring Security, Spring-Boot, Tutorial on May 30, 2014 by Justin. 15 Comments

Part 4: Internationalization in Spring Boot

One of the many features provided by Spring Boot's automatic configuration is a ResourceBundleMessageSource. This is the foundation of the support for internationalization provided by Spring as part of Spring Boot. However, resolving the user's current locale, and being able to switch locales on demand is a little trickier, and is something that I had a hard time finding a comprehensive example of. In this installment, we'll cover how to set up a resource bundle for messages, how to name them to support locales, and finally, how to wire a LocaleResolver and LocaleChangeInterceptor into our test application so that we can implement and test Spring's internationalization support.

As usual, for this installment, I've created a copy of the code from Part 3 and created a new project called Part 4. It's committed to Github (https://github.com/jrodenbostel/beyond-the-examples), ready for cloning.

The Resource Bundle
First, we'll create a resource bundle with message files for English and Spanish in our sample application. When provided a locale, the auto-configured message source can dynamically look up a message file using a default base file name – for example, the default messages file will be named "messages.properties", whereas the file for Spanish messages will be called "messages_es.properties". Likewise, the file for English language messages will be "messages_en.properties". Basically, the

message source resolves the file name in which to look for message properties by concatenating Spring's default base file name "messages", with an underscore and the locale name. The default location for these files, as specified by the auto-configuration, is "/src/main/resources".

Create two files in /src/main/resources: messages_en.properties, and messages_es.properties.

We'll add a couple of messages to each file, and we'll use them to change the labels on our sample application's based on the client's locale.

Update "/src/main/resources/messages_en.properties", adding the following properties and values:

/src/main/resources/messages_en.properties:

```
1  field1 = Field 1
2  field2 = Field 2
```

and likewise, for our Spanish clients, update "src/main/resources/messages_es.properties" to include the Spanish versions of the same properties and values (note that I do not speak Spanish. I typed these names into Google Translate. I think they are accurate enough for this example)

/src/main/resources/messages_es.properties:

```
1  field1 = El Campo 1
2  field2 = El Campo 2
```

Updating The View Template
The view template will need to be updated with placeholders containing the keys that Thymeleaf can use to swap in values from our message files. We've seen usages of ${} and *{} in these tutorials. The next one we'll use is #{}, which is the placeholder Thymeleaf uses to bind messages, and can be used to do general string manipulation within the view.

Update our view template to include two new placeholders such that:

/src/main/resources/hello.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
<head lang="en">
    <meta charset="UTF-8" />
    <title>HELLO</title>
</head>
<body>
<p th:text="${message}"></p>
<form id="gizmo-form" role="form" th:action="@{/save}" method="post" th:object
    <div>
        <label for="field1" th:text="#{field1}"></label>
        <input type="text" id="field1" name="field1" th:field="${gizmo.field1}
    </div>
    <div>
        <label for="field2" th:text="#{field2}"></label>
        <input type="text" id="field2" name="field2" th:field="${gizmo.field2}
    </div>
    <div>
        <ul>
            <li th:each="item, stat : *{children}" class="itemRow">
                <div>
                    <label th:for="${'childField1-'+stat.index}">Field 1</labe
                    <input type="text" class="form-control quantity" name="chi
                           th:field="*{children[__${stat.index}__].childField1

                    <label th:for="${'childField2-'+stat.index}">Field 2</labe
                    <input type="text" class="form-control quantity" name="chi
                           th:field="*{children[__${stat.index}__].childField2
                </div>
            </li>
        </ul>
    </div>
    <div>
        <button type="submit">Save</button>
    </div>
</form>
</body>
</html>
```

You can see we've removed the label text (formerly "Field 1" and "Field 2" for the 'field1' and 'field2' properties on the Gizmo object and added new th:text references containing our new binding types. In these cases, our goal will be to replace #{field1} with the value of the "field1" property in the appropriate messages file, and to replace the value of #{field2} in a similar fashion. If you were to start the server at this point, the solution would still not work. The last step ties the views to the message files.

Configuring The LocaleResolver

In order for Spring to know which message file's values to make available to Thymeleaf, the application needs to be able to determine which locale the application is currently running in. For this, we need to configure a LocaleResolver (http://docs.spring.io/spring/docs/4.0.0.RC1/javadoc-api/org/springframework/web/servlet/LocaleResolver.html).

In 'src/main/java/com.rodenbostel.sample.Application.java' file, configure a new LocaleResolver bean.

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver slr = new SessionLocaleResolver();
    slr.setDefaultLocale(Locale.US);
    return slr;
}
```

