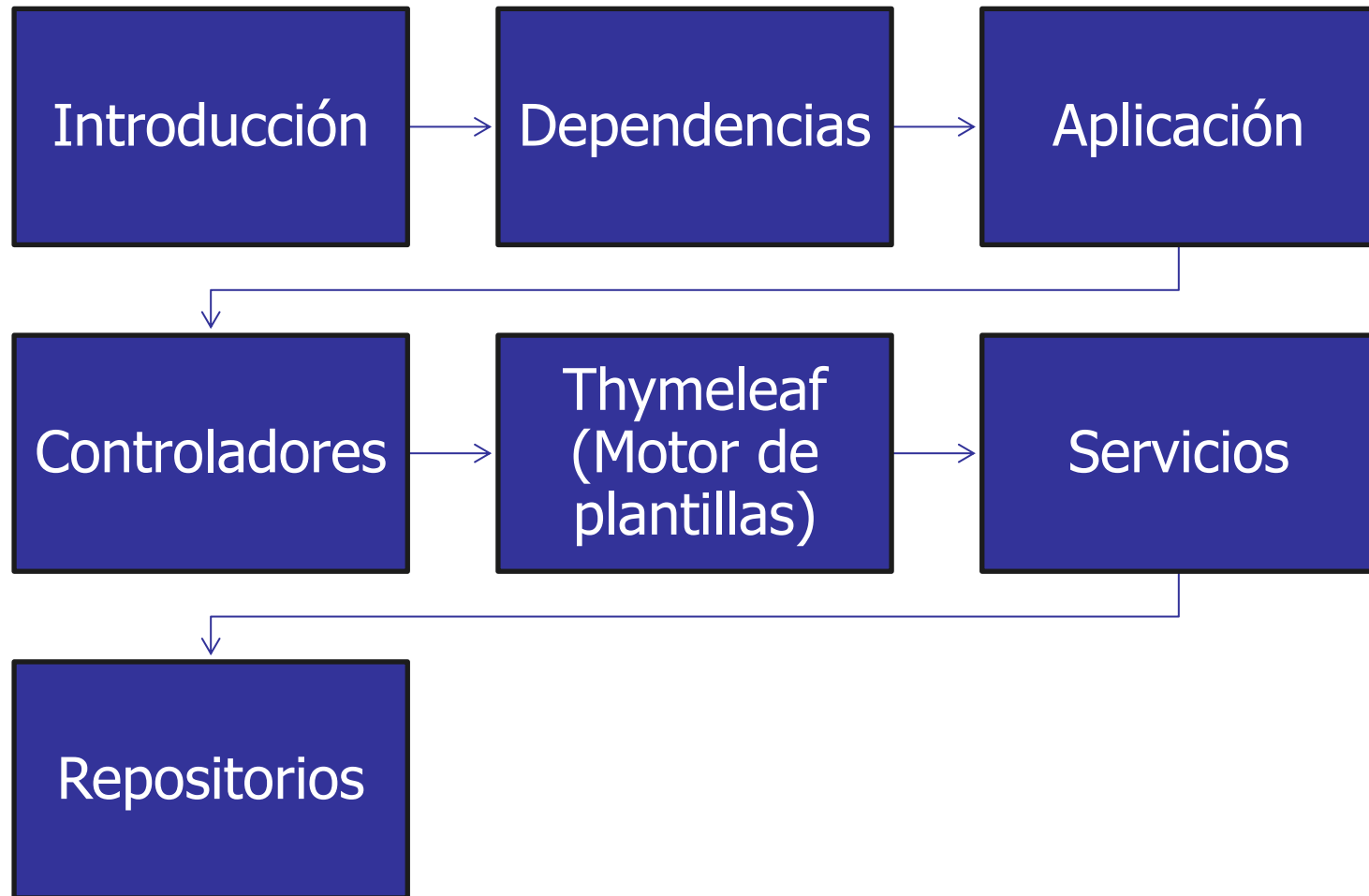




Sistemas Distribuidos e Internet

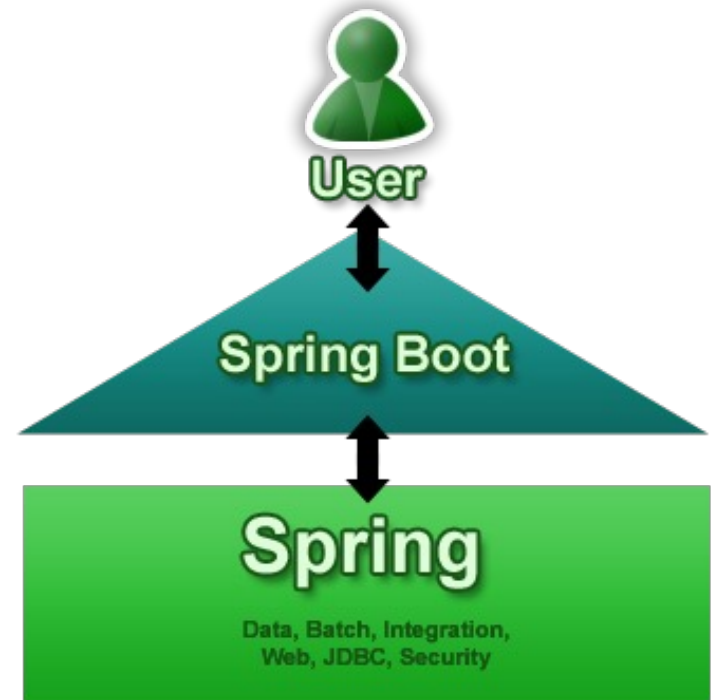
Tema 2
Spring Boot 1

Contenidos



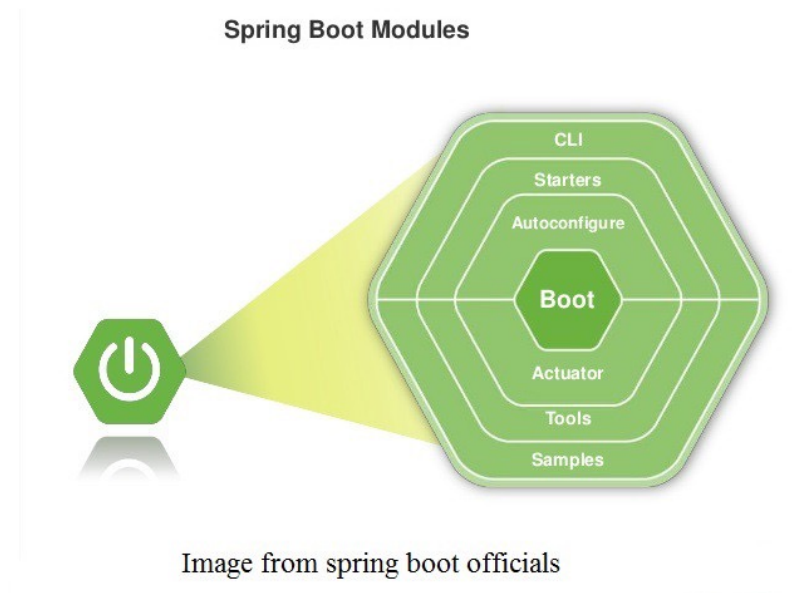
Qué es Spring y que es Spring Boot

- Spring es un framework basado en JEE que usa el patrón MVC para el desarrollo de aplicaciones web cuya característica principal es el uso de un modelo POJO (clases planas sin herencia de otras)
- Spring Boot no es un framework es una forma fácil de desarrollar aplicaciones Spring pero facilitando los aspectos duros de Spring (configuración, generación de código, servidor embebido, ...).



