



Curso de Spring Framework

Tema 1

Desarrollo web con Spring

Micael Gallego

micael.gallego@gmail.com
@micael_gallego

Patxi Gortázar

patxi.gortazar@gmail.com
@fgortazar

- **Java Enterprise Edition**
- Spring
- Maven
- Spring MVC
- Inyección de dependencias
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

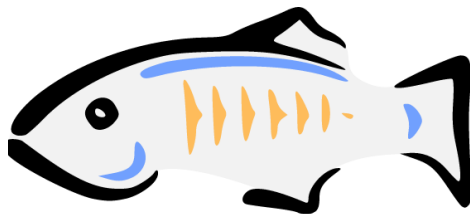
- **Java Enterprise Edition** (anteriormente conocido como **J2EE**) apareció a finales de los 90 para el desarrollo de aplicaciones **empresariales** con tecnología **Java**
- Actualmente, las aplicaciones empresariales más extendidas son las aplicaciones **web**
- Es la tecnología oficial soportada por **Oracle, IBM, RedHat**, etc..
- Las aplicaciones **Java EE** necesitan una versión estándar de Java (**Java SE**) para ejecutarse
- La versión actual es **Java EE 7**, que puede ejecutarse sobre **Java SE 7** o **Java SE 8**.

- Java EE proporciona una plataforma para construir aplicaciones **transaccionales, seguras, interoperables y distribuidas** con los siguientes servicios
 - **Servicios básicos**
 - ▢ Creación de páginas web con **Servlets y Java Server Pages (JSP)**
 - ▢ Acceso a bases de datos relacionales con **Java Database Connectivity (JDBC)**
 - ▢ Sistema de mapeo objeto-relacional con **Java Persistence API (JPA)**
 - **Servicios avanzados**
 - ▢ Gestión de componentes con **Enterprise Java Beans (EJB)**
 - ▢ Interfaz de usuario con **Java Server Faces (JSFs)**
 - ▢ Gestionar transacciones con **Java Transaction API (JTA)**
 - ▢ Envío y recepción de mensajes con **Java Message Service (JMS)**
 - Y muchos más

- **Estándares e implementaciones**
 - Java EE se ha hecho tan popular porque define **estándares** sobre cómo tienen que ser las librerías (**interfaz**) y cualquier empresa puede hacer una implementación **compatible** con el estándar
 - Esto permite que los desarrolladores puedan cambiar de implementación sin tener que modificar su aplicación (evita el *vendor locking*)
 - Por ejemplo, existen varias implementaciones del estándar **JPA**, siendo las más conocidas **Hibernate** y **EclipseLink**

- **Servidores Java EE**
 - Una aplicación Java EE se ejecuta dentro de un **contenedor** (*container*)
 - Los contenedores más usados son los servidores web: **Tomcat, Jetty, Glassfish, Websphere, JBoss, etc**
 - Existen **dos tipos** de servidores:
 - ▮ **Servidores de aplicaciones:** Contienen todos los servicios definidos en el estándar Java EE
 - ▮ **Servidores web:** Contienen los servicios esenciales de Java EE (servlets y JSP)

- Servidores Java EE
 - **Servidor de aplicaciones:** Contenedor que cumple con la especificación **Java EE**



Glassfish 4.0
(Java EE 7)

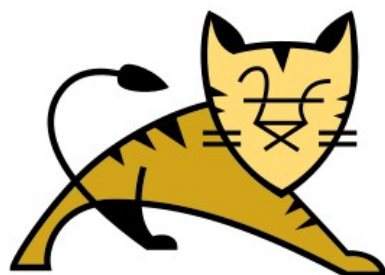


WildFly 8.0 (Jboss)
(Java EE 7)

- Servidores Java EE

- Servidores web:

- ▢ Contenedor que ofrece la APIs básicas de Servlets y JSPs
 - ▢ Se le pueden añadir otras librerías Java EE complementarias
 - ▢ Son mucho más **ligeros que los servidores Java EE**
 - ▢ La mayoría de las veces son suficientes



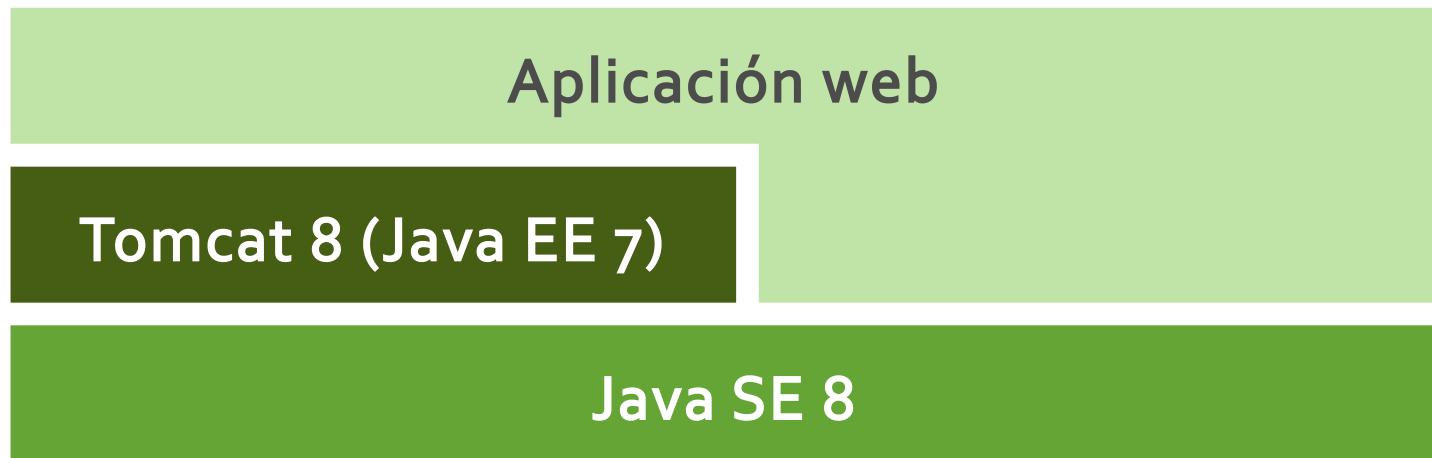
Apache Tomcat 8
(Servlets 3.1 y JSPs 2.3)



Eclipse Jetty 9.1
(Servlets 3.1 y JSPs 2.3)

- **Servidor web externo vs Servidor web embebido**
 - **Servidor web externo**
 - ▮ El servidor web se instala en el sistema operativo
 - ▮ Se puede configurar vía web
 - ▮ Puede ejecutar varias aplicaciones web a la vez
 - **Servidor web embebido**
 - ▮ El servidor web está incluido en la aplicación
 - ▮ La aplicación se instala en el sistema operativo
 - ▮ Sólo existe una aplicación (que incluye al servidor)

- Esquema aplicación web Java EE



Para implementar la **aplicación web** se pueden usar librerías de **Java SE 8** y las librerías de **Java EE 7** proporcionadas por **Tomcat 8**

- Java Enterprise Edition
- **Spring**
- Maven
- Spring MVC
- Inyección de dependencias
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

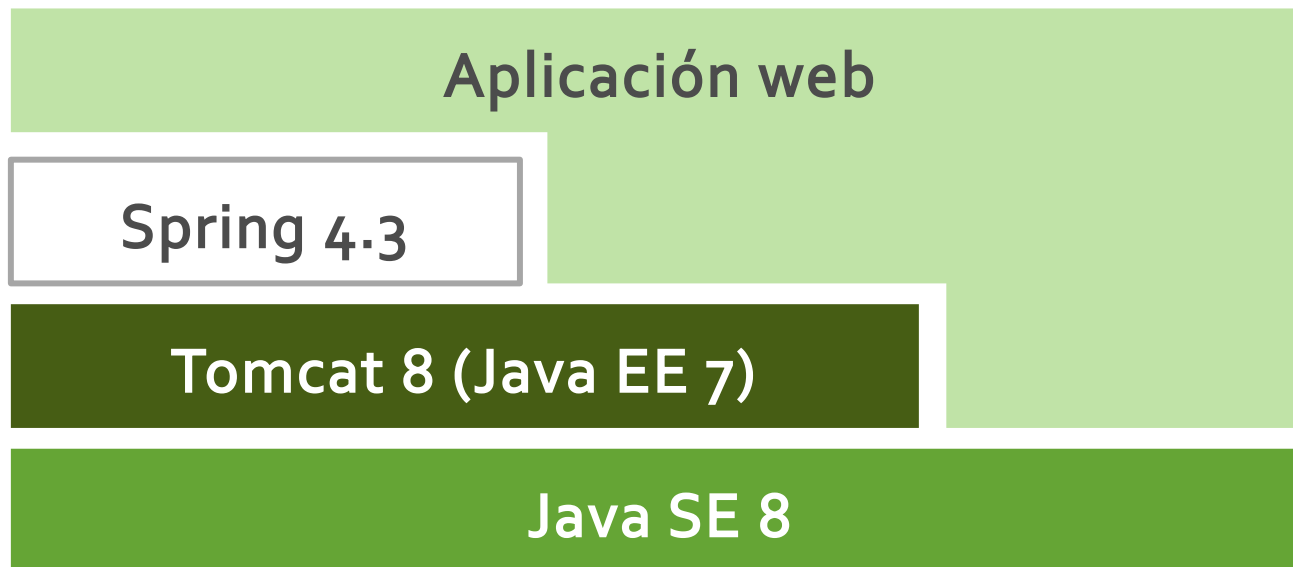
- Spring es un framework de desarrollo de **aplicaciones empresariales** basado en tecnologías Java
- La mayoría de sus partes están implementadas sobre **Java Enterprise Edition**



