Tema 4. Tecnologías del servidor. Spring MVC y Thymeleaf

Programación web

Boni García

Curso 2016/2017

Índice

1. Java EE y Spring
2. Spring MVC y Thymeleaf
3. Bases de datos con Spring Data y JPA
4. Seguridad con Spring Security
5. Pruebas con JUnit y Selenium

Índice

1. Java EE y Spring
2. Spring MVC y Thymeleaf
   - Spring MVC
   - Thymeleaf
   - Envío de información al servidor
   - Gestión de datos de sesión
   - Soporte de internacionalización (I18N)
   - *Layouts*
3. Bases de datos con Spring Data y JPA
4. Seguridad con Spring Security
5. Pruebas con JUnit y Selenium

Spring MVC

Introducción

- El modelo vista controlador (**MVC**) es un patrón de diseño que permite separar los datos y la lógica de negocio de una aplicación de la interfaz de usuario.
- El esquema del patrón MVC en Spring es:

http://spring.io/guides/gs/serving-web-content/

Spring MVC

Controladores

- Los controladores (*Controllers*) son clases Java encargadas de atender las peticiones web
- Procesan los datos que llegan en la petición (parámetros)
- Hacen peticiones a la base de datos, usan diversos servicios, etc…
- Definen la información que será visualizada en la página web (el modelo)
- Determinan que vista será la encargada de generar la página HTML

Spring MVC

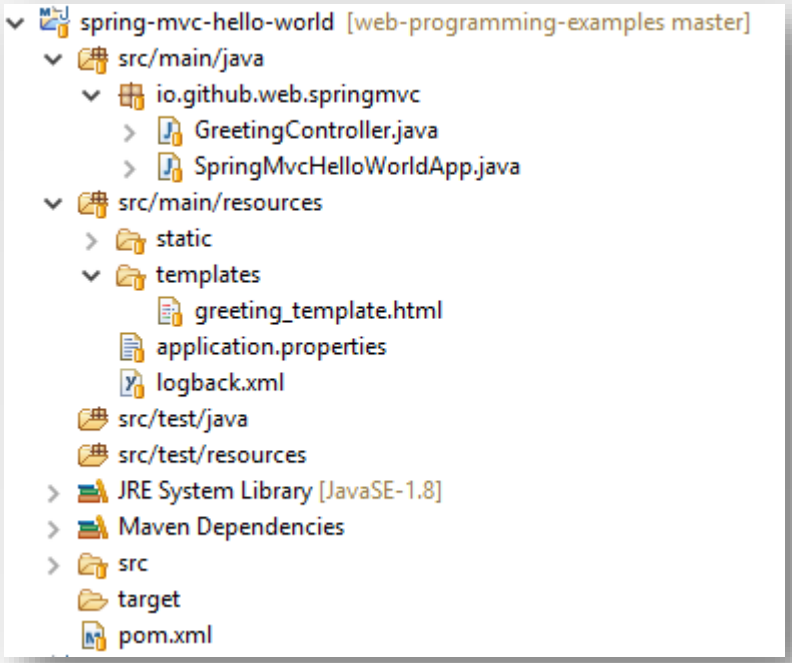
Vistas

- Las vistas en Spring MVC se implementan como tecnologías de plantillas HTML
- Generan HTML partiendo de una plantilla y la información que viene del controlador (modelo)
- Existen diversas tecnologías de plantillas que se pueden usar con Spring MVC: JSP, Thymeleaf, FreeMarker, etc…
- Nosotros usaremos **Thymeleaf**

Spring MVC

Ejemplo

- Estructura de la aplicación vista desde Eclipse:



Spring MVC

Ejemplo

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>io.github.web</groupId>
   <artifactId>spring-mvc-hello-world</artifactId>
   <version>1.0.0</version>

   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.5.2.RELEASE</version>
   </parent>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <java.version>1.8</java.version>
   </properties>

   <dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
     </dependency>
   </dependencies>
</project>
```

Proyecto padre del que se hereda la configuración (Spring Boot)

Java 8

Tipo de proyecto Spring Boot

Spring MVC

Ejemplo

GreetingController.java

```
package io.github.web.springmvc;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class GreetingController {

   @RequestMapping("/greeting")
   public ModelAndView greeting() {
      return new ModelAndView("greeting_template").addObject("name", "World");
   }
}
```

Se indica la URL con la que se ejecutan

Se indica el nombre de la plantilla que generará la página HTML

Se incluye la información en el modelo

Spring MVC

Ejemplo

- Vistas (implementado con **Thymeleaf**)

greeting_template.html