This will configure a locale resolver that fits the following description, according to the Spring API:

"Implementation of LocaleResolver that uses a locale attribute in the user's session in case of a custom setting, with a fallback to the specified default locale or the request's accept-header locale"

The only item remaining is how to switch the locale without updating the configuration of the bean above and restarting our server or changing our default or accept-header specified locale.

Configuring a LocaleChangeInterceptor
Configuring an interceptor that is responsible or swapping out the current locale allows for easy testing by a developer, and also gives you the option of including a select list in your UI that lets the user pick the locale they prefer. Add the following bean to your 'src/main/java/com.rodenbostel.sample.Application.java' file:

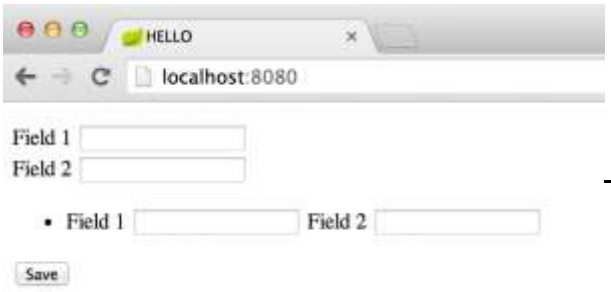
```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("lang");
    return lci;
}
```

As you can see, this interceptor will look for a request parameter named 'lang' and will use its value to determine which locale to switch to. For example, adding 'lang=en' to the end of any request will render the messages from default English locale's message file. Changing that parameter to 'lang=es' will render the Spanish version. For any interceptor to take effect, we need to add it to the application's interceptor registry. In order to do that, we need to get a handle and override Spring Boot Web's addInterceptor configuration method.

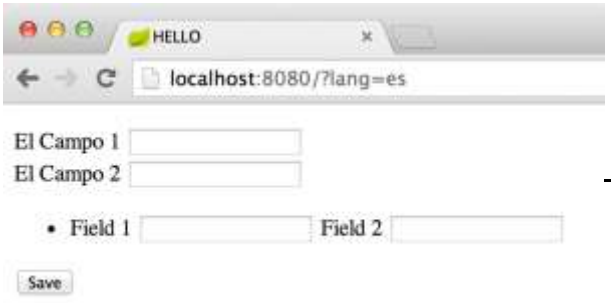
In order to do that, we need to update our 'src/main/java/com.rodenbostel.sample.WebApplication.java' file to extend WebMvcConfigurerAdapter (http://docs.spring.io/spring/docs/4.0.0.M1/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurerAdapter.html). The WebMvcConfigurerAdapter's existence is based on around activities such as this. It provides hooks to override base Spring configuration. Let's update the class to extend this super class and add our interceptor:

```java
package com.rodenbostel.sample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapt
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

import java.util.Locale;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application extends WebMvcConfigurerAdapter {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver slr = new SessionLocaleResolver();
        slr.setDefaultLocale(Locale.US);
        return slr;
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
        lci.setParamName("lang");
        return lci;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }

}
```

Testing
Start your server and observe the default locale in action:



(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-13-at-5-20-00-pm.png)
then, add our request parameter and refresh the page to change the locale so that our Spanish language file is loaded:



(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-13-at-5-20-52-pm.png)
As Spring Boot matures, I'm sure the available examples will be updated to include end to end samples such as this. In the mean time, I hope you found this simple example useful. Check back soon for Spring Security integration in Part 5.