<http://spring.io/>

- **Java EE y Spring**
 - **Java EE** es un conjunto de librerías/frameworks estándar en Java
 - **Spring** es un framework software libre que se apoya en algunos estándares Java EE
 - Hay desarrolladores que sólo usan **Java EE** y otros que **combinan Spring y Java EE** en sus aplicaciones

- Java EE y Spring



Para implementar la **aplicación web** se pueden usar librerías de **Java SE 8**, las librerías de **Java EE 7** proporcionadas por **Tomcat 8** y la librería **Spring 4.2**

- Spring permite el desarrollo de **aplicaciones de servidor**:
 - **Aplicaciones web, servicios REST y websockets**
 - Análisis de datos big data
 - Procesado de tareas por lotes (Batch)
 - Integración de sistemas



INTEGRATION

Channels, Adapters, Filters,
Transformers



BATCH

Jobs, Steps, Readers, Writers



BIG DATA

Ingestion, Export, Orchestration,
Hadoop



WEB

Controllers, REST, WebSocket

- Tiene un completo soporte de **acceso a bases de datos** de diferentes tipos:
 - Relacionales (SQL)
 - No relacionales (No SQL)



- La versión actual es **Spring 4.3** y está diseñada para **Java 8** y **Java EE 7**
- Soporte de lenguajes dinámicos con **Groovy**
- Soporte de arquitecturas reactivas con **Reactor**



SPRING FRAMEWORK

Provides core support for dependency injection, transaction management, web apps, data access, messaging and more.



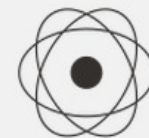
SPRING SECURITY

Protects your application with comprehensive and extensible authentication and authorization support.



GROOVY

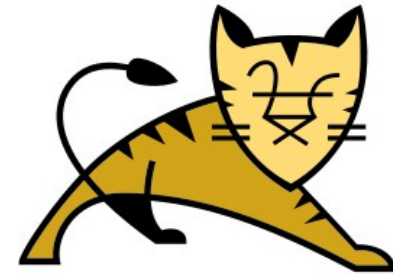
Brings high-productivity dynamic language features to the JVM.



REACTOR

A foundation for reactive fast data applications on the JVM.

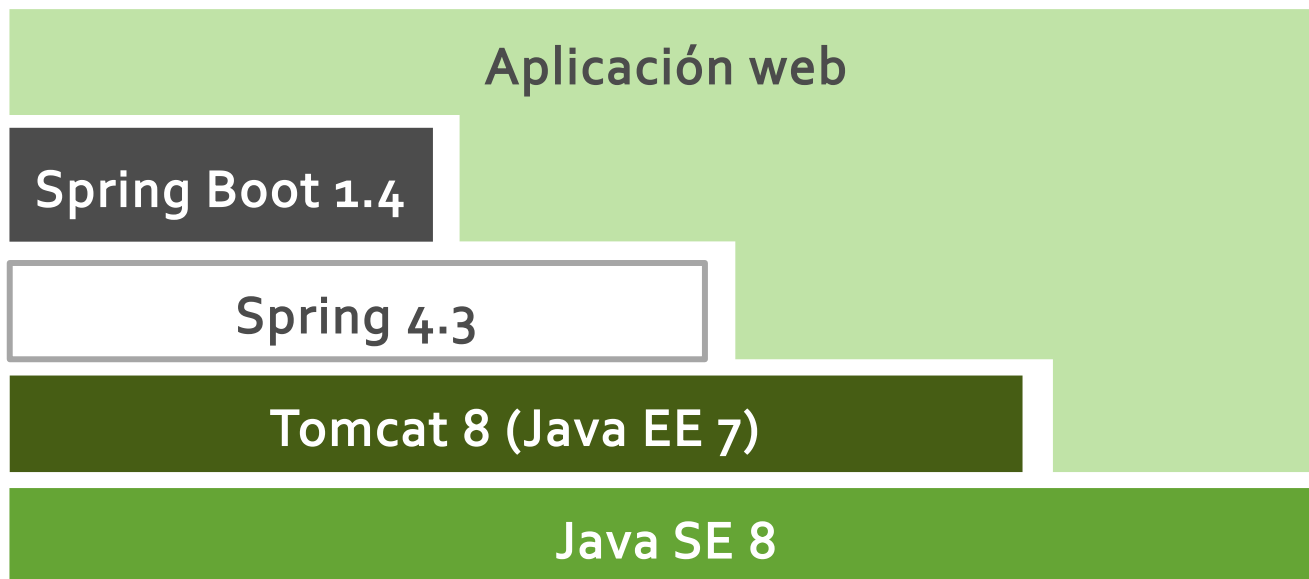
- **Servidor Java EE**
 - Una aplicación **Spring** puede ejecutarse tanto en un **servidor de aplicaciones (Java EE)** como en un **servidor web**
 - Es bastante habitual que las aplicaciones **Spring** se ejecuten en **Tomcat** o **Jetty**



- **Spring Boot**

- Es una librería que **facilita** el desarrollo de aplicaciones con Spring
- Permite usar el **servidor web Tomcat embebido** en la aplicación
- **Simplifica la configuración** y acelera el desarrollo
- Es una librería reciente, antes se implementaban las aplicaciones usando **directamente Spring**

- Spring Boot



Para implementar la **aplicación web** se pueden usar librerías de **Java SE 8**, las librerías de **Java EE 7** proporcionadas por **Tomcat 8**, la librería **Spring 4.3** y la librería **Spring Boot 1.4**

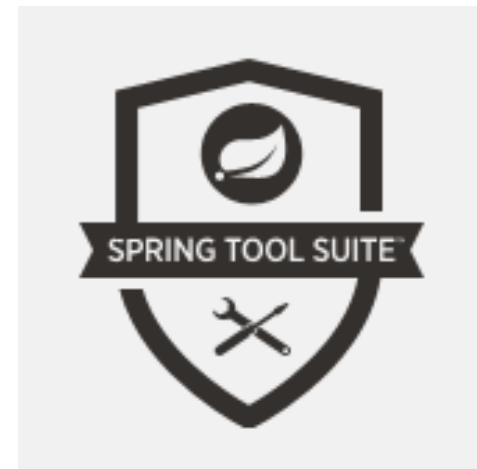
- Java Enterprise Edition
- Spring
- **Maven**
- Spring MVC
- Inyección de dependencias
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

- Sistema de **gestión de dependencias** (*librerías*) y sus versiones
- Sistema de **construcción de proyectos** (de código a entregable .zip)
- Estructura única de proyecto compatible con todos los **entornos de desarrollo** y sistemas de **integración continua**

- Maven gestiona muchos aspectos de la **construcción** de la aplicación
 - Descarga automática de dependencias
 - Compilación
 - Ejecución de tests
 - Publicación de la aplicación construida (binarios)
 - Documentación
 - ...

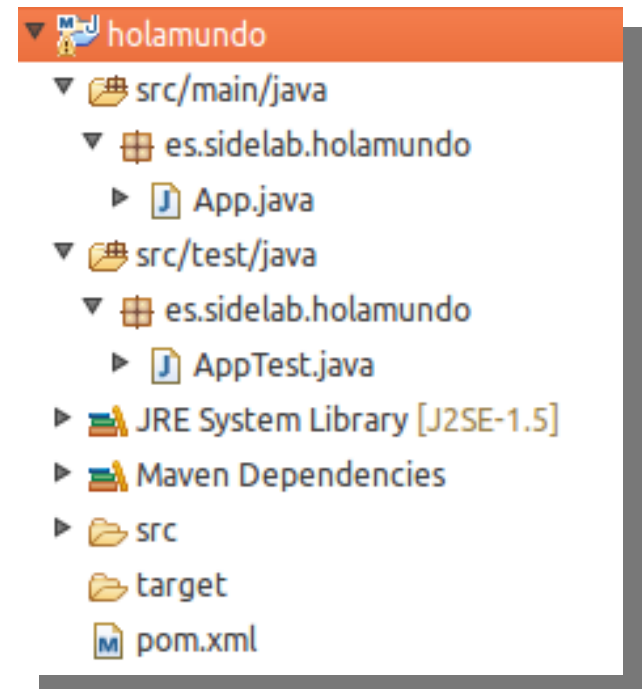
- **¿Cómo usar Maven?**
 - **Desde el entorno de desarrollo:** Maven está integrado en los IDEs Java más importantes (eclipse, netbeans, IntelliJ...)
 - **Desde la línea de comandos:** Sin necesidad de usar un IDE. Ideal para construcción de proyectos de forma automatizada

- **Spring Tool Suite (STS)**
 - En la asignatura usaremos una versión de **Eclipse** diseñada para desarrollo de aplicaciones **Spring**
 - <http://spring.io/tools>
 - Proporciona facilidades para trabajar con aplicaciones **Spring**
 - Está basado en la última versión de Eclipse