```
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <p>Hello <span th:text="${name}">Friend</span>!</p>
</body>
</html>
```

Se accede a los objetos que el **controlador** ha puesto en el **modelo**

El valor del objeto *name* sustituirá a la palabra *Friend* cuando se genere el HTML

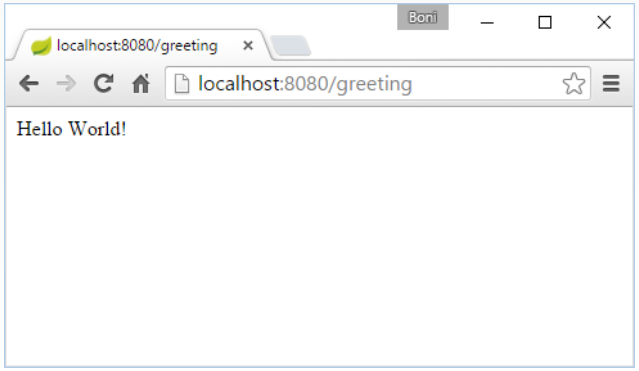
Spring MVC

Ejemplo

- La aplicación se ejecuta como una app Java normal
- En Eclipse: botón derecho proyecto > Run as… > Java Application…

SpringMvcHelloWorldApp.java

```
@SpringBootApplication
public class SpringMvcHelloWorldApp extends WebMvcConfigurerAdapter {

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



Thymeleaf

Introducción

- Spring MVC se apoya en alguna tecnología de plantillas para la generación de páginas HTML:

<FreeMarker>

http://freemarker.org/

http://www.oracle.com/technetwork/java/javaee/jsp/

thymeleaf

http://www.thymeleaf.org/

http://velocity.apache.org/

Thymeleaf

Introducción

- Todas estas tecnologías son conceptualmente similares
  - Los ficheros HTML se generar con **plantillas** que contienen código HTML junto con referencias a variables y funciones
  - Ejemplo implementado con FreeMarker:

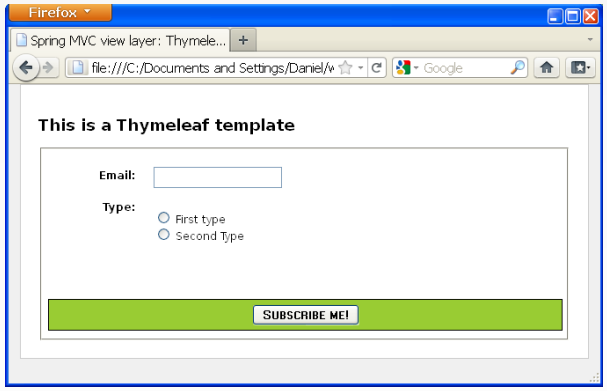
```
<html>
<body>
    <p>Hello ${name}!</p>
</body>
</html>
```

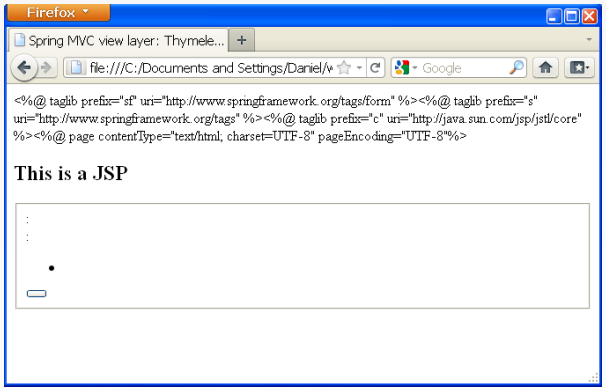
- **Thymeleaf** se diferencia de las demás en que las plantillas son ficheros HTML válidos que pueden verse en un navegador sin necesidad de servidor web (***natural templating***)
- Esta característica es ideal para la separación de roles: diseñadores y desarrolladores

Thymeleaf

Introducción

- La tecnología de plantillas JSP (Java Server Page) es la más extendida
- Ambas pueden ser usadas en conjunción con Spring MVC
- Thymeleaf permite realizar la maquetación HTML sin necesidad de que intervenga el servidor





- Más información: http://www.thymeleaf.org/doc/articles/thvsjsp.html

Thymeleaf

Introducción

- Thymeleaf está totalmente Integrado con Spring (MVC, Security)
- Soporta de dos tipo de lenguajes de expresiones (EL, *Expression Language*) para el acceso a objetos Java:
  - OGNL (*Object-Graph Navigation Language*):

```
${myObject.property}
```

```
${myObject.method()}
```