Posted in Spring and tagged Internationalization, Locale Support, ResourceBundleMessageResource, Spring, Spring MVC, Spring-Boot, Thymeleaf, Tutorial on May 13, 2014 by Justin. 22 Comments

Part 3: Form Binding

Welcome back. Now that we have Thymeleaf view templates rendering in our Spring Boot test app, the next logical topic is learning how to bind java objects to forms in these views for processing. In this installment, we'll cover binding a simple object to a form and then take a step deeper and bind an object with a list of children to a form. These are basic examples, but should show enough to get you started in most cases, and should be step beyond what's available in the examples from Thymeleaf and Spring.

As usual, for this installment, I've created a copy of the code from part 2 and created a new project called Part 3. It's committed to Github (https://github.com/jrodenbostel/beyond-the-examples), ready for cloning.

The Gizmo
To get started, we'll create a simple object that will eventually back a form in our app. We'll call it gizmo, and we'll give it two String fields with accessors and mutators.

/src/main/java/com.rodenbostel.sample.Gizmo.java:

```
1  package com.rodenbostel.sample;
2
3  public class Gizmo {
4
5      private String field1;
6      private String field2;
7
8      public String getField1() {
9          return field1;
10     }
11
12     public void setField1(String field1) {
13         this.field1 = field1;
14     }
15
16     public String getField2() {
17         return field2;
18     }
19
20     public void setField2(String field2) {
21         this.field2 = field2;
22     }
23 }
```

Passing Gizmo To The View
Next, we'll make a small change to our SampleController that will allow us to pass a gizmo instance to the browser in place of our previous message.

For this, change the previous version:

src/main/java/com.rodenbostel.sample.SampleController.java:

```
1  @RequestMapping("/")
2  public String index(Model model) {
3      model.addAttribute("message", "HELLO!");
4      return "hello";
5  }
```

to this:

```
1  @RequestMapping("/")
2  public String index(Model model) {
3      model.addAttribute("gizmo", new Gizmo());
4      return "hello";
5  }
```

Updating The View
Next, in place of displaying static text, we'll update our view template so that it contains a form to which we can bind the instance of Gizmo. I'll walk through the important bits after the code sample.

src/main/resources/hello.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
<head lang="en">
    <meta charset="UTF-8" />
    <title>HELLO</title>
</head>
<body>
<form id="gizmo-form" role="form" th:action="@{/save}" method="post" th:objecI
    <div>
        <label for="field1">Field 1</label>
        <input type="text" id="field1" name="field1" th:field="${gizmo.field1}
    </div>
    <div>
        <label for="field2">Field 2</label>
        <input type="text" id="field2" name="field2" th:field="${gizmo.field2}
    </div>
    <div>
        <button type="submit">Save</button>
    </div>
</form>
</body>
</html>
```

Notice we've create a form and dropped in two "th" attributes – action and object. The th:action attribute signifies to what path the form's contents will be submitted, and th:object tells Thymeleaf where to find the object to be bound in the response parameters – in our case, in the Model we're returning with every controller method call.

Also notice the body of the form – we're using the th:field attribute on the input elements we've created. This tells Thymeleaf which fields on the th:object to bind to these inputs.

Last, you may have realized that we haven't create a method in our controller with an @ResponseMapping that will wire to the th:action we've specified… We need somewhere to post this form to.

Adding The Controller Method
Add a new method to your SampleController that will respond to a post issued from our new form

/src/main/java/com.rodenbostel.SampleController:

```
@RequestMapping("/save")
public String save(Gizmo gizmo) {
    System.out.println(gizmo.getField1());
    System.out.println(gizmo.getField2());
    return "hello";
}
```

Notice that we don't do anything special to bind the incoming request to our Gizmo model object on the server side. If the properties on the bound object match the request parameter structure, the request will bind automatically. Specialized binding can be accomplished using the @RequestParam (http://docs.spring.io/spring/docs/3.0.x/api/org/springframework/web/bind/annotation/RequestParam.html) annotation.

Also notice that we're simply printing out the contents of the Gizmo object and rendering the same view again with the same form contents. Alternatively, you can issue a redirect to the "/" path to clear the form, following the route that is
used when the app is started.

Instead of:

```
1 | return "hello";
```

redirect using:

```
1 | return "redirect:/";
```

Start up your server to view the work we've done so far. You should see an empty form that looks similar to this:

(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2014-05-12-at-5-25-52-pm.png)

and when you complete the form and submit it, you should see output in your console similar to the following:
2014-05-12 17:26:50.780 INFO 23927 — [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 9 ms
hello
world

Binding Child Objects
A common use case involves binding child objects to the same form a parent object is bound to – items on an order, addresses on a user profile, etc. The syntax for accomplishing this in Thymeleaf is a bit messy, mostly due to the fact that this type of binding requires nested, escaped quotation marks to accomplish.

Let's add a new object to the mix, one that will be the child of Gizmo.

/src/main/java/com.rodenbostel.sample.GizmoChild:

```java
package com.rodenbostel.sample;

public class GizmoChild {
    private String childField1;
    private String childField2;

    public String getChildField1() {
        return childField1;
    }

    public void setChildField1(String childField1) {
        this.childField1 = childField1;
    }

    public String getChildField2() {
        return childField2;
    }

    public void setChildField2(String childField2) {
        this.childField2 = childField2;
    }
}
```

Another simple class with simple fields. Next, let's add an iterable collection of these objects to the Gizmo object as a property.

/src/main/java/com.rodenbostel.Gizmo:

```java
package com.rodenbostel.sample;

import java.util.ArrayList;
import java.util.List;

public class Gizmo {

    private String field1;
    private String field2;
    private List<GizmoChild> children;

    public String getField1() {
        return field1;
    }

    public void setField1(String field1) {
        this.field1 = field1;
    }

    public String getField2() {
        return field2;
    }

    public void setField2(String field2) {
        this.field2 = field2;
    }

    public List<GizmoChild> getChildren() {
        if(children == null) {
            children = new ArrayList<GizmoChild>();
        }
        return children;
    }

    public void setChildren(List<GizmoChild> children) {
        this.children = children;
    }
}
```

Updating The View (again)

We'll add a new section to our form so that we can iterate through the collection of GizmoChildren, creating a row of inputs for each element in the collection, and dynamically naming them so they bind correctly on the server side.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
<head lang="en">
    <meta charset="UTF-8" />
    <title>HELLO</title>
</head>
<body>
<form id="gizmo-form" role="form" th:action="@{/save}" method="post" th:objecI
    <div>
        <label for="field1">Field 1</label>
        <input type="text" id="field1" name="field1" th:field="${gizmo.field1}
    </div>
    <div>
        <label for="field2">Field 2</label>
        <input type="text" id="field2" name="field2" th:field="${gizmo.field2}
    </div>
    <div>
        <ul>
            <li th:each="item, stat : *{children}" class="itemRow">
                <div>
                    <label th:for="${'childField1-'+stat.index}">Field 1</labe
                    <input type="text" class="form-control quantity" name="chi
                           th:field="*{children[__${stat.index}__].childField1
                    <label th:for="${'childField2-'+stat.index}">Field 2</labe
                    <input type="text" class="form-control quantity" name="chi
                           th:field="*{children[__${stat.index}__].childField2
                </div>
            </li>
        </ul>
    </div>
    <div>
        <button type="submit">Save</button>
    </div>
</form>
</body>
</html>
```

You can see the unordered list's first child has a "th:each" attribute. That's Thymeleaf's comprehension mechanism for collections. The first two parameters – item and stat basically represent the current item in the collection and the state of the iterator at the time the element is being rendered. We can use this stat variable to get the index of the current item being displayed. We also see "*{children}". This is slightly different from earlier examples and shows how Thymeleaf can scope binding to a parent object. In this case, our form is bound to gizmo, and we can reference it's property named 'children' by using an asterisk instead of a dollar sign.

For each item in the collection, we're creating two input fields with ids of childField1-[index] and childField2-[index]. You'll also see the "th:field" attribute used again on these new inputs. In this case we're building the binding names for these fields dynamically. It's easiest explained by looking at the generated html.

Start your server and let's look at the output.

The Rendered Product
You should see a form similar to the following after you start your server.