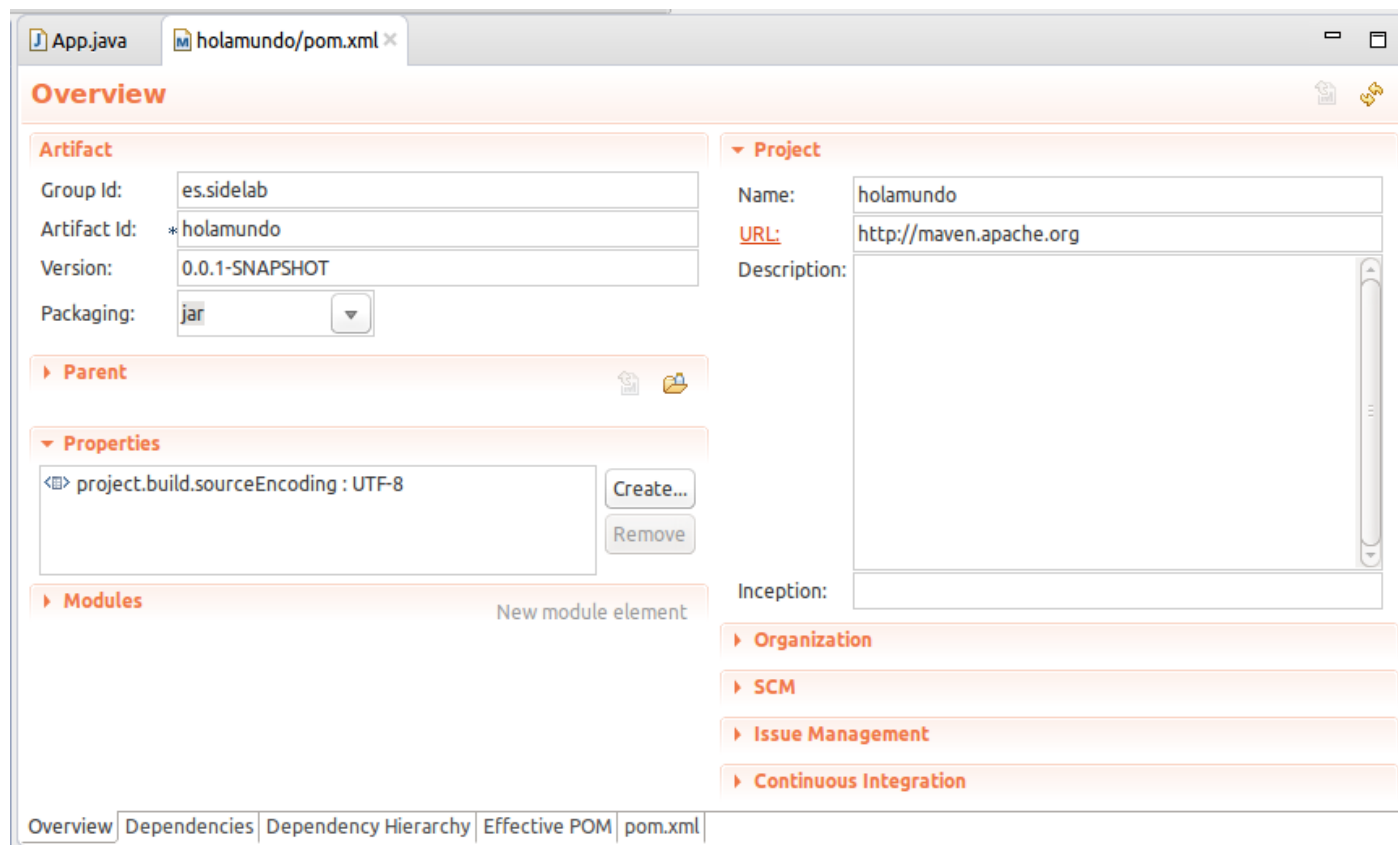


- **Cómo crear un proyecto Maven en Eclipse**
 - Nuevo proyecto > Maven > Maven Project
 - Dejar la plantilla que aparece por defecto seleccionada (**maven-archetype-quickstart**)
 - Indicar el nombre del proyecto:
 - GroupId: es.sidelab
 - ArtifactId: holamundo

- La mayoría de los proyectos Maven tienen la siguiente **estructura**
 - **src/main/java**: Código de la aplicación
 - **src/test/java**: Código de los tests
 - **pom.xml**: Fichero de descripción del proyecto (nombre, dependencias, configuraciones, etc...)



- **pom.xml**: Configuración del proyecto



- pom.xml: Configuración del proyecto



```
1<?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4
5    <groupId>es.sidelab</groupId>
6    <artifactId>holamundo</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>holamundo</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
21       <version>3.8.1</version>
22       <scope>test</scope>
23     </dependency>
24   </dependencies>
25 </project>
```

Poner la vista de código fuente

- **pom.xml: Configuración del proyecto**
 - **groupId:** Organización, familia
 - **artifactId:** Nombre del proyecto
 - **version:** Versión del proyecto (especialmente útil para librerías)
 - **packaging:** Tipo de aplicación (jar es una aplicación normal)
 - **name:** Nombre “bonito” del proyecto (para documentación)
 - **url:** Web del proyecto (para documentación)

```
<groupId>es.sidelab</groupId>  
<artifactId>holamundo</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<packaging>jar</packaging>
```

```
<name>holamundo</name>  
<url>http://maven.apache.org</url>
```

- **pom.xml:** Configuración del proyecto
 - **properties:**
 - ▮ Configuraciones generales del proyecto
 - ▮ Versión de Java (Si no se pone nada es la 1.5)
 - ▮ Codificación de los ficheros fuente

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

- **pom.xml: Configuración del proyecto**
 - **dependencies:**
 - ▮ Dependencias (**librerías**)
 - ▮ Cada librería está identificada por su **groupId**, **artifactId** y **versión** (coordenadas)
 - ▮ Se pueden poner tantas dependencias como se quiera

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```


- **pom.xml:** Configuración del proyecto
 - **Cuidado!** Algunos cambios en el fichero pom.xml no se reflejan en eclipse de forma automática
 - Cuando se hace un cambio y eclipse no se actualiza con esos cambios, se tiene que indicar explícitamente
 - ▮ Botón derecho proyecto > Maven > Update Project...

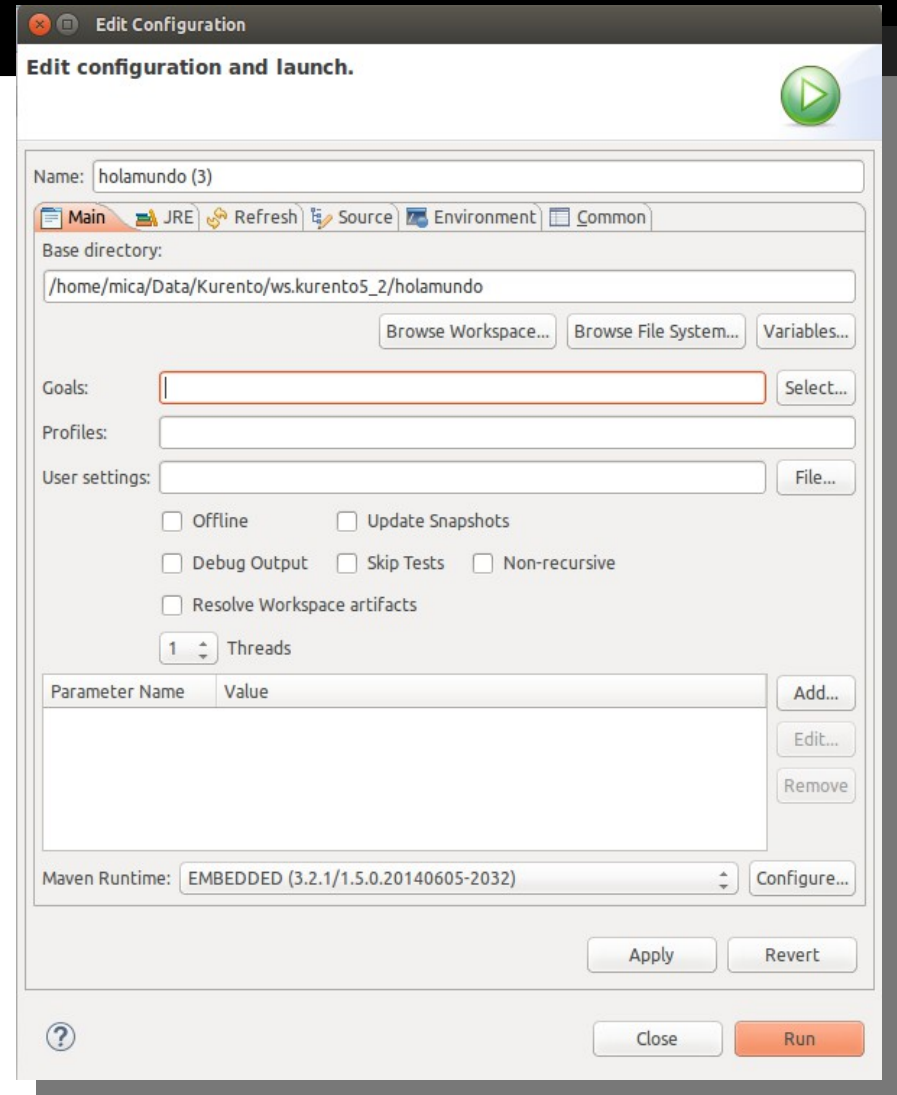
- **Construir un proyecto Maven**
 - **De forma automática:**
 - ▮ Eclipse se descarga las librerías y compila el código.
 - ▮ Eclipse no empaqueta la aplicación en un .jar
 - **De forma manual:**
 - ▮ Se pueden ejecutar operaciones Maven desde eclipse o desde la línea de comandos (si está maven instalado en el sistema)

- **Tareas básicas de Maven**

- **compile:** Compila el código
- **test:** Ejecuta los tests del proyecto
- **package:** Empaqueta el proyecto, habitualmente generando un fichero .jar
- **install:** Instala el paquete generado para que esté accesible a otros proyectos de la misma máquina
- **deploy:** Publica el paquete generado para que esté accesible para otros desarrolladores del equipo o públicamente en Internet
- **clean:** Limpia la carpeta de los ficheros generados (binarios, zips, etc...)