  - Spring EL (*Spring Expression Language*)

```
${@myBean.method()}
```

- Tutorial: http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

Thymeleaf

Introducción

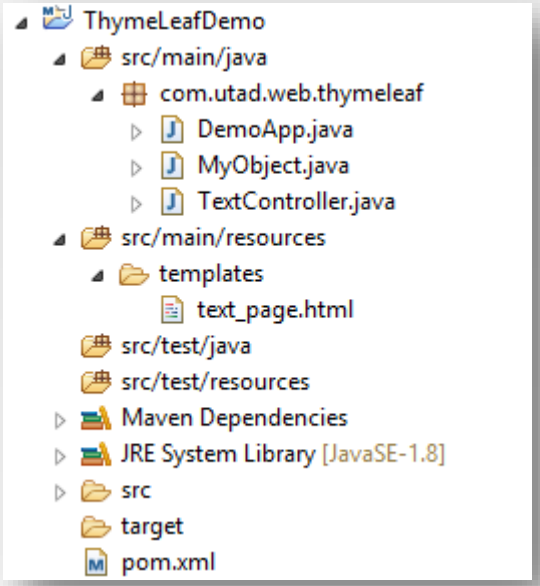
- La sintaxis de las plantillas Thymeleaf se define en las páginas HTML mediante la etiqueta `th`
- Los navegadores ignorarán el espacio de nombre que no entienden (`th`) con lo que la página seguirá siendo válida

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
   <p>Hello <span th:text="${name}"></span></p>
</body>
</html>
```

El atributo `xmlns` define el espacio de nombres (*XML Namespace*) para `th`. Un espacio de nombres permite definir nombres de elementos y atributos únicos en un documento XML (o HTML)

Thymeleaf

Ejemplo



pom.xml


```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>io.github.bonigarcia.web</groupId>
   <artifactId>ThymeLeafDemo</artifactId>
   <version>1.0.0</version>

   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.5.2.RELEASE</version>
   </parent>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <java.version>1.8</java.version>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-thymeleaf</artifactId>
      </dependency>
   </dependencies>
</project>
```

Thymeleaf

Ejemplo

SpringMvcThymeleafApp.java

```
package io.github.web.thymeleaf;

import org.springframework.boot.*;
import org.springframework.web.servlet.config.annotation.*;

@SpringBootApplication
public class SpringMvcThymeleafApp extends WebMvcConfigurerAdapter {

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

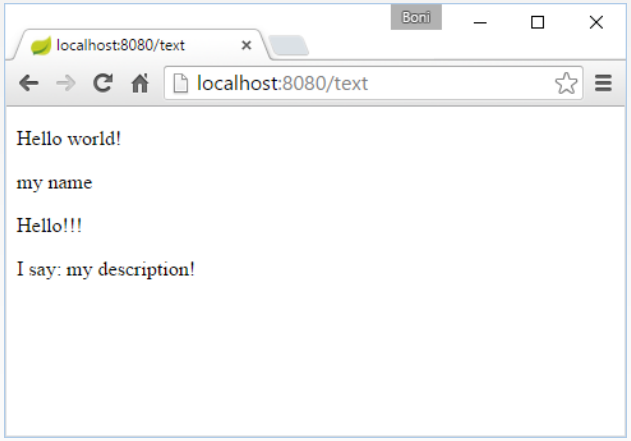
MyObject.java

```
package io.github.web.thymeleaf;

public class MyObject {

   private String name;
   private String description;

   public MyObject(String name, String description) {
      this.name = name;
      this.description = description;
   }

   public String sayHello() {
      return "Hello!!!";
   }
   // Getters and setters

}
```

Thymeleaf

Manejando texto

- Para mostrar texto en la plantilla usamos la etiqueta `th:text`



TextController.java

```
package io.github.web.thymeleaf;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class TextController {
   @RequestMapping("/text")
   public ModelAndView text() {
      MyObject myObject = new MyObject("my name", "my description");
      return new ModelAndView("text_page").addObject("greetings",
         "Hello world!").addObject("myobj", myObject);
   }
}
```