(https://justinrodenbostel.files.wordpress.com/2014/05/screen-shot-2015-01-02-at-9-50-20-am.png)

And when we view the source for the same page, we should see something similar to this within our form:

```
<form id="gizmo-form" role="form" method="post" action="/save">
    <div>
        <label for="field1">Field 1</label>
        <input type="text" id="field1" name="field1" value="" />
    </div>
    <div>
        <label for="field2">Field 2</label>
        <input type="text" id="field2" name="field2" value="" />
    </div>
    <div>
        <ul>
            <li class="itemRow">
                <div>
                    <label for="childField1-0">Field 1</label>
                    <input type="text" class="form-control quantity" name="chi

                    <label for="childField2-0">Field 2</label>
                    <input type="text" class="form-control quantity" name="chi
                </div>
            </li>
        </ul>
    </div>
    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

You can see the names produced by Thymeleaf fit the standard Spring MVC compliant naming for child objects. They are identified uniquely as they would be an plain object notation.

Let's update our controller to add a GizmoChild instance to our Gizmo on initial request, and print out our new child's output on response:

/src/main/java/com.rodenbostel.sample.SampleController.java:

```java
package com.rodenbostel.sample;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class SampleController {

    @RequestMapping("/")
    public String index(Model model) {
        Gizmo gizmo = new Gizmo();
        gizmo.getChildren().add(new GizmoChild());
        model.addAttribute("gizmo", gizmo);
        return "hello";
    }

    @RequestMapping("/save")
    public String save(Gizmo gizmo) {
        System.out.println(gizmo.getField1());
        System.out.println(gizmo.getField2());
        for(GizmoChild child : gizmo.getChildren()) {
            System.out.println(child.getChildField1());
            System.out.println(child.getChildField2());
        }
        return "redirect:/";
    }
}
```

As you can see, in our index method, we're adding a child object to our Gizmo instance prior to adding to our model for display on the screen. Then, in the 'save' method, we're looping through the collection of children and printing the output to our console. With our server running, by completing the form and submitting, we should see first our Gizmo's properties, followed by the values for the fields on each child object.

Conclusion

Although this is a simplified example, this can easily be expanded to accommodate more complex use cases. Maybe you're reading your Gizmos from a database, or maybe you need to create new Gizmos from scratch. The child collection needs to be instantiated for the Thymeleaf template to render correctly – it can be empty though. Having one child element in a collection in a new Gizmo object gives Thymeleaf a blank object to render. However, the fields Thymeleaf renders for that blank object could be created by jQuery (or another javascript library), and can be further added/removed from there.

Posted in Spring and tagged Redpoint, Spring, Spring MVC, Spring-Boot, Thymeleaf, Tutorial on May 12, 2014 by Justin. 11 Comments

Part 2: Adding Views Using Thymeleaf (and JSP if you want)

In Part 1 (https://justinrodenbostel.com/2014/04/08/part-1-getting-started-again-build-a-web-app-with-spring-boot/) of this series, we set up a sample application using Spring Boot and some of it's default configuration options. Now that we've got the shell of an application, we can start adding content to our application. By adding a single dependency to our project and using Spring Boot's auto configuration settings, we get a free Thymeleaf (http://www.thymeleaf.org/index.html) configuration that's ready to go.

For this installment, I've created a copy of the code from part 1 and created a new project. It's committed to Github (https://github.com/jrodenbostel/beyond-the-examples), ready for cloning.

Adding The Dependency

In the dependencies section of your build.gradle file, add the following entry:

```
compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE”)
```

The end result should leave your build.gradle following looking like this:

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-snapshot (http://repo.spring.i
        mavenLocal()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.1.RE
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'war'

jar {

    baseName = 'part-2'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
    maven { url "http://repo.spring.io/libs-snapshot (http://repo.spring.io/li
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.thymeleaf:thymeleaf-spring4:2.1.2.RELEASE")

    testCompile("junit:junit")
}

task wrapper(type: Wrapper) {
    gradleVersion = '1.11'
}
```

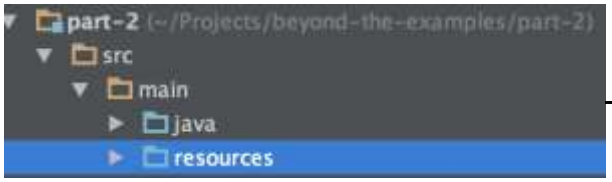
Running a simple 'gradle build' from the CLI will install this new dependency.