- Las tareas tienen **dependencias** entre ellas. Por ejemplo, al ejecutar **install**, se ejecuta antes automáticamente **compile**, **test** y **package**.
- La tarea **clean** es **independiente** de las demás

- Ejecutar tareas Maven desde eclipse
 - Botón derecho > Run As
 - **mvn clean**
 - **mvn install**
 - **mvn build...:** Permite crear una tarea nueva configurable
 - **mvn build:** Permite seleccionar una tarea previamente ejecutada

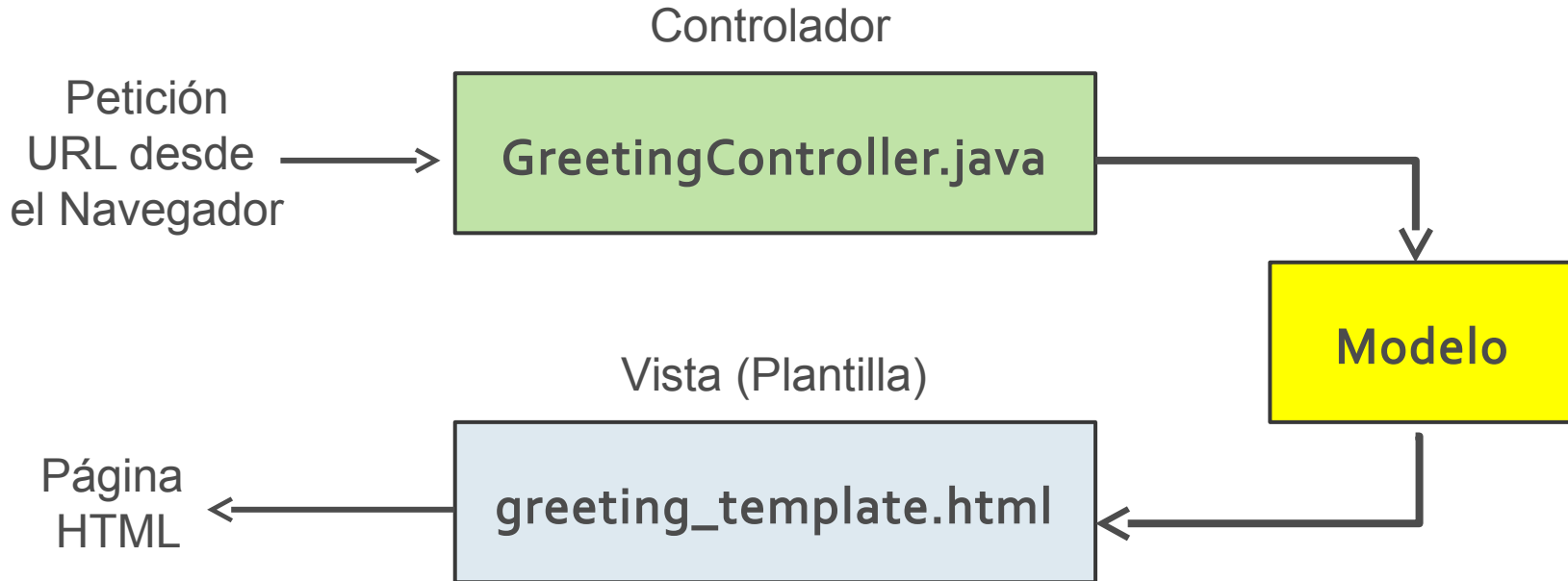


- **Repositorio de dependencias (librerías)**
 - Cuando Maven **construye un proyecto**, las dependencias se cargan desde un repositorio
 - **Repositorio en la red**
 - ▮ Existen varios repositorios públicos en Internet (**Maven Central Repository**)
 - ▮ Es habitual que las empresas tengan un **repositorio privado** para toda la organización en su red local
 - **Repositorio en la máquina de desarrollo ~/.m2/repository**
 - ▮ Guarda las librerías que se han **descargado** para que estén disponibles para construir de nuevo el proyecto o para **otros proyectos**
 - ▮ Guarda las librerías que **construyen** para que estén disponibles para otros proyectos

- Java Enterprise Edition
- Spring
- Maven
- **Spring MVC**
- Inyección de dependencias
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

- **Spring MVC** es una parte de Spring para la construcción de aplicaciones web
- Sigue la arquitectura **MVC** (*Model View Controller*)
- **Permite estructurar la aplicación en:**
 - **Model:** Modelos de datos (objetos Java)
 - **View:** Plantilla que genera la página HTML
 - **Controller:** Controlador que atiende las peticiones http que llegan del navegador

- Aplicación web básica Spring MVC



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

- **Controlador**

- Los **controladores** (*Controllers*) son las clases encargadas de atender las peticiones web
 - 1) **Manipulan los datos** que llegan con la petición (hacen peticiones a la BBDD, utilizan servicios externos...)
 - 2) **Obtienen los datos** que se visualizará en la página (**modelo**)
 - 3) Deciden qué **plantilla generará el HTML** partiendo de esos datos (**vista**)

- Controlador

GreetingController.java

```
@Controller
public class GreetingController {

    @RequestMapping("/greeting")
    public String greeting(Model model) {

        model.addAttribute("name", "World");

        return "greeting_template";
    }
}
```

Se indica qué **URL** debe llevar la petición para ejecutar el controlador

Se añade al parámetro model la **información** que será visualizada en la página web

El método devuelve el **nombre de la plantilla** que será usada para generar el HTML partiendo del modelo

- **Vista (Plantillas)**

- Las vistas en **SpringMVC** se implementan como plantillas HTML definidas en base a la información del modelo
- Existen diversas tecnologías de plantillas: **JSP (estándar), Mustache, Thymeleaf, FreeMarker, etc...**
- Nosotros usaremos **Mustache**

ejem1

- Vista (Plantillas)

greeting_template.html

```
<html>
<body>
  <p>Hello, {{name}}</p>
</body>
</html>
```

En las plantillas se indican los elementos del modelo para que sean sustituidos por sus valores al generar el HTML

Plantilla implementada con la librería **Mustache**

<https://mustache.github.io/>

SpringMVC

- pom.xml

Proyecto padre
para aplicaciones
SpringBoot

Dependencias
necesarias para
implementar
aplicaciones web
Spring MVC con
Mustache y
SpringBoot

```
<?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>es.urjc.code.daw</groupId>
  <artifactId>ejeml</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.1.RELEASE</version>
    <relativePath/>
  </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-mustache</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

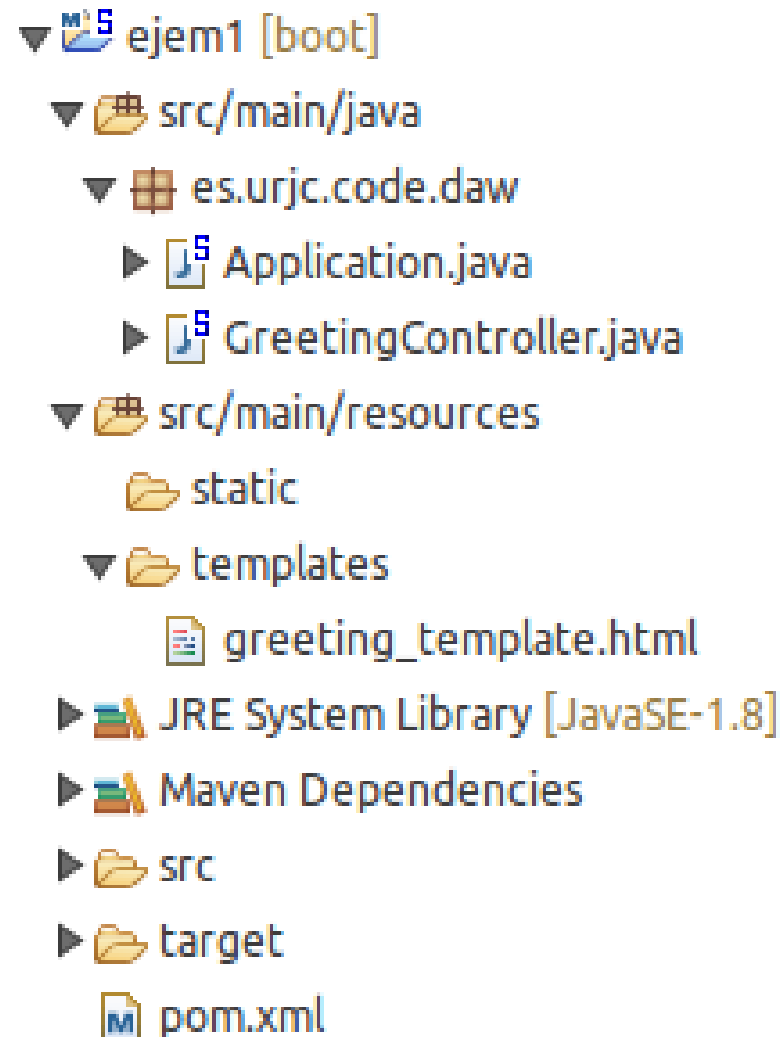
</project>
```