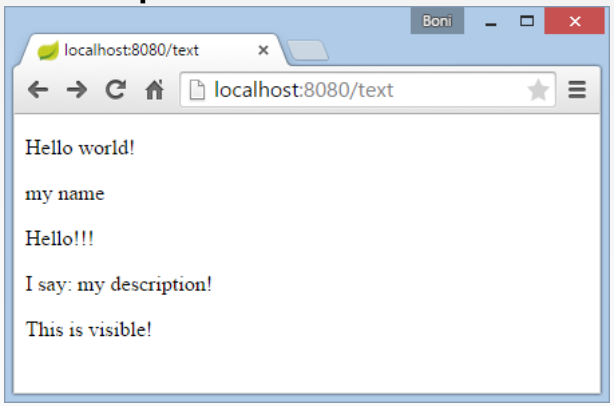
text_page.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<p th:text="${greetings}"></p>
<p th:text="${myobj.name}"></p>
<p th:text="${myobj.sayHello()}"></p>
<p th:text="|I say: ${myobj.description}!|"></p>
</body>
</html>
```

Thymeleaf

Condicionales

- Etiqueta `th:if`



TextController.java

```
package com.utad.web.thymeleaf;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class TextController {

   @RequestMapping("/text")
   public ModelAndView text() {
      MyObject myObject = new MyObject("my name", "my description");
      return new ModelAndView("text_page")
        .addObject("greetings", "Hello world!")
        .addObject("myobj", myObject).addObject("hidden", false);
   }
}
```

text_page.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<p th:text="${greetings}"></p>
<p th:text="${myobj.name}"></p>
<p th:text="${myobj.sayHello()}"></p>
<p th:text="|I say: ${myobj.description}!|"></p>
<p th:if="${not hidden}">This is visible!</p>
</body>
</html>
```

Thymeleaf

Iteración

- Etiqueta `th:each`

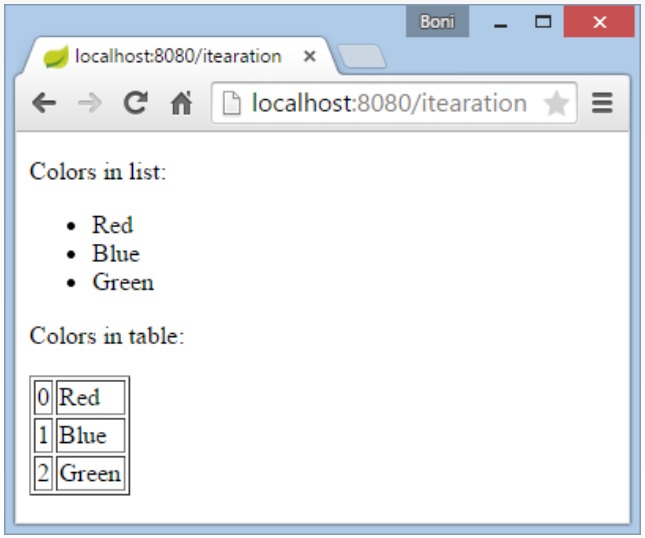
TextController.java

```
@RequestMapping("/itearation")
public ModelAndView iteration() {
   List<String> colors = Arrays.asList("Red", "Blue", "Green");
   return new ModelAndView("iteration_template").addObject("colors", colors);
}
```

iteration_template.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<p>Colors in list:</p>
<ul>
<li th:each="color : ${colors}" th:text="${color}">Color</li>
</ul>
<p>Colors in table:</p>
<table border="1">
<tr th:each="color, it : ${colors}">
<td th:text="${it.index}">1</td>
<td th:text="${color}">Color</td>
</tr>
</table>
</body>
</html>
```

Se puede declarar una variable adicional para guardar información de la iteración



Thymeleaf

Soporte HTML5

- Por defecto, las plantillas Thymeleaf tiene que se válidas a nivel XML
- Para evitar este problema potencial, simplemente hay que añadir que definir el modo a `LEGACYHTML5` y añadir una dependencia más a nuestro pom.xml

pom.xml

```
<dependency>
    <groupId>net.sourceforge.nekohtml</groupId>
    <artifactId>nekohtml</artifactId>
</dependency>
```

application.properties

```
spring.thymeleaf.mode=LEGACYHTML5
```

Envío de información al servidor

- Formas de enviar información del navegador al servidor:
  - **Mediante formularios HTML:** La información la introduce manualmente el usuario
  - **Insertando información en la URL en enlaces:** La información la incluye el desarrollador para que esté disponible cuando el usuario pulsa el enlace
- Acceso a la información en el servidor
  - La información se envía como pares clave=valor
  - Se accede a la información como parámetros en los métodos del controlador

Envío de información al servidor

Envío mediante formulario

- Ejemplo de formulario HTML (parte cliente) y controlador (parte servidor):

```
<!DOCTYPE html>
<html>
<body>
   <form action="processForm" method="post">
   <h1>Form</h1>
   <label for="input">Input</label>
   <input type="text" name="input"
      id="input">
   <input type="submit">
   </form>
</body>
</html>
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class FormController {