Building On Part 1

At the conclusion of Part 1, we had a single controller in our project configured as a @RestController, which returned a simple string. To start using our new Thymeleaf installation, we need to create a view, and make minor changes to our controller.

The auto-configured Thymeleaf install looks for Thymeleaf templates to be available as resources on the classpath in a folder named 'templates'.

Create a resources folder, and a folder named 'templates' within it, such that your project's directory structure appears similar to the following:



(https://justinrodenbostel.files.wordpress.com/2014/04/screen-shot-2014-04-15-at-9-51-50-pm.png)

Creating A Template

Thymeleaf templates are intended to be natural – as close to straight HTML as possible, and easily interpreted by browsers and developers alike. Thymeleaf is packed with several dialects, and takes the place of JSP in the default stack provided by Spring.

Inside of our new templates folder (/src/main/resources/templates/), we need to create a new template that we'll use as our first view. Create a file named "hello.html" that contains the following text:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml (http://www.w3.org/1999/xhtml)" xmlr
<head lang="en">
    <meta charset="UTF-8" />
    <title>HELLO</title>
</head>
<body>
<p th:text="${message}"></p>
</body>
</html>
```

There are two important parts of this simple example: First, in the html tag, you'll see two new attributes that set up template to be processed by Thymeleaf. This will give the page access to use the family of "th" attributes in html tags, thereby having access to the features available in the basic Thymeleaf dialect. Second, in the body of the page, there is a "p" tag with a simple "th" attribute. We'll see the usage of this shortly.

Modifying The Controller

As previously mentioned, our @RestController must change to return a processed view template instead of a simple String, as it does now. First, open our existing controller class, SampleController.java, and change the marker interface from @RestController on it so that it reflects a simple controller

(@Controller).

```
@Controller
public class SampleController {
```

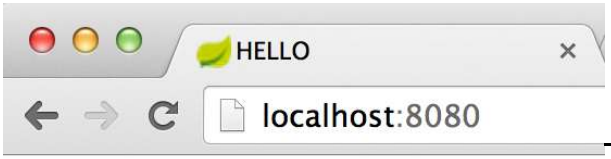
In order to provide our Thymeleaf template with data to display, we need to change our method's signature to accept a Model, which we can then begin to populate. Model is much like it was in previous versions of Spring MVC – an arbitrary collection of java objects that can be displayed and bound to views. The framework will provide our controller method with an instance of a Model object for us populate. Change the "index" method on the SampleController to have the following signature:

```
public String index(Model model) {
```

Next, we'll populate the "message" attribute of the Model instance so that we can see our message in our newly created view. Update the SampleController to do just that:

```
@RequestMapping("/")
    public String index(Model model) {
        model.addAttribute("message", "HELLO!");
        return "hello";
    }
```

Start your server to see our new page in action.



(https://justinrodenbostel.files.wordpress.com/2014/04/screen-shot-2014-04-15-at-10-53-53-pm.png)

HELLO!

This is a very basic example, but you can see that the "${message}" element in our Thymeleaf template has been replaced by the element that we put in the attribute named "message" in our UI Model in our controller. Look at the page source to see the end result.

A Step Further

Let's say you want to change the path to templates or revert to JSPs. The settings provided by Spring Boot can be easily overridden by configuring an InternalResourceViewResolver in our Application.java config file. In the example below, you can see a change to the template path as well as a change to the template technology being used.

```
@Bean
    public InternalResourceViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolve
        resolver.setPrefix("WEB-INF/jsp/");
        resolver.setSuffix(".jsp");
        resolver.setViewClass(JstlView.class);
        return resolver;
    }
```

Of course, if you've changed to JSP, you'll also have to add the appropriate dependencies to your build.gradle file. Both of the dependencies in question are available with embedded Tomcat, so they'll be 'providedRuntime' dependencies.