- **Aplicación principal**
 - La aplicación se ejecuta como una **app Java normal**
 - Botón derecho proyecto > Run as... > Java Application...

```
@SpringBootApplication
public class Application {

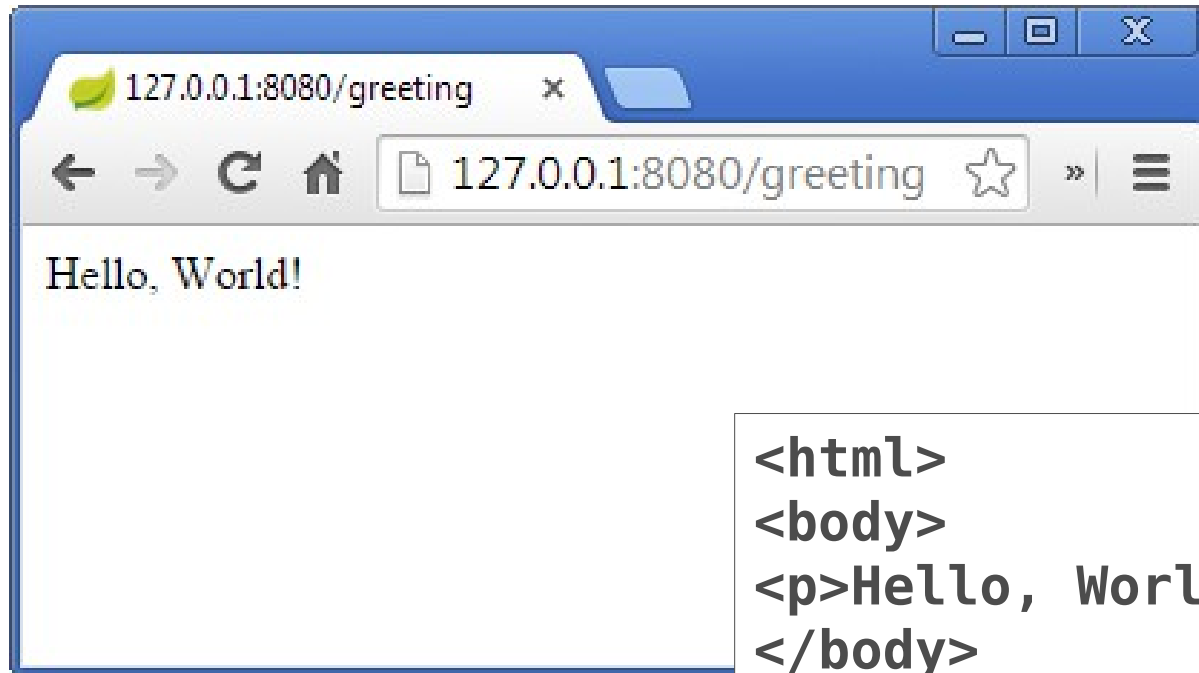
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- Estructura de la aplicación



ejem1

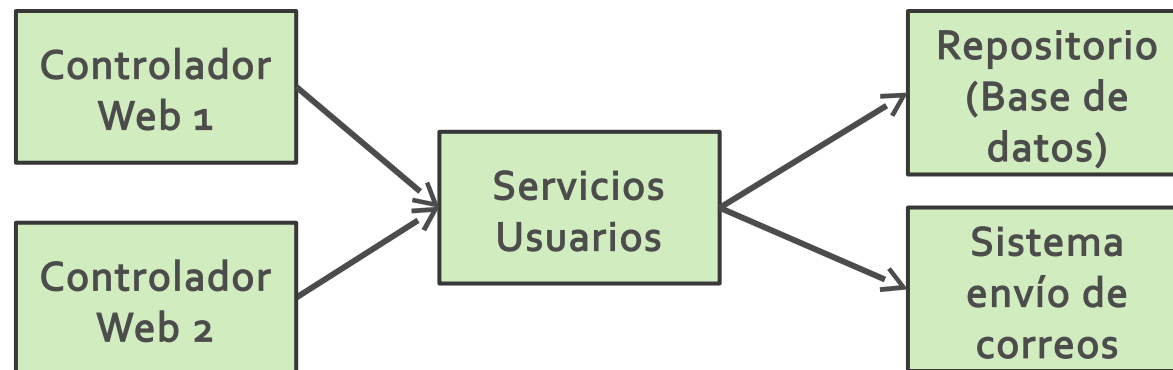
- Ejecución de la aplicación



```
<html>
<body>
<p>Hello, World!</p>
</body>
</html>
```


- Java Enterprise Edition
- Spring
- Maven
- Spring MVC
- **Inyección de dependencias**
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

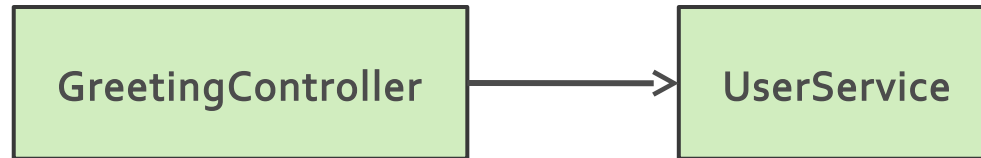
- Las aplicaciones se suelen dividir en **módulos de alto nivel**
- Algunos **módulos** ofrecen servicios a otros módulos
- **Ejemplo:** Diseño modular de una aplicación web con SpringMVC



- ¿Cómo se **implementa un módulo**?
- ¿Cómo se **conecta un módulo** a otro módulo?
- La **inyección de dependencias** es una técnica que permite especificar un módulo y sus dependencias
- Cuando se inicia la aplicación, el framework crea todos los módulos e **inyecta las dependencias** en los módulos que las necesitan
- **Spring** dispone de un sistema de inyección de dependencias interno

DESARROLLO WEB CON SPRING

Inyección de dependencias



```
@Controller
public class GreetingController {

    @Autowired
    private UsersService usersService;

    @RequestMapping("/greeting")
    public String greeting(Model model) {

        model.addAttribute("name",
            usersService.getNumUsers()+" users");

        return "greeting_template";
    }
}
```

```
@Component
public class UsersService {

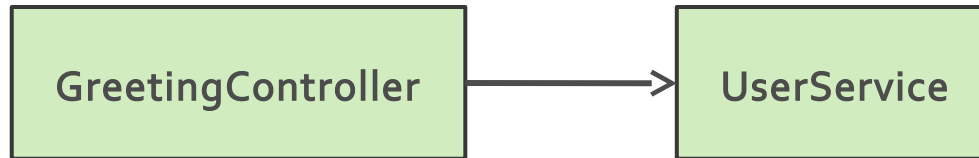
    public int getNumUsers(){
        return 5;
    }
}
```

- A los módulos de la aplicación Spring se los denomina **beans** o **componentes**
- Para que una clase se considere un componente, tiene que anotarse con **@Component** **@Controller** o **@Service**
- Si un componente depende otro, puede poner la anotación **@Autowired** (auto enlazado) en un atributo, un constructor o un método setter

<http://docs.spring.io/spring/docs/4.3.3.RELEASE/spring-framework-reference/htmlsingle/#beans>

DESARROLLO WEB CON SPRING

Inyección de dependencias



```
@Controller
public class GreetingController {

    @Autowired
    private UserService userService;

    @RequestMapping("/greeting")
    public String greeting(Model model) {

        model.addAttribute("name",
            userService.getNumUsers()+" users");

        return "greeting_template";
    }
}
```

```
@Component
public class UserService {

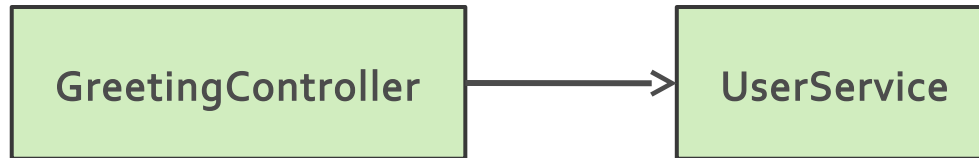
    public int getNumUsers() {
        return 5;
    }
}
```

- Existen casos en los que los componentes de una aplicación vienen en librerías (no los podemos modificar) pero tienen que ser **configurados en la aplicación**
- Existen otros casos en los que existen **varias implementaciones** disponibles de un componente y la aplicación tiene que **seleccionar** la implementación concreta
- En estos casos, la aplicación puede **configurar los componentes** de la aplicación

- En las aplicaciones **SpringBoot**, la clase principal de la aplicación se utiliza para **configurar los componentes**
- Por cada componente que se quiera **configurar**:
 - Se **quita la anotación** `@Component` del componente
 - Se **añade un método** en la clase principal que devuelva un nuevo componente configurado

DESARROLLO WEB CON SPRING

Inyección de dependencias



```
@Controller
public class GreetingController {

    @Autowired
    private UserService userService;

    @RequestMapping("/greeting")
    public String greeting(Model model) {

        model.addAttribute("name",
            userService.getNumUsers()+" users");

        return "greeting_template";
    }
}
```

```
public class UserService {

    private int numUsers;

    public UserService(int numUsers){
        this.numUsers = numUsers;
    }

    public int getNumUsers() {
        return numUsers;
    }
}
```

DESARROLLO WEB CON SPRING

Inyección de dependencias

ejem3

```
@SpringBootApplication
public class Application {

    @Bean
    public UserService userService() {
        return new UserService(10);
    }