   @RequestMapping("/processForm")
   public ModelAndView process(@RequestParam String input) {
      return new ModelAndView("result").addObject("result",
        input);
   }
}
```

- Recordatorio: Existen dos métodos para enviar datos desde un formulario HTML: GET y POST

Envío de información al servidor

Envío mediante formulario

- Algo habitual es que un formulario tenga muchos campos
- En lugar de recoger uno a uno estos campos con `@RequestParam`, podemos definir un clase simple Java (“POJO”)
  - El nombre de los atributos de la clase coincide con los campos del formulario
- Recogemos esos datos en el controlador usando la anotación `@ModelAttribute` asociada al POJO que acabamos definir

Envío de información al servidor

Envío mediante formulario

- Ejemplo:

```
<!DOCTYPE html>
<html>
<body>
    <form action="processForm2" method="post">
        <label for="input">Information</label><br>
        <input type="text" name="info1" id="info2"><br>
        <input type="text" name="info2" id="info2">
        <input type="submit">
    </form>
</body>
</html>
```

```
@Controller
public class TextController {

    @RequestMapping("/processForm2")
    public ModelAndView process(@ModelAttribute MyForm info) {
        return new ModelAndView("result").addObject("result",
                info.getInfo1() + " and " + info.getInfo2());
    }

}
```

```
public class MyForm {

    private String info1;
    private String info2;

    // Getters and setters

    public String getInfo1() {
        return info1;
    }

    public void setInfo1(String info1) {
        this.info1 = info1;
    }

    public String getInfo2() {
        return info2;
    }

    public void setInfo2(String info2) {
        this.info2 = info2;
    }

}
```

Envío de información al servidor

Envío mediante URL (opción 1)

- La primera opción de enviar datos mediante URL consiste en simular el envío GET de un formulario:
  - Se incluyen al final de la URL separados con ? (*query*)
  - Los parámetros se separan entre sí con &
  - Cada parámetro se codifica como `nombre=valor`
- Ejemplo:

  http://my-server.com/path?option=web&view=category&lang=es

Envío de información al servidor

Envío mediante URL (opción 1)

- Para acceder a la información se usa el mismo mecanismo que para leer los campos del formulario
- Ejemplo:

  http://my-server.com/path?option=web&view=category&lang=es

```
@RequestMapping("/path")
public ModelAndView path(@RequestParam String option,
      @RequestParam String view, @RequestParam String lang) {
   // Create and return model
}
```

Envío de información al servidor

Envío mediante URL (opción 2)

- La información también se pueden incluir como parte de la propia URL, en vez de cómo parámetros
- Ejemplo:

  http://my-server.com/path/web/category/es

```
@RequestMapping("/path/{option}/{view}/{lang}")
public ModelAndView path(@PathVariable String option,
      @PathVariable String view, @PathVariable String lang) {
   // Create and return model
}
```

Envío de información al servidor

URLs en Thymeleaf

- Debido a su importancia, las URLs son “ciudadanos de primera clase” en Thymeleaf y tienen una sintaxis especial
- Los enlaces también se pueden construir en una plantilla con la información del modelo
- Ejemplo: Modelo con un objeto *id* :

```
<!-- Will produce '/order/details?orderId=3'  -->
<a href="details.html" th:href="@{/order/details(orderId=${id})}">view</a>

<!-- Will produce '/order/3/details'  -->
<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${id})}">view</a>
```

La etiqueta `th:href` generará el atributo `href` del enlace

Se usa con el símbolo `@`

Gestión de datos de sesión

- Es habitual que las aplicaciones web gestionen información diferente para cada usuario que está navegando. Por ejemplo:
  - Amigos en Facebook
  - Lista de correos en Gmail
  - Carrito de la compra en Amazon
- Se puede gestionar la información del usuario en dos ámbitos diferentes:
  - Información que se utiliza durante la navegación del usuario, durante la **sesión** actual
  - Información que se guarda mientras que el usuario no está navegando y que se recupera cuando el usuario vuelve a visitar la página web (**información persistente**)

Gestión de datos de sesión

- **Sesión**: Mantener información mientras el usuario navega por la web
  - Cuando el usuario pasa cierto tiempo sin realizar peticiones a la web, la sesión finaliza automáticamente (caducidad)
  - El tiempo para que caducidad es configurable
  - La información de sesión se guarda en memoria del servidor web
- **Información persistente**: Guardar información entre distintas navegaciones por la web
  - Para que podamos guardar información del usuario en el servidor, es necesario que el usuario se identifique al acceder a la página
  - La información se suele guardar en el servidor web en una BBDD
  - La lógica de la aplicación determina a qué información de la BBDD puede acceder cada usuario

Gestión de datos de sesión

- Vamos a ver dos técnicas posibles para gestionar datos de sesión:

1. Objeto `HttpSession`
   - Es la forma básica de gestión de sesiones en Java EE
   - Existe un objeto `HttpSession` por cada usuario que navega por la web
   - Se puede almacenar información en una petición y recuperar la información en otra petición posterior
   - Es de más bajo nivel
2. Componente específico para cada usuario
   - Cada usuario guarda su información en uno o varios componentes Spring
   - Existe una instancia por cada usuario (cuando lo habitual es tener una única instancia por componente)
   - Es de más alto nivel

Gestión de datos de sesión

Objeto `HttpSession`

- La sesión se representa como un objeto del interfaz `javax.servlet.http.HttpSession`
- El framework Spring es el encargado de crear un objeto de la sesión diferente para cada usuario
- Para acceder al objeto de la sesión del usuario que está haciendo una petición, basta incluirlo como parámetro en el método del controlador