Introducción > Spring Boot

- **Spring Boot** aumenta la agilidad del desarrollo de aplicaciones en **Spring**
 - Provee opciones de **configuración por defecto** para evitar las excesivas configuraciones de Spring
 - Uso opcional de **POMs** para simplificar configuraciones Maven
 - **Evita la generación de código** y las **configuraciones XML** presentes en Spring
 - Permite crear aplicaciones **stand-alone** (servidor embebido)
 - También el despliegue en servidores de aplicaciones



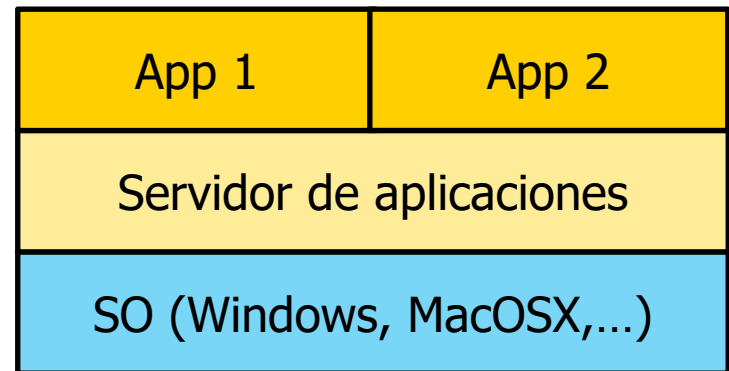
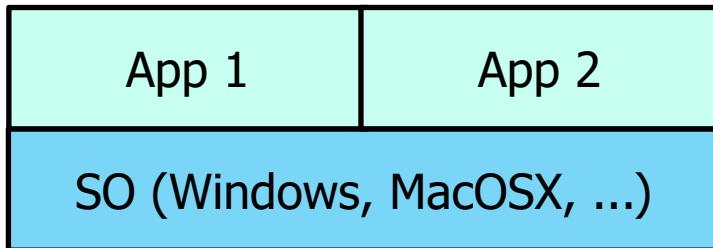
Introducción > Conceptos

- **Framework Spring**, algunas características:
 - Aplicaciones basadas en el patrón MVC
 - Soporte completo al desarrollo de aplicaciones de empresariales basadas en POJOs
 - POJO: objeto plano Java, no extiende ni implementa clases específicas del framework
 - Sistema de inyección de dependencias basado en el IoC container (Inversion of Control)
 - Menor consumo de recursos que los EJB
 - Gran cantidad de módulos con funcionalidad reutilizable
 - Traducción de excepciones específicas a genéricas (Ej JDBC, Hibernate, etc.)

Introducción > Conceptos

■ **Standalone** web applications

- Se ejecutan como una aplicación estándar sobre el propio sistema operativo
- Las aplicaciones web “tradicionales” se ejecutan en un servidor de aplicaciones



Introducción > Entorno de desarrollo

- Pueden desarrollarse aplicaciones en cualquier entorno **Java** con **Apache Maven**
- El entorno oficial es el **Spring Tool Suite**
 - Versión modificada de eclipse
 - <https://spring.io/tools/sts>
- También se puede emplear **JetBrains IntelliJ**



Introducción > Maven

- Maven <http://maven.apache.org>
 - Permite especificar procesos para muchas acciones relativas al desarrollo de software
 - Validaciones, compilación, despliegue, pruebas, etc.
 - Gestión de dependencias / artefactos ("superlibrerías")
- Spring Boot utiliza Maven con ficheros **POM**
 - Project Object Model, representación de Maven en XML
 - <https://maven.apache.org/pom.html>

Introducción > Maven

■ Ejemplo de fichero **POM** (Maven) – Parte 1

```
<?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>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

Propiedades de la aplicación

```
  <!-- Inherit defaults from Spring Boot -->
```

```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.3.RELEASE</version>
  </parent>
```

Configuración
por defecto
Spring Boot

Introducción > Maven

■ Ejemplo de fichero **POM** (Maven) – Parte 2

```
<!-- Add typical dependencies for a web application -->
```

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

Dependencias
dependencia:
starter-web

```
<!-- Package as an executable jar -->
```

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

Instrucciones
para construir
el jar (build)

```
</project>
```

Dependencias > Gestión de dependencias

- Las dependencias se especifican en el fichero **POM** (Maven)
- La **dependencia principal** de la aplicación Spring Boot es: **spring-boot-starter-web**

```
<!-- Add typical dependencies for a web application -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Dependencias > Gestión de dependencias

- La mayor parte de librerías externas/dependencias se gestionan utilizando el fichero **POM**
- Una aplicación compleja suele **tener muchas dependencias**
- Dependencias comunes:
 - spring-boot-starter-data-jpa -> JPA
 - spring-boot-starter-thymeleaf -> Motor de plantillas Thymeleaf
 - spring-boot-starter-security -> Spring Security
 - spring-boot-starter-social-Facebook -> login con facebook
 - Etc.

Dependencias > Gestión de dependencias

- Una vez se incluye una dependencia:
 - En código se descarga e indexa al proyecto
 - La actualización del POM es automática en eclipse
 - En otros entornos puede requerir hacer un build del proyecto o ejecutar un comando
 - IntelliJ: File>Settings>Build,Execution,Deployment>Maven>
- Para **eliminar** dependencias se **elimina la etiqueta XML** correspondiente

Aplicación > Main

- Las aplicaciones **stand-alone** deben definir una clase de inicio
- La clase de inicio implementa un método **main** que ejecuta la aplicación
 - Invoca a **SpringApplication.run()**
- **@SpringBootApplication** es la anotación que se recomienda incluir en la clase principal

Aplicación > Main

- Ejemplo clase principal para **NotaneitorApplication**

```
@SpringBootApplication
public class NotaneitorApplication {

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

Aplicación > SprintBootApplication

- **@SpringBootApplication** fusiona tres anotaciones:
 - **@Configuration** indica que esta clase puede definir elementos de configuración
 - **@EnableAutoConfiguration** habilitar auto-configuración
 - No requiere especificar configuraciones
 - Realiza acciones de configuración por defecto
 - Crea instancias de una serie de objetos en forma de Beans, (Serán utilizados automática o manualmente en diferentes partes de la aplicación)
 - **@ComponentScan** se debe escanear la aplicación en busca de componentes implementados (controladores, servicios, repositorios, etc.)
 - Al detectar esos componentes los registra como **Beans**

Aplicación > EnableAutoConfiguration

- **@EnableAutoConfiguration** configura de forma automática muchas funciones básicas y avanzadas de la aplicación (incluso funciones relativas a dependencias)
 - WebMvcAutoConfiguration
 - JpaRepositoriesAutoConfiguration
 - DataSourceAutoConfiguration
 - DataSourceTransactionManagerAutoConfiguration
 - MongoAutoConfiguration
 - HibernateJpaAutoConfiguration
 - SecurityAutoConfiguration
 - EmbeddedServletContainerAutoConfiguration
 - ServerPropertiesAutoConfiguration
 - Etc

Aplicación > Autoconfiguración

- Parte del fichero **spring-boot-autoconfigure-jar** utilizado por **@EnableAutoConfiguration**

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongodb.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\
```

Aplicación > Autoconfiguración

- Cualquier autoconfiguración puede ser **redefinida** si nos interesa personalizarla
- El atributo **exclude** permite **excluir partes** de la autoconfiguración y que dicha parte no sea aplicada.