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

En la clase de la aplicación se configura el componente

Se implementa un método anotado con **@Bean** que devuelve el componente ya configurado

Desarrollo Web con Spring

- Java Enterprise Edition
- Spring
- Maven
- Spring MVC
- Inyección de dependencias
- **Generación de HTML con Mustache**
- Proceso de formularios y enlaces
- Sesión: Datos de cada usuario

- Las **plantillas** y los **controladores** permiten la generación de páginas **HTML**
- Existen muchas tecnologías de plantillas, nosotros usaremos **Mustache**

GreetingController.java

```
@Controller
public class GreetingController {

    @RequestMapping("/greeting")
    public String greeting(Model model) {

        model.addAttribute("name", "World");

        return "greeting_template";
    }
}
```

greeting_template.html

```
<html>
<body>
    <p>Hello, {{name}}</p>
</body>
</html>
```

- Con **SpringMVC** pueden usarse diversas tecnologías de generación de páginas HTML

<FreeMarker>

<http://freemarker.org/>



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



<http://www.thymeleaf.org/>



<http://velocity.apache.org/>



<http://mustache.github.io/>

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

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

Ejemplo implementado con **Mustache**

- **Mustache**

- Esta formato de plantillas se puede usar en múltiples lenguajes de programación (Ruby, C++, Rust, ASP, C...)
- También se puede usar en el navegador porque hay implementaciones para JavaScript



<http://mustache.github.io/>

- **Funcionalidades básicas**
 - Uso de variables del modelo
 - Generación de HTML si una expresión es true
 - Generación de listas o tablas HTML con el contenido de objetos del modelo de tipo lista

Tutorial sobre Mustache

<https://mustache.github.io/mustache.5.html>

Documentación de la implementación para Java

<https://github.com/samskivert/jmustache>


```
@RequestMapping("/basic")
public String basic(Model model) {


    model.addAttribute("name", "World");
    model.addAttribute("silent", true);

    return "basic_template";
}
```

```
<html>
<body>
    <p>Hello, {{name}}</p>
    {{#silent}}
    <p>Hello!</p>
    {{/silent}}
</body>
</html>
```

Uso de atributos del modelo

Generación condicional. Sólo se muestra si el objeto **silent** es **false**



```
<html>
<body>
    <p>Hello, World</p>
    <p>Hello!</p>
</body>
</html>
```

```
@RequestMapping("/list")
public String iteration(Model model) {

    List<String> colors = Arrays.asList("Red", "Blue", "Green");
    model.addAttribute("colors", colors);
    return "list_template";
}
```

```
<html>
<body>
    <p>Colors in list:</p>
    <ul>
        {{#colors}}
            <li>{{.}}</li>
        {{/colors}}
    </ul>
    <p>Colors in table:</p>
    <table>
        {{#colors}}
            <tr>
                <td>{{-index}}</td>
                <td>{{.}}</td>
            </tr>
        {{/colors}}
    </table>
</body>
</html>
```

Repite la etiqueta por cada elemento de la lista

El **punto** hace referencia al objeto en esa iteración

Se pueden usar variables especiales como **-index** que tienen el índice de la iteración

```
<html>
<body>
  <p>Colors in list:</p>
  <ul>
    {{#colors}}
      <li>{{.}}</li>
    {{/colors}}
  </ul>
  <p>Colors in table:</p>
  <table>
    {{#colors}}
      <tr>
        <td>{{-index}}</td>
        <td>{{.}}</td>
      </tr>
    {{/colors}}
  </table>
</body>
</html>
```



```
<html>
  <body>
    <p>Colors in list:</p>
    <ul>
      <li>Red</li>
      <li>Blue</li>
      <li>Green</li>
    </ul>
    <p>Colors in table:</p>
    <table>
      <tr>
        <td>1</td>
        <td>Red</td>
      </tr>
      <tr>
        <td>2</td>
        <td>Blue</td>
      </tr>
      <tr>
        <td>3</td>
        <td>Green</td>
      </tr>
    </table>
  </body>
</html>
```

ejem4

Desarrollo Web con Spring

- Java Enterprise Edition
- Spring
- Maven
- Spring MVC
- Inyección de dependencias
- Generación de HTML con Mustache
- **Proceso de formularios y enlaces**
- Sesión: Datos de cada usuario

- **Formas de enviar información del navegador al servidor**
 - **Mediante formularios HTML:** La información la introduce manualmente el usuario
 - **Insertando información en la URL de enlaces:** La información la incluye el desarrollador para que esté disponible cuando el usuario pulse 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

- Creación de formularios en HTML

- La etiqueta `<form>` contiene los elementos del formulario
- Puede contener otros elementos HTML

```
<form action='url_controlador'>  
...  
</form>
```

- **action:** URL del controlador que será ejecutado al enviar los datos al servidor pulsando el botón de enviar (*submit*)

- Creación de formularios en HTML

- Tiene que existir al menos un **botón para enviar** los datos del formulario

```
<form action='url_controlador'>  
  ...  
  <input type='submit' value='Enviar'>  
</form>
```

Botón con texto

```
<form action='url_controlador'>  
  ...  
  <input type='image' src='imagen.gif'>  
</form>
```

Botón gráfico

- Creación de formularios en HTML

- Campo de texto

```
<input type='text' name='nombreParametro'>
```

- Área de texto

```
<textarea name='nombreParametro' rows=5 cols=40>  
Texto del cuadro de texto </textarea>
```

- Casilla de verificación (checkbox)

```
<input type='checkbox' name='nombreParametro'  
value='valorOpcion'>Texto Opción
```

- Acceso a los datos desde el controlador
 - Los valores se recogen con parámetros del método del controlador con anotaciones **@RequestParam**

src/main/resources/static/index.html

```
<form action="greeting">
  <p>Saludar a :</p>
  <input type='text' name='name' />
  <input type='submit' value='Enviar' />
</form>
```

Parámetro con el valor
del campo de texto del
formulario

```
@RequestMapping("/greeting")
public String greeting(Model model, @RequestParam String name) {

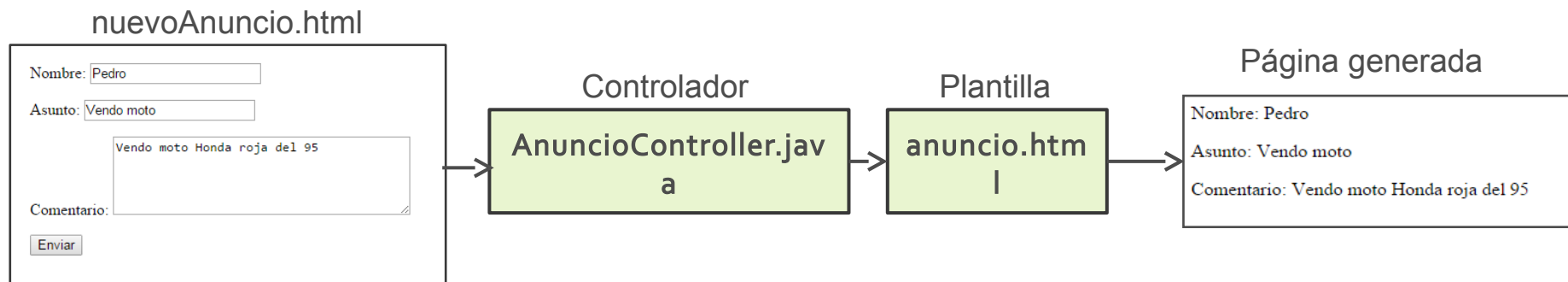
    model.addAttribute("name", name);

    return "greeting_template";
}
```

Ejercicio 1

- Crear una página **html estática** que muestre un **formulario** para enviar al servidor información de nuevos anuncios.
- En el formulario debe aparecer:
 - Campo de texto para el **nombre** del usuario
 - Campo de texto para el **asunto** del mensaje
 - Área de texto para el **cuerpo** del mensaje
 - **Botón de envío**
- Implementar un **controlador** que sea ejecutado al pulsar el botón de envío del formulario y recoja los datos del formulario
- Diseñar una **plantilla** que muestre el anuncio que se ha enviado al servidor

Ejercicio 1



Proceso de formularios y enlaces

- Existen dos formas de enviar la información de un formulario al servidor
 - Método GET*

```
<form method='get'  
action='url_controlador'  
...  
</form>
```

- Método POST

```
<form method='post'  
action='url_controlador'  
...  
</form>
```

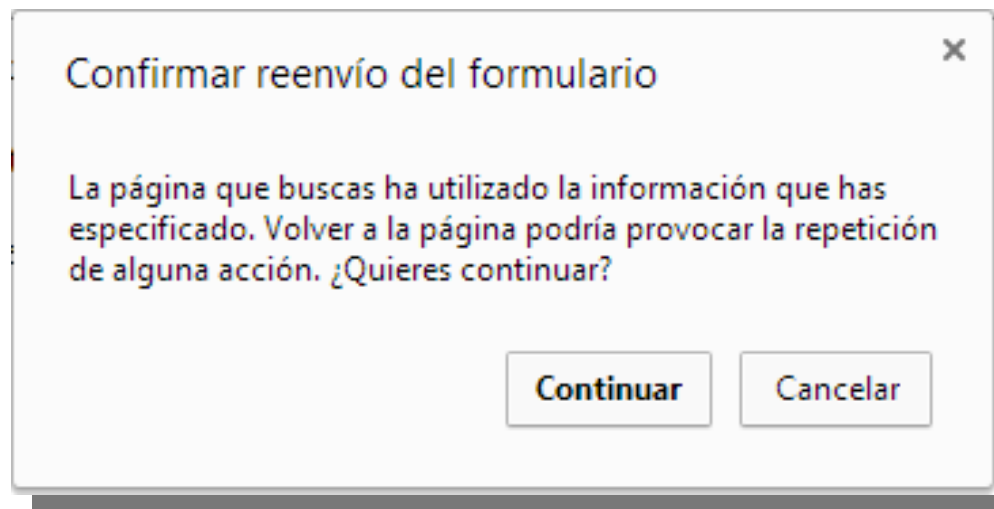
* El método GET se usa por defecto si no se especifica ningún método