```
@RequestMapping("/mypath")
public ModelAndView process(HttpSession session, ...) {
    Object info = ...;
    session.setAttribute("info", info);
    return new ModelAndView("template");
}
```

Gestión de datos de sesión

Objeto `HttpSession`

- Métodos de `HttpSession`:
  - `void setAttribute(String name, Object value)`: Asocia un objeto a la sesión identificado por un nombre
  - `Object getAttribute(String name)` : Recupera un objeto previamente asociado a la sesión
  - `boolean isNew()` : Indica si es la primera página que solicita el usuario (sesión nueva)
  - `void invalidate()` : Cierra la sesión del usuario borrando todos sus datos. Si visita nuevamente la página, será considerado como un usuario nuevo
  - `void setMaxInactiveInterval(int seconds)`: Configura el tiempo de inactividad para cerrar automáticamente la sesión del usuario.

Gestión de datos de sesión

Objeto `HttpSession`

- Vamos a ver el funcionamiento de `HttpSession` mediante un ejemplo:
  - La aplicación recoge la información de formulario y la guarda de dos formas:
    - Atributo del controlador (compartida)
    - Atributo de la sesión (usuario)
  - Una vez guardada la información, se puede acceder a ella y generar una página
  - Si dos usuarios visitan esta página a la misma vez, se puede ver cómo la información del controlador es compartida (la que guarda el último usuario es la que se muestra), pero la que se guarda en la sesión es diferente para cada usuario

Gestión de datos de sesión

Objeto `HttpSession`

- Ejemplo:

```
@Controller
public class SessionController {

    private String sharedInfo;

    @RequestMapping(value = "/processFormSession")
    public ModelAndView processForm(@RequestParam String info,
            HttpSession sesion) {

        sesion.setAttribute("userInfo", info);
        sharedInfo = info;
        return new ModelAndView("info_session");
    }

    @RequestMapping("/showDataSession")
    public ModelAndView showData(HttpSession session) {
        String userInfo = (String) session.getAttribute("userInfo");

        return new ModelAndView("data_session").addObject("userInfo", userInfo)
                .addObject("sharedInfo", sharedInfo);
    }

}
```

Gestión de datos de sesión

Objeto `HttpSession`

- Ejemplo:

```
<!DOCTYPE html>
<html>
<body>
    <form action="processFormSession" method="post">
        <label for="input">Information</label><br>
        <input type="text" name="info" id="info">
        <input type="submit">
    </form>
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    Info has been stored. Click
    <a href="/showDataSession">here</a>
    to show it.
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <p th:text="|User info: ${userInfo}|"></p>
    <p th:text="|Shared info: ${sharedInfo}|"></p>
</body>
</html>
```

Gestión de datos de sesión

Gestión de la sesión en Spring

- En Spring existe una forma de más **alto nivel** para gestionar sesiones
- Consiste en crear un `@Component` especial que se asociará a cada usuario y hacer `@Autowire` del mismo en el controlador que se utilice
- Ejemplo:

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
public class User {

   private String info;

   public void setInfo(String info) {
      this.info = info;
   }

   public String getInfo() {
     return info;
   }
}
```

La anotación `@Scope` con estos valores hace que haya un componente por cada usuario

