**SpringBoot in28minutes**

Spring **Initializr:** el **groupId** es como el package base del proyecto. **artifactId** es como un nombre para el proyecto.

Creamos un proyecto con starter-web y dev-tools. Típicamente en proyectos web tendríamos un empaquetado war para ser desplegado en un servidor como Tomcat. Con SpringBoot el empaquetado es jar y ya incluye embebido un servidor Tomcat para ejecutar la aplicación.

Vemos que el proyecto creado con Initializr tiene su pom que hereda de spring-boot-starter-parent. En otros proyectos puede tenerse un pom padre distinto.

SpringBoot starter-web habilita tanto las aplicaciones web como los servicios RESTful.

Maven automáticamente descarga todas las dependencias incluidas en el pom. Además de estas, existen las dependencias transitivas, que son aquellas de las que dependen las dependencias declaradas explícitamente en el pom.

En la clase que contiene el método para arrancar el proyecto, observamos la anotación **@SpringBootApplication**. Esta anotación nos indica: **@Configuration, @EnableAutoconfiguration, @ComponentScan**.

Primero inicializa el Spring framework (Component Scan), y después inicializa SpringBoot (Auto Configuration).

Ofrece el método:

public static void main(String[] args) {

    SpringApplication.run(NombreDeLaClase.class, args);

}

SpringApplication.run(NombreDeLaClase.class, args) devuelve el context del que se pueden extraer todos los @Bean declarados y los @Component.

En **application.properties** podemos configurar facetas como el nivel de logging, el puerto en que queremos que corra la aplicación, etc.

Step 02: Part 1 First Spring MVC Controller, @ResponseBody, @Controller

Por defecto, los métodos de un @Controller retornan el nombre de una vista. Si un método va a devolver texto u otra cosa, se debe marcar el método como @**ResponseBody**, para que el **ViewResolver** no intente buscar una vista.

Con esta anotación, cualquier cosa que devuelve un método con mapeo la convierte a json o xml para devolverla en la respuesta HTTP (message converters).

Solo si definimos la clase como @Controller, Spring va a considerar los mappings declarados asignados a los métodos. En el arranque se puede ver en el log que relaciona las url con los métodos de la clase.

Para ver en el log información del flujo de peticiones web desde el navegador, habilitamos el nivel DEBUG pero no para todo, porque sería demasiado profuso. Solo para el paquete web de org.springframework:

**logging.level.org.springframework.web=DEBUG**

Step 02: Part 2 Understanding HTTP Request Flow

Guía de preguntas comunes relacionadas con Spring y SpringBoot:

https://github.com/in28minutes/in28minutes-initiatives/blob/master/The-in28Minutes-TroubleshootingGuide-And-FAQ/quick-start.md

Step 03: Demystifying some of the Spring Boot magic

Spring Boot Starter Parent tiene como parent pom **spring-boot-dependencies**. Una de las cosas llamativas e importantes de ese pom es que tiene todos los **números de versión** de las dependencias empleadas. Es un conjunto de versiones de cosas que están probadas para trabajar bien juntas. Eso en spring tiene un nombre: *set of well curated dependencies.*

Este pom tambien tiene configurados plugins, como maven integration para test de integración. Otro plugin de maven para crear jar files, otro para test-unitarios, etc. Tiene la configuración de todos los plugin de maven importantes para una aplicación spring-boot.

Versión de Java también está en el parent pom. Se puede sobrescribir en nuestro pom concreto.

**Spring Boot Starter Web** tiene las dependencias para web app: **spring core, spring mvc, bean, validation, binding, hibernate validator, logging, Jackson, etc**. Todas las dependencias están en starter web. **Embedded Tomcat**: lo recibimos también con spring-starter-web. Core, websocket, EL...

**Dev Tools**: reinicio auto con los cambios. Busca cambios en las carpetas del proyecto, y cuando los detecta reinicia, pero solo recarga las partes dinámicas. Por ejemplo, no revisa las dependencias, pero recompila las clases con cambios. Es mucho más rápido que un reinicio completo de la aplicación.

Step 04: Redirect to Login JSP - @ResponseBody and View Resolver

Java es para la lógica de negocio, pero lo que se muestra pertenece a la vista (html): jsp, etc.

Para usar vistas y tenerlas en una carpeta determinada, y no tener que escribir el nombre de la carpeta, hay que **configurar** un prefijo para el nombre de la vista.

Para no tener que escribir la extensión del archivo (jsp en el ejemplo), tenemos que definir un **sufijo**.

Creamos la carpeta dentro de **src/main/: webapp/WEB-INF/jsp/**

La carpeta webapp es como un nombre por defecto para aplicaciones web, y **Tomcat lo reconoce** y **no necesita** que se le especifique en la parte de prefijo.

**spring.mvc.view.prefix=/WEB-INF/jsp/**

**spring.mvc.view.sufix=.jsp**

Además, hace falta una dependencia más para que Tomcat embebido soporte jsp: **tomcat-embed-jasper**

<dependency>

    <groupId>org.apache.tomcat.embed</groupId>

    <artifactId>tomcat-embed-jasper</artifactId>

    <version>9.0.16</version>

</dependency>

Resultado de lanzar la petición **GET /server/loginpage:**

o.s.web.servlet.DispatcherServlet : **GET "/loginpage",** parameters={} s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped to com.carregal.springbootlockdown.controller.LoginController#getLogin() o.s.w.servlet.view.InternalResourceView : View name 'login', model {} o.s.w.servlet.view.InternalResourceView : Forwarding to **[/WEB-INF/jsp/login.jsp]** o.s.web.servlet.DispatcherServlet : **Completed 200 OK**

Step 05: Show userid and password on welcome page - ModelMap and @RequestParam

El siguiente objetivo es crear una vista con algo de contenido dinámico. Un ejemplo sencillo es mostrar un parámetro que se mande con la petición. Se recoge en el controller como parámetro, y se adjunta al modelo (se pasa como objeto accesible por el modelo, para la renderización del resultado).

En el método del controller, en su firma, se incluye **@RequestParam String nombreParam**. También se incluye el Model, que es como un contenedor de información al que se pueden añadir atributos de forma dinámica para que se renderice la vista haciendo uso de esa información.

El controller controla el flujo. El modelo va a contener la información generada por la lógica de negocio, y es el conductor de esa información para que pueda generarse la vista. Estos son los componentes del patrón MVC.

**Para obtener el Model en SpringMVC ponemos como segundo parámetro del método del controlador ModelMap model, y así el framework inyecta el model en el controller.**

En la vista tenemos que hacer uso del **EL** (expression language). Un primer ejemplo, en el jsp de prueba, es: **${name}**

@RequestMapping("/loginpage")

public String getLogin(@RequestParam(required = false) String name, ModelMap model) {

    model.put("name", name);

    return "login";

}

Step 06: DispatcherServlet and Spring MVC Flow

**DispatcherServlet** recibe la petición http e identifica el controlador adecuado en base a la url solicitada. El controlador ejecuta la lógica de negocio. Después el controlador le devuelve el Model y el nombre de una vista al DispatcherServlet, que identifica la vista mediante el **ViewResolver**, pone el Model disponible para la vista y finalmente la ejecuta. El DispatcherServlet emite entonces la respuesta HTTP.

FRONT CONTROLLER es el DispatcherServlet, configurado automáticamente por el framework.

Step 07: Your First HTML form

Formularios: por defecto el submit va contra la misma url. Se cambia con el atributo **action** del elemento form. Por defecto el método http de submit es GET, con lo que los parámetros se envían en la url, de forma abierta. Se cambia por post usando el atributo **method** del form.

<form method="post" action="login">

    Name: <input name="name" type="text" />

    <input type="submit" />

</form>

Para la petición GET habilitamos este método del controller:

@RequestMapping(value = "/login", method = RequestMethod.GET)

public String getLogin(@RequestParam(required = false) String name, ModelMap model) {

    return "login";

}

Y para la petición POST que se ejecuta con el Submit desde el form, habilitamos otra entrada en el controller, que es la que recibirá los parámetros enviados por el formulario.

@PostMapping("/login")

public String login(@RequestParam String name, @RequestParam String password, ModelMap model) {

    model.put("name", name);

    model.put("password", password);

    return "welcome";

}

Adicionalmente, creamos un jsp llamado "welcome.jsp" que se renderiza con los datos recibidos.

Step 08: Add hard-coded validation of userid and password

Crear un **@Service** dummy para que devuelva true si el user y pwd es un par determinado. Todo para dirigir la petición de login.

@PostMapping("/login")

public String login(@RequestParam String name, @RequestParam String password, ModelMap model) {

    if (loginService.validUser(name, password)) {

        model.put("name", name);

        model.put("password", password);

        return "welcome";

    } else {

        model.put("error", "Usuario o passwod incorrectos");

        return "login";

    }

}

Al usar un framework de inyección de dependencias, no necesitamos crear instancias de servicios, repositorios, componentes, o beans en general. Los gestiona el framework, muchos de ellos serán singleton y algún otro prototype.

Step 09: Magic of Spring

Spring ha facilitado mucho el testing de aplicaciones por su facilidad para hacer elementos mock y crear test unitarios. Antes todo había que probarlo desplegando en un servidor y ejecutando.

**@Component, @Service, @Repository, @Controller, @Bean**

Con todas estas anotaciones le decimos al framework que gestione esas clases y que tenga al menos una instancia disponible en el contexto.

**@Controller** es una especialización de Component para clases de implementación para que sean autodetectadas a la hora de buscar **mapeos** entre url y métodos.

**@Service** se emplean para contener la lógica de negocio. Se pueden mockear para probar controllers.

**@Repository**, data store, guardar datos etc. Se pueden mockear para probar services.

**@Autowired**: solicitar la inyección del *bean* en el atributo de la clase.

**@ComponentScan**: paquete en el que se buscan los bean. Por defecto se buscar en el paquete donde está la **@SpringBootApplication**.

@SpringBootApplication

@ComponentScan(value = { "com.carregal.springbootlockdown", "com.carregal.other" })

public class SpringbootlockdownApplication {

    public static void main(String[] args) {

        SpringApplication.run(SpringbootlockdownApplication.class, args);

    }

}

Step 10: Create TodoController and list-todos view. Make TodoService a @Service

Definimos un pojo Todo y un Dao service al que anotamos como @Service, que maneja una colección en memoria (no bd). La colección se inicia con un bloque estático de código.

Creamos un método de controller para procesar la petición de la lista de todos.

@GetMapping("/list")

public String manageTodosList(ModelMap model) {

    List<Todo> todos = todoService.findAllTodos("daniel");

    model.put("todos", todos);

    return "todos";

}

Todavía echamos a faltar las variables de session, de modo que el usuario para mostrar los *todo* todavía está *hardcoded*.

Creamos una página para mostrar la lista de Todos, todavía sin ningún formato (mediante EL **${todos}**).

Cosas por hacer:

* Dar formato a la lista de TODOS con Bootstrap.
* no se debe poder acceder a la lista de todos sin estar logueado. Al loguearse, acceder a tu propia lista de todos.
* quitar los hardcode, como el de usuario, para obtener los todos.