- Existen dos formas de enviar la información de un formulario al servidor
 - **Método GET (por defecto)**
 - ▮ Se usa el método GET del protocolo http
 - ▮ El navegador incluye la información del formulario en la URL que solicita al servidor
 - **Método POST**
 - ▮ Se usa el método POST del protocolo http
 - ▮ El navegador incluye la información del formulario en el cuerpo de la petición (no es visible por el usuario)

- En el servidor podemos especificar si un método debe ser invocado por GET o POST
 - Spring < 4.3
 - @RequestMapping(...,
method=RequestMethod.POST)
 - @RequestMapping(...,
method=RequestMethod.GET)
 - Spring >= 4.3
 - @PostMapping(...)
 - @GetMapping(...)

- **Se recomienda usar el método POST en formularios**
 - Las URLs quedan más limpias (porque no incluyen los parámetros)
 - La página que aparece al enviar el formulario se puede añadir a la lista de marcadores del navegador

- Se recomienda usar el método POST en formularios
 - Si se intenta recargar la página que aparece al enviar el formulario, el navegador muestra un aviso al usuario de que es posible que recargar la página tenga efectos no deseados



- **Insertando información en la URL de los enlaces**
 - Es habitual que los desarrolladores incluyan información en la URL para que esté disponible en el servidor cuando el usuario pulsa el enlace
 - Formato:

URL con parámetros

```
http://www.traumainforma.com/nuevoUsuario?
option=com_weblinks&view=category&lang=es
```

- ▮ Los parámetros se incluyen al final de la URL separados con ? (query)
- ▮ Los parámetros se separan entre sí con &
- ▮ Cada parámetro se codifica como nombre=valor

- Insertando información en la URL de los enlaces
 - Formato: Codificación de los nombres y los valores
 - ▢ Los caracteres alfanuméricos "a" hasta "z", "A" hasta "Z" y "0" hasta "9" se quedan igual
 - ▢ Los caracteres especiales ".", "-", "*", y "_" se quedan igual
 - ▢ El carácter espacio " " es convertido al signo "+"
 - ▢ Todos los otros caracteres son codificados en uno o más bytes. Después cada byte es representado por la cadena de 3 caracteres "%xy", donde xy es la representación en hexadecimal del byte

- Insertando información en la URL de los enlaces

Parámetros:

Nombre: direccion

Valor: C\ Pepe, nº 3

Nombre: poblacion

Valor: Madrid

URL con parámetros:

<http://www.micasa.com/nueva?direccion=C%5C+Pepe%2C+n%BA+3&poblacion=Madrid>

- Insertando información en la URL de los enlaces
 - Para acceder a la información se usa el mismo mecanismo que para leer los campos del formulario

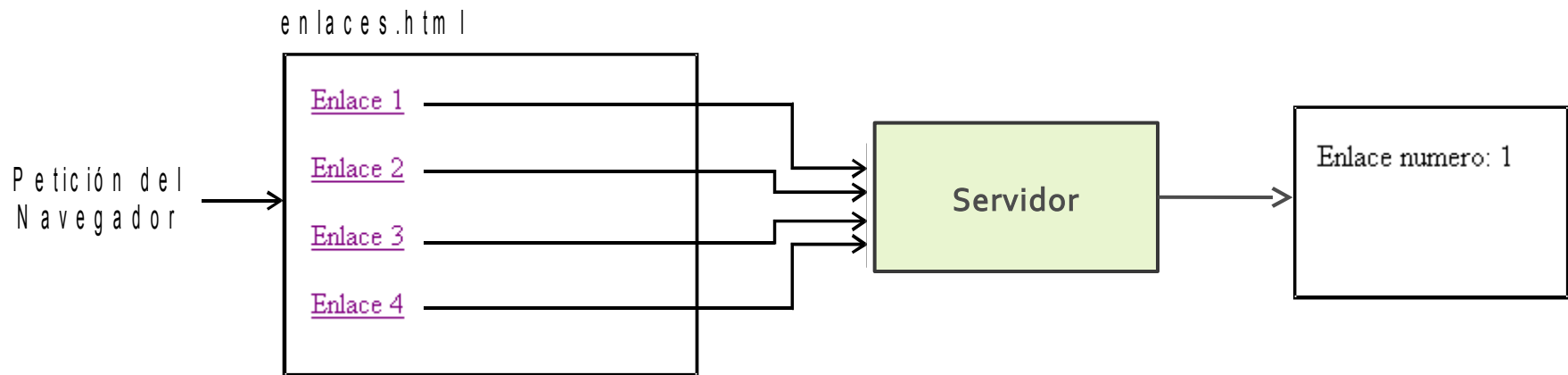
<http://www.micasa.com/ofertas?poblacion=Madrid>

```
@RequestMapping("/ofertas")  
public String ofertas(  
    Model model, @RequestParam String poblacion) {  
    ...  
}
```

Ejercicio 2

- Crear una página **html** con cuatro enlaces
- Todos los enlaces hacen referencia a un mismo controlador
- En la URL de cada enlace incluimos un parámetro llamado “**nenlace**” con valores 1,2,3 y 4
- Implementar un **controlador** que sea llamado al pulsar cualquiera de los enlaces
- Diseñar una **plantilla** que muestre el número del enlace que ha sido pulsado

Ejercicio 2



- Insertando información en la URL de los enlaces
 - La información también se pueden incluir como parte de la propia URL, en vez de cómo parámetros
 - La anotación es `@PathVariable`

<http://www.micasa.com/ofertas/Madrid/>

```
@GetMapping("/ofertas/{poblacion}")
public String ofertas(
    Model model, @PathVariable String poblacion) {
    ...
}
```


Ejercicio 3

- Implementa la misma funcionalidad que el ejercicio 2 pero incluye la información en la propia URL en vez de cómo parámetros

Ejercicio 4

- Crear una aplicación web para gestionar un tablón de anuncios con varias páginas
- La página principal muestra los anuncios existentes (sólo nombre y asunto) y un enlace para insertar un nuevo anuncio
- Si pulsamos en la cabecera de un anuncio se navegará a una página nueva que muestre el contenido completo del anuncio

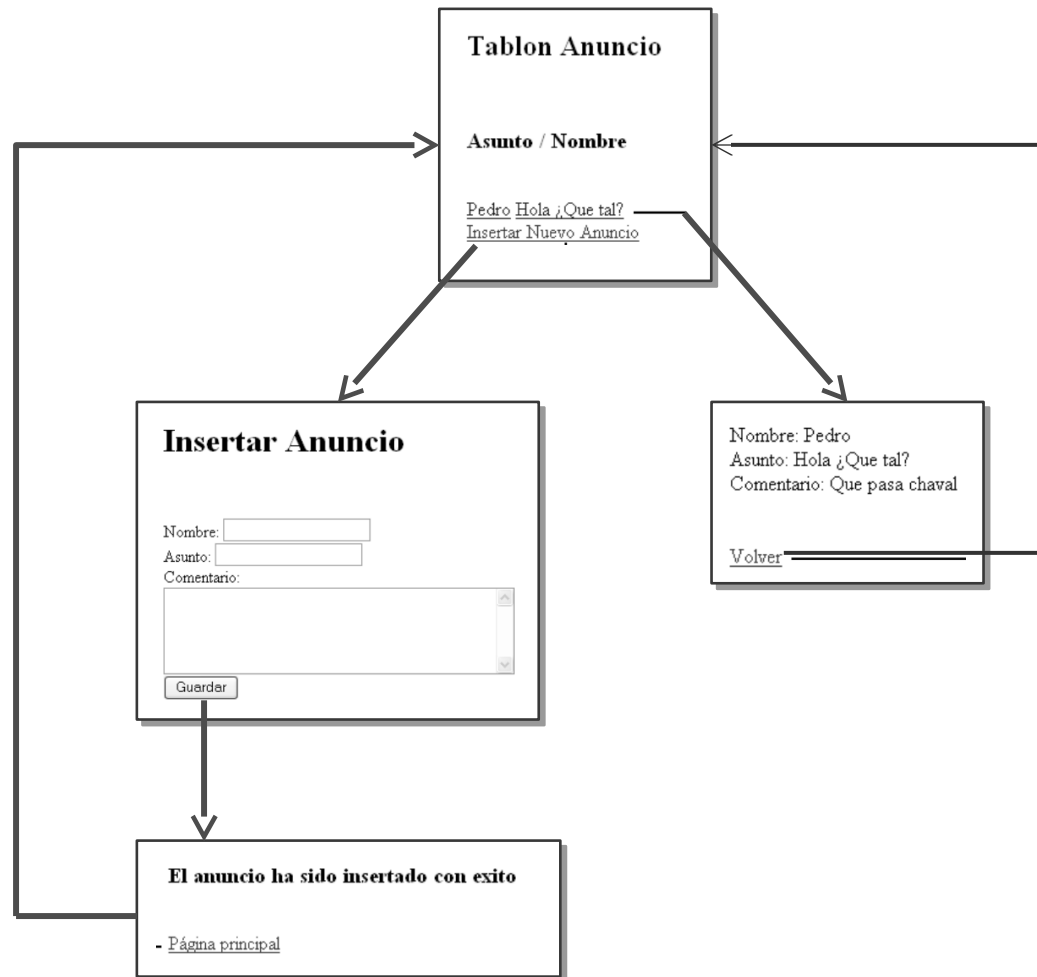
Ejercicio 4

- Si se pulsa el enlace para añadir el anuncio se navegará a una nueva página que contenga un formulario
- Al enviar el formulario se guardará el nuevo anuncio y se mostrará una página indicando que se ha insertado correctamente y un enlace para volver