```
providedRuntime("javax.servlet:jstl")
providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
```

Conclusion

Hopefully, this installment sheds additional light on the basic configuration of Thymeleaf and points you in the correct direction if you need to configure your application beyond what is provided out of the box by Spring Boot Web. Next up, we'll review objecting binding and validation in Thymeleaf, getting deeper into the details of the Thymeleaf dialects and the functionality they provide. Again, find the complete source code for Part 2 on Github (https://github.com/jrodenbostel/beyond-the-examples)!

Posted in Spring and tagged Redpoint, Spring, Spring MVC, Spring-Boot, Thymeleaf, Tutorial on April 16, 2014 by Justin. Leave a comment

Part 1: Getting Started (again): Build A Web App With Spring Boot

As the latest in a long line of enhancement to the Spring framework aimed at eliminating boilerplate code and decreasing the time spent by developers setting up a new project, Spring Boot serves its purpose well. In this installment, I'll walk through getting a project off the ground, rehashing much of what is covered in the first Spring-Boot starter guide on the Spring.io site (http://spring.io/guides/gs/spring-boot/). This will serve as the basis for upcoming installments. In this series, I'll be using Java 1.7, Gradle 1.11, and Spring Boot 1.0.1.

Setting Up

There are Spring Boot starter versions of several different projects: web projects, data projects, integration projects, etc. Here we'll be starting with the Spring Boot web project.

The first step is configuring your project's directory structure. Spring Boot, whether you're using Gradle or Maven, both follow the classic convention for directory naming and source structure. Create an example project using the structure below, substituting values in square brackets with your own:

[project-name]/src/main/java/[package-name]

In the root of our project, we'll create a new build.gradle file:

[project-name]/build.gradle:

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-snapshot (http://repo.spring.i
        mavenLocal()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.1.RE
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'

jar {
    baseName = 'getting-started'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
    maven { url "http://repo.spring.io/libs-snapshot (http://repo.spring.io/li
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")

    testCompile("junit:junit")
}


task wrapper(type: Wrapper) {
    gradleVersion = '1.11'
}
```

Several items to mention here:

The 'buildscript' closure sets up the build itself. This is where the plugins used by the build are declared, along with the repositories in which they can be found.

Following that, there are several plugin applications.

- The java plugin provides the basic tasks related to a simple java project – compiling and packaging, among others.
- The eclipse and idea plugins allow project files to be created for eclipse and Intellij, respectively.
- The spring-boot plugin contains tasks that build executable jars and execute them using embedded tomcat using tasks
  like 'bootRun', which we'll use often in these tutorials.

Next the 'repositories' closure is used to declare where the project's dependencies will be found.

After that, there is a 'dependencies' closure that contains the jars required by the project

Last, there is a standard gradle pattern – the task wrapper – which is used to enforce the version of gradle being used on the project, as well as allow easy execution of the build by users who do not have gradle installed. More information on that found here (http://www.gradle.org/docs/current/userguide/gradle_wrapper.html).

Application Configuration

In the source root of our project ([project-name]/src/main/java/[package-name]), create a file called 'Application.java'. This file mirrors the one found in the first Spring-Boot tutorial, but since it serves as the foundation for upcoming tutorial sections, I'll call out a few things already explained on the Spring.io site (http://spring.io/guides/gs/spring-boot/):

[project-name]/src/main/java/[package-name]/Application.java:

```
1  package com.rodenbostel.sample;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5  import org.springframework.context.annotation.ComponentScan;
6  import org.springframework.context.annotation.Configuration;
7
8  @Configuration
9  @EnableAutoConfiguration
10 @ComponentScan
11 public class Application {
12
13     public static void main(String[] args) {
14         SpringApplication.run(Application.class, args);
15     }
16
17 }
```

You'll notice two major components of this file: the annotations, and the main method.