Capas en una aplicacion JEE:

**Web**: tanto la exposición de servicios como la interacción con UIs web. La **V** y la **C** del patrón MVC están en la **capa web** casi siempre. **Spring MVC, Struts**

**View Tech**: **JSP/EL/JSTL**, Freemarker, Velocity, JSF o exponer **RESTServices** y consumirlos con un frontend **AngularJs**

**Model 1**: Directamente hacia varios **JSP**, que acceden al **model**. La UI apunta a distintos JSP. En esta aproximación los JSP tenían la lógica de controlador y la de vista. Muy complicado.

**Model 2:** El navegador apunta contra **servlets**, que acceden al modelo y crean la vista con los datos. Existen **distintos servlets** a los que hacer peticiones distintas. Esta es la arquitectura JEE sin front controller

**Model 2** **front controller**: Desde el browser siempre se envía la petición a un único **FrontController** (un servlet), que **redirige** la petición a uno de entre todos los **Controllers** de la aplicación.

Step 11: Architecture of Web Applications

En **SpringMVC** el frontController se llama **DispatcherServlet**. Este esquema es el más popular porque permite hacer más sencillo cosas como la **seguridad**, que va a través del frontcontroller. Toda la lógica centralizada se puede añadir a través del FC.

**Business**: lógica del negocio.

**Data**: persistencia

**Integration**: comunicación con otras aplicaciones o APIs

Step 12: Session vs Model vs Request - **@SessionAttributes**

Scopes: session, request, y como se relacionan con el model.

Los **parámetros de una request** sólo están disponibles durante el **ciclo de vida de esa petición** en concreto, y no en las sucesivas.

Como ejemplo, en el flujo que tenemos: login page, welcome page, list todos, si entre login y welcome se pasa name y password en el model, provenientes de la petición, en list-todo ya no están disponibles esos datos, pues la petición http ya no es la misma.

Session es la forma en la que puedes guardar datos para que estén disponibles en multiples peticiones.

Los datos del model también son válidos durante una sola petición, de modo que la forma para solventar esto es la session.

Para poner un valor en la session desde un @Controller hay que **anotar la clase del controlador** con

**@SessionAttributes({"lista","variables","en","session"})**

Para acceder a esos valores guardados en session en los métodos del controller, se hace un **get por nombre sobre el modelo ModelMap**, dando por hecho que existe. Para que exista, debe haberlo puesto "alguien" antes allí.

A esto se le llama **conversational storage**: valores persistidos a través de varias peticiones.

**@Controller @SessionAttributes("name") public class LoginController {**

Hay que tener mucho **cuidado** con lo que se guarda en session, porque imaginemos que tenemos una aplicación con millones de usuarios. Eso sería equivalente a tener en memoria millones de usernames. Algo de conversational state está bien, pero se debe limitar a lo imprescindible únicamente.

Step 13: Add new todo

Añadimos un enlace en la página de listado de todos para crear un nuevo todo, así que:

* Creamos el método en el controller que dirige a la vista adecuada
* Creamos el jsp add-todo.jsp que se retornará cuando se reciba una petición en /add-todo: tiene un form con method POST y **sin action, por lo que va a la misma url, pero por POST**. Si no existe mapping en el controller encontraremos un error de método no soportado.

Request method 'POST' not supported

org.springframework.web.HttpRequestMethodNotSupportedException: Request method 'POST' not supported

Podemos acceder al nombre de usuario desde el propio controller asumiendo que está en session:

@PostMapping("/post-todo")

public String postTodo(ModelMap model, @RequestParam String desc) {

   Todo todo = new Todo(0, (String) model.get("name"), desc, new Date(), false);

   todoService.saveTodo(todo);

   return "redirect:/list";

}

Hay que hacer **redirect** para que se vuelva a producir una petición a través del controller al método adecuado, que pondrá en el Model la información necesaria para renderizar la página. En caso contrario, se muestra el JSP **pero no tiene** los TODOS.

Step 14: Display Todos in a table using JSTL Tags

Haciendo uso de las etiquetas **JSTL**. Hay que añadir una dependencia al proyecto, y como es un proyecto maven, añadimos al pom:

<dependency>

    <groupId>jstl</groupId>

    <artifactId>jstl</artifactId>

</dependency>

En cada jsp hay que añadir el namespace: **<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>**

<tbody>

    <c:forEach var="todo" items="${todos}">

        <tr>

            <td>${todo.id}</td>

            <td>${todo.user}</td>

            <td>${todo.desc}</td>

            <td>${todo.targetDate}</td>

            <td>${todo.isDone}</td>

        </tr>

    </c:forEach>

</tbody>

25. Step 15: Bootstrap for Page Formatting using webjars

Añadiremos **Bootstrap** y **jQuery** al proyecto en forma de **webjars** en vez de archivos estáticos. Esto va en el pom. Es autoconfigurado por SpringBoot:

o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/\*\*] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

Las dependencias para jQuery y Bootstrap son:

<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>bootstrap</artifactId>

    <version>3.3.6</version>

</dependency>

<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>jquery</artifactId>

    <version>1.9.1</version>

</dependency>

o.s.w.s.handler.SimpleUrlHandlerMapping  : Patterns [/webjars/\*\*, /\*\*] in 'resourceHandlerMapping'

En una aplicación Spring habría que configurar el controller que se haría cargo de las peticiones /webjars/\*\*, pero con SpringBoot esto forma parte de la **autoconfiguración**.

Los archivos **js** de **jQuery**y de **Bootstrap**se descargan a una carpeta dentro de resources. Para incluirlos en un jsp, es una buena práctica ponerlos al final de la jsp.

<script src="webjars/jquery/1.9.1/jquery.min.js"></script>

<script src="webjars/bootstrap/3.3.6/js/bootstrap.min.js"></script>

El enlace a la hoja de estilos **css**se pone en el head de la pagina

<link href="webjars/bootstrap/3.3.6/css/bootstrap.min.css" rel="stylesheet">

Las rutas que se ven en los enlaces son las ubicaciones de los archivos dentro de los webjars.

Al añadir a la página del listado de todos los js y el css, si desde la perspectiva developer del **navegador** analizamos la **petición** a la página, vemos que con la **respuesta** html se obtienen también el archivo css y los dos js desde nuestro lado back.

Haremos un pequeño formato de la página: **div.container** conteniendo el contenido de todo el body, poner el link **Add Todo** dentro de un **div** y **a.button** al elemento a href. Asignarle la clase **.table.table-striped** a la tabla.

Step 16: Let's delete a Todo

Borrado de un TODO. Hay que añadir el th en la cabecera, y un td más con un enlace **a type="button"** y **class="btn btn-warning"**.

Pregunta: ¿cómo incluir en la petición de borrar TODO el ID del todo que se quiere realmente borrar? Una forma es **incluirlo en el path de href**. Otra forma es poner la url del href como sigue: **"/delete?id=${todo.id}"**, o sea, pasar el dato como **parámetro query**.

@GetMapping("/delete-todo-param")

public String deleteTodoParam(@RequestParam(name = "id") Integer id) {

    todoService.deleteTodoById(id);

    return "redirect:/list";

}

@GetMapping("/delete-todo-path/{id}")

public String deleteTodoPath(@PathVariable Integer id) {

    todoService.deleteTodoById(id);

    return "redirect:/list";

}

Step 17: Format Add Todo Page and Adding Basic HTML5 form validation

Formateando la página ADD-TODO con bootstrap: tenemos los includes en la jsp. Creamos el div container. No creamos fila ni columna, tenemos en formulario dentro de un div container. Las filas solo valen para crear columnas dentro, si no se van a crear columnas las filas son irrelevantes.

Queremos formatear la página y añadir un poco de html validation. Bootstrap para formularios: todos los campos tienen que estar en lo que se llama **fieldset**. En un fieldset cada elemento tiene su **<label>**

Al fieldset se le añaden class y al input también. Al fieldset se le pone **class="form-group"**. Al **input** se le da **class="form-control"**.

De esta forma se ve el label en negrita, y el campo de form ocupa todo el ancho del container y tiene formato redondeado.

El botón de submit aparece en otra línea, ya no está en la misma que el input. El botón de submit ya **no lo ponemos como un input**, sino como un **button type="submit"**. Vamos a darle **class="btn-success"**, lo que lo muestra coloreado.

<div class="container">

    <form method="post">

        <fieldset class="form-group">

            <label>Description</label>

            <input type="text" name="desc" class="form-control" />

        </fieldset>

        <fieldset class="form-group">

            <button type="submit" class="btn btn-success">Send Todo</button>

        </fieldset>

    </form>

</div>

Validación HTML: poner en el input **required="required"**, y entonces es el navegador el que impide que se haga el submit de los datos si falla la validación. Se muestra un aviso de que el campo debe ser cubierto.

<fieldset class="form-group">

    <label>Description</label>

    <input type="text" name="desc" class="form-control" required="required" />

</fieldset>

Step 18: Part 1 Validations with Hibernate Validator - Using Command Bean

org.apache.jasper.JasperException: /WEB-INF/views/todo.jsp (line: [20], column: [4]) Unable to find setter method for attribute: [commandName] at org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:42) ~[tomcat-embed-jasper-9.0.26.jar:9.0.26]

The solution is to use **modelAttribute** instead of **commandName**.

Lets use a command bean for Todo! **org.hibernate:hibernate-validator**

Veremos ahora JSR303 y JSR349 para las validaciones, y parece que haremos uso de una **taglib** nueva.

Vamos a utilizar algo que se llama **commandBean**, y después otra cosa que se llama **modelAttribute**. Hibernate provee una implementación para esos standards: **hibernate validator**, que no tiene que ver con la implementación de JPA, sino que es otra faceta: **Bean Validation API**. Nos da herramientas para permitir dar valores a un bean (validación).

Implementar validación del lado del servidor: **Form backing bean** (tener un bean que corresponda con los campos del form). Añadir la validación, usar validación en el Controller y mostrar los errores en la vista.

Hasta ahora tenemos un formulario con un solo campo, pero ¿y si tenemos 50? **¿Vamos a declarar un @RequestParam para cada campo?** No tiene mucho sentido. Spring da solución a eso con **CommandBean (form backing bean)**. Podemos mapear lo que ser recibe en la petición directamente con un objeto.

Una vez que tenemos el pojo al que mapear podemos hacer uso de la Validation API y añadir las validaciones sobre los campos el bean.

Se habilita la validación en el Controller. El propio método del Controller enviará el conjunto de errores a la vista a través del **ModelMap**, si **no** se pasa alguna de las validaciones.

Tenemos que usar **spring form tags** en el jsp. Esto para que la **info** del **front** se convierta en **form backing bean** en el **backend**.

<%@taglib uri="http://www.springframework.org**/tags/form**" **prefix="form"**%>

Cambia: <form> **🡪** <form:form method="post" commandName="todo"> (ModelAttribute)

commandName="todo" así es como se llamará el parámetro recibido en el controller.