Gestión de datos de sesión

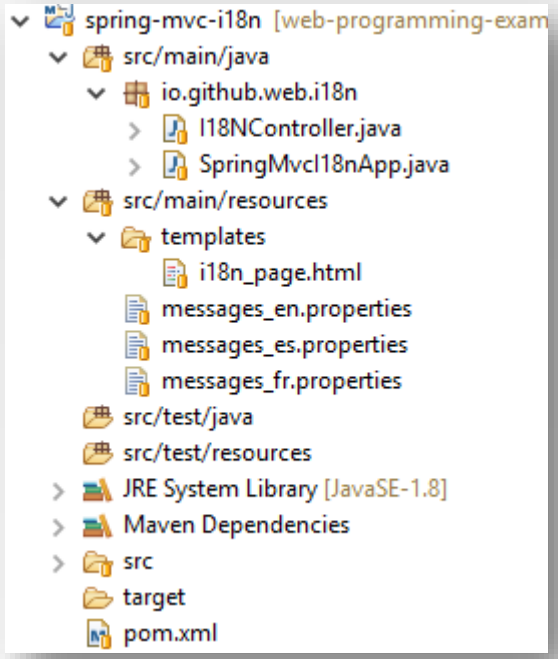
Gestión de la sesión en Spring

- Ejemplo:

```
@Controller
public class SessionController2 {

    @Autowired
    private User user;

    private String sharedInfo;

    @RequestMapping(value = "/processFormSession2")
    public ModelAndView processForm(@RequestParam String info) {
        user.setInfo(info);
        sharedInfo = info;
        return new ModelAndView("info_session2");
    }

    @RequestMapping("/showDataSession2")
    public ModelAndView showData() {
        String userInfo = user.getInfo();
        return new ModelAndView("data_session").addObject("userInfo", userInfo)
                .addObject("sharedInfo", sharedInfo);
    }
}
```

Soporte de internacionalización (I18N)

- Spring MVC con Thymeleaf tiene soporte nativo para aplicaciones multi-idioma (internacionalización, I18N)
- Vamos a estudiar este soporte mediante un ejemplo:



`messages_en.properties`

```
welcome=Welcome to my web!
```

`messages_es.properties`

```
welcome=Bienvenido a mi web!
```

`messages_fr.properties`

```
welcome=Bienvenue sur mon site web!
```

Soporte de internacionalización (I18N)

Nombre del fichero que contiene los mensajes de I18N

Región (locale) por defecto

Parámetro con el que cambiar la región desde URL

SpringMvcI18nApp.java

```
@SpringBootApplication
public class SpringMvcI18nApp extends WebMvcConfigurerAdapter {
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("messages");
        return messageSource;
    }
    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver sessionLocaleResolver = new SessionLocaleResolver();
        sessionLocaleResolver.setDefaultLocale(Locale.ENGLISH);
        return sessionLocaleResolver;
    }
    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor result = new LocaleChangeInterceptor();
        result.setParamName("lang");
        return result;
    }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

Soporte de internacionalización (I18N)

I18NController.java

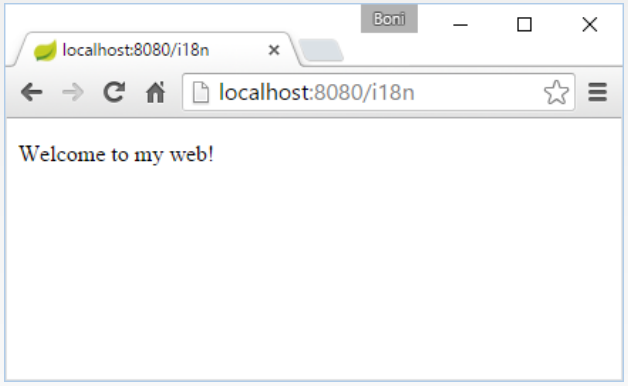
```java
package io.github.web.springmvc;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class I18NController {

   @RequestMapping("/i18n")
      public ModelAndView i18n() {
         return new ModelAndView("i18n_page");
   }