The annotations are marker interfaces used to alert the framework that this is a Spring config file (@Configuration), that you'd like to scan for beans to load in the current and child packages (@ComponentScan) and that you'd like to use auto configuration settings (@EnableAutoConfiguration). Auto configuration in Spring Boot takes the concept of convention over configuration to an almost Rails or Grails-like level. It provides basic configuration of an application – where to find properties files, how properties files are named when using Spring Profiles, configuration a DispatcherServlet (note the lack of web.xml), and much more. It is worth it to look further into what auto configuration is providing to insure your application does not conflict with it, especially if you're planning on deploying to containers that also provide libraries on the class path by default (I'm looking at you, Websphere).

The main method simply allows the application to be executed from the command line. This includes the startup of your embedded container (in our case, Tomcat). It's worth mentioning here that the return type of SpringApplication.run is an ApplicationContext object which can be further manipulated.

Creating a Controller

Again using the similar source to that of the Spring Boot demos on the Spring.io site (http://spring.io/guides/gs/spring-boot/), we'll begin developing our app using a sample controller configuration as a RestController to verify our app is functioning correctly. In our source root ([project-

name]/src/main/java/[package-name]), create a file called 'SampleController.java'.

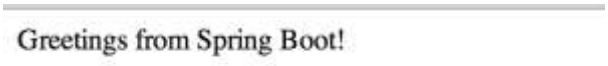
[project-name]/src/main/java/[package-name]/SampleController.java:

```
1  package com.rodenbostel.sample;
2
3  import org.springframework.web.bind.annotation.RequestMapping;
4  import org.springframework.web.bind.annotation.RestController;
5
6  @RestController
7  public class SampleController {
8
9      @RequestMapping("/")
10     public String index() {
11         return "Greetings from Spring Boot!";
12     }
13
14 }
```

Here we are using constructs familiar since Spring MVC 3.5-ish. The RestController interface declares just that – a RESTful controller. This controller will return strings instead of views by default. The RequestMapping annotation on the only method in this class should be familiar, too. This calls for the controller to respond to requests at "/" by executing the index() method. In this case, we're responding with a simple String.

Execution

For this introduction, and the tutorials that follow, I'll be using 'gradle bootRun' to execute the app. Upon executing 'gradle bootRun', in a browser, visit 'http://localhost:8080/', and admire you're work. You should be looking at a simple string: 'Greetings from Spring Boot!'.



(https://justinrodenbostel.files.wordpress.com/2014/04/screen-shot-2014-04-07-at-9-21-09-pm.png)

Preparing for Part 2

Again, this post re-hashed most of what is found on the Spring.io site (http://spring.io/guides/gs/spring-boot/) on the same topic. I hope this has included some additional insight that will help get you started. Since this will be the foundation for Parts 2 and beyond, the code up to this point can be found on Github (https://github.com/jrodenbostel/beyond-the-examples). The next installment will start where this post left off. Check back soon for the next installment: Adding views using Thymeleaf.

Posted in Spring and tagged Getting Started, Gradle, Redpoint, Spring, Spring-Boot, Tutorial on April 8, 2014 by Justin. 1 Comment

Beyond The Examples

In the last several years, rapid web application frameworks have been most developers first choice when starting a new project. Rails and Grails have taken the concept of convention of configuration to extremes, eliminating boilerplate code, and allowing developers to create web applications more quickly than ever. The latest incarnation of Spring and its related tools go far in emulating this kind of productivity. Spring Boot and it's related starter projects make creating a Spring application possible with just a few lines of code. With Spring Boot recently hitting it's 1.0 release, and Spring 4 having just been released in December, it's an exciting time to get reacquainted with an old stand-by framework and tool set. The Getting Started projects at Spring.io serve their purpose extremely well, but as they are only intended to get a project up and running, they leave much to be discovered. The individual tutorials are great, but I had a hard time finding something that put it all together into an app similar to what would be required in the real world. I've recently started a new project using Spring Boot and Spring 4 and hope to share some of my experiences to date in a series of posts.

Here are the topics I hope to cover in each part:

Part 1: Getting Started (again): Build A Web App With Spring Boot
Part 2: Adding Views Using Thymeleaf
Part 3: Form Binding
Part 4: Locale Support
Part 5: Spring Security

Posted in Spring and tagged Getting Started, Redpoint, Spring, Spring-Boot on April 8, 2014 by Justin. 1 Comment