Al método del controller que devuelve la vista para Add Todo (nuevo *todo*) le hacemos una modificación conveniente: obtenemos el **ModelMap** y establecemos en el un **new Todo()** con **id=0 y valores default**. Uno de los valores **default** es el **user**, que tenemos almacenado en **session**.

De esta manera, cuando se hace submit del formulario se está enviando el todo creado (un objeto nuevo).

@GetMapping("/post-todo")

public String showPostTodo(ModelMap model) {

    model.put("todo", new Todo(0, (String) model.get("name"), "", new Date(), false));

    return "post-todo";

}

@PostMapping("/post-todo")

public String postTodo(ModelMap model, Todo formTodo) {

    todoService.saveTodo(formTodo);

    return "redirect:/list";

}

Cambiamos también las tag de **label** por **form:label**. En el input, hemos de cambiar al atributo **name** por **path**, que contiene el nombre del campo. En los **form:label** hay que poner también el atributo **path** correspondiente.

Más sobre la taglib de spring haciendo control click sobre alguno de los tags. Se llama **Spring Framework JSP Tag Library**.

Para que se pueda hacer el **binding** entre el **form** y el **backend** el POJO debe tener un **constructor sin parámetros por defecto**.

<form:form method="post" commandName="todo">

    <fieldset class="form-group">

        <form:label path="desc">Description</form:label>

        <form:input type="text" path="desc" class="form-control" required="required" />

    </fieldset>

    <fieldset class="form-group"> <button type="submit" class="btn btn-success">Send Todo</button> </fieldset>

</form:form>

Step 18: Part 2 Using JSR 349 Validations

Validaciones, doble binding: **form-bean** y **bean-form**.

Cuando creamos una vista con el model y tenemos un formulario que declara un commandName o un **modelAttribute** (en versiones recientes de SpringBoot).

Validaciones en backend con JSR: Bean validation API.

@Size(min=10, message="Enter at least 10 Characters") para validar la longitud de un campo cadena

En el Controller hay que poner **@Valid** en el parámetro.

¿Cómo sabemos si la validación ha sido buena? **binding result**. **Inyectando** ese objeto en el método de controller, podemos acceder a su atributo **result.hasErrors()**. Volvería a la misma página, **pero entonces en el modelo se habrán inyectado los errores.**

El objeto **BindingResult** se tiene que inyectar en la lista de parámetros del método del controller **(puerta del controller) DESPUÉS del objeto @Valid Todo (modelAttribute).**

<form:errors path="desc" cssClass="text-danger"></form:errors> 🡪 ¿qué html genera?

<form id="todo" action="/post-todo" method="post">

    <fieldset class="form-group"> <label for="desc">Description</label> <input id="desc" name="desc" type="text"

            class="form-control" required="required" value="Default"> <span id="desc.errors" class="text-danger">A

            little too short description</span> </fieldset>

    <fieldset class="form-group"> <button type="submit" class="btn btn-success">Send Todo</button> </fieldset>

</form>

Otras funciones de validación:

1. **DecimalMin 1**
2. **DecimalMax 100**
3. **Max**
4. **Min**
5. **NotNull**
6. **Null**
7. **Past (fechas)**
8. **Future**
9. **Pattern (valores que cumplen regexp)**
10. **Size**

31. Step 19: Updating a todo

Ahora queremos actualizar. Hace falta:

1. botón update 🡪 se va a la página de add-todo.
2. manejar la petición GET (muestra el detalle del todo elegido)
3. manejar la petición POST (guardar los cambios).

Botón, copiando del de delete. Crear el método, que será muy parecido al de borrar. Hace falta el método de getById y el de updateById en el DaoService.

Una cosa que hay que hacer es tener el campo, aunque sea **oculto**, del **id** en el **formulario**. El framework va a tratar de hacer **binding** de todo lo que tenga, pero es necesario para el **update** disponer del campo id.

**<form:hidden path="id"/>**

Cogemos también el **user** como campo **hidden**, lo cogemos de la **session** (del **modelMap**).

Step 20: Let's add a Target Date for Todo - Use **initBinder** to Handle Date Fields

Añadiremos el **targetDate**, tanto para crear un todo como para actualizarlo.

Primero lo añadimos al JSP, o sea que replicamos un fieldset con todo el código (**fieldset.form-group, form:label, form:input, form:errors**)

Hay que setear un formato de fechas en la aplicación: **initBinder** (bean que se inyecta en el controller):

@InitBinder //ver DOC:inicializa org.springframework.web.bind.WebDataBinder

public void initBinder(){

    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/YYYY");

    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

}

Eso hace que para la clase Date, en los *databinding*, use ese formatter.

Para formatear la fecha en el JSP: JSTL format

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

usaremos el <fmt:formatDate value="${todo.targetDate}" pattern="dd/MM/YYYY"/>

No es el mismo formato por defecto que tenemos en la vista de detalle que la que tenemos en la vista de la lista. Ponemos el patrón con el atributo **pattern**. Usamos el mismo de InitBinder.

Pero ahora queremos usar un **datepicker**, no queremos escribir la fecha manualmente.

**webjars, bootstrap datepicker**, es una dependencia nueva.

<dependency>

    <groupId>org.webjars</groupId>

    <artifactId>bootstrap-datepicker</artifactId>

    <version>1.0.1</version>

</dependency>

Hay que añadir el **<script src="webjars/"** del datepicker. Podemos ver el path entrando en dependencias del datepicker y buscar el .css y el .js. Hay que añadir los dos, o al menos el .js

<fieldset class="form-group">

    <form:label path="targetDate">Target date</form:label>

    <form:input type="text" path="targetDate" class="form-control" required="required" />

    <form:errors path="targetDate" cssClass="text-danger"></form:errors>

</fieldset>

…

<script src="webjars/bootstrap-datepicker/1.0.1/js/bootstrap-datepicker.js"></script>

<script>

    $('#targetDate').datepicker({ format: 'dd/mm/yyyy' });

</script>

Hay que configurar el datepicker de dos maneras: a qué campo de formulario va a aplicar y cuál va a ser el formato. Se inicializa en el bloque script sobre este párrafo:

Step 21: Jsp Fragments

Barra superior de navegación haciendo uso de **JSP fragments**. Tenemos que eliminar la duplicidad de código de las áreas comunes de la aplicación: **head**, **body**, **footer**, **javascripts** que añadimos cada vez y que son iguales.

Cogemos el snippet de la página del curso para la cabecera:

<nav role="navigation" class="navbar navbar-default">

    <div class="">

        <a href="http://www.in28minutes.com" class="navbar-brand">in28Minutes</a>

    </div>

    <div class="navbar-collapse">

        <ul class="nav navbar-nav">

            <li class="active"><a href="/login">Home</a></li>

            <li><a href="/list-todos">Todos</a></li>

        </ul>

    </div>

</nav>

Se lo ponemos en principio a listTodos. Aquí hemos observado diferencias entre Bootstrap 3 y 4.

Carpeta **common** para **header.jspf** (la f es de fragment), **footer** y **navigation**. Todo hasta la etiqueta **<body>** incluida lo metemos en **header.jspf.** Todo desde la etiqueta de cierre **</body>** lo metemos en **footer.**

**<%@ include file="common/header.jspf"%>**

**<%@ include file="common/navigation.jspf"%>**

🡨html de la página 🡪

**<%@ include file="common/footer.jspf"%>**

**Ejercicios:**

* Acomodar las páginas **login** y **welcome** (con los fragmentos).
* Realza el elemento correcto en el menú de la cabecera.

Step 22: Preparing for Spring Security

Eliminamos la parte de **login** como la teníamos, de modo que cuando tengamos un formulario de login será el de Spring Security. Ahora la página Home no es **login**, sino la de **welcome**. Establecemos, de momento, en session el usuario de forma *hardcode*.

Sacamos a un método esta línea, porque se usa mucho. Servirá para después refactorizar a Spring Security.

private String getLoggedInUserName(ModelMap model) {

    return (String) model.get("username");

}

Step 23: Setup for Spring Security

Añadimos el starter **spring-boot-starter-security**. Esto provoca que se añada algo de autoconfiguración relacionada y que todas las peticiones a la aplicación tienen el interceptor de seguridad de spring security.

**Ldap** o **base de datos** para tener user id y password. Creamos un package security y la clase **SecurityConfiguration**. Usamos la anotación **@Configuration** más **@Bean**.

Clase **WebSecurityConfgurerAdapter**, de esa vamos a extender para *override* algún método de interés:

* Queremos crear usuarios.
* Queremos definir a qué página redirigir a nuestra propia pagina de login.

Cuando definimos un método en una clase de **@Configuration** que ha de recibir **parámetros** inyectados mediante DI por el framework, los **métodos** se anotan con **@Autowired**.

Copiamos snippet de github in28minutes:

@Autowired

public void configureGlobalSecurity(AuthenticationManagerBuilder auth) throws Exception {

  auth.inMemoryAuthentication().withUser("daniel").password("dani01").roles("USER", "ADMIN");

}

Con este fragmento sobrescribimos el user por defecto que genera spring. Ahora queremos habilitar nuestra propia página de login. Recetita 2. Copiamos otro snippet para eso:

@Override

protected void configure(HttpSecurity http) throws Exception {

    http.authorizeRequests().antMatchers("/login").permitAll().antMatchers("/", "/\*todo\*/\*\*")

            .access("hasRole('USER')").and().formLogin();

}

Este fragmento permite acceder a todos al path /login. Solo permite acceder a los otros paths (antMatchers) a usuarios con el rol de USER, y si no lo tienen los redirige al formulario de login, habilitado por **formLogin()**.

Esta fórmula ha funcionado de una manera irregular. Ha funcionado bien cuando he incluido un @Bean

@Bean

public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

}

y he cambiado a:

@Autowired

public void configureGlobalSecurity(AuthenticationManagerBuilder auth) throws Exception {

    auth.inMemoryAuthentication().withUser("daniel").password(passwordEncoder().encode("dani01")).roles("USER",

            "ADMIN");

}

Además, he configurado las rutas, de momento, así:

@Override

protected void configure(final HttpSecurity http) throws Exception {

    http.csrf().disable().authorizeRequests().antMatchers("/").hasRole("USER").antMatchers("/login\*").permitAll()

            .anyRequest().authenticated().and().formLogin().defaultSuccessUrl("/", true);

}

Es interesante la página de Baeldung al respecto de spring security y login:

**https://www.baeldung.com/spring-security-login**

Step 24: Añadir logoff functionality y usar el usuario logueado con Spring Security para listar *todos*.

En Spring Security, un intento de usuario logueado se llama ***principal***.

SecurityContextHolder.getContext().getAuthentication().getPrincipal()

Puede ser un **UserDetails** cuando ha encontrado un usuario que se corresponde con él. Creamos el método en el Controller. **SecurityContextHolder** es la clase importante.

En los métodos del controller puedo hacer uso de HttpServletRequest y HttpServletResponse.

Para hacer logout tengo que hacer uso de la clase SecurityContextLogoutHandler.