}
```

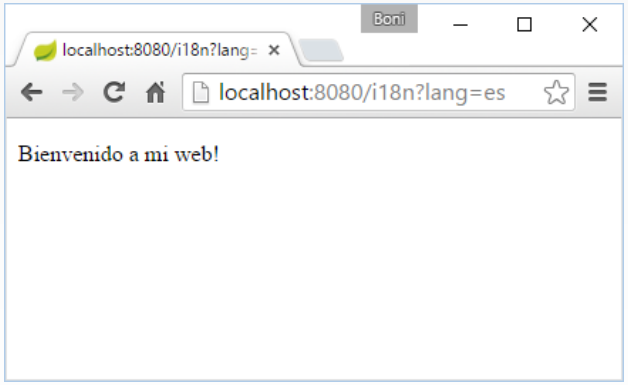
i18n_page.html

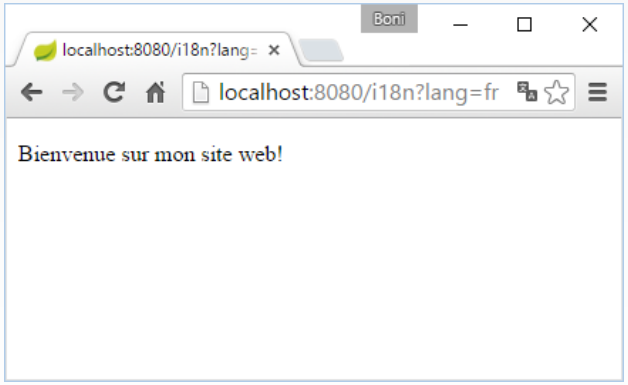
```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<p th:text="#{welcome}">Default greetings</p>
</body>
</html>
```

El símbolo # hace que la plantilla se rellene con los datos de I18N

Soporte de internacionalización (I18N)







Layouts

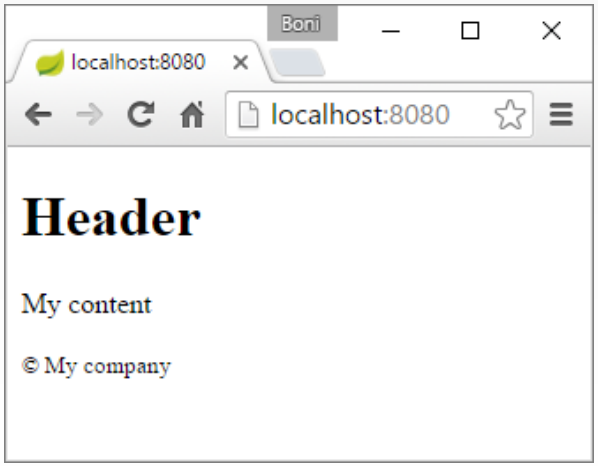
- Normalmente las aplicaciones web tienen partes comunes (cabecera, pie de página, menú, …)
- Thymeleaf proporciona mecanismos para realizar la disposición de los elementos de una página (*layout*) para evitar la duplicidad de estructura y código en las páginas web
- Las etiquetas Thymeleaf que nos permiten jugar con el *layout* son:
  - `th:fragment` : Permite definir un fragmento de página
  - `th:replace` : Permite sustituir un fragmento de página
  - `th:include` : Permite reemplazar un fragmento de página (la diferencia con `th:replace` es que con `th:include` incluye el contenido del fragmento previamente definido)

http://www.thymeleaf.org/doc/articles/layouts.html

Layouts

- Ejemplo:

```
@Controller
public class MyController {

    @RequestMapping(value = "/")
    public ModelAndView index() {
        return new ModelAndView("home");
    }

}
```



fragments.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<body>
    <div th:fragment="header">
        <h1>Header</h1>
    </div>

    <div th:fragment="footer">
        <small>&copy; My company</small>
    </div>
</body>
</html>
```

home.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<body>
    <p th:replace="fragments :: header"></p>

    <p>My content</p>

    <p th:include="fragments :: footer"></p>
</body>
</html>
```

Mediante esta notación (`::`) seleccionamos el fragmento definido dentro de una determinada plantilla

Layouts

- Todavía podemos reducir aún más la duplicación de código en el *layout* de nuestra aplicación web
- Para ello podemos crear una plantilla única para el *layout* general, variando el contenido de la misma en base
- En este caso vamos a usar las siguientes etiquetas Thymeleaf:
  - `layout:decorator` : Para definir la plantilla (*layout*) que vamos a usar
  - `layout:fragment` : Para definir el fragmento de la plantilla que redefinimos

Layouts

- Ejemplo:

```
@Controller
public class MyController {

 @RequestMapping(value = "/page1")
    public ModelAndView layout1() {
        return new ModelAndView("page1");
    }

    @RequestMapping(value = "/page2")
    public ModelAndView layout2() {
        return new ModelAndView("page2");
    }

}
```

layout.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<body>
    <div class="container">
        <div class="header">
            <h1>Header</h1>
        </div>
        <div class="content">
            <div class="nav">
                Section 1<br> Section 2<br> Section 3
            </div>
            <div class="main" layout:fragment="content"></div>
        </div>
        <div class="footer">Copyright &copy; Company.com</div>
    </div>
</body>
</html>
```


page1.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.
nz/thymeleaf/layout"
      layout:decorator="layout">
<body>
    <div layout:fragment="content">
        <h1>This is the page 1</h1>
        <p>Lorem ipsum ...</p>
    </div>
</body>
</html>
```