

Teoria-Primer-Parcial.pdf



Anónimo



Sistemas Distribuidos e Internet



3º Grado en Ingeniería Informática del Software



Escuela de Ingeniería Informática
Universidad de Oviedo

antes

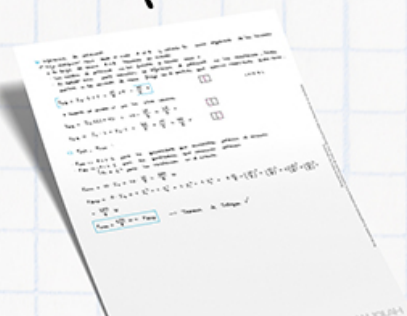


**Descarga sin publi
con 1 coin**



Después

WUOLAH



Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

perdo
espacio



Necesito
concentración

ali ali ooh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial

Alejandro Antuña Alonso

SDI – TEORÍA PRIMER PARCIAL

1. Introducción a patrones para la Web:

Patrones:

Patrón: repetición de las mejores prácticas de lo que funciona en cualquier dominio. Tipos de patrones:

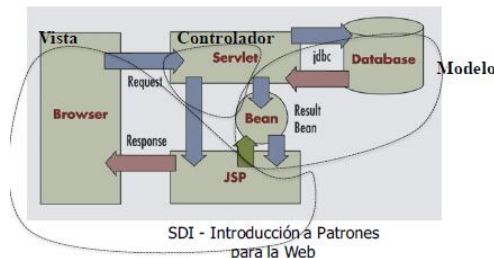
- Arquitectónicos: relacionados con el diseño a gran escala y de granularidad gruesa.
- Diseño: relacionados con el diseño de objetos y frameworks de pequeña y mediana escala.
- Estilos: soluciones de diseño de bajo nivel orientadas a la implementación o al lenguaje.

Servlets y JSPs:

- Servlets: buenos ejecutando lógica de negocio, pero no presentando información.
- JSPs: buenos presentando información, pésimos introduciendo lógica programática.
- Combinación Servlet/JSPs era lo más común en el desarrollo de aplicaciones web antes de la aparición de los frameworks MVC.
- Dos arquitecturas:
 - Model-1.5: JSPs para representación y control y JavaBeans para lógica.
 - Model-2: MVC.

Patrón MVC:

Evolución del modelo 1.5:



Descripción de roles:

- Modelo: representación específica del dominio del problema. Este rol también incluiría la capa de datos. La lógica de dominio añade significado a los datos.
- Vista: presenta el modelo en un formato adecuado para interactuar, usualmente un elemento de interfaz de usuario.
- Controlador: código navegacional. Recibe eventos, usualmente acciones del usuario, invoca al modelo y a la vista.

Controlador:

- Servlet central recibe peticiones, procesa URL recibida y delega procesamiento a JavaBeans.
- Servlet guarda resultado de procesamiento realizado por JavaBeans en el contexto de la petición, la sesión o la aplicación.
- Servlet transfiere control a un JSP que lleva a cabo la presentación de resultados

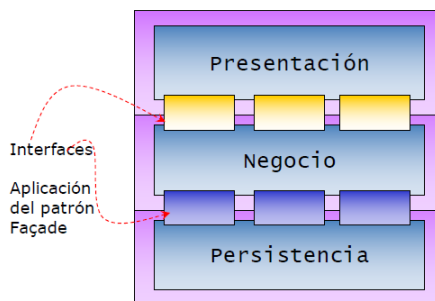
Modelo:

- JavaBeans (o EJBs para aplicaciones más escalables) juegan el rol de modelo:
 - o Algunos beans ejecutan lógica.
 - o Otros guardan datos.
- Normalmente:
 - o Servlet controlador invoca un método en bean lógico y éste devuelve un bean de datos.
 - o Autor de JSP tiene acceso a bean de datos.

Vista:

- Rol ejecutado por JSPs.
- Servlet Controlador transfiere control al JSP después de haber guardado en un contexto el resultado en forma de un bean de datos.
- JSP usa `jsp:useBean` y `jsp:getProperty` para recuperar datos y formatear respuesta en HTML o XML.

Patrón Capas:



SDI - Introducción a Patrones para la Web

Presentation Layer, Business Layer, Persistence Layer.

- Layer: capa arquitectónica de la aplicación software.
- Tier: capa física de la arquitectura de despliegue del hardware.
- Las "layers" se despliegan sobre las "tiers".

Arquitectura en capas:

Las capas se comunican a través de interfaces:

- Las implementaciones están ocultas al exterior.
- Una factoría sirve una implementación para cada interfaz.
- La capa superior se comunica con la inferior, no al revés.
- Las capas no se pueden saltar.

Patrones para la arquitectura en capas:

















Presentación	Negocio	Persistencia
MVC	Fachada Factoría	DAO DTO Factoría Active Record

Capa de presentación:

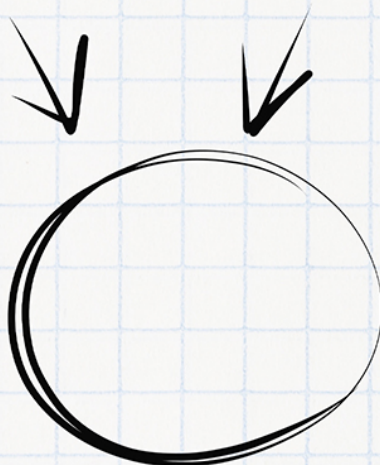
- Resuelve la interacción con el usuario.
- Incluye algo de lógica (pero de presentación):
 - o Internacionalización (i18N).
 - o Informar de los errores lógicos y de ejecución (errores internos).

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	 PLAN TURBO	 PLAN PRO	 PLAN PRO+
 Descargas sin publi al mes	10 	40 	80 
 Elimina el video entre descargas			
 Descarga carpetas			
 Descarga archivos grandes			
 Visualiza apuntes online sin publi			
 Elimina toda la publi web			
 Precios Anual <input type="checkbox"/>	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo,
¿Qué nota vas a sacar?



WUOLAH

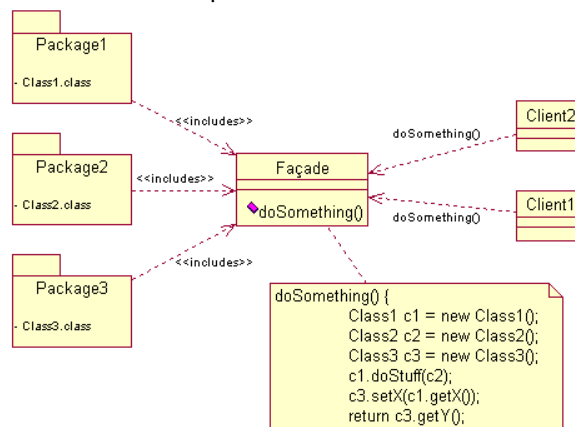
- Controlar la navegación entre pantallas.
- Algunas reglas de negocio pueden ser responsabilidad de esta capa:
 - Ocultar/deshabilitar determinado dato/control si se da una circunstancia.
- Puede estar dividida en subcapas.
- Patrones habituales:
 - MVC.
 - ServiceLocator o Factory.

Patrón Fachada:

Interfaz único y simplificado de los servicios más generales de un subsistema.

Cuando se utiliza:

- Se busca una interfaz simple para un subsistema complejo.
- Hay muchas dependencias entre clientes y clases que implementan una abstracción.
- Se desea obtener una división en capas de nuestros subsistemas.



Capa de negocio:

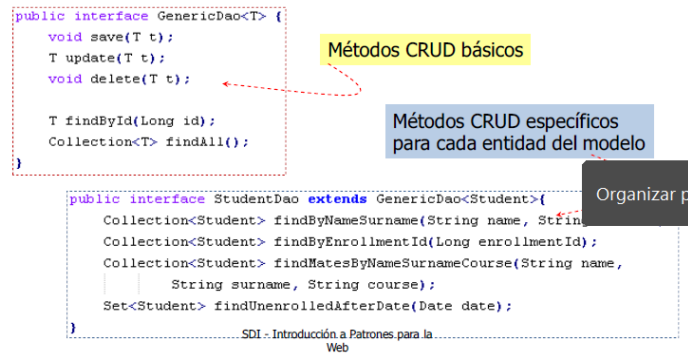
Debería poderse ejecutar fuera de cualquier entorno.

Patrón Factoría:

- Una factoría es un objeto encargado de la creación de otros objetos.
- Utilizados en ocasiones en las que hacerse con un objeto implica algo más complejo que crearlo.
- El cliente no conoce el tipo concreto de objeto a crear.

Capa de persistencia:

- Ofrece interfaz a la capa superior.
- Las distintas implementaciones de la persistencia no deben ser perceptibles por la capa de lógica --> independencia.
- Uso de patrones:
 - DTQ: Utilizado para transferir datos entre subsistemas:
 - Para reducir el número de llamadas a método.
 - Su único comportamiento viene dado básicamente por getters y setters.
 - DAO: DAO proporciona una interfaz única de acceso a los datos, de forma independiente a dónde se hallen almacenados. Independiza la lógica de negocio del acceso a los datos.



- Active Record.

2. Introducción a JEE:

JEE: especificación de Sun para una plataforma basada en APIs de Java 2 que permiten construir aplicaciones empresariales. Una aplicación Java EE no depende de una implementación particular.

Servidores de aplicaciones:

- Programa que provee la infraestructura necesaria para aplicaciones web empresariales.
- Es una capa intermedia que se sitúa entre el servidor web y las aplicaciones.
- Motivación:
 - Surgen cuando queda patente que las aplicaciones cliente/servidor no van a ser escalables a un gran número de usuarios.
 - Diseñados para gestionar de forma centralizada el modo en que los clientes podían acceder a los servicios con los que tenían que interoperar.

CGI:

- Necesario para dotar de dinamismo e interactividad a las aplicaciones web.
- Se apoya en comandos HTTP para establecer la comunicación entre servidor y aplicación.
- Se utiliza para conseguir páginas web dinámicas e interactivas.

Servlet:

- Módulos/componentes que amplían los servidores orientados a petición/respuesta.
- Son la respuesta en lenguaje Java a los CGI para construir páginas dinámicas.

Ventajas Servlet vs CGI:

- Eficiencia: JVM.
- Facilidad de uso y aprendizaje.
- Potentes.
- Portables.
- Lenguaje Java.

Contenedor de Servlets:

- Define un ambiente estandarizado de ejecución que provee servicios específicos a los servlets: dan servicio a peticiones de clientes, realizando procesamiento y devolviendo un resultado.

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

pierdo espacio



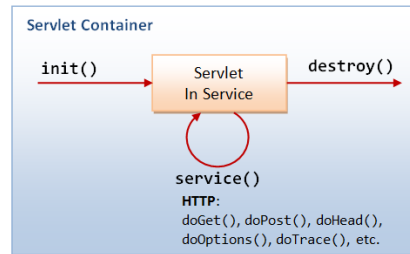
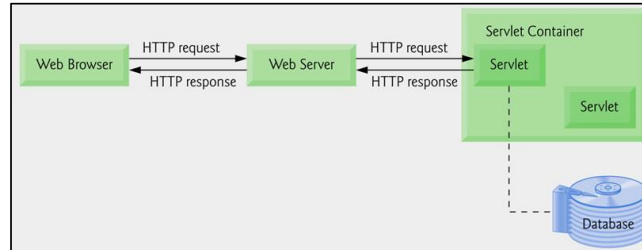
Necesito concentración

ali ali ooh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial
Alejandro Antuña Alonso

- Los contratos son interfaces Java que implementan los Servlets.



Tipos de peticiones HTTP:

Un navegador puede enviar la información al servidor de varias formas:

- Get:** paso de parámetros en la propia URL de acceso al método o recurso del servidor. Método doGet del servlet.
- Post:** idéntico a get pero los parámetros no van en la URL. Método doPost del servlet.
- Put...**

Métodos **doGet()** y **doPost()** son llamados desde el método **service()**. Reciben interfaces instanciadas:

- HttpServletRequest: canal de entrada con información enviada por el usuario.
- HttpServletResponse: canal de salida.

DOGET() CONSULTA MIENTRAS QUE DOPOST() MODIFICA DATOS.

Servlets. Respuesta en HTML:

La salida del servlet es, habitualmente, un documento HTML. Dos pasos:

- Indicar en la cabecera de la respuesta el tipo de contenido que vamos a retornar (método setContentType).
- Crear y enviar código HTML válido.

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Hola Mundo!</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("Bienvenido a mi primera página Gúev!");
    out.println("</BODY></HTML>");
```

Servlets. Registro en el descriptor de despliegue:

Insertamos en web.xml la declaración del servlet y del servlet-mapping.
http://<server>/HolaMundoCordial

```
<servlet>
    <servlet-name>HolaMundo</servlet-name>
    <servlet-class>uo.sdi.servlet.HolaMundoServlet</servlet-class>
</servlet>
```

Servlets. Registro con anotaciones:

Insertamos antes de la clase servlet la anotación con el nombre del servlet y url de mapeo.
http://<server>/HolaMundoCordial

```
@WebServlet(name="HolaMundo",urlPatterns={"/HolaMundoCordial"})
public class HolaMundoServlet extends HttpServlet { ...
```

HTTPServletRequest:

Parámetros que llegan al request:

- Request es tipo "HTTPServletRequest".
- Método getParameter("nombre").

Son los parámetros de QueryString o de campos del formulario.

Siempre nos devuelve objetos de tipo String:

- "" si no hay valor.
- null si no existe el parámetro.
- El valor en caso de haber sido establecido.

```
<form ACTION="http://server/app/HelloWorld" METHOD="POST">
    Nombre: <INPUT TYPE="TEXT" NAME="NombreUsuario" SIZE="15">
    <INPUT TYPE="Submit" VALUE="Aceptar">
</FORM>

public class HelloWorld extends HttpServlet {
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        String nombre = (String) request.getParameter("NombreUsuario");
    }
}
```

Servlets. Ciclo de vida:

1) Inicialización:

- Única llamada al método init por parte del contenedor de servlets.
public void init(ServletConfig config)
- Se pueden recoger parámetros concretos con getInitParameter. Estos parámetros están especificados en web.xml.

2) Peticiones:

- Primera petición se ejecuta en un thread que invoca a service.
- El resto de las peticiones se invocan en un nuevo hilo mapeado sobre service.

3) Destrucción:

- Cuando todas las llamadas desde el cliente cesan o un temporizador lo indique.
public void destroy()

Servlets. Seguimiento de sesión:

Los servlets proporcionan como solución la API HttpSession, una interfaz de alto nivel construida sobre las cookies y la reescritura de URLs. Permite almacenar objetos.

Para utilizar HttpSession asociado a una petición HTTP se utiliza el método getSession() que devuelve null si no hay una sesión asociada.

Añadir y recuperar información de una sesión:

- **getAttribute("nombre_variable"):**
 - Devuelve una instancia de object en caso de que la sesión ya tenga algo asociado con la etiqueta nombre_variable.
 - Null si no hay nada asociado.
- **setAttribute("nombre_variable", referencia):**
Coloca el objeto referenciado por referencia en la sesión del usuario bajo el nombre nombre_variable. Si el objeto ya existiera, lo sobrescribe.
- **getAttributeNames():**
Retorna un Enumeration con los nombres de todos los atributos de la sesión.
- **getId():**
Devuelve un identificador único generado para cada sesión.
- **isNew():**
True si el cliente nunca ha visto la sesión. False para sesión preexistente.
- **getCreationTime():**
Devuelve la hora, en ms, en la que se creó la sesión.
- **getLastAccessedTime():**
La hora en la que la sesión fue enviada por última vez al cliente.

Caducidad de la sesión: automáticamente el servidor web invalida la sesión tras un periodo de tiempo sin peticiones o manualmente utilizando el método invalidate.

Servlets. Contexto de la aplicación:

Se trata de un saco común a todas las sesiones del usuario activas en el servidor. Nos permite compartir información y objetos entre distintos usuarios. Se accede por medio del objeto ServletContext.

Para colocar o recuperar objetos del contexto:

- Añadir atributo: método `setAttribute()`.
- Recuperar atributo: método `getAttribute()`.

JSP:

Java Server Pages: tecnología para crear páginas web dinámicas. Están construidas sobre servlets.

Beneficio: incluir mucha lógica de programación en una página web no es mucho mejor que generar HTML por programa. JSP proporciona acciones que son como etiquetas HTML, pero representan código reutilizable. Se puede invocar a otras clases Java del servidor.

JSP permite una mayor separación entre la presentación de la página y la lógica de la aplicación.

Elementos JSP:

- **Scripting**: permite insertar código Java que será ejecutado en el momento de la petición.
 - `<% ... %>` Scriptlet. Encierra código Java.
 - `<%= ... %>` Expresión. Permite acceder al valor devuelto por una expresión en Java e imprimirlo en OUT.
 - `<%! ... %>` Declaración. Usada para declarar variables y métodos en la clase correspondiente a la página.
- **Directivas**: permiten especificar información acerca de la página que permanece constante para todas las peticiones.
- **Acciones**: permiten ejecutar determinadas acciones sobre información que se requiere en el momento de la petición JSP.

```
<%!  
private java.util.Date hora = new java.util.Date();  
  
java.util.Date getHora(){  
    return hora;  
}  
  
String getSaludo() {  
    String saludo;  
    if (hora.getHours() < 13) {  
        saludo = "Buenos días";  
    } else if (hora.getHours() < 20) {  
        saludo = "Buenas tardes";  
    } else {  
        saludo = "Buenas noches";  
    }  
    return saludo;  
}  
%>  
<HTML>  
<HEAD>  
<TITLE>Hola Mundo!</TITLE>  
</HEAD>  
<BODY>  
<center>Bienvenido a mi primera página Web!</center>  
<p>Son las <%=getHora() %>, <%=getSaludo() %></p>  
</BODY>  
</HTML>
```

Objetos predefinidos:

El código Java incrustado en JSP tiene acceso a los mismos objetos predefinidos que tenían los servlets.

- **Request**. Permite acceder a:
 - Parámetros de la petición.
 - Tipo de petición.
 - Cabeceras HTTP entrantes.Se permite que la petición sea una subclase de `ServletRequest` distinta de `HttpServletRequest`, si el protocolo de la petición es distinto de HTTP.
- **Response**.
- **Out**: `PrintWriter` utilizado para enviar la salida al cliente. Esta es una versión con buffer `PrintWriter` llamada `JspWriter`.
- **Session**: las sesiones se crean automáticamente. Se puede utilizar el atributo sesión para desactivar las sesiones.
- **Application**: `ServletContext` obtenido mediante `getServletConfig().getContext()`.

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

pierdo espacio



Necesito concentración

ali ali ooh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial

Alejandro Antuña Alonso

Directivas JSP:

Son mensajes para el contenedor JSP.

- **<%@ page ... %>** Permite importar clases Java, especificar el tipo de la respuesta ("text/html" por omisión), etc.
 - Import=... permite especificar paquetes que deben ser importados.
 - ContentType=... especifica el tipo MIME de salida. Por defecto HTML.
 - isThreadSafe: indica procesamiento de servlet normal, múltiples peticiones pueden procesarse con un solo ejemplar del servlet. Si es false, indica que el servlet debe implementar SingleThreadMode.
 - session: true indica que la variable predefinida debería unirse a la sesión existente. Si no existe se debe crear una nueva para incluirla. Si es false, no se utilizarán sesiones.
 - buffer: tamaño del buffer para JSPWriter out.
- **<%@ include ... %>** Permite incluir otros ficheros antes de que la página sea traducida a un servlet.
 - **<%@ include file="copyright.html" %>**
- **<%@ taglib ... %>** Declara una biblioteca de etiquetas con acciones personalizadas para ser utilizadas en la página. Identifica la librería por su URL y le asocia un prefijo para usar en código JSP.

Acciones JSP:

Utilizan construcciones de sintaxis XML para controlar el comportamiento del motor de servlets.

Tipos de acciones: estándar, a medida, JSTL.

Elemento	Descripción
<jsp:useBean>	Permite usar un <i>JavaBean</i> desde la página
<jsp:getProperty>	Obtiene el valor de una propiedad de un componente <i>JavaBean</i> y lo añade a la respuesta
<jsp:setProperty>	Establece el valor de una propiedad de un <i>JavaBean</i>
<jsp:include>	Incluye la respuesta de un servlet o una página JSP
<jsp:forward>	Redirige a otro servlet o página JSP
<jsp:param>	Envía un parámetro a la petición redirigida a otro servlet o JSP mediante <jsp:include> o <jsp:forward>
<jsp:plugin>	Genera el HTML necesario para ejecutar un <i>applet</i>

Encadenamiento de Servlets/JSPs:

- Desde un Servlet concatenar otro Servlet:
 - RequestDispatcher/forward/include.
- Desde un JSP:
 - **<jsp:forward>**
 - **<jsp:include>**

3. Spring Boot:

- **Spring:** framework basado en JEE que utiliza el patrón MVC para el desarrollo de aplicaciones web cuya característica principal es el uso de un modelo POJO. Otras características:
 - Sistema de inyección de dependencias basado en el IoC container (Inversion of control).
 - Gran cantidad de módulos con funcionalidad reutilizable.
 - Traducción de excepciones específicas a genéricas.
- **Spring Boot:** forma fácil de desarrollar aplicaciones Spring, pero facilitando los aspectos duros de Spring (configuración, código, servidor embebido).

Spring Boot **incrementa la agilidad** de desarrollo de aplicaciones en Spring:

- 1) Provee opciones de configuración por defecto para evitar excesivas configuraciones de Spring.
- 2) Uso opcional de POMs para simplificar configuraciones Maven.
- 3) Evita generación de código y configuraciones XML.
- 4) Permite crear aplicaciones stand-alone.

Maven:

- Permite especificar procesos para muchas acciones relativas al desarrollo software.
- Gestión de dependencias / artefactos.
- Dependencias: se especifican en el fichero POM:

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

 - Una vez se incluye una dependencia el código se descarga e indexa al proyecto. La actualización del POM es automática.

Aplicación:

Clase principal Main: la clase de inicio implementa un método main que ejecuta la aplicación.

- Invoca a SpringApplication.run().
- Se recomienda que la clase principal incluya la etiqueta @SpringBootApplication.
- **@SpringBootApplication** fusiona:
 - @Configuration: esta clase puede definir elementos de configuración.
 - @EnableAutoConfiguration: habilita auto-configuración. Configura de forma automática muchas funciones básicas y avanzadas de la aplicación. Crea instancias de una serie de objetos en forma de Beans.
 - Cualquier auto-configuración puede ser redefinida si nos interesa:

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```
 - @ComponentScan: se debe escanear la aplicación en busca de componentes implementados. Una vez detectados se registra como Beans.

Propiedades: se incluyen en el fichero application.properties.

Aunque es menos común, la configuración de las propiedades se puede definir desde:

- Las clases, al iniciar la aplicación.
- Con anotaciones específicas.
- Fichero de propiedades application.yml.
- Parámetros en la línea de comandos al ejecutar la aplicación.

Elementos principales:

- **Componentes:**
 - **Componente:** componente genérico sin propósito específico.
 - **Controladores:** definen peticiones que van a ser recibidas por la aplicación. Suelen utilizar varios servicios y retornan respuestas.
 - **Servicios:** definen funcionalidades disponibles en la capa de lógica de **negocio**. Suelen utilizar repositorios.
 - **Repositorios:** definen acceso al SGDB.
 - **Configuración:** definen funcionalidades de configuración transversal, no relativa a la lógica de negocio.
- **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.

Controladores, Servicios y Repositorios son estereotipos de Componentes:

- Todos son componentes.
- Algunos añaden funcionalidad adicional al componente.
- Cada uno se utiliza con propósitos muy diferentes.

Todos los componentes se procesan internamente, son instanciados y registrados como Beans.

Existe la declaración explícita de beans **@Bean**:

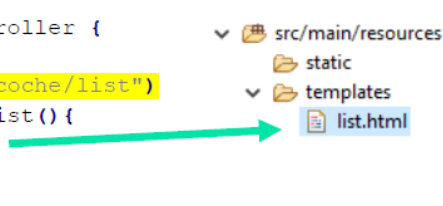
- Para beans de configuración o funcionalidad transversal.
- Se crea manualmente la instancia de una clase y se registra como Bean.
- Pueden ser inyectados en el código de la aplicación.

Controladores: componentes que procesan peticiones realizadas por los clientes. Pueden invocar a la lógica de negocio y generar una respuesta.

- **@Controller** indica que la clase es un componente de tipo controlador.
- **@RequestMapping** indica que el método responderá a peticiones.
- El retorno es el nombre de una vista.

```
@Controller
public class CocheController {

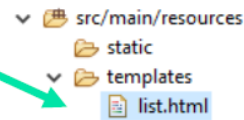
    @RequestMapping("/coche/list")
    public String getList() {
        return "list";
    }
}
```



The diagram illustrates the mapping between the return value of the `getList()` method and the corresponding view file. A green arrow points from the `return "list";` line in the code to the `list.html` file located in the `templates` directory under `src/main/resources`.

- Suelen invocar lógica de negocio definida en los servicios:

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



Mapeo de peticiones: @RequestMapping.

Atributos que admite:

- value, especifica la URL.
- method, especifica el tipo de petición HTTP:
 - RequestMethod.POST, RequestMethod.GET.

Peticiones 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;
}
```

@RequestParam debe especificar el tipo de dato. Si los tipos de datos no coinciden se produce una excepción.

Los parámetros pueden ser opcionales:

- required = false
- Podrían tomar valor null.

O valores por defecto:

- value = valor

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

@ModelAttribute construye automáticamente un objeto en base a los parámetros recibidos. La clase utilizada como @ModelAttribute debe definir un constructor sin parámetros y métodos get para sus atributos.

Al recibir una petición se crea un objeto. Después:

- 1) Se completan los atributos en los que haya coincidencia de nombres.
- 2) 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) {
    ...
}
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

perdo
espacio



Necesito
concentración

ali ali ooh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial
Alejandro Antuña Alonso

Respuestas:

Retornan la ruta para localizar una plantilla. El motor de plantillas se encarga de generar la vista.

@ResponseBody hace que una respuesta sea un objeto en lugar de una plantilla:

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

@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.

Thymeleaf. Motores de plantillas:

Permiten componer respuestas de forma dinámica y ágil. Concebido para la definición de vistas. Por defecto, las plantillas se almacenan en la carpeta /templates y combinan lenguajes web y Thymeleaf.

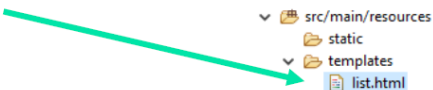
Modelos y atributos:

Los controladores pueden enviar un modelo de datos a la plantilla. El modelo contiene atributos que se identifican por claves. model.addAttribute(clave, objeto) agrega el 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";
}
```



1. Inserta los valores de los atributos en el HTML:

- Acceso al atributo \${<clave>}
- Incluir en HTML th:<propiedad_HTML>="expresión"

```
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>
```

c. Se puede modificar cualquier atributo HTML:

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

Genera: Enlace

d. Admite el uso de literales, operaciones y utilidades:

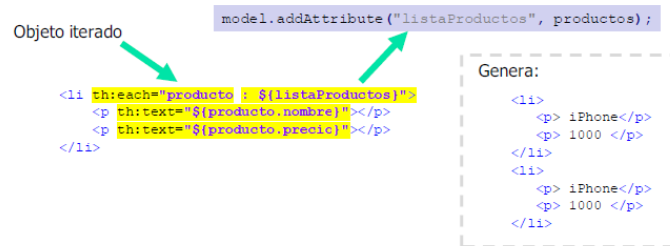
```
<a th:href="${ /producto/detalles"+identificador}">Enlace</a>
```

Genera: Enlace

e. Si el atributo es un objeto se accede a sus atributos y métodos con el operador ': \${producto.nombre}

2. Atributos en estructuras de control:

a. Bucle `th:each`:



b. Condiciones `th:if`. El bloque HTML sólo se incluye si cumple la petición:

```
<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>
```

3. Atributos en JavaScript:

a. Sintaxis: `[[${param.clave_atributo_del_modelo}]]`

b. El script debe:

i. Incluir `th:inline="javascript"`

ii. Encapsular código en `/*<![CDATA[*] [*]]>*/`

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

Link URL Expressions:

El operador `@` ofrece funcionalidad para gestión de parámetros en URLs. Es útil para insertar propiedades `src` y `href`.

Sintaxis: `@{literales y variables$(<clave_parámetros>=valores)}`

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

Genera href="/detalles?id=322"

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

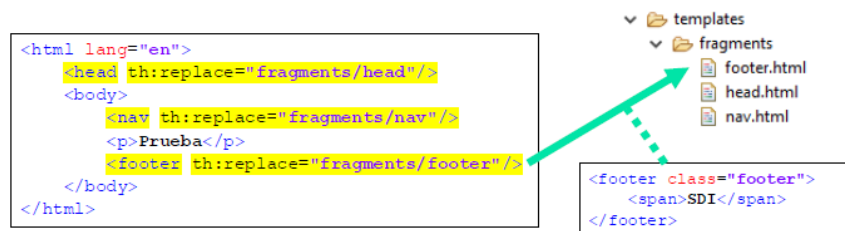
Genera href="/detalles/322/"

Incluir y reemplazar:

Varias posibilidades para componer plantillas a partir de otras. Evitan replicar partes comunes en varias vistas.

Forma más rápida de utilizar:

- o `th:include="ruta plantilla"`: incluye el contenido de una plantilla dentro del tag HTML.
- o `th:replace`: sustituye el tag HTML por el contenido de la plantilla.



Servicios: componentes que contienen la lógica de negocio. Suelen ser utilizados desde los controladores o desde otros servicios.

La anotación **@Autowired** se asocia a los atributos:

- Permite inyectar una dependencia sin necesidad de configuración adicional.
- La inyección es una alternativa a instanciar un objeto.

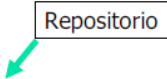
🚦 **Funcionamiento básico de los Servicios:**

- Implementan métodos de lógica de negocio.
- Suelen acceder a repositorios de datos:

```
@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.findById(id);
        coche.agregarRevision(revision);
        ...
    }
}
```



🚦 **Inyección:**

- Inyección de dependencias es una forma de “Inversión de control”.
- Inyectamos objetos en lugar de instanciarlos para:
 - Evitar que las clases tengan que saber cómo instanciar/obtener el objeto.
 - Código más modular, desacoplado y reusable.
 - Simplifica pruebas de componentes / test unitarios.
 - Muchos frameworks ofrecen opciones avanzadas.

🚦 **@PostConstruct:**

- Permite especificar que un método se ejecutará una vez construido el componente.

🚦 **@PreDestroy:**

- Permite especificar que un método se ejecutará justo antes de destruir el componente.

🚦 **Ámbito:**

- Por defecto los servicios tienen un ámbito singleton: 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 que se inyecta el componente.
- Lógica que almacena IDs de productos agregados al carrito de compra para cada sesión:

```
@SessionScope
@Service
public class CarritoService {

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

Repositorios: componentes que acceden a la BBDD. Suelen ser utilizados desde la capa de servicios.

Notación **@Repository** indica que una clase es un componente de tipo repositorio.

Los repositorios tienen habilitado por defecto la PersistenceExceptionTranslationPostProcessor, que traduce errores generados en cualquier proceso de persistencia en objetos DataAccessException.

Spring Boot suele integrarse con **JPA**.

Datasource:

- Incluir entradas en application.properties.
- Importar dependencia org.hsqldb.

Funcionamiento básico de los Repositorios:

- `CrudRepository <Clase_entidad, tipo clave primaria >`
`import org.springframework.data.repository.CrudRepository;`
`public interface CochesRepository extends CrudRepository<Coche, Long>{`
- Extensión del CrudRepository:
 - Sin consulta: métodos estándar por atributos de la entidad:
 - Nombre del método debe coincidir con el del atributo.
 - No requieren especificar la consulta.`Coche findByMatricula(String matricula);`
`Iterable<Coche> findAllByModelo(String modelo);`
 - Con consulta: métodos con consultas específicas @Query y JPQL:
`@Query("SELECT c FROM Coche c WHERE c.caballos >= ?1")`
`Iterable<Coche> cochesConCaballos(int caballos);`
- En la definición de entidades:
 - @Entity: la clase es una entidad.
 - @Id: atributo es PK.
 - @GeneratedValue: atributo se genera automáticamente.
 - @Column(unique=true): atributo único.
 - @Transient: no queremos guardar el atributo en la BBDD.
- Se inyectan como un componente, @Autowired en la capa de Servicio:

```
@Service
public class CochesService {

    @Autowired
    CochesRepository cochesRepository;

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

Inyección de repositorio

Uso del repositorio

Datos de prueba:

- Servicio @Autowired InsertSampleDataService y una función @PostConstruct para agregar datos.

Thymeleaf. Fragmentos: para ganar fluidez y eficiencia en ocasiones no se puede retornar una plantilla/página completa. Una alternativa es retornar sólo una o varias partes (Fragmentos) de la plantilla/página que se sustituyen o insertan en ella.

Atributo `th:fragment` determina un fragmento en una plantilla. Se suele dar una id al nodo padre de `th:fragment` que facilita referenciarlo desde JavaScript.

```
<table class="table table-hover" th:fragment="tableMarks" id="tableMarks">
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

pierdo espacio



Necesito concentración

ali ali oohh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial
Alejandro Antuña Alonso

Controladores:

- Los controladores referencian los fragments con:
 - <ruta plantilla> :: <nombre fragmento>

```
@RequestMapping("/mark/list/update")
public String updateList(Model model){
    model.addAttribute("markList", marksService.getMarks() );
    return "mark/list :: tableMarks";
}
```

Sólo retorna el fragmento tableMarks de la vista.

Cliente:

- Incluimos código jQuery en las vistas/plantillas para:
 - Obtener el fragmento.
 - Incluirlo dentro de un elemento de la página actual.
- \$(<selector>).load(<url>) permite obtener y cargar el fragmento:

```
<button type="button" id="updateButton">Actualizar</button>
<script>
    $( "#updateButton" ).click(function() {
        $("#tableMarks").load('/mark/list/update');
    });
</script>
<div class="table-responsive">
```

Configuración: la funcionalidad relativa a la configuración es genérica, no tiene que ver directamente con la lógica de negocio. Algunas funciones comunes de configuración son:

- Configuración del sistema de seguridad.
- Del sistema de paginación.
- Del sistema de internacionalización.

Estas clases incluyen la notación **@Configuration**. Suelen heredar de una clase de configuración del framework. Estas clases base se pueden utilizar de diferentes formas, lo más común:

- Sobrescribiendo métodos para personalizar funcionamiento.
- Utilizando métodos definidos en la clase configuración.

@Bean:

- Las clases de configuración instancian objetos como Beans.
- Estos objetos definen:
 - Funcionalidad necesaria para la propia configuración.
 - Funcionalidad común que será utilizada en otras partes de la aplicación.
- Lo más común es que sean clases del framework.
- Al estar en una clase @Configuration, los Beans se registran al iniciar la aplicación.

Internacionalización:

Consiste en adaptar una aplicación a diferentes idiomas o regiones. Se trata de traducir todos los contenidos y estándares a los propios de un idioma o región.

Configuración:

- Incluir internacionalización requiere ampliar la configuración de la aplicación.
- Se añade una clase @Configuration que extienda de WebMvcConfigurerAdapter: una de las clases más genéricas de configuración.

Interceptores:

- Interceptores: pueden procesar peticiones antes de que lleguen al controlador.
- Implementación de interceptores:

```
public class MiInterceptor extends HandlerInterceptorAdapter {

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object object) throws Exception {

        System.out.println("Interceptar la petición");
        // Buscar el parámetro "limite"
        Integer parametroLimite =
            ServletRequestUtils.getIntParameter(request, "limite", 0);


        return true;
    }
}
```

✚ **LocaleChangeInterceptor** es un interceptor implementado en el framework:

- Permite definir un parámetro para realizar cambios de localización.
- Si la petición contiene el parámetro se cambia la localización.
- El interceptor está activo sobre las URLs del sitio.
`http://www.ejemplo.com/index?lang=es`
- Se crea un Bean con la instancia de LocaleChangeInterceptor. Se sobrescribe el método `addInterceptors` y se registra el Bean Interceptor:

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

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



✚ **LocaleResolver**: objeto del framework que permite hacer cambios automáticos de idioma. Para habilitar su funcionalidad se debe registrar una instancia de LocalResolver como `@Bean`.

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("es", "ES"));
    return localeResolver;
}
```

✚ **Mensajes**:

- Se definen en ficheros de propiedades:
`messages_es -----> welcome.message=Bienvenidos a la página principal`

✚ **Cambio de idioma**:

- Se define un parámetro en la instancia de LocaleChangeInterceptor:

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

- El sistema selecciona los mensajes del fichero correspondiente al idioma actual:
`?lang=en` se usa el fichero `messages_en.properties`

Spring Security:

✚ **Autenticación**:

- Proceso para validar la identidad de un usuario.
- Procesos más comunes verifican si existe coincidencia para un identificador único de usuario y una contraseña.
- Datos de autenticación: username, contraseña, rol (grupo).

🚩 Encriptación:

- El servicio que guarda los usuarios debe encriptar la contraseña.
- BCryptPasswordEncoder soporta la encriptación.

```
@Service
public class UserService {
    ...
    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public void addUser(User user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        usersRepository.save(user);
    }
}
```

🚩 Configuración:

- La clase de configuración de Spring Security debe ser hija de WebSecurityConfigurerAdapter. Incluimos las anotaciones @Configuration y @EnableWebSecurity:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

🚩 Autorización:

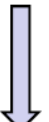
- Definición de autorizaciones: authorizeRequests()

- Se anidan funciones antMatchers y anyRequests.
- antMatchers("URLS"): especifica la URL(s).
- anyRequest especifica la autorización:
 - permitAll(): cualquier petición puede acceder.
 - authenticated(): usuario autenticado.
 - hasAuthority("ROLES"): para acceder el usuario debe tener el rol especificado.

```
http
.authorizeRequests()
    .antMatchers("/css/**", "/img/**", "/script/**").permitAll()
    .antMatchers("/principal", "/", "/registrarse").permitAll()
    .antMatchers("/notas/misnotas").authenticated()
    .antMatchers("/dellesApp") .hasAuthority("ROLE_ADMIN","ROLE_PROFESOR")
    .antMatchers("/usuarios/**").hasAuthority("ROLE_ADMIN")
    .anyRequest().authenticated()
```

- Se basa en un orden de prioridad. La petición /notas/misnotas coincide con 3 matchers: sería gestionada por el primero ("ROLE_PROFESOR"):

```
http
.authorizeRequests()
    .antMatchers("/css/**", "/img/**", "/script/**").permitAll()
    .antMatchers("/principal", "/", "/registrarse").permitAll()
    .antMatchers("/notas/misnotas").hasAuthority("ROLE_PROFESOR")
    .antMatchers("/notas/**").authenticated()
    .antMatchers("/usuarios/**").hasAuthority("ROLE_ADMIN")
    .anyRequest().authenticated()
```



- Formulario de autenticación/login:

- Función formLogin() --> NO incluido en el authorizeRequests().
- Dentro se anidan:
 - loginPage("URL"): URL formulario de login.

- Tipo de autenticación
(`permitAll()`, `authenticated()`...).
- `defaultSucessUrl("URL")`: url que se carga después de la autenticación válida.

```
http
.authorizeRequests()
.antMatchers("/css/**", "/img/**", "/script/**", "/", "/signup").permitAll()
.anyRequest().authenticated()
.and()
.formLogin()
.loginPage("/login")
.permitAll()
.defaultSuccessUrl("/principal")
```

Concatenador `and()`

○ **Sistema logout:**

- Función `logout()`.
- Por defecto utiliza la URL `/logout`.
- Dentro se anidan:
 - Tipo de autenticación
(`permitAll()`, `authenticated()`...).
 - `logoutSucessUrl("URL")`: redirección después de cerrar sesión.

```
.formLogin()
.loginPage("/login")
.permitAll()
.defaultSuccessUrl("/home")
.and()
.logout()
.permitAll()
.logoutSucessUrl("/despedida")
```

Concatenador `and()`

🚩 **CSRF:**

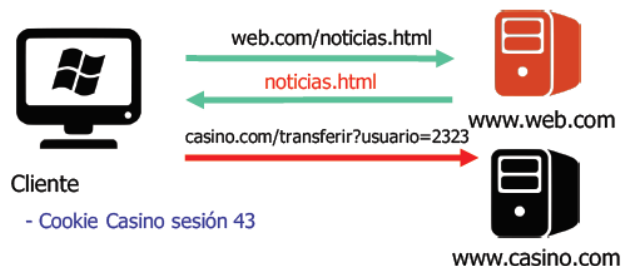
- Activada por defecto. Para desactivarla se utiliza `disable()`:

```
.logout()
.permitAll()
.logoutSucessUrl("/despedida")
.and()
.csrf().disable();
```

- Falsificación de petición en sitios cruzados CSRF. Basado en que un cliente envía peticiones no intencionadas y de carácter malicioso a un sitio web en el que confía:

▪ **Ejemplo**

- El cliente continua navegando en varias páginas
 - Por ejemplo, un link compartido en un comentario www.web.com/noticias.html
- La página **noticias.html** se carga en el navegador del usuario
 - Contiene código JavaScript con una petición www.casino.com/transferir?usuario=2323
 - Seguramente una petición en background no perceptible
- El **cliente** ejecuto **realizo sin saberlo una petición no deseada**
 - Al estar autenticado en www.casino.com la petición fue aceptada



Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

pierdo espacio



Necesito concentración

ali ali oohh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial

Alejandro Antuña Alonso

UserDetails:

- Información del usuario relativa a la autenticación se almacena en la BBDD de forma estándar: username, contraseña y rol.
- Spring Security no procesa usuarios de la lógica, sino objetos UserDetails que contienen únicamente la información de autenticación.

UserDetailsService:

- Encargado de crear los UserDetails.
- Sobrescribe la función `loadByUsername("username")`, que debe:
 - 1) Obtener el usuario asociado al username del repositorio.
 - 2) Crear una colección de tipo `GrantedAuthority`. Aquí se almacenan los roles.
 - 3) Crear un objeto de tipo `UserDetails`.
 - 4) Retornar el objeto `UserDetails`.

Controladores:

- Un esquema común en Spring Security podría ser:
 - GET /signup: muestra vista de registro.
 - POST /signup: registra al usuario en la aplicación.
 - GET /login: muestra vista de inicio de sesión.

Fuerza bruta:

- Detectar cuando un usuario intenta identificarse repetidamente sin éxito debe lanzar una acción de seguridad.



```
@Component
public class AuthenticationFailureListener implements
    ApplicationListener<AuthenticationFailureBadCredentialsEvent> {

    @Override
    public void onApplicationEvent(AuthenticationFailureBadCredentialsEvent e) {
        String username = e.getAuthentication().getName();

        WebAuthenticationDetails detalles =
            (WebAuthenticationDetails) e.getAuthentication().getDetails();
        String ip = detalles.getRemoteAddress();
        String idSession = detalles.getSessionId();
    }
}
```

Thymeleaf. Autenticación:

Acceso a los elementos de Spring Security desde las plantillas.

-  Atributo `sec:authentication` contiene información sobre el cliente autenticado.
-  Atributo `sec:authorize` contiene información sobre autorizaciones. Se utiliza para mostrar HTML condicional a nivel de autorización.

Validación de datos:

Los datos introducidos por los usuarios deben ser válidos. Consiste en comprobar que los datos son adecuados: formato, longitud, reglas de negocio...

Tipos de validaciones:

- En el cliente: código script que se ejecuta en el navegador. Valida los datos antes de enviarlos al servidor, evitan enviar al servidor una petición inválida. Son fáciles de desactivar.
- En el servidor: comprobaciones realizadas en el servidor. Validan datos al recibir la petición. No pueden ser desactivadas por el cliente.

Clase Validator:

- Se recomienda implementar un validator por proceso/formulario.
- Implementación:
 - Debe implementar la interfaz Validator.
 - Se gestionan como Bean.
 - Sobrescribir la función validate(Object, Errors):
 - Object: objeto creado con los campos del formulario.
 - Errors: mensajes de error.

```
@Override
public void validate(Object target, Errors errors) {
    User user = (User) target;
    if (user.getDni().length() < 5 || user.getDni().length() > 24)
        errors.rejectValue("dni", "El DNI debe tener entre 5 y 24 caracteres");
    if (userService.getUserByDni(user.getDni()) != null) {
        errors.rejectValue("dni", "El DNI ya está siendo usado");
    }
}
```

Agregar el validator:

- Vinculación de una petición con un validator: inyectamos el Bean validator en un controlador:

```
@Autowired
private SignUpFormValidator signUpFormValidator;
```

- El controlador debe recibir información adicional de la validación

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute @Validated User user, BindingResult result) {
    signUpFormValidator.validate(user, result);
    if (result.hasErrors()) {
        return "signup";
    }
    user.setRole(rolesService.getRoles()[0]);
    userService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home"; //Se accede la vista de operaciones autorizadas/privadas
}
```

Sesión:

La aplicación crea una sesión para cada nuevo cliente, cada navegador que accede genera una nueva sesión identificada con un ID único.

La aplicación envía el ID del cliente en una cookie que posteriormente el cliente envía con ese ID en todas las peticiones al servidor.

La aplicación puede almacenar/recuperar datos de las sesiones. Son datos de carácter temporal, se destruyen automáticamente después de un tiempo de inactividad.

Tiene una configuración por defecto que define su funcionamiento.

HttpSession:

- Permite acceder a la sesión.
- Funciona como una tabla hash.

Thymeleaf:

Plantillas pueden acceder a la sesión:

- Indirectamente: atributos del modelo enviados desde el controlador a la plantilla.
- Directamente: utilizando el objeto {\${session}}

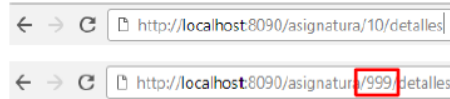
Beans:

- Por defecto se gestionan con singleton.
- Anotación @SessionScope: una instancia distinta para cada sesión.

Datos y acciones sensibles:

Ejemplo

- El usuario "**profesor1**" se autentica en la aplicación
- Su lista de **asignaturas** muestra las asignaturas: 10 , 11 y 12.
- Abre la URL de la asignatura 10, modifica la URL manualmente



- La aplicación permite el acceso, porque
 - **Sí** comprueba que el usuario esta autenticado
 - **No** comprueba que es el dueño de asignatura 999

```
public Asignatura getAsignatura(Long id){
    Asignatura asignatura = asignaturasRepository.findOne(id);
    return asignatura;
}
```

Las URLs con datos/acciones sensibles sólo pueden ser utilizadas por ciertos usuarios. Se deben identificar todos los datos sensibles de la aplicación e incluir las comprobaciones necesarias en la implementación: comprobación en el servicio, recibe id de asignatura y de profesor.

```
public Asignatura getAsignatura(Long id, Long prof){
    Asignatura asignatura
        = asignaturasRepository.getAsignaturaDeProf(id, prof);
    //@Query("SELECT a FROM Asignatura a WHERE a.id = ?1 AND a.prof = ?2")
    return asignatura;
}
```

DecisionManager:

- La protección de URLs se puede realizar también en la configuración de Spring Security.
- WebSecurityConfigurerAdapter declara diferentes políticas de acceso:
 - authenticated(), hasAuthority().
 - accesDecicisionManager(manager): crea políticas de acceso específicas (ejecutan lógica de negocio).

AccessDecision:

- Las comprobaciones de acceso se realizan en una clase que implementa AccesDecisionVoter<FilterInvocation>.
- El retorno de vote(authentication, filter, atributes) determina si habrá acceso. Tipos de retorno:
 - ACCESS_DENIED.
 - ACCESS_GRANTED.
 - ACCESS_ABSTAIN.

```
public class EjemploVoter implements AccessDecisionVoter<FilterInvocation> {

    @Override
    public int vote(Authentication authentication, FilterInvocation filter,
        Collection<ConfigAttribute> attributes) {

        System.out.println("Petición a : "+filter.getRequestUrl());
        if ( authentication.getName().startsWith("SDI")) {
            return ACCESS_GRANTED;
        } else {
            return ACCESS_DENIED;
        }
    }
}
```

- La interfaz `AccessDecisionVoter` requiere sobrescribir los métodos `supports()`.
- Para aplicar los `AccessDecisionVoter` en la configuración de seguridad:
 - Se agrupan los `AccessDecisionVoter` en un `AccessDecisionManager`.
 - Se incluye la política `.accessDecisionManager(manager)` para un conjunto de URLs.

```
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
    public AccessDecisionManager accessDecisionManager() {  
        List<AccessDecisionVoter<? extends Object>> decisionVoters  
            = Arrays.asList(new EjemploVoter(), new EjemploVoter2(), new EjemploVoter3());  
        return new UnanimousBased(decisionVoters);  
    }  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http  
            .authorizeRequests()  
            .antMatchers("/us/streaming/*").permitAll().accessDecisionManager(accessDecisionManager())  
            .antMatchers("/mark/add").hasAuthority("ROLE_PROFESSOR")  
    }  
}
```

Paginación:

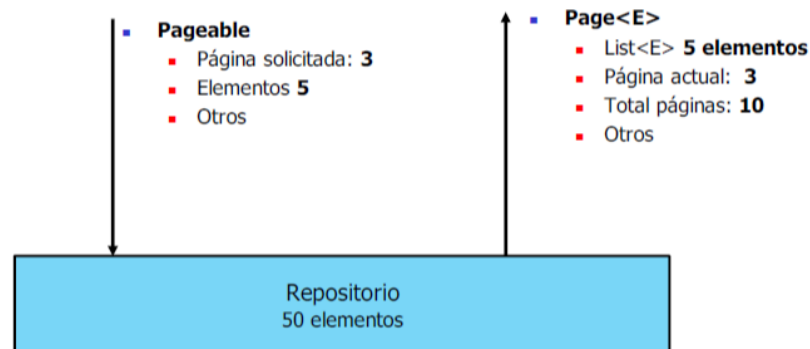


NO se deben manejar colecciones con muchos recursos/entidades. Cargar muchos en la misma página es costoso para el servidor y perjudica la experiencia de usuario.

Page<E>:

- Objeto del sistema de paginación. Colección de datos similar a una lista.
- El uso de la paginación afecta a los repositorios:
 - Afectará a los métodos que retornan colecciones.
 - Se usará `Page<E>` para retornar los datos obtenidos.
 - Deben recibir un parámetro adicional `Pageable` que encapsula información sobre la página solicitada: nº pag. solicitada, cantidad de elementos que debe contener...

```
Page<Mark> findAllByUser(Pageable pageable, User user);  
Page<Mark> findAll(Pageable pageable);
```



- Se puede utilizar también en los servicios y controladores que manejen paginación en lugar de otras colecciones Java.
- En caso de requerir una colección Java: `Page<E>.getContent()`.
- `getNum()` -> número de página a la que pertenecen los registros. **EMPIEZA EN 0.**
- `getTotalPages()` -> número de páginas totales.
- El parámetro pageable debe ser enviado desde los controladores a los repositorios:

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato
→ Planes pro: más coins

pierdo espacio



Necesito concentración

ali ali oohh
esto con 1 coin me
lo quito yo...

WUOLAH

SDI Primer Parcial
Alejandro Antuña Alonso

Los **Controladores** que manejan listados con paginación

- Deben recibir un nuevo parámetro **Pageable**
 - Encapsula información sobre la página solicitada
 - Debe llegar al repositorio, para que sepa como realizar la consulta

```
@RequestMapping("/mark/list")  
public String getList(Model model, Pageable pageable){
```

- La URL pasa a admitir parámetros de paginación (automático)
 - **page**: número de página a mostrar (por defecto 0)