El método **logout(request, response, auth)** necesita tres parámetros que son los que vemos en el siguiente código:

@GetMapping("/logouch")

public String logout(HttpServletRequest request, HttpServletResponse response) {

    Authentication auth = SecurityContextHolder.getContext().getAuthentication();

    if (auth != null) {

        new SecurityContextLogoutHandler().logout(request, response, auth);

    }

    return "redirect:/";

}

He creado un mapping algo distinto porque */logout* ya tiene automáticamente la funcionalidad de deslogado.

Resumen: principal es un intento de login, se obtiene de SecurityContextHolder.getContext().getAuthentication().getPrincipal()

Es un método estático.

Para deslogar hace falta una INSTANCIA de SecurityContextLogoutHandler y de su método **logout**.

Step 25: Exception Handling

La gestión de excepciones es lo que se llama un cross-cutting concern. Tener un montón de try catch no es una buena idea, y hoy en día hay soluciones mejores.

Se puede crear un controller con el el path "error" y luego métodos anotados con

@ExceptionHandler(RuntimeException.class)

public ModelAndView handlerError() {

}

Se pueden pasar los típicos parámetros que le pasarías a cualquier método de controller. Vamos a poblar el hasErrors y devolver una vista, ese es el objetivo.

En un objeto de la clase **ModelAndView** podemos tener detalles del model y de la vista.

Creamos una instancia. Tiene métodos como **addObject**, **setViewName**. Es una forma alternativa a la que hemos usado hasta ahora, en la que solo devolvíamos un nombre de vista. Creamos error.jsp con un simple mensaje.

@Controller("error")

public class ErrorController {

    @ExceptionHandler(RuntimeException.class)

    public ModelAndView handleError(HttpServletRequest request, HttpServletResponse response, Exception e) {

        ModelAndView mv = new ModelAndView();

        mv.addObject("exception", e.getStackTrace());

        mv.addObject("url", request.getRequestURL());

        mv.setViewName("errorpage");

        return mv;

    }

}

No funciona. Supuestamente poniendo "error" como path en en Controller, se informa al framework que es un controlador para manejar los errores, cada uno con @ExceptionHandler…

Moviendo el código al controlador concreto que produce la excepcion **SI** que ha funcionado como esperaba.

package com.carregal.springbootlockdown.controller;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.List;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.propertyeditors.CustomDateEditor;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.WebDataBinder;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.InitBinder;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.SessionAttributes;

import org.springframework.web.servlet.ModelAndView;

import com.carregal.springbootlockdown.dao.TodoDaoService;

import com.carregal.springbootlockdown.entity.Todo;

import com.carregal.springbootlockdown.exception.TodoNotFoundException;

@Controller

@SessionAttributes("name")

public class AppController {

    @InitBinder

    // ver DOC:inicializa org.springframework.web.bind.WebDataBinder

    public void initBinder(WebDataBinder binder) {

        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

    }

    // @Autowired

    // private LoginService loginService;

    @Autowired

    private TodoDaoService todoService;

    // @Autowired

    // private DummyService helloService;

    @RequestMapping(value = "/", method = RequestMethod.GET)

    public String getLogin(ModelMap model) {

        model.put("name", getLoggedUserName());

        return "welcome";

    }

    @GetMapping("/list")

    public String manageTodosList(ModelMap model) {

        List<Todo> todos = todoService.findAllTodos(getLoggedUserName());

        model.put("todos", todos);

        return "todos";

    }

    @GetMapping("/post-todo")

    public String showPostTodo(ModelMap model, @RequestParam(required = false) Integer id) {

        if (id == null) {

            model.put("todo", new Todo(0, getLoggedUserName(), "Default Description", new Date(), false));

        } else {

            Todo todo = todoService.findTodoById(id);

            if (todo == null) {

                throw new TodoNotFoundException("No existe el elemento");

            }

            model.put("todo", todo);

        }

        return "post-todo";

    }

    @PostMapping("/post-todo")

    public String postTodo(ModelMap model, @Valid Todo formTodo, BindingResult result) {

        if (result.hasErrors()) {

            // los errores se inyectan solos en el modelo, no los pongo yo

            return "post-todo";

        }

        todoService.saveTodo(formTodo);

        return "redirect:/list";

    }

    @GetMapping("/delete-todo-param")

    public String deleteTodoParam(@RequestParam(name = "id") Integer id) {

        todoService.deleteTodoById(id);

        return "redirect:/list";

    }

    @GetMapping("/throwit")

    public String produceError() {

        throw new TodoNotFoundException("error a propósito");

    }

    @GetMapping("/delete-todo-path/{id}")

    public String deleteTodoPath(@PathVariable Integer id) {

        todoService.deleteTodoById(id);

        return "redirect:/list";

    }

    private String getLoggedUserName() {

        Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

        if (principal instanceof UserDetails) {

            return ((UserDetails) principal).getUsername();

        } else {

            return principal.toString();

        }

    }

    @ExceptionHandler(RuntimeException.class)

    public ModelAndView handleError(HttpServletRequest request, HttpServletResponse response, Exception e) {

        ModelAndView mv = new ModelAndView();

        mv.addObject("exception", e.getStackTrace());

        mv.addObject("url", request.getRequestURL());

        mv.setViewName("errorpage");

        return mv;

    }

}

**Springboot Advanced, deep dive.**

Step 1: Set up an Maven Project with Eclipse.

Refer https://github.com/in28minutes/SpringMvcStepByStep/blob/master/Step15.md to understand the sort of stuff - web.xml, dispatcher servlet configuration, maven dependency management and plugins - that are need to launch a typical web application without Spring Boot!

Spring Boot does a lot of magic. This magic is called Auto Configuration. We will discuss about different terms related to Spring Boot - Starter Parent, Starter projects, Auto configuration - in depth during our first 10 steps.

Hay un plugin en el pom de los proyectos spring boot que es el que habilita a correrlo con Maven: Run as.

Springboot-starter-web incluye tomcat e hibernate-validator, jackson databind, ...

El pom de springboot parent:

<parent>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-parent</artifactId>

    <version>1.4.0.RELEASE</version>

</parent>

tiene **properties** definidas que podemos sobrescrbir en la sección **properties** de nuestro propio pom. Por ejemplo: **java.version**, **hibernate-validator.version**, **plugins** y cosas así.

Reescribiendo versión de java:

<properties>

    <java.version>1.8</java.version>

</properties>

El parent.pom define las versiones de los starter projects que se incluyan en las dependencias. Curated dependencies.

Step2: first Rest Controller

* @Component
* @Autowired
* @RestController

Step3: some about springboot

First installment of revealing how magic happens with Spring Boot. As a Spring Boot developer, you need to understand what's happening beneath the hood of Spring Boot!

spring-boot-starter-web : starter for building applications with Spring MVC. Tomcat is default embedded container. Spring Boot Starter Web brings all dependencies needed to build normal and RESTful web applications. Look at the dependency tree.

All the dependencies are added in because of spring-boot-starter-web. Also look at /META-INF/spring provides inside the spring-boot-starter-web.jar

Spring Boot Starter Web auto configures things needed to startup a web application. Look at the log

Mapping servlet: 'dispatcherServlet' to [/]

Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController .error(javax.servlet.http.HttpServletRequest)

Mapped URL path [/webjars/\*\*] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

Look at package org.springframework.boot.autoconfigure.web in spring-boot-autoconfigure-\*.jar

Go to url <http://localhost:8080/some-non-existing-url> --> Will map to /error

Useful Snippets:

In /src/main/resources/application.propertiesc 🡪

logging.level.org.springframework: DEBUG

Step4: Understand Starter Parent

* spring-boot-starter-web-services
* spring-boot-starter-test
* spring-boot-starter-jdbc
* spring-boot-starter-security
* spring-boot-starter-data-jpa
* spring-boot-starter-data-rest

More at https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#using-boot-starter

Step5: Spring Boot vs Spring

**Spring:** Spring is just a dependency injection framework. Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

First half of the 2000 decade! EJBs. EJBs were NOT easy to develop. Write a lot of xml and plumbing code to get EJBs running. Impossible to Unit Test. Alternative - Writing simple JDBC Code involved a lot of plumbing. Spring framework started with aim of making Java EE development simpler.

Goals: Make applications testable. i.e. easier to write unit tests. Reduce plumbing code of JDBC and JMS. Simple architecture. Minus EJB. Integrates well with other popular frameworks.

Applications with Spring Framework:

Over the next few years, a number of applications were developed with Spring Framework. Testable but lot of configuration (XML and Java). Developing Spring Based application need configuration of a lot of beans!

Integration with other frameworks need configuration as well! In the last few years, focus is moving from monolith applications to microservices. We need to be able to start project quickly. Minimum or Zero start up time.

Framework Setup: Deployment - Configurability Logging, Transaction Management, Monitoring, Web Server Configuration.

Spring Boot: Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”. We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss.

Example Problem Statements

You want to add Hibernate to your project. You dont worry about configuring a data source and a session factory. SpringBoot will do if for you!

Goals:

* Provide quick start for projects with Spring.
* Be opinionated but provide options.
* Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).

Message converters: ¿cómo se convierte la lista de objetos devuelta en un **json**? Si **Jackson** está en la lista de **dependencias**, se autoconfiguran los **messageConverters.Jackson2HttpMessageConverter:**

**json** 🡨🡪 **object** y viceversa

**xml** 🡨🡪 **object** y viceversa

Se producen por los messageConverters, que son autoconfigurados por **jackson-data-bind** (que es una dependencia **transitiva** de starter-web).

**REST**: cliente y servidor pueden ser tecnologías completamente independientes y evolucionar por separado. Siempre es una ventaja tener sistemas poco acoplados.

Otra de las características de los servicios REST es que deberían poder dar resultados cacheables en la lista de routers que hay entre el cliente y el servidor final.

El cliente no debe asumir una conexión directa con el service provider. Esto es una de las *constraints* de la arquitectura REST. Y la última es que un recurso puede tener múltiples representaciones.

Existe un verbo http que es **PATCH** que se usa para **updates** muy parciales, como de **un campo** o así.

Devtools: se puede configurar EXCLUSIONES: **spring.devtools.restart.exclude=static/,public/**

Se pueden configurar nuevas rutas: **spring.devtools.restart.additional-paths**

**Cosas del RESTful:**

Cuando se crea un recurso, la respuesta debe contener la **URI del nuevo recurso creado**, y, además, tener el status **201 Created**, y no el 200 OK genérico.

Queremos la URI del recurso, para lo que hace falta el Id generado al crearlo, pero no queremos tener que hardcodear esa URI.

Se devuelve un **ResponseEntity<?>** (Java generics) mediante un método estático:

**return ResponseEntity.created(location).build();** o en caso de **no crear** el recurso:

**return ResponseEntity.noContent().build();**

ResponseEntity es la clase que nos permite devolver una respuesta con el **http status** que queremos.

Y para obtener URI de nuestro servicio se hace con **ServletUriComponentsBuilder**.

URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExppand(createdTodo.getId())

            .toUri();

En el ejemplo se trata de un servicio de encuestas y preguntas. Si se invoca un POST para crear el recurso de una pregunta nueva en una encuesta que no existe:

**POST localhost:8080/surveys/{surveyId}/questions**

la respuesta más adecuada sería **204 Not Created** en lugar de **500 Internal Server Error**. Esto se hace así: **return ResponseEntity.noContent().build();**

Claves:

* @RequestBody
* ServletUriComponentsBuilder
* ResponseEntity.