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

Aplicación > Propiedades

- El fichero **application.properties** permite modificar las propiedades por defecto / definir nuevas
- Algunas propiedades comunes

- Nombre de la aplicación

```
app.name = Mi aplicación
```

- Puerto del servidor

```
server.port = 8090
```

- Configuración de conexión al datasource

```
spring.datasource.url=jdbc:hsqldb:hsq1://localhost:9001  
spring.datasource.username=SA  
spring.datasource.password=  
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

Aplicación > Propiedades

- Aunque es menos común las propiedades de configuración se pueden definir desde:
 - Las **clases**, al iniciar la aplicación
 - Con anotaciones específicas
 - Fichero de propiedades **application.yml**

```
server:  
  port: 8081
```

- Parámetros en la **línea de comandos** al ejecutar la aplicación

```
java -jar Miaplicacion --server.port=8081
```

Aplicación > Elementos principales

■ Componentes

- ***Componente:** componente genérico sin un propósito específico
- **Controladores:** definen las **peticiones** que van a ser recibidas por la aplicación
Suelen utilizar los **servicios** y retornan **respuestas**
- **Servicios:** definen las funcionalidades disponibles en la capa de lógica de negocio.
Suelen utilizar **repositorios**
- **Repositorios:** definen el acceso al sistema de gestión de bases de datos.
- **Configuración:** definen funcionalidades de configuración transversal, no relativa a la lógica de negocio
 - Encriptación, seguridad, internacionalización, etc.
- **Entidades:** clases que representan las entidades con las que trabaja la aplicación.
- **Vistas:** documentos que pueden ser utilizados para componer **respuestas** de forma más sencilla o eficiente

Aplicación > Elementos principales

- **Controladores, Servicios, Repositorios y Configuración** son estereotipos de **Componentes**
 - Todos son **componentes**
 - Algunos añaden **funcionalidad adicional** al componente
 - Ej: un controlador es un componente que incluye funcionalidad de enrutado para responder peticiones
 - Cada uno se usa con **propósitos muy diferentes** y definidos
 - Pueden ser procesados por herramientas específicas
 - Ej: existen herramientas que analizan los repositorios

Aplicación > Elementos principales

- Todos los **componentes** se procesan internamente como **Beans**
 - Los beans registrados contienen una instancia de un objeto
 - Pudiendo ser inyectados en diferentes partes de la aplicación
- Los **componentes** son instanciados y registrados como **Beans**
 - Automáticamente al iniciar la aplicación (IoC - Inversion of Control)
 - No creamos manualmente las instancias de los componentes
 - Indicado por la anotación **@SpringBootApplication** (que contenía la funcionalidad de **@ComponentScan**)

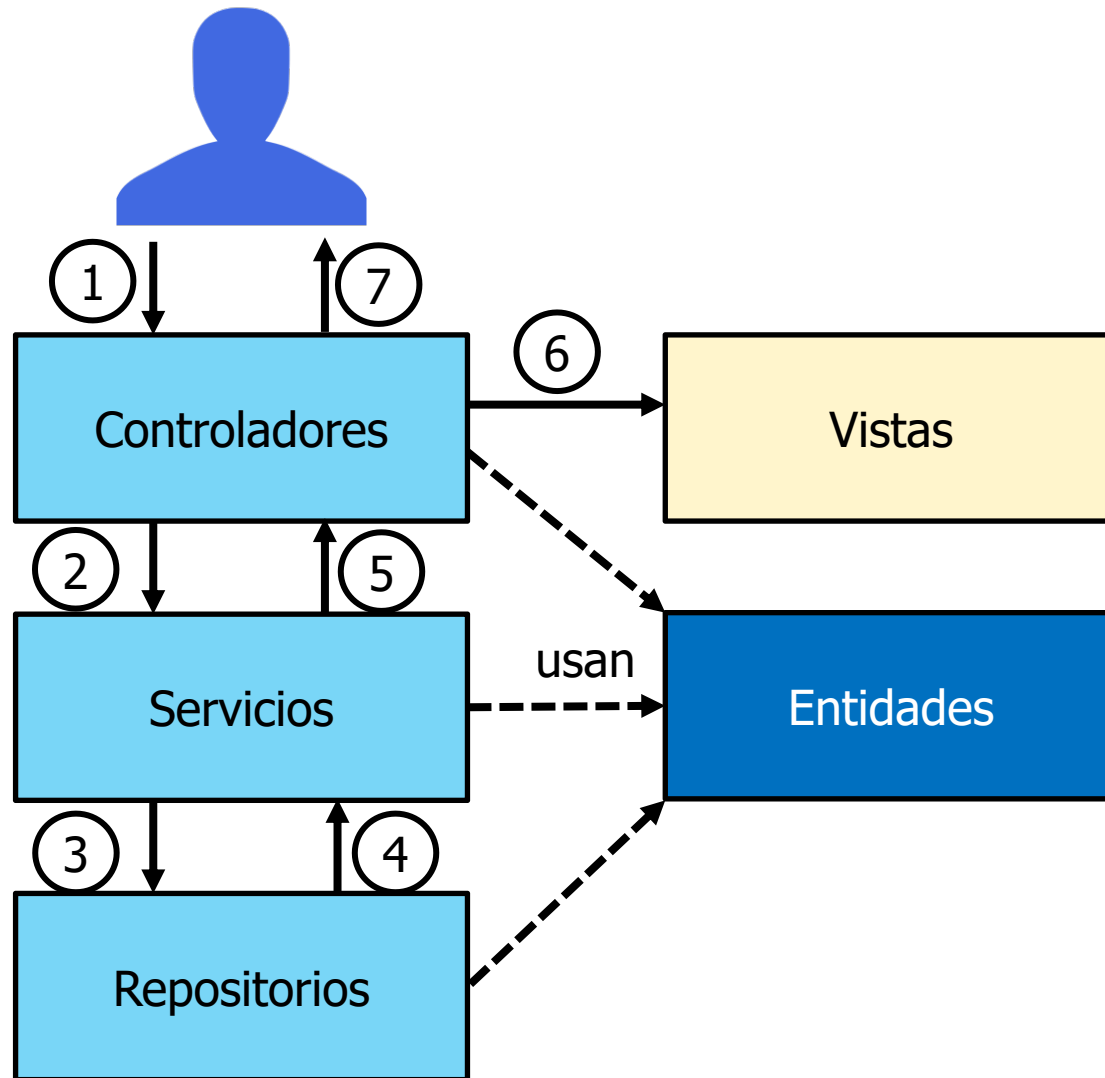
Aplicación > Elementos principales

- Existe la declaración explícita de beans **@Bean**
 - Principalmente para Beans de **configuración** o **funcionalidad transversal**
 - Se crea manualmente la instancia de una clase y se registra como **Bean**
 - En gran parte de los casos se usan **clases del framework**
 - El framework define gran cantidad de funcionalidad transversal
 - Ej: Un **Bean** creado a partir de la clase **BCryptPasswordEncoder** define funciones de encriptación
 - Los **Beans** registrados pueden ser inyectados en el código de la aplicación

Elementos scaneables

- @Bean → No Scaneable
 - @Component --> Si lo son.
 - @Controller
 - @Service
 - @Repository
 - @Configuration

Aplicación > Ejemplo de arquitectura



Aplicación > Ejemplo de arquitectura

- Una aplicación base para gestionar una entidad **coche** definiría los siguientes elementos
 - *Aunque podría ser mucho más compleja