- **Implementación**

- Se recomienda usar un único controlador con varios métodos (cada uno atendiendo una URL diferente)
- El controlador tendrá como atributo una lista de objetos Anuncio
- Ese atributo será usado desde los diferentes métodos

Ejercicio 4



Desarrollo Web con Spring

- Java Enterprise Edition
- Spring
- Maven
- Spring MVC
- Inyección de dependencias
- Generación de HTML con Mustache
- Proceso de formularios y enlaces
- **Sesión: Datos de cada usuario**

- Es habitual que las aplicaciones web gestionen **información diferente para cada usuario** que está navegando:
 - Amigos en Facebook
 - Lista de correos en Gmail
 - Carrito de la compra en Amazon

- Aunque un usuario no se identifique en la página (no haya hecho *login*) es posible gestionar información específica para él que los demás usuarios no podrán consultar
- En la mayoría de las ocasiones, si el usuario se **identifica en la página**, puede disfrutar de más funcionalidades:
 - Carga de sus datos que están en el servidor
 - Guardado de información: Carrito de la compra, lista de deseos, mensajes leídos...

- 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**)

- **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 (la sesión **caduca**).
 - El tiempo de caducidad es configurable (los bancos suelen tener un tiempo muy pequeño por seguridad)
 - La información de sesión (generalmente) 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 (nombre y clave)
 - 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 la sesión en Spring**
 - Existen dos técnicas principales
 - Objeto HttpSession
 - Una instancia de un componente específica para cada usuario
 - Se pueden combinar estas dos técnicas en una misma aplicación

- **Gestión de la sesión en Spring**
 - 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

<http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html>

- **Gestión de la sesión en Spring**
 - 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 en toda la aplicación, *singleton*)
 - ▮ Es de más alto nivel

SESIÓN: DATOS DE CADA USUARIO

Objeto HttpSession

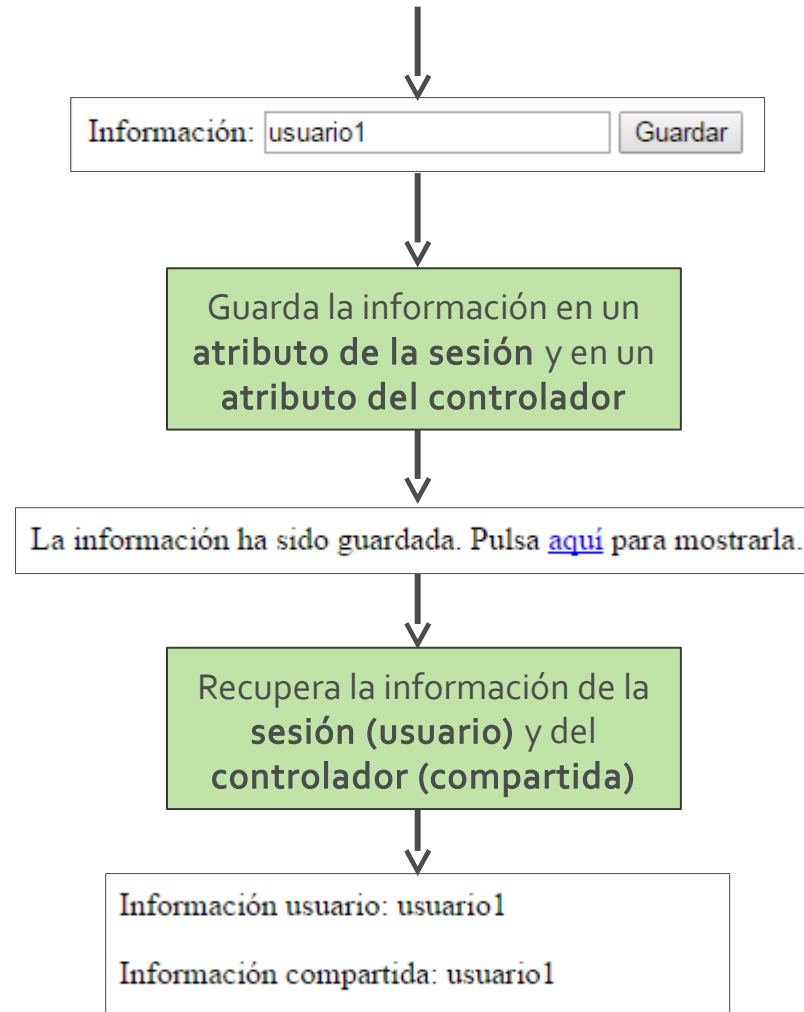
- La sesión se representa como un objeto del interfaz `javax.servlet.http.HttpSession`
- El framework Spring es el encargado de crear un **objeto de la sesión** diferente para cada **usuario**
- Para acceder al **objeto de la sesión** del usuario que está haciendo una petición, basta incluirlo como parámetro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

```
@RequestMapping("/ruta_controlador")
public String procesarFormulario(HttpSession sesion, ...) {
    Object info = ...;
    sesion.setAttribute("info", info);
    return "template";
}
```

SESIÓN: DATOS DE CADA USUARIO

Objeto HttpSession

- Ejemplo



- **Ejemplo**

- Una aplicación recoge la información de un formulario y la guarda en dos lugares:
 - **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
- Para simular dos usuarios en el mismo ordenador, se puede usar el **modo normal** y el **modo incógnito** de Google Chrome.

SESIÓN: DATOS DE CADA USUARIO

Objeto HttpSession

ejem6

```
@Controller
public class SesionController {

    private String infoCompartida;

    @PostMapping(value = "/procesarFormulario")
    public String procesarFormulario(@RequestParam String info, HttpSession session) {

        session.setAttribute("infoUsuario", info);
        infoCompartida = info;


        return "resultado_formulario";
    }

    @GetMapping("/mostrarDatos")
    public String mostrarDatos(Model model, HttpSession session) {

        String infoUsuario = (String) session.getAttribute("infoUsuario");

        model.addAttribute("infoUsuario", infoUsuario);
        model.addAttribute("infoCompartida", infoCompartida);

        return "datos";
    }
}
```



A green box with a black border contains the text "HttpSession como parámetro". Two green arrows point from this box to the "HttpSession session" parameter in the method signatures of the "procesarFormulario" and "mostrarDatos" methods in the code above.

- **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. Si la sesión es 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 segundos):** Configura el tiempo de inactividad para cerrar automáticamente la sesión del usuario.

- En Spring existe una forma de más **alto nivel** para asociar información al **usuario**
- Consiste en crear un **@Component** especial que se asociará a cada usuario y hacer **@Autowired** del mismo en el controlador que se utilice
- Internamente Spring hace bastante **magia** para que la información se gestione de forma adecuada

- Componente para el usuario

```
@Component
@SessionScope
public class Usuario {

    private String info;

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

    public String getInfo() {
        return info;
    }

}
```

La anotación
@SessionScope
hace que haya una
instancia del
componente por
cada usuario

SESIÓN: DATOS DE CADA USUARIO

Componente específico para cada usuario

ejem7

```
@Controller
public class SesionController {

    @Autowired
    private Usuario usuario;

    private String infoCompartida;

    @PostMapping(value = "/procesarFormulario")
    public String procesarFormulario(@RequestParam String info) {

        usuario.setInfo(info);
        infoCompartida = info;

        return "resultado_formulario";
    }

    @GetMapping("/mostrarDatos")
    public String mostrarDatos(Model model) {

        String infoUsuario = usuario.getInfo();

        model.addAttribute("infoUsuario", infoUsuario);
        model.addAttribute("infoCompartida", infoCompartida);

        return "datos";
    }
}
```

Se accede al objeto usuario
con **@Autowired**
(inyección de dependencias)

Se utilizan métodos
del objeto

- **Gestión de la sesión en Spring**
 - Ambas técnicas se pueden combinar
 - El objeto **HttpSession** se utilizará para controlar el ciclo de vida de la sesión (si es nueva, invalidarla, etc...)
 - El **componente** se usará para gestionar la información asociada al usuario

Ejercicio 5

- Modificar el Tablón de anuncios (ejercicio 4) para que la primera vez en la sesión que un usuario cargue la página principal le salga un mensaje de Bienvenida
- En las siguientes visitas a la página principal no tiene que aparecer el mensaje

Ejercicio 5

- Cuando el usuario cree un anuncio por primera vez en la sesión, introducirá su nombre.
- Cuando vaya a crear más anuncios durante la sesión, el nombre le debe aparecer ya escrito (aunque con la posibilidad de modificarlo)
- Además, cada vez que vaya a incluir un anuncio se le debe indicar cuántos anuncios que lleva creados en la sesión