Content Negotiation: accept Header **Accept:application/xml**. Por defecto, springboot con spring-starter-web **no responde con xml**. Hay que añadir una dependencia más. En caso de no hacerlo, la respuesta es **406 Not Acceptable.**

<dependency>

    <groupId>com.fasterxml.jackson.dataformat</groupId>

    <artifactId>jackson-dataformat-xml</artifactId>

</dependency>

Actuator:

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.data</groupId>

    <artifactId>spring-data-rest-hal-browser</artifactId>

</dependency>

Metadata Services. Hal browser es una UI para actuator. Facetas que muestra actuator:

* **/beans**: se pueden ver todos los bean configurados en la aplicación. localhost:8080/beans.
* /**heapdump** para problemas, trazas de error.
* /**configprops**: propiedades que están configuradas, pudiendo estar sobrescritas en application.properties. Por ejemplo, propiedades de spring.mvc: default local, staticPathPattern, localeResolver...
* /**autoconfig** muestra características que han sido habilitadas por autoconfig por los starters, etc.
* /**trace**: muestra las últimas peticiones (request, response, headers, etc).
* /**heatlh**: disco libre, memoria libre, etc.
* /**mappings**: paths ligadas a métodos de controladores.
* /**env**: java, classpath, profiles
* /**metrics**: memoria disponible, libre, threads, urls invocadas, uso de cpu
* /**counter** status: estado de respuesta http 415, 12 veces. Etc.

Step 14: embedded servers.

Servlet containers. Tomcat se usa por defecto, está en el pom que genera Initializr. Se puede cambiar, por supuesto, pero está en el parent de starter-web, por lo que, si se quiere configurar otro, lo que hay que hacer es poner una exclusión en el pom (en la sección de starter-web).

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

    <exclusions>

        <exclusion>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-tomcat</artifactId>

        </exclusion>

    </exclusions>

</dependency><dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

Se puede cambiar el puerto en el que arranca la aplicación en **application.properties** con

**server.port=9000**

Step 15: configuracion dinámica para la aplicación.

Opciones para configurar por entorno. Configuracion con yml.

logging.level.org.springframework: INFO

igualmente podemos definir propiedades nosotros:

welcome.message=Welcome message from properties file!

Para coger un valor de los definidos en las aplicacion properties, lo hacemos mediante la anotación

@Value("${welcome.message}")

private String message;

Otra forma de configurar propiedades es pasarlas como **argumentos de programa**:

**--welcome.message="Welcome message from arguments"**. Estos parámetros de programa obviamente pueden ser distintos en los diferentes entornos de ejecución de la aplicación, con lo que podrían usarse parámetros distintos en producción y en desarrollo, por ejemplo.

Los parámetros de programa tienen **mayor prioridad** que los definidos en **>** **application.properties**.

En actuator, si vamos a **/env** podemos ver las **CommandLine Arguments**, y ver el que acabamos de crear. De la misma forma podemos ver las propiedades configuradas por archivo de properties.

Una forma distinta es **configurar el path hacia el archivo de propiedades** por parámetro de programa:

**--spring.config.location=classpath:/default.properties.**

Profiles es otra forma.

Placeholders: se pueden usar los valores de una propiedad para definir los valores de otras.

Ejemplo:

application.name=in28Minutes

welcome.message=Welcome to ${application.name}

En actuator, si vemos las propiedades definidas podríamos esperar ver la propiedad de dos formas, traducida (evaluada) o tal como la hemos definido. En la realidad se comprueba que no se muestra evaluada, sino planteada.

Configuracion en yaml, es más legible:

logging:

 level:

  org.springframewrok: INFO

  org.springframework.web.servlet: DEBUG

Step 16: Profiles

La forma más fácil de establecer un **profile** es, en applicacion.properties, poner (por ejemplo):

spring.profiles.active=prod

Otra forma de pasar el **profile** es como parámetro de la VM (máquina virtual) de la siguiente forma en **Run Configurations**:

**VM Arguments**

-Dspring.profiles.active=prod

Entonces, si tenemos en el classpath un properties llamado **application-prod.properties**, las propiedades definidas allí serán las que se apliquen.

Podemos tener cosas como la seguridad habilitada en un entorno pero no en otros, por ejemplo.

Si no se encuentra un application.properties de un profile, se hace uso de los **valores por defecto** para las properties, que son los definidos en applicacion.properties.

Forma dinámica. Creación de beans:

@Profile("dev")

@Bean

public String dummy(){

    return 'dummy';

}

El bean NO se crea en otros profile si estuviera activo. De esta manera, puedes por ejemplo tener un env para test unitarios que usen implementaciones distintas de las empleadas en otros entornos.

Por ejemplo, la base de datos. O cache distribuida para producción, cache de menos nivel para pre, y nada de cache para dev.

Step 17: advanced configuration.

**type safe configuration properties**. Vamos a ver una alternativa que puede evitar tener que definir un atributo por cada propiedad, anotado con @Value.

Se trata de definir un bean que contenga esos atributos, claro. Una clase **BasicConfiguration**.

Se declara como @Component para que sea escaneado.

@Component

@ConfigurationProperties("**basic**")

public class BasiConfiguration {

    private String message;

    private int number;

    private boolean value;

}

De esta forma, podemos acceder a la las properties **basic.message**, **basic.number**, **basic.value**.

Hay que tener un @Autowired en la clase de servicio donde queremos hacer uso de las properties, y hacer uso de ellas accediendo a sus atributos. La alternativa sería tener de definir un value por cada property, lo cual no es demasiado practico si son muchas propiedades.

Se dice que es **type safe**, pues la aplicación no arranca en absoluto si no se puede convertir la propiedad al tipo de dato del atributo que la ha de transportar.

**/confogprops** en actuator, allí podemos ver el bean definido para contener datos de configuración.

Step 18: Spring Data Jpa.

Creando un ejemplo sencillo de Repositorio. Incluye el starter **spring-data-jpa**. Hace falta además una base de datos, que puede ser por ejemplo H2.

En un primer intento, al no incluir ninguna, el contexto no se levanta.

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

    <groupId>com.h2database</groupId>

    <artifactId>h2</artifactId>

</dependency>

Típicamente se utilizan en test unitarios y TDD las bases de datos en memoria.

En el step actual ha creado una Entity y un Repository para crear y persistir algunos datos en un **CommandLineRunner**.

Se define una **Interface** que EXTIENDE de **CrudRepository<User, Long>**. La implementación es obra del framework, y no tenemos ese código de forma explícita.

¿Qué métodos ofrece la interfaz CrudRepository?

* Save
* Exists
* findOne
* findById
* findAll
* delete
* deleteById
* …

@Component

public class UserCommandLineRunner implements CommandLineRunner {

    @Override

    public void run(String... args) {

        sysout("hola from command line runner");

    }

}

El starter **spring-data-jpa** incluye **spring-jdbc**, **hibernate-jpa**, **hibernate-core**

La autoconfiguración al incluir **spring-data-jpa** crea beans como **jdbcTemplate**, **dataSource**, **entityManagerFactory**

**/autoconfig** en actuator se ve mucha información relativa a todo lo que se autoconfigura.

Sobre la base de datos **H2**, se puede configurar la url para acceder a ella en las **properties**:

spring.datasource.url=jdbc:h2:mem:testdb

También hay que habilitar el acceso a la consola, en algunas versiones de springboot al menos.

Step 19: http://localhost:8080/h2-console

Para cambiar H2 por mysql, en las properties definimos los valores necesarios:

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql//localhost:3306/test

spring.datasource.username=root

spring.datasource.password=admin

spring.datasource.initialize=true

spring-jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

Para los CrudRepository de SpringDataJpa, necesitamos exponer en la interfaz todos los métodos que queremos que estén en la implementación y que no sean los típicos. Existe un convenio de nombres para eso, pero como decimos, para que el framework lo proporcione HAY QUE DECLARAR EL METODO EN LA INTERFAZ. Ver la documentación para conocer la nomenclatura de los **query methods**.

**findByRole** (por ejemplo).

Step 20: Spring data Rest

Ofrecer servicios **REST directamente desde los repositorios**, este sería el objetivo de la dependencia **spring-data-rest**.

Es bueno para prototyping, pero no para producción, por que expone muchas más cosas que las que tal vez deseamos hacer disponibles.

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

El objetivo es exponer **rest services** sobre los métodos del **UserRepository.**

Se hace una copia de la interfaz creada antes UserRepository, y la llama **UserRestRepository**. En lugar de extender de Crudrepository ahora extendemos de **PagingAndSortingRepository** (tiene paginado y ordenación además de lo expuesto por CrudRepository). En esta es donde vamos a experimentar con el **DataRest**.

Se anota la clase con

**@RepositoryRestResource(path="users", collectionResourceRel="users")**

Funciona para peticiones GET pero también para peticiones POST para crear un nuevo usuario.

Es necesario incluir la cabecera **Content-type: application/json** si se enviará la información en este formato, pero por lo demás funciona como se espera.

En caso de GET, la cabecera necesaria es **Accept: application/json** (o lo que corresponda). Es más, configura correctamente la devolución de enlace al recurso creado y devuelve la entity creada.

**collectionResourceRel="usuarios"**: así es como se llama la colección con los resultados de la query en el json de respuesta.

{

    "\_embedded": {

        "usuarios": [

            {

                "name": "Ranga",

                "role": "Admin",

                "\_links": {

                    "self": {

                        "href": "http://localhost:8080/users/1"

                    },

                    "user": {

                        "href": "http://localhost:8080/users/1"

                    }

                }

            },…

        ]

    },

    "\_links": {

        "self": {

            "href": "http://localhost:8080/users"

        },

        "profile": {

            "href": "http://localhost:8080/profile/users"

        },

        "search": {

            "href": "http://localhost:8080/users/search"

        }

    },

    "page": {

        "size": 20,

        "totalElements": 4,

        "totalPages": 1,

        "number": 0

    }

}

**RepositoryRestResource** también expone otras url como:

http://localhost:8080/users/1 (Obtiene el elemento por su Id)

http://localhost:8080/users/?size=1 (Cuántos elementos queremos obtener. También da enlaces a las páginas de la paginación)

**HATEOAS** viene automáticamente, enlaces a los otros elementos, a la lista completa, a documentación del API (profile).

Por ejemplo, que proporcione enlaces a la siguiente página, a la primera página, a la página anterior. Son enlaces super útiles para cualquier consumidor de ese servicio.

<http://localhost:8080/users/?sort=name,desc> Otra de las funcionalidades que expones es **ordenación**.

Documentación de Spring Data Rest:

https://docs.spring.io/spring-data/rest/docs/current/reference/html/#reference

En el repositorio definido en el step anterior, que era una interface que extendía CrudRepository, teníamos definido un método **findByRole**. Para exponerlo como servicio REST tenemos que recibir como parámetro el id por el que buscamos. Así, en nuestra @RepositoryRestResource interface, definimos el método findByRole, pero en el parámetro del método usamos también una anotación:

la url para la invocación es **/users/search/findByRole/?role=Admin**

@RepositoryRestResource(path = "users", collectionResourceRel = "usuarios")

public interface UserRestRepository extends PagingAndSortingRepository<User, Long> {

    List<User> findByRole(@Param("role") String role);

}

Todos los recursos que hemos visto y comentado han sido configurados automáticamente por SpringBoot.

Lo recomienda el instructor para un rápido prototyping, pero a la vez advierte que expone tantas cosas de forma automática que para usarlo en una aplicación en producción hay que ser muy cauto, que puede ser más aconsejable tener más control sobre lo que se expone.

Step 21: unit testing & integration testing

Una de las palabras polisémicas que puede ser relevante aclarar es la palabra servicio o service. A la vez se usa en este documento para referirse a cualquiera de los componentes manejados por el framework anotados con @Service, y a la vez se usa para referirse a un **endpoint** de un servicio REST (de ahí el término servicio en este otro caso). En el segundo caso, como sabemos, el servicio es uno de los métodos de alguna de las clases anotadas como **@RestController**.

**Integration test**: va a través de todas las capas de la aplicación: **controller**, **service** y **persistence**.

En un **unit test** solo probamos **una cosa**, o el controlador, o el servicio, etc.

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>

</dependency>

Importante NOTAR que en la dependencia ha añadido un scope, así que la dependencia solo estará disponible si el **scope test** está activo. No es buena práctica tener código de test desplegado en producción.

Seguimos las convenciones de Junit para la creación de test, con lo que el paquete será el mismo que el de la clase que probamos, pero partiendo de **src/test/paquete** (otra source folder).

Vamos a la clase del Controller SurveyController y copiamos el package. Creamos el pachage en src/test/

**New -> Junit Test Case**. Se indica la clase a probar, que va a ser nuestro **controller**.

Para hacer un test de integración hay dos partes: inicialización de la aplicación e invocación de la lógica de negocio que queremos probar evaluando su resultado.

Para correr el contexto de Spring en un test una herramienta es **SpringRunner.class**. SpringJUnit4ClassRunner es la clase, SpringRunner es un alias. Entonces, en la clase del test:

Junit runners to start spring context: **@RunWith(SpringRunner.class)**

**@SpringBootTest** ayuda a lanzar una aplicación SpringBoot en un test, que es la clase que tenemos anotada como **@SpringBootApplication.**

Tiene otras opciones, entre ellas una que nos permitirá levantar la aplicación en un puerto random, y después obtener esa propiedad y conocer su valor.

@RunWith(SpringRunner.class)

@SpringBootTest(classes = Application.class,

        webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class SurveyControllerIT {

    @LocalServerPort

    private int port;

…

}

Se usa en conjunto con @LocalServerPort para inyectar el puerto en el Test, necesario para conocer la **url** e invocarla. Necesitamos el puerto para montar la url para llamar al controlador.

String url = "http://localhost:" + port + "/surveys/Survey1/questions/Question1";

Step 22: primer test de integración

Para invocar la url hacemos uso de TestRestTemplate.

import org.springframework.boot.test.web.client.TestRestTemplate;

TestRestTemplate restTemplate = new TestRestTemplate();

Primero ejecutamos el método **getForObject**:

String output = restTemplate.getForObject(url, String.class);

Si queremos **content negotiation** necesitamos incluir **cabeceras**. Entonces no se puede hacer uso de **getForObject** y hay que usar **exchange**. Para la primera prueba crearemos una requestEntity null, o sea, sin cabeceras ni body. Estaríamos en el mismo caso anterior.

ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET, requestEntity, String.class);

Para crear las cabeceras hacemos uso de **HttpHeaders headers = new HttpHeaders();**

Aunque creo que se puede hacer así: **headers.add("Accept", "application/json");** la forma concreta de hacer esto es:

headers.setAccept(Arrays.asList(MediaType.APPLICATION\_JSON));

Entonces podemos crear un objeto HttpEntity y con él hacer uso del método exchange de la RestTemplate.

HttpEntity<Object> requestEntity = new HttpEntity<Object>(null, headers);

ResponseEntity<String> response = restTemplate.exchange(url, HttpMethod.GET, requestEntity, String.class);

String body = response.getBody();

System.out.println("Response : " + body);

// {"id":"Question1","description":"Largest Country in the

// World","correctAnswer":"Russia","options":["India","Russia","United

// States","China"]}

assertTrue(body.contains("\"id\":\"Question1\""));

assertTrue(body.contains("\"correctAnswer\":\"Russia\""));

**JSONAssert** es una herramienta mucho mejor para hacer asserts cuando el resultado que se obtiene es una cadena JSON. No se fija en espacios o comillas en los nombres o valores de los atributos del objeto json.

**JSONAssert.assertEquals**, puede ser estricta o laxa. Por ejemplo:

JSONAssert.assertEquals("{date: 1}", "{\"date\": 1}", false);

El primer valor es ***expected***. El segundo valor es ***actual***. Como el valor obtenido cumple con que contiene todos los atributos expected y los mismos valores, no fallaría por el hecho de tener información adicional.

En cambio, en modo estricto tienen que coincidir **todos** los atributos recibidos con los esperados. Por contra, no importan los espacios es blanco o las comillas.

@Test

public void testRetrieveSurveyQuestion() {

    String url = createUrl("/surveys/Survey1/questions/Question1");

    System.out.println(url);

    HttpEntity<Object> requestEntity = new HttpEntity<Object>(null, headers);

    ResponseEntity<String> response = template.exchange(url, HttpMethod.GET, requestEntity, String.class);

    String body = response.getBody();

    System.out.println("Response : " + body);

    // {"id":"Question1","description":"Largest Country in the

    // World","correctAnswer":"Russia","options":["India","Russia","United

    // States","China"]}

    assertTrue(body.contains("\"id\":\"Question1\""));

    assertTrue(body.contains("\"correctAnswer\":\"Russia\""));

    String expected = "{id:Question1,description:Largest Country in the World,correctAnswer:Russia}";

    JSONAssert.assertEquals("{date:1}", "{date: 1}", false);

    JSONAssert.assertEquals(expected, response.getBody(), false);

}

Ahora para probar la petición de todas las Question de una Survey:

@Test

public void testRetrieveSurveyQuestions() {

    String url = createUrl("/surveys/Survey1/questions/");

    System.out.println(url);

    HttpEntity<Object> requestEntity = new HttpEntity<Object>(null, headers);

    @SuppressWarnings("rawtypes")

    ResponseEntity<ArrayList> response = template.exchange(url, HttpMethod.GET, requestEntity, ArrayList.class);

    @SuppressWarnings("unchecked")

    ArrayList<Question> body = response.getBody();

    System.out.println("Response : " + body);

}

El fragmento anterior sirve, pero el instructor utiliza:

@Test

public void testRetrieveSurveyQuestions2() {

    String url = createUrl("/surveys/Survey1/questions/");

    System.out.println(url);

    HttpEntity<Object> requestEntity = new HttpEntity<Object>(null, headers);

    ResponseEntity<List<Question>> response = template.exchange(url, HttpMethod.GET, requestEntity,

            new ParameterizedTypeReference<List<Question>>() {

            });

    Question assertQuestion = new Question("Question1", "Largest Country in the World", "Russia",

            Arrays.asList("India", "Russia", "United States", "China"));

    assertTrue(response.getBody().contains(assertQuestion));

}

ParameterizedTypeReference: The purpose of this class is to enable capturing and passing a generic Type. In order to capture the generic type and retain it at runtime,you need to create a subclass as follows:

ParameterizedTypeReference<List<String>> typeRef = new ParameterizedTypeReference<List<String>>() {};

The resulting typeReference instance can then be used to obtain a Type instance that carries parameterized type information.For more information on "super type tokens" see the link to Neal Gafter's blog post

Probar el servicio POST de una nueva Question será la siguiente cuestión a estudio. Ahora en la HttpEntity no podemos enviar un body **vacío**, sino que precisamente ese cuerpo de la petición será la **Question** que vamos a postear, guardar o registrar.

El siguiente fragmento ilustra la primera parte del test, el envío efectivo de la Question para que el controlador invoque la lógica de negocio que guarde un nuevo elemento.

@Test

public void testAddSurveyQuestions() {

    String url = createUrl("/surveys/Survey1/questions");

    System.out.println(url);

    Question newQuestion = new Question("Question10", "Daniel pants size", "34",

            Arrays.asList("33", "34", "35", "36"));

    HttpEntity<Question> requestEntity = new HttpEntity<Question>(newQuestion, headers);

    ResponseEntity<String> response = template.exchange(url, HttpMethod.POST, requestEntity, String.class);

}

Ahora la parte de aserción, necesaria para automatizar los test y ver de manera más gráfica el resultado. Sabemos que este servicio no devuelve el registro creado, sino que devuelve una **cabecera** y un status **201 created**.

La cabecera se llama **location**, y se puede obtener a partir de la response. Incluimos la comprobación también del status (**201 created**)

@Test

public void testAddSurveyQuestions() {

    String url = createUrl("/surveys/Survey1/questions");

    System.out.println(url);

    Question newQuestion = new Question("Question10", "Daniel shoe size", "43",

            Arrays.asList("43", "44", "45", "46"));

    HttpEntity<Question> requestEntity = new HttpEntity<Question>(newQuestion, headers);

    ResponseEntity<String> response = template.exchange(url, HttpMethod.POST, requestEntity, String.class);

    List<String> headers = response.getHeaders().get(HttpHeaders.LOCATION);

    String location = headers.get(0);

    assertTrue(location.contains("/surveys/Survey1/questions/"));

    assertEquals(response.getStatusCode(), HttpStatus.CREATED);

}

Step 23: refactor para eliminar las duplicidades: cabeceras, template...

Poner la creación de la url en un método. Creación de las cabeceras y del template en un método init() anotado con @Before (se ejecuta antes de cualquiera de los @Test)

Step 24: unit testing

Los test unitarios no son iguales que los de integración (pues se prueba uno solo de los componentes, y en los de integración la prueba incluye todas las capas de la aplicación). Hacen uso de **mocking** o stubs (rudimentario y desaconsejado el ultimo).

Se hará a continuación un test para probar el obtener una Question específica de una de las Survey (solo tenemos una).

Las clases de tests unitarios se suelen llamar por el nombre la **clase que queremos probar + 'Test'**. Es una convención. Hacemos una copia de la clase en la que tenemos los test de integración, pero borramos prácticamente todo para quedarnos con lo único que nos hace falta de momento:

package com.in28minutes.springboot.controller;

import org.junit.runner.RunWith;

import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)

public class SurveyControllerTest {

}

Seguimos usando **SpringRunner** para correr el contexto (aunque a mí me da que para el concepto de test unitario no es necesario eso) En un test de integración levantamos toda la aplicación. En la clase anterior esa parte era provista por la anotación

@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

Solo queremos cargar una clase, y para eso vamos a **mockear** las dependencias que necesitemos.