```
com
+- Aplicación
  +- Aplicación.java
  |
  +- Entidades
    +- Coche.java
    |
    +- Repositorios
      +- CochesRepository.java
      |
      +- Servicios
        +- CochesService.java
        |
        +- Controladores
          +- CochesController.java
```

Controladores > Introducción

- Los controladores son **componentes** que **procesan peticiones** realizadas por los clientes
- Pueden invocar a la lógica de negocio y generar una respuesta

com

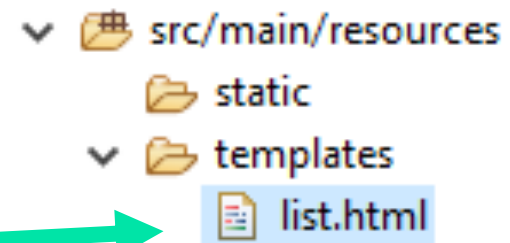
```
+-- Aplicación
    +- Aplicación.java
    |
    +- Entidades
    |   +- Coche.java
    |
    +- Repositorios
    |   +- CochesRepository.java
    |
    +- Servicios
    |   +- CochesService.java
    |
    +- Controladores
    |   +- CochesController.java
```

Controladores > Funcionamiento básico 1

- **@Controller** indica que la clase es un **componente** de tipo controlador
 - Recibe peticiones, suele incluir **@RequestMapping** en sus metodos
- **@RequestMapping** indica que un método responderá a peticiones
 - Su parámetro principal es la URL de la petición
- El retorno es el **nombre de una vista**
 - *Las vistas suelen estar definidas en la carpeta **/templates**


```
@Controller
public class CocheController {

    @RequestMapping("/coche/list")
    public String getList() {
        return "list";
    }
}
```



Controladores > Funcionamiento básico 2

- **@Controller** es un **@Component**
 - Lo más recomendado es usar la anotación **@Controller**



```
@Controller
public class CocheController {

    @RequestMapping("/coche/list")
    public String getList(){
        return "list";
    }
}

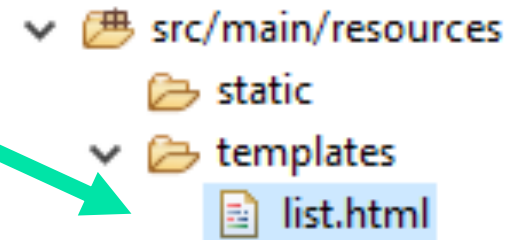
@Component
@RequestMapping
public class CocheController {

    @RequestMapping("/coche/list")
    public String getList(){
        return "list";
    }
}
```

Controladores > Funcionamiento básico 3

- Suelen invocar lógica de negocio, definida en los **servicios**
- Las **vistas** pueden recibir un **Modelo** de datos con **atributos**
 - Ej: modelo con atributo con clave `listaCoches`


```
@RequestMapping("/coche/list")  
public String getList(Model model){  
    List<Coche> coches = cochesService.getCoches();  
    model.addAttribute("listaCoches", coches);  
    return "list";  
}
```



Controladores > Funcionamiento básico 4

- Ejemplo de vista con **motor de plantillas thymeleaf**, combinan:
 - Código web estándar (HTML, Css, JavaScript)
 - Lenguaje propio que les permite:
 - Manejar atributos del modelo recibido como parámetro
 - Realizar procesamiento: if , else, for, etc.

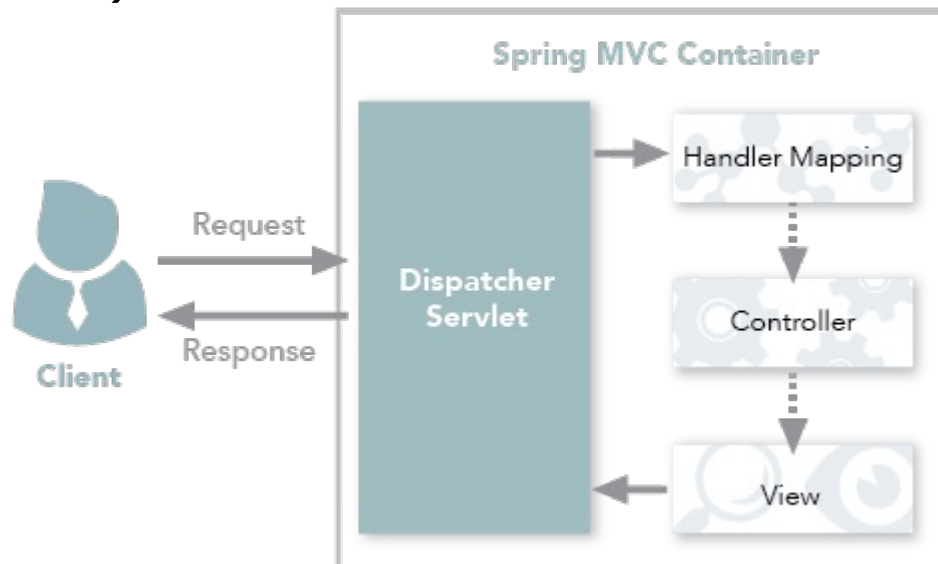
```
model.addAttribute("listaCoches", coches);
```



```
<table class="table table-hover">
  <thead>
    <tr>
      <th>Modelo</th>
      <th>Matricula</th>
      <th>Año</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="coche : ${listaCoches}">
      <td th:text="${coche.modelo}">Modelo</td>
      <td th:text="${coche.matricula}">Matricula</td>
      <td th:text="${coche.año}">Año</td>
    </tr>
  </tbody>
</table>
```

Controladores > Procesamiento en Spring

- Los **@Controller** se escanean y registran en el **RequestMappingHandlerMapping** de Spring
- Cuando el **DispatcherServlet** de Spring recibe una petición, busca la URL en el **Handler Mapping**
- Sí hay coincidencia con la petición HTTP, la delega en el **controlador**
- Los controladores “estándar” referencian una vista **HTML** como retorno (**View**)



Controladores > Mapeo de peticiones

- **@RequestMapping** indica que se debe responder a una petición especificada
 - Parámetro principal de la anotación: URL

```
@RequestMapping("/coche/list")  
public String getList(){  
    return "list";  
}
```

Controladores > Mapeo de peticiones

- **@RequestMapping** admite más **atributos**:
 - **value** especifica la URL
 - **method** especifica el tipo de petición HTTP
 - `method=RequestMethod.POST`, `RequestMethod.POST`
 - `* RequestMethod.GET` es el valor por defecto

```
@RequestMapping(value="/coche/agregar", method=RequestMethod.POST)  
public String setCoche(){...}
```

```
@RequestMapping(value="/coche", method=RequestMethod.GET )  
public String getCoche(){...}
```

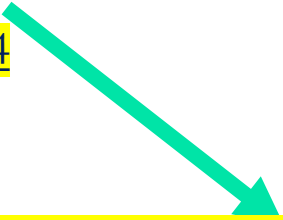
Controladores > Petición y parámetros

- Las **peticiones** pueden contener **parámetros**
- Para obtenerlos:
 1. Incluimos el parámetro en la función
 2. Colocamos **@RequestParam** delante del parámetro

<http://localhost:8090/coche/detalles?id=4>

<http://localhost:8090/coche/detalles?año=2000&id=4>

```
@RequestMapping("/coche/detalles" )  
public String getDetalles(@RequestParam Long id) {  
    String frase = " Detalles del coche : "+id;  
}
```



Controladores > Petición y parámetros

- **@RequestParam** es valido para parámetros:
 - GET (URL)
 - POST (Cuerpo de la petición)

Modelo:

Matrícula:



Request URL: **/coche/agregar**

Request Method: POST

Body: modelo=audi&matricula=34

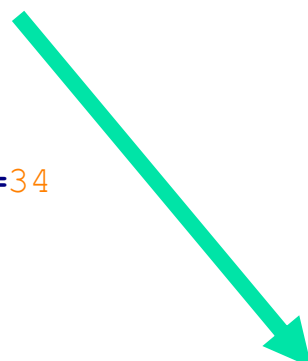
```
<form method="post" action="/coche/agregar">
  Modelo:<br>
  <input type="text" name="modelo"></br>
  Matrícula:<br>
  <input type="text" name="matricula" ></br>

  <input type="submit" value="Send">
</form>
```

Controladores > Petición y parámetros

- **@RequestParam** debe especificar el **tipo de dato**
- Por defecto los parámetros son **obligatorios**

Request URL: /coche/agregar
Request Method: POST
Body: modelo=audi&matricula=34



```
@RequestMapping(value="/coche/agregar", method=RequestMethod.POST)
public String setCoche(@RequestParam String modelo, @RequestParam String matricula){
    ...}
```

Controladores > Petición y parámetros

- La petición podría **no contener el parámetro** tal y como esperamos
 - No incluir parámetros obligatorios produce una **excepción**

GET http://localhost:8080/saludar

¿nombre?

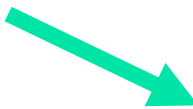
```
@RequestMapping("/saludar")  
public String saludar(@RequestParam String nombre){
```

There was an unexpected error (type=Bad Request, status=400).
Required String **parameter 'nombre' is not present**

Controladores > Petición y parámetros

- Si los tipos de los datos no encajan se produce una **Excepción**

<http://localhost:8080/sumar?a=5&b=hola>



```
@RequestMapping("/sumar")  
public String sumar(@RequestParam long a,@RequestParam long b){  
    long result = a + b;  
}
```

There was an unexpected error (type=Bad Request, status=400).
Failed to convert value of type 'java.lang.String' to required type 'long'; nested exception is java.lang.NumberFormatException: For input string: "hola"

Controladores > Petición y parámetros

- Los parámetros pueden ser **opcionales**
 - `required = false;`
 - *En este caso podrían tomar valor **null**
- Pueden tener **valores por defecto**
 - `value = valor`

```
@RequestMapping("/saludar")  
public void saludar(@RequestParam(value = "sdi", required=false) String  
nombre) {
```

Controladores > Petición y parámetros

■ Variables en URL

- Son parámetros **GET** sin clave en la URL
- La clave se determina por su **posición** en la URL

■ Ejemplos

- Variable en URL:

<http://localhost:8090/coche/4/>

- Parámetro get:

<http://localhost:8090/coche?id=4>

- Variables en URL:

<http://localhost:8090/coche/audi/4/>

- Parámetros get:

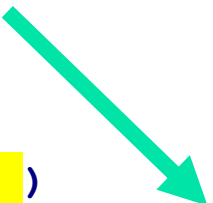
<http://localhost:8090/coche?modelo=audi&id=4>

Controladores > Petición y parámetros

■ Implementación

- Agregar un parámetro a la función con la anotación **@PathVariable**
- Incluir la variable en la URL con el patrón { <clave> }

<http://localhost:8090/mark/details/4/>




```
@RequestMapping("/mark/details/{id}")  
public String getDetail(@PathVariable Long id) {  
    return " Getting Detail: "+id;  
}
```

Controladores > Petición y parámetros

- **@ModelAttribute** construye automáticamente un objeto en base a los parámetros recibidos
- La clase utilizada como **@ModelAttribute** debe definir:
 - Constructor sin parámetros
 - Métodos Get para los atributos
- Al recibir la petición, se crea un objeto, después:
 - Se completan los atributos en los que haya coincidencia de nombres
 - Los atributos no contenidos se quedan sin valor

Request URL: /coche/agregar
Request Method: POST
Body: modelo=audi&matricula=34

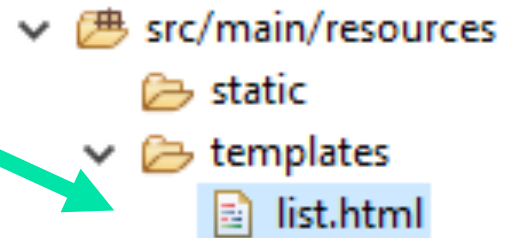


```
@RequestMapping(value="/coche/agregar", method=RequestMethod.POST)
public String setCoche(@ModelAttribute Coche coche) {
    ...
}
```

Controladores > Respuestas

- Los **@Controller** retornan la ruta para localizar una **plantilla**
- El **motor de plantillas** se encarga de generar la vista
 - Debe tener configurado un motor de plantillas (Ejemplo: thymeleaf)

```
@RequestMapping("/coche/list")  
public String getList(Model model){  
    List<Coche> coches = cochesService.getCoches();  
    model.addAttribute("listaCoches", coches);  
    return "list";  
}
```



Controladores > Respuestas

- **@ResponseBody** hace que una respuesta sea **un objeto** en lugar de una **plantilla**
 - Utilizado para implementar Servicios web / pruebas, que retorna: cadenas, XML, JSON, etc.
 - Ejemplo: retorna la cadena "Hola SDI"
 - "Hola SDI" no es la clave de ninguna plantilla
- Petición: `/saludar?nombre="Pepe"`

```
@RequestMapping("/saludar")
@ResponseBody
public void saludar(@RequestParam String nombre) {
    return "Hola "+nombre;
}
```

Controladores > Respuestas

- Un **@RestController** es un **@Controller** específico
 - Añade de forma transparente **@ResponseBody** en todos los métodos
 - Todos retornan **objetos**, no usan plantillas
 - Utilizados comúnmente en servicios web y pruebas

```
@RestController
public class SaludadorController {

    @RequestMapping("/saludar")
    public void saludar(@RequestParam String nombre) {
        return "Hola "+nombre;
    }
}
```


Thymeleaf > Motores de plantillas

- Permiten componer respuestas de forma **dinámica y ágil**
 - Concebido para la definición de vistas
- Existen muchos motores compatibles con Spring
 - Las funcionalidades básicas son comunes a todos
 - Algunos incluyen funcionalidades muy avanzadas
- **Thymeleaf** es uno de los motores más popular en Spring
 - <http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html>



Thymeleaf > Introducción

- Desarrollado por Daniel Fernández
- Licencia Apache 2.0
- Implementado en Java
- Pensado para HTML5/XML/HTML y extensible a otros formatos
- Ofrece módulos adicionales para integrarse con dependencias de **Spring**, como SpringSecurity
- Soporta:
 - Acceso a atributos del modelo (objetos enviados desde el controlador)
 - Acceso a request y otros elementos http
 - Definición de lógica en la plantilla: iteraciones, condiciones, variables, etc.
 - Spring WebFlox (eventos AJAX)
 - Otros.

Thymeleaf > Instalación

- Incluir la dependencia base de **thymeleaf** en **pom.xml**

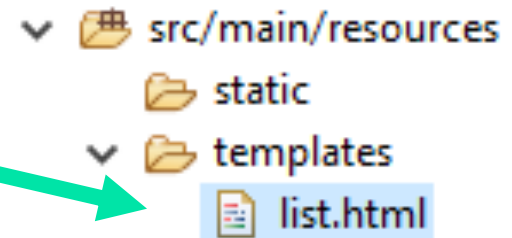
```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>
```

- Por defecto las plantillas se almacenan en la carpeta **/templates**
 - Configurable mediante propiedades
- Las plantillas combinan **lenguajes web y Thymeleaf**
 - La funcionalidad ofrecida es **muy amplia** revisaremos algunas características básicas

Thymeleaf > Modelo y atributos

- Los controladores pueden enviar un **modelo de datos a la plantilla**
 - El **modelo** contiene atributos (Strings, listas, otros objetos, etc.)
 - Los atributos se identifican por claves
 - **model.addAttribute(clave, objeto)** agregar objeto al modelo

```
@RequestMapping(value="/articulos")  
public String getMark(Model model){  
    model.addAttribute("tienda", "Mi Tienda");  
  
    List<Articulo> articulos = articulosService.getArticulos();  
    model.addAttribute("lista", articulos);  
  
    // Plantilla:  
    return "list";  
}
```



Thymeleaf > Modelo y atributos

- La plantilla tiene **acceso a los atributos del modelo**
- Acciones comunes con estos atributos:
 1. Insertar sus valores en el HTML
 2. Usarlos en estructuras de control y utilidades de Thymeleaf (condiciones, bucles, etc.)
 3. Insertar sus valores en JavaScript (funciona de diferente forma a HTML)


Thymeleaf > Modelo y atributos

1 Insertar los valores de los atributos **en el HTML**

- Acceso al atributo **`${<clave_atributo_del_modelo>}`**
- El **\$** manipula **expresiones con variables**
 - Otros operadores `#`, `@`, `*`...
- Incluir en el nodo HTML **`th:<propiedad_HTML> = "expresión"`** crea un nuevo atributo con ese valor
 - Ejemplo colocar un texto **`th:text`** un nodo **`<p>`**

```
model.addAttribute("tienda", "Mi Tienda");
```

```
<!DOCTYPE html>
<html lang="en">
<body>
  <div>
    <p th:text="${tienda}"></p>
  </div>
</body>
</html>
```



Genera:

```
<p>Mi Tienda</p>
```

Thymeleaf > Modelo y atributos

1 Insertar los valores de los atributos **en el HTML**

- Se puede modificar cualquier atributo HTML
 - Etiquetas: **th:text, th:id, th:src, th:href, etc.**
 - Depende de los atributos HTML que queramos componer
 - Sí el atributo ya existía se sustituye por el nuevo

```
<a th:href="${identificador}" th:id="${identificador}">Enlace</a>
```

```
Genera: <a href="3434" id="3434" >Enlace</a>
```

- Admite **uso de literales, operaciones y utilidades**

```
<a th:href="${'/producto/detalles' + identificador}">Enlace</a>
```

```
Genera: <a href="/producto/detalles/3434">Enlace</a>
```

Thymeleaf > Modelo y atributos

1 Insertar los valores de los atributos **en el HTML**

- Si el atributo es un **objeto** se accede a sus datos y métodos con el operador `.`

```
<li th:text="{producto.nombre}"> Ver </li>
```

```
Genera: <li> iPhone </li>
```

```
<li th:text="{producto.getUnidades()}"> Ver </li>
```

```
Genera: <li> 111 </li>
```


Thymeleaf > Modelo y atributos

2 Atributos en estructuras de control (condiciones, bucles, etc.)

- **Bucle `th:each`** . El bloque HTML se repite tantas veces como elementos contenga la colección
 - Para procesar colecciones

Objeto iterado

```
model.addAttribute("listaProductos", productos);
```

```
<li th:each="producto : ${listaProductos}">
  <p th:text="${producto.nombre}"></p>
  <p th:text="${producto.precio}"></p>
</li>
```

Genera:

```
<li>
  <p> iPhone</p>
  <p> 1000 </p>
</li>
<li>
  <p> iPhone</p>
  <p> 1000 </p>
</li>
```

Thymeleaf > Modelo y atributos

2 Atributos en estructuras de control (condiciones, bucles, etc.)

- **Condiciones th:if** . El bloque HTML solo se incluye si se cumple la condición

- Expresiones lógicas (* Soporta varios formatos de expresiones)

```
<p th:if="{producto.nuevo == true}"> Nuevo </p>
<p th:if="{producto.precio <= 5}"> oferta </p>
<li th:if="{page.getNumber () -1 >=0}"> Primera </li>
```

- Expression Utility Objects. Algunos ejemplos comunes
- El **#** manipula **objetos de utilidad y mensajes (internacionalización)**
 - `{#dates.formatISO(miFecha)}` transforma fecha a formato ISO
 - `{#dates.minute(miFecha)}` obtiene los minutos de una fecha
 - `{#numbers.formatInteger(miNumero,3)}` formatea a número de 3 dígitos
 - `{#strings.isEmpty(miCadena)}` comprueba si la cadena es vacía
 - `{#lists.isEmpty(miLista)}` comprueba si la lista es vacía
 - `{#lists.size(miLista)}` obtiene el tamaño de la lista
 - Referencia completa: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html-expression-utility-objects>

```
<p th:if="{#strings.isEmpty(producto.descripcion)}">Sin info</p>
```

Thymeleaf > Modelo y atributos

3 Atributos en JavaScript

- Los atributos del modelo pueden insertarse en JavaScript
- Sintaxis **[[\${param.clave_atributo_del_modelo}]]**
- El Script debe:
 - Incluir **th:inline="javascript"** en la declaración
 - Encapsular el código en **/*<![CDATA[*] código [*]]>*/**
 - Hace que ignore el parseador de XML
- Ejemplo

```
<script th:inline="javascript">
    /*<![CDATA[*]
    var listaClientes = [[${clientes}]];
    $( "#resendButton[[${producto.id}]]" ).click(function() {
        ...
    });
    /*]]>*/
</script>
```

Thymeleaf > Link URL Expressions

- El operador @ ofrece funcionalidad para **gestión de parámetros en URLs**. En ocasiones es útil para insertar propiedades **src y href**

- @{**literales y variables** \$(<clave_parámetros> = valores) }
- Ej, parámetros con clave

```
<a th:href="@{/detalles(id=${producto.id})}">ver</a>
```

Genera href="/detalles?id=322"

- Ej, parámetros embebidos en la URL

```
<a th:href="@{/detalles/{id}/{id=${producto.id}}}">ver</a>
```

Genera href="/detalles/322/"

*También se pueden componer únicamente con variables \$ y literales

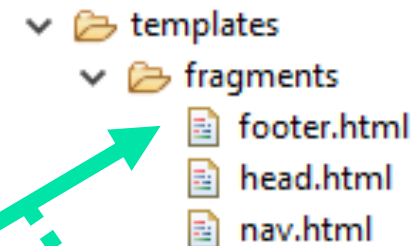
```
<a th:href="${'/detalles?id='+producto.id}">ver</a>
```

Genera href="/detalles?id=322"

Thymeleaf > Incluir y reemplazar

- Varias posibilidades para componer plantillas a partir de otras
- Evitan replicar partes comunes en varias vistas
 - Cabeceras, menús, pies de página, etc.
- Mejoran la arquitectura y el mantenimiento
- Una de las formas más rápidas es usar
 - **th:include="ruta plantilla"** Incluye el contenido de una plantilla dentro del tag HTML
 - **th:replace** sustituye el tag HTML por el contenido de la plantilla

```
<html lang="en">
  <head th:replace="fragments/head"/>
  <body>
    <nav th:replace="fragments/nav"/>
    <p>Prueba</p>
    <footer th:replace="fragments/footer"/>
  </body>
</html>
```



```
<footer class="footer">
  <span>SDI</span>
</footer>
```

Thymeleaf > Otros

- Permite acceder a multitud de **objetos de la aplicación**

- Como por ejemplo:

- **#locale** propiedades de localización

```
<p th:text="${#locale.country}"></p>
```

- **#HttpServletRequest** la petición

```
<p th:text="${#HttpServletRequest.getAttribute('id')}"></p>
```

- **#httpSession** al objeto sesión

```
<p th:text="${#httpSession.getAttribute('total')}"></p>
```

- La gran funcionalidad del core puede ser **extendida con dependencias**

- Mediante dependencias, Ej: **thymeleaf-extras-springsecurity4** da Acceso a objetos de Spring-Security.

Servicios > Introducción

- Los servicios son **componentes** que contienen lógica de negocio
- Suelen ser utilizados desde los **controladores** o desde otros **servicios**

```
com
+- Aplicación
  +- Aplicación.java
  |
  +- Entidades
    +- Coche.java
    |
    +- Repositorios
      +- CochesRepository.java
      |
      +- Servicios
        +- CochesService.java
        |
        +- Controladores
          +- CochesController.java
```

Servicios > Introducción

- Los **servicios** son estereotipos de un **componente**
 - Este estereotipo indica que el componente pertenece a la **capa de servicios / lógica de negocio**
 - Los **componentes** tienen un sistema de autodetección y autoconfiguración basado en **Beans**
 - Todos los componentes son registrados al iniciar la aplicación como **Beans** (luego podrán ser inyectados)
 - Indicado en la anotación **@SpringBootApplication** (que contenía la funcionalidad de **@ComponentScan**)

Servicios > Introducción

- La anotación **@Autowired** se asocia a los atributos
 - Permite **inyectar una dependencia**, sin necesidad de ninguna configuración adicional
 - La inyección es una alternativa a instanciar un objeto
 - Spring instancia los componentes, cuando una clase los necesita los inyecta
- En varias tecnologías la inyección se hace a través del constructor o un método set.
 - En Spring Boot también podríamos hacerlo, pero **@Autowired** es más directo

Servicios > Funcionamiento básico

- Implementan métodos de lógica de negocio
- Suelen acceder a **repositorios** de datos
 - Capa de persistencia de la aplicación

@Service

```
public class CochesService {
```

```
    public List<Coche> getCoches () {
```

```
        List<Coche> coches = cochesRepository.findAll ();
```

```
        return coches;
```

```
    }
```

```
    public void agregarRevision (Long idCoche, String revision) {
```

```
        Coche coche = cochesRepository.findAll (id);
```

```
        coche.agregarRevision (revision);
```

```
        ...
```

```
    }
```

Repositorio



Servicios > Funcionamiento básico

- Son comúnmente **inyectados** e utilizados en **controladores** y otros **servicios**

```
@Controller  
public class CocheController {
```

```
    @Autowired
```

```
    CochesService cochesService;
```

Inyección de servicio



```
    @RequestMapping("/coche/list")
```

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

```
        List<Coche> coches = cochesService.getCoches();
```

```
        model.addAttribute("listaCoches", coches);
```

```
        return "list";
```

```
    }
```

Servicios > Inyección

- La inyección de dependencias es una forma de “inversión de control”.
- ¿Por qué inyectamos objetos en lugar de instanciarlos?
 - Evitar que las clases tengan que saber como instanciar/obtener el objeto
 - Podría reducir el tiempo de desarrollo
 - Código más modular, desacoplado y reusable
 - Simplifica las pruebas de los componentes / test unitarios
 - Posibilidad de inyección de objetos de prueba
 - Muchos frameworks ofrecen **opciones avanzadas**
 - Ej: modificar dependencias en tiempo de ejecución

Servicios > Procesamiento en Spring

- Spring tiene un **contenedor de dependencias (The IoC Container - Inversion of Control)**
- Se encarga de:
 - Escanear el código y localizar los **componentes**
 - Instancia como Beans
 - Cuando una clase solicita el Bean se inyecta
- El sistema de inyección es altamente configurable
 - Por defecto: se usa la misma instancia en todas las inyecciones
 - Otras opciones: instancia por petición (request), por usuario (session).
 - Otras

Servicios > postConstruct

- **@PostConstruct** permite especificar que un método se ejecutará una vez construido el componente
 - Antes de inyectarlo en cualquier clase

```
@PostConstruct
public void init() {
    cochesService.add(new Coche("Audi", "1111"));
}
```

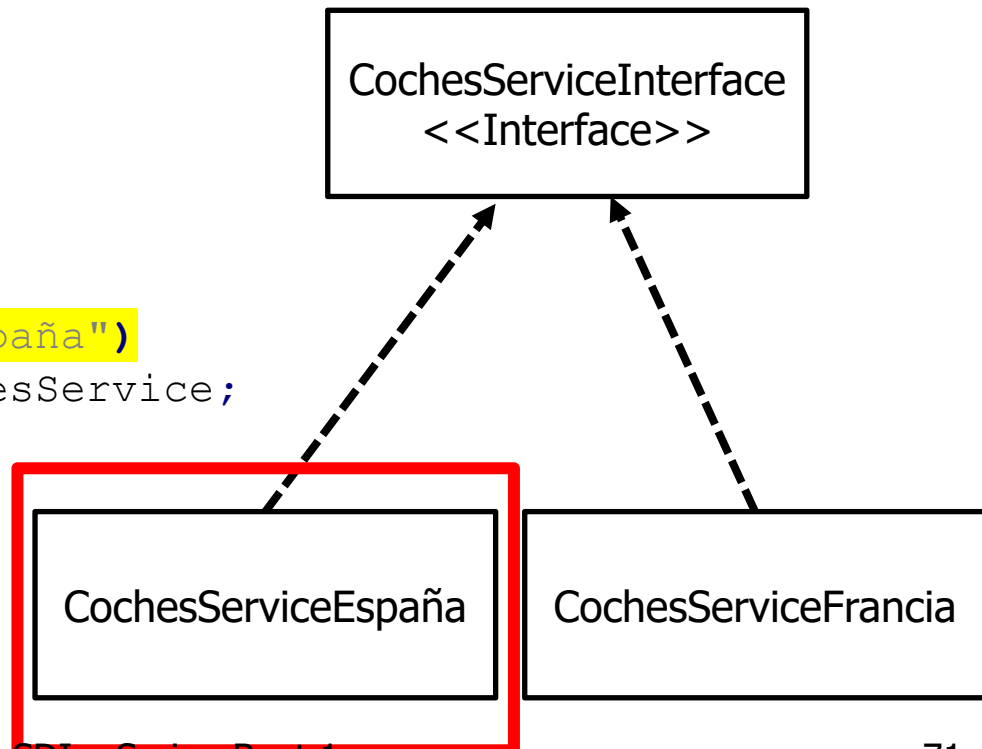
- **@PreDestroy** permite especificar que un método se ejecutará justo antes de destruir el componente

Servicios > Interfaces

- Para inyectar una implementación de una Interfaz se combina **@Autowired** con **@Qualifier("<nombre_de_la_implementación>")**

```
@Controller
public class CocheController {

    @Autowired
    @Qualifier("CochesServiceEspaña")
    CochesServiceInterface cochesService;
```



Servicios > Ámbito

- Por defecto los servicios tienen un ámbito (Scope) **singleton**
 - La misma instancia del servicio se usa en todas las dependencias
- Otros ámbitos comunes definen una **instancia distinta** para:
 - **@RequestScope** cada petición HTTP
 - **@SessionScope** cada sesión HTTP / cliente/navegador.
 - **@Scope("prototype")** cada clase en la que se inyecta el componente

Servicios > Ámbito

- Ejemplo, lógica que almacena las IDs de los productos agregados al carrito de la compra **para cada sesión**

```
@SessionScope
@Service
public class CarritoService {

    List<String> idProductos = new LinkedList<String>();

    public List<String> getIdsProductos() {
        return idProductos;
    }

    public void addIdProducto(String id) {
        idProductos.add(id);
    }

}
```

Repositorios > Introducción

- Los repositorios son **componentes** que acceden a bases de datos
- Suelen ser utilizados desde la capa de **servicios**

```
com
+- Aplicación
  +- Aplicación.java
  |
  +- Entidades
    +- Coche.java
    |
    +- Repositorios
      +- CochesRepository.java
    |
    +- Servicios
      +- CochesService.java
    |
    +- Controladores
      +- CochesController.java
```

Repositorios > Introducción

- La anotación **@Repository** indica que una clase es un componente de tipo repositorio
 - Es un estereotipo, componentes que pertenecen a la capa de acceso a datos
 - Estos componentes tienen habilitado por defecto la **PersistenceExceptionTranslationPostProcessor**
 - En otros frameworks debe especificarse manualmente
 - Esta funcionalidad **traduce** errores generados en cualquier proceso de persistencia (HibernateExceptions, PersistenceExceptions) en objetos **DataAccessException**

Repositorios > Introducción

- El repositorio puede utilizar multitud de **APIs / librerías** para acceder a las bases de datos
 - Ejemplo: **JPA – Java Persistente API** para acceder a una base de datos **hsqldb-2.4.0**
- Spring Boot puede integrarse con **JPA** (dependencia **spring-boot-starter-data-jpa**)
 - Multitud de clases de alto nivel que abstraen JPA
 - Para utilizarlo se debe agregar la dependencia

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>
```

Repositorios > Datasource y Jpa

- Configuración del datasource en **application.properties**
 - **spring.datasource.url** : Dirección del datasource
 - **spring.datasource.username** : nombre del usuario para la conexión
 - **spring.datasource.password** : contraseña del usuario para la conexión
 - **spring.datasource.driver-class-name** : nombre del driver de conexión (debe de estar incluido en el proyecto)
 - ***spring.jpa.hibernate.ddl-auto** : modo de inicialización:.
 - **create** : elimina todos los datos anteriores, **validate** no los elimina

```
server.port = 8090
```

```
spring.datasource.url=jdbc:hsqldb:hsqldb://localhost:9001
```

```
spring.datasource.username=SA
```

```
spring.datasource.password=
```

```
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

```
spring.jpa.hibernate.ddl-auto=create
```

```
#spring.jpa.hibernate.ddl-auto=validate
```

Repositorios > Datasource

- En muchos casos habrá que instalar el driver
 - Se instala mediante una dependencia
 - Algunos drivers comunes ya están incluidos
 - Dependencia para el caso anterior **org.hsqldb / hsqldb**

```
<dependency>  
  <groupId>org.hsqldb</groupId>  
  <artifactId>hsqldb</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

- *En caso de que fuera una base de datos Oracle7 se usaría **com.Oracle / ojdbc7** , etc.

Repositorios > Funcionamiento básico

- Una vez configurado el **datasource** e incluida la dependencia **org.hsqldb / hsqldb** implementamos componentes **@Repository**
- Un enfoque común es:
 - Extender una clase con operaciones CRUD incluidas
 - Ej: **org.springframework.data.repository.CrudRepository**
 - Define los métodos de CRUD para una entidad (Crear, Leer, Actualizar y Borrar)
 - Declarar otros métodos **específicos** (no incluidos en la interfaz)
 - Ej: obtener solo los coches con caballos ≥ 100 .

Repositorios > Funcionamiento básico

- **CrudRepository< Clase_entidad , tipo clave primaria >**

- *Incluye la etiqueta **@Repository**
- Ejemplo:

```
import org.springframework.data.repository CrudRepository;
```

```
public interface CochesRepository extends CrudRepository<Coche, Long>{  
  
}
```

- CochesRepository incluye implementaciones:
 - save(Coche)
 - saveAll(Iterable <Coche>) : Iterable<Coche>
 - exist(Long id) : boolean
 - findAll() : Iterable<Coche>
 - findOne(Long id) : Coche
 - findAll(Iterable<Long>) : Iterable<Coche>

Repositorios > Funcionamiento básico

- CochesRepository incluye implementaciones:
 - count() : long
 - delete(Long id) : void
 - delete(Coche) : void
 - delete(Iterable<Coche>) : void
 - deleteAll() : void

Repositorios > Funcionamiento básico

- Extensión del **CrudRepository**
 - Sin consulta: métodos **estándar por atributos de la entidad**
 - El nombre del método debe coincidir con el atributo
 - No requieren especificar la consulta

```
public interface CochesRepository extends CrudRepository<Coche, Long>{  
  
    Coche findByMatricula(String matricula);  
    Iterable<Coche> findAllByModelo(String modelo);  
}
```

- Con consulta: métodos con consultas específicas **@Query y lenguaje JPQL**

```
public interface CochesRepository extends CrudRepository<Coche, Long>{  
  
    @Query("SELECT c FROM Coche c WHERE c.caballos >= ?1")  
    Iterable<Coche> cochesConCaballos(int caballos);  
}
```

Repositorios > Funcionamiento básico

- Definición de **Entidades**, anotaciones comunes:
 - **@Entity** indica que una clase es una entidad
 - **@Id** el atributo es clave primaria
 - **@GeneratedValue** el atributo se genera automáticamente al salvarlo
 - **@Column(unique=true)** el atributo es único
 - **@Transient** indica que no queremos guardar el atributo en la base de datos
 - Por defecto todos los atributos sin anotación se almacenan
 - *Debemos incluir métodos **get/set** a todos los atributos

Repositorios > Funcionamiento básico

- Definición de **Entidades**, ejemplo Coche:

```
@Entity
public class Coche {
    @Id
    @GeneratedValue
    private Long id;
    private String modelo;
    @Column(unique=true)
    private String matricula;
    private int caballos;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getMatricula() {
        return matricula;
    }
}
```

javax.persistence



. . . Otros Get y Set

Repositorios > Funcionamiento básico

- Se inyecta como un **componente**, anotación **@Autowired**
- Se utilizan desde la capa de **servicios**

```
@Service
public class CochesService {

    @Autowired
    CochesRepository cochesRepository;

    public List<Coche> getCoches() {
        List<Coche> coches = cochesRepository.findAll();
        return coches;
    }

    public void agregarRevision (Long idCoche, String revision){
        Coche coche = cochesRepository.findAll(id);
        coche.agregarRevision(revision);
        ...
    }
}
```

The diagram illustrates the flow of repository injection and usage in the `CochesService` class. A box labeled "Inyección de repositorio" has a green arrow pointing to the `@Autowired CochesRepository cochesRepository;` line. Another box labeled "Uso del repositorio" has a green arrow pointing to the `cochesRepository.findAll()` call in the `getCoches()` method and the `cochesRepository.findAll(id)` call in the `agregarRevision` method.