Para crear un **contexto solo con el controlador** que queremos probar, en lugar de lo que teníamos antes **@SpringBootTest**, ahora tenemos **@WebMvcTest(SurveyController.class).**

Necesitamos el service **SurveyService**, que de momento podemos **mockear** teniendo un atributo de este tipo y anotado con **@MockBean**, lo que inyectará una instancia mock del mismo.

**SpringRunner** al ver la anotación **@MockBean** va a crear una instancia mock y la va a inyectar en la clase de **@WebMvcTest(SurveyController.class).**

import org.springframework.test.context.junit4.SpringRunner;

import com.in28minutes.springboot.service.SurveyService;

@RunWith(SpringRunner.class)

@WebMvcTest(SurveyController.class)

public class SurveyControllerTest {

    @MockBean

    private SurveyService service;

    @Test

    public void firstTest() {

        System.out.println("hello");

    }

}

Vemos que al especificar **@WebMvcTest** un **controlador** **concreto** (no se hace ComponentScan, que es lento), **solo se cargan sus mapeos** (mucho más rápido de ejecutar) al **iniciar** ese **mini-contexto**:

Mapped "{[/surveys/{surveyId}/questions],methods=[GET]}"

onto public java.util.List<com.in28minutes.springboot.model.Question> com.in28minutes.springboot.controller.SurveyController.retrieveQuestions(java.lang.String)

Mapped "{[/surveys/{surveyId}/questions/{questionId}],methods=[GET]}"

onto public com.in28minutes.springboot.model.Question com.in28minutes.springboot.controller.SurveyController.retrieveDetailsForQuestion(java.lang.String,java.lang.String)

Mapped "{[/surveys/{surveyId}/questions],methods=[POST]}"

onto public org.springframework.http.ResponseEntity<java.lang.Void> com.in28minutes.springboot.controller.SurveyController.addQuestionToSurvey(java.lang.String,com.in28minutes.springboot.model.Question)

El primer test será sobre el método: **retrieveDetailsForQuestion**. Los test tienen 3 partes:

1. setup
2. invocación
3. assert

El setup es **montar el mock** como queremos: para tal conjunto de parámetros de entrada tienes que dar **tal resultado dummy**. Entonces llegará la parte de la prueba en sí, que en nuestro caso lo es del controller.

Para hacer llamadas al controller, **@WevMvcTest** provee de un bean llamado **mockMvc** que nos permite hacer llamadas web (como un **restTemplate** o parecido). Hay que declararlo e inyectarlo con **@Autowired**:

@Autowired

private MockMvc mockMvc;

Tiene un método que se llama **perform** que necesita una **requestBuilder** que montamos por medio del patrón Builder con la interfaz **RequestBuilder**.

**MockMvcRequestBuilders.get(url).accept(MediaType.APPLICATION\_JSON);**

Queremos probar la path **/surveys/{surveyId}/questions/{questionId}**, y construimos una request a esa url.

Sobre **mockMvc.perform** añadimos una llamada a **.andReturn()** 🡪 Return the result of the executed request for direct access to the results (MvcResult)

**mvcResult.getResponse().getContentAsString()** es el JSON del body de la respuesta obtenida. Obviamente para compararla con la cadena expected lo hacemos con JSONAssert.assertEquals y strict = false.

En la consola se muestra información jugosa:

MockHttpServletRequest:

      HTTP Method = GET

      Request URI = /surveys/Survey1/questions/Question1

       Parameters = {}

          Headers = {Accept=[application/json

    ]

}

Handler:

             Type = com.in28minutes.springboot.controller.SurveyController

           Method = public com.in28minutes.springboot.model.Question com.in28minutes.springboot.controller.SurveyController.retrieveDetailsForQuestion(java.lang.String,java.lang.String)

Async:

    Async started = false

     Async result = null

Resolved Exception:

             Type = null

ModelAndView:

        View name = null

             View = null

            Model = null

FlashMap:

       Attributes = null

MockHttpServletResponse:

           Status = 200

    Error message = null

          Headers = {Content-Type=[application/json;charset=UTF-8

    ]

}

     Content type = application/json;charset=UTF-8

             Body = {

    "id": "Question1",

    "description": "Largest Country in the World",

    "correctAnswer": "Russia",

    "options": [

        "India",

        "Russia",

        "United States",

        "China"

    ]

}

    Forwarded URL = null

   Redirected URL = null

          Cookies = []

Este es el código del test completo (que funciona OK):

@Test

public void retrieveDetailsForQuestionTest() throws Exception {

    Question question1 = new Question("Question1", "Largest Country in the World", "Russia",

            Arrays.asList("India", "Russia", "United States", "China"));

    // ajustando el mock. Lo que probamos es el controller.

    // when(service.retrieveQuestion("Survey1", "Question1")).thenReturn(question1);

    when(service.retrieveQuestion(Mockito.anyString(), Mockito.anyString())).thenReturn(question1);

    // tenemos que llamar con GET y ver los datos que recibimos. El controlador ya

    // tiene inyectado el service.

    RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/surveys/Survey1/questions/Question1")

            .accept(MediaType.APPLICATION\_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andReturn();

    String expected = "{id:Question1,description:Largest Country in the World,correctAnswer:Russia}";

    String contentAsString = result.getResponse().getContentAsString();

    JSONAssert.assertEquals(expected, contentAsString, false);

}

Step 25: tests para POST y GET cuando se recupera una colección

El siguiente test sirve para probar el caso de uso de obtener todas las Question de una Survey, por lo que se compara con un array expected. Además se introduce el uso de los **ResultMatcher** al poner la restricción de que la respuesta del servicio tiene que ser **200 OK**.

StatusResultMatchers org.springframework.test.web.servlet.result.MockMvcResultMatchers.status() : Access to response status assertions.

Ejemplo:

MvcResult result = mockMvc.perform(requestBuilder).andExpect(status().isOk()).andReturn(); //status() tiene el import static de org.springframework.test.web.servlet.result.MockMvcResultMatchers.status()

El código completo del @Test es:

@Test

public void retrieveQuestionsTest() throws Exception {

    Question question1 = new Question("Question1", "Largest Country in the World", "Russia",

            Arrays.asList("India", "Russia", "United States", "China"));

    Question question2 = new Question("Question2", "Smallest Country in the World", "San Marino",

            Arrays.asList("San Marino", "Monaco", "Batican", "Andorra"));

    when(service.retrieveQuestions(Mockito.anyString())).thenReturn(Arrays.asList(question1, question2));

    RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/surveys/SurveyDummyId/questions")

            .accept(MediaType.APPLICATION\_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andExpect(status().isOk()).andReturn();

    String expected = "[{id:Question1,description:Largest Country in the World,correctAnswer:Russia},{id:Question2,description:Smallest Country in the World,correctAnswer:San Marino}]";

    String contentAsString = result.getResponse().getContentAsString();

    JSONAssert.assertEquals(expected, contentAsString, false);

}

Un último test para esta sección será probar el servicio (= endpoint del controller) de creación de una Question para una Survey. Hay que utilizar el método **post de mockMvc** y además hacer uso del método **content(String content)** para poner la String json con la Question que queremos postear. También especificar el ContentType con contentType(MediaType.APPLICATION\_JSON). Vemos todo esto resumido en el código a continuación:

@Test

public void addQuestionToSurveyTest() throws Exception {

    Question mockQuestion = new Question("1", "Smallest Number", "1", Arrays.asList("1", "2", "3", "4"));

    String questionJson = "{\"description\":\"Smallest Number\",\"correctAnswer\":\"1\",\"options\":[\"1\",\"2\",\"3\",\"4\"]}";

    when(service.addQuestion(Mockito.anyString(), Mockito.any(Question.class))).thenReturn(mockQuestion);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/surveys/SurveyDummyId/questions")

            .content(questionJson).contentType(MediaType.APPLICATION\_JSON).accept(MediaType.APPLICATION\_JSON);

    MvcResult result = mockMvc.perform(requestBuilder).andExpect(status().isCreated()).andReturn();

    String contentAsString = result.getResponse().getContentAsString();

    String location = result.getResponse().getHeader(HttpHeaders.LOCATION);

    System.out.println(location);

}

Step 26: Spring Security basics

Aplicación de los conceptos de autenticación y autorización. Quién eres y a qué tienes acceso. Hay que añadir un starter al pom y ya se hace necesaria la autenticación para cualquier petición, en virtud de los jars presentes en el classpath y la autoconfiguración de spring-boot.

Para acceder a actuator también se auto-habilita. Todo lo que está expuesto necesita auth básica ahora.

Al arrancar el contexto de aplicación, se puede ver un mensaje en el log que muestra la password auto-generada en el arranque (el username es implicito y es siempre 'user')

**Using default security password: ea74c381-7383-461b-870f-e5a76560c35c**

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>

Las dependencias del parent pom de spring boot relacionadas con **spring-boot-starter-security** son **web-security** y **security-config**.

Al intentar hacer una petición sin cabeceras:

{

    "timestamp": 1591899066427,

    "status": 401,

    "error": "Unauthorized",

    "message": "Full authentication is required to access this resource",

    "path": "/surveys/Survey1/questions/Question1"

}

Step 26-2: Configurar roles y actualizar los test de integración y unitarios

Vamos a crear dos roles. User y Admin. Creamos clase de configuración que extiende de **WebSecurityConfigurerAdapter** para hacer override a algunos métodos.

Dando un vistazo a la clase WebSecurityConfigurerAdapter vemos que para habilitar algunos usuarios y tenerlos en memoria cargados tenemos que hacer override del método **configure(AuthenticationManagerBuilder auth).** Copiamos código de la propia clase para poner nuestros propios usuarios (linea 134 de la clase):

/\*

\* For example, the following configuration could be used to register in memory

 \* authentication that exposes an in memory {@link UserDetailsService}:

 \*/

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    // enable in memory based authentication with a user named user and admin

    auth.inMemoryAuthentication().withUser("user").password("password").roles("USER").and().withUser("admin")

            .password("password").roles("USER", "ADMIN");

}

Con esto tendríamos la primera parte que queremos abordar. Ahora hay que definir los accesos de cada uno de los roles. Se usan antMatchers. Se copia también de la clase WebSecurityConfigurerAdapter:

/\*\*

 \* Override this method to configure the {@link HttpSecurity}. Typically

 \* subclasses should not invoke this method by calling super as it may override

 \* their configuration. The default configuration is:

 \*

 \* <pre>

 \* http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();

 \* </pre>

 \*

 \* @param http the {@link HttpSecurity} to modify

 \* @throws Exception if an error occurs

 \*/

// @formatter:off

protected void configure(HttpSecurity http) throws Exception {

    logger.debug("Using default configure(HttpSecurity). If subclassed this will potentially override subclass configure(HttpSecurity).");

    http

        .authorizeRequests().anyRequest().authenticated().and().formLogin().and()

        .httpBasic();

}

Nuestro método adaptado:

// 2.-autorizacion de los usuarios role --> accesss

// survey -> USER

protected void configure(HttpSecurity http) throws Exception {

    http.httpBasic().and().authorizeRequests().antMatchers("/survey/\*\*").hasRole("USER").antMatchers("/users/\*\*")

            .hasRole("USER").antMatchers("/\*\*").hasRole("ADMIN").and().csrf().disable().headers().frameOptions()

            .disable();

}

Las dos últimas líneas se incluyen **para que pueda funcionar hal-browser** de actuator, pero son de una profundidad en la que de momento no entraremos en este documento.

https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html This section discusses Spring Security's Cross Site Request Forgery (CSRF) support.

https://developer.mozilla.org/es/docs/Web/HTTP/Headers/X-Frame-Options : El encabezado de respuesta HTTP X-Frame-Options puede ser usado para indicar si debería permitírsele a un navegador renderizar una página en un <frame>, <iframe> o <object> . Las páginas webs pueden usarlo para evitar ataques de clickjacking , asegurándose que su contenido no es embebido en otros sitios.

Tras probar actuator y hacer distintas peticiones con POSTMAN se ha comprobado que el sistema de roles funciona como era de esperar.

NOTA

Disabling Security for 2.2.0.RELEASE in Unit Tests needs a small change

Spring Boot 2.2.0.RELEASE+

@WebMvcTest(value = SurveyController.class, **secure = false**)

**secure=false** is deprecated. Instead, you would need to use **@WithMockUser**

Add this to pom.xml

<dependency>

    <groupId>org.springframework.security</groupId>

    <artifactId>spring-security-test</artifactId>

    <scope>test</scope>

</dependency>

Use **@WithMockUser** to use a mock user as follow:

@WithMockUser(username = "user1", password = "secret1")

@Test

public void yourTest() throws Exception {...}

Step 27: modificar los test unitarios y de integración para que funcionen con spring-security-login

Tal como están, los test fallan por no llevar autenticación. Es necesaria una dependencia y el uso de un par de anotaciones, como se expone en la nota anterior.

Se debe establecer la cabecera "Authorization". Después se debe cifrar el usuario y la clave en Base64, y crear la cadena **"Basic " + Base64Enconding(userId+":"+password)**

String auth = "user"+":"+"secret1";

byte[] encondeAuth = Base64.encode(auth.getBytes(Charset.forName("US-ASCII")));

String headerValue = "Basic " + new String(encondeAuth);

headers.add("Authorization", headerValue);

Este código lo ponemos en un método para que cree las cabeceras en todos los test de integración.

@Before

public void init() {

    template = new TestRestTemplate();

    headers = createHeaders("user", "secret1");

}

private HttpHeaders createHeaders(String userId, String password) {

    HttpHeaders headers = new HttpHeaders();

    headers.setAccept(Arrays.asList(MediaType.APPLICATION\_JSON));

    String auth = "user" + ":" + "secret1";

    byte[] encondeAuth = Base64.encode(auth.getBytes(Charset.forName("US-ASCII")));

    String headerValue = "Basic " + new String(encondeAuth);

    headers.add("Authorization", headerValue);

    return headers;

}

Para los test unitarios queremos deshabilitar la seguridad, pues NO es lo que se está probando y hay que centrarse en probar aquello que es objeto de la prueba únicamente, en nuestro caso el controller de survey.

Para deshabilitar la seguridad, @WebMvcTest tiene un parámetro "secure" que podemos establecer a false. En versiones posteriores a SpringBoot 2.2.0 esto no se hace así pues el parámetro estaría deprecado, y hay que proceder como indica la nota al respecto que ya hemos visto.

/\*\*

 \* If Spring Security's {@link MockMvc} support should be auto-configured when it is

 \* on the classpath. Defaults to {@code true}.

 \* @return if Spring Security's MockMvc support is auto-configured

 \*/

@AliasFor(annotation = AutoConfigureMockMvc.class, attribute = "secure")

boolean secure() default true;

Step 28: Deep dive into spring autoconfiguration.

Al arrancar spring-boot, en función de las clases presentes en el classpath se configuran y se hacen disponibles un número de Beans. Todo esto sucede de forma automática y ahora veremos algunos detalles de qué sucede en el proceso.

Ejemplos: **JdbcTemplateAutoConfiguration**, **HttpMessageConvertersAutoConfiguration**.

* ¿Qué beans se configuran?
* ¿Qué beans NO se configuran?

Se puede obtener mucha información viendo el log en **modo DEBUG**.

Configurados:

DeviceResolverAutoConfiguration did not match

- required @ConditionalOnClass classes not found: org.springframework.mobile.device.DeviceResolverHandlerInterceptor,org.springframework.mobile.device.DeviceHandlerMethodArgumentResolver (OnClassCondition)

DispatcherServletAutoConfiguration.DispatcherServletConfiguration#multipartResolver did not match

- @ConditionalOnBean (types: org.springframework.web.multipart.MultipartResolver; SearchStrategy: all) found no beans (OnBeanCondition)

EhCacheCacheConfiguration did not match

- required @ConditionalOnClass classes not found: net.sf.ehcache.Cache,org.springframework.cache.ehcache.EhCacheCacheManager (OnClassCondition)

JpaBaseConfiguration#entityManagerFactoryBuilder matched

- @ConditionalOnMissingBean (types: org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder; SearchStrategy: all) found no beans (OnBeanCondition)

JpaBaseConfiguration#jpaVendorAdapter matched

- @ConditionalOnMissingBean (types: org.springframework.orm.jpa.JpaVendorAdapter; SearchStrategy: all) found no beans (OnBeanCondition)

JpaBaseConfiguration#transactionManager matched

- @ConditionalOnMissingBean (types: org.springframework.transaction.PlatformTransactionManager; SearchStrategy: all) found no beans (OnBeanCondition)

WebMvcAutoConfiguration#httpPutFormContentFilter matched

- @ConditionalOnMissingBean (types: org.springframework.web.filter.HttpPutFormContentFilter; SearchStrategy: all) found no beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#defaultViewResolver matched

- @ConditionalOnMissingBean (types: org.springframework.web.servlet.view.InternalResourceViewResolver; SearchStrategy: all) found no beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#requestContextFilter matched

- @ConditionalOnMissingBean (types: org.springframework.web.context.request.RequestContextListener,org.springframework.web.filter.RequestContextFilter; SearchStrategy: all) found no beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#viewResolver matched

- @ConditionalOnBean (types: org.springframework.web.servlet.ViewResolver; SearchStrategy: all) found the following [defaultViewResolver, beanNameViewResolver, mvcViewResolver] @ConditionalOnMissingBean (names: viewResolver; types: org.springframework.web.servlet.view.ContentNegotiatingViewResolver; SearchStrategy: all) found no beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.FaviconConfiguration matched

- matched (OnPropertyCondition)

WebSocketAutoConfiguration matched

- @ConditionalOnClass classes found: javax.servlet.Servlet,javax.websocket.server.ServerContainer (OnClassCondition)

- found web application StandardServletEnvironment (OnWebApplicationCondition)

WebSocketAutoConfiguration.TomcatWebSocketConfiguration matched

- @ConditionalOnClass classes found: org.apache.catalina.startup.Tomcat,org.apache.tomcat.websocket.server.WsSci (OnClassCondition)

WebSocketAutoConfiguration.TomcatWebSocketConfiguration#websocketContainerCustomizer matched

- Required JVM version 1.7 or newer found 1.8 (OnJavaCondition)

- @ConditionalOnMissingBean (names: websocketContainerCustomizer; SearchStrategy: all) found no beans (OnBeanCondition)

...

: Returning cached instance of singleton bean 'resourceHandlerMapping'

: Returning cached instance of singleton bean 'beanNameHandlerMapping'

: Returning cached instance of singleton bean 'faviconHandlerMapping'

: Returning cached instance of singleton bean 'requestMappingHandlerMapping'

: Returning cached instance of singleton bean 'endpointHandlerMapping'

: Returning cached instance of singleton bean 'beanNameHandlerMapping'

: Returning cached instance of singleton bean 'resourceHandlerMapping'

: Returning cached instance of singleton bean 'faviconHandlerMapping'

: Returning cached instance of singleton bean 'requestMappingHandlerMapping'

En **Actuator** también podemos ver todos los beans que están configurados (**HAL Browser**).

**Autoconfig**: se ven todos los **positive matches**. Toda la información del log está presente en actuator también, y de modo más manejable.

La **lógica de autoconfiguración** está en un jar: **spring-boot-autoconfigure-1.4.0-RELEASE.jar**. El jar tiene un montón de packages. Cassandra, Cloud, JMS, JDBC... Todas las clases de configuración están en el jar, no vienen del parent pom. En el jar hay también un archivo que se llama **spring.factories** en la carpeta **META-INF**.

Tiene un listado de las **clases de autoconfiguración** con sus packages (full name). Initializers, Application Listeners, AutoConfigure...

Una de esas clases es **JDBCAutoConfiguration**. ¿Qué clase de código tiene dentro?

/\*\*

 \* {@link EnableAutoConfiguration Auto-configuration} for {@link JdbcTemplate}

 \* and {@link NamedParameterJdbcTemplate}.

 \*

 \* @author Dave Syer

 \* @author Phillip Webb

 \* @author Stephane Nicoll

 \* @author Kazuki Shimizu

 \* @since 1.4.0

 \*/

@Configuration

@ConditionalOnClass(DataSource.class)

@ConditionalOnSingleCandidate(DataSource.class)

@AutoConfigureAfter(DataSourceAutoConfiguration.class)

public class JdbcTemplateAutoConfiguration {

Vemos que es condicional a que haya otra clase en el classpath. Y además que solo haya un candidato. El otro ejemplo que comentamos es HttpMessageConvertersAutoConfiguration.

    /\*\*

     \* {@link EnableAutoConfiguration Auto-configuration} for

     \* {@link HttpMessageConverter}s.

     \*

     \* @author Dave Syer

     \* @author Christian Dupuis

     \* @author Piotr Maj

     \* @author Oliver Gierke

     \* @author David Liu

     \* @author Andy Wilkinson

     \* @author Sebastien Deleuze

     \* @author Stephane Nicoll

     \* @author EddÃº MelÃ©ndez

     \*/

@Configuration

@ConditionalOnClass(HttpMessageConverter.class)

@AutoConfigureAfter({GsonAutoConfiguration.class,JacksonAutoConfiguration.class,JsonbAutoConfiguration.class})

@Import({JacksonHttpMessageConvertersConfiguration.class,GsonHttpMessageConvertersConfiguration.class,JsonbHttpMessageConvertersConfiguration.class})

public class HttpMessageConvertersAutoConfiguration {

static final String PREFERRED\_MAPPER\_PROPERTY = "spring.http.converters.preferred-json-mapper";

private final List<HttpMessageConverter<?>> converters;

public HttpMessageConvertersAutoConfiguration(

        ObjectProvider<HttpMessageConverter<?>> convertersProvider) {

    this.converters = convertersProvider.orderedStream().collect(Collectors.toList());

}

@Bean

@ConditionalOnMissingBean

public HttpMessageConverters messageConverters() {

    return new HttpMessageConverters(this.converters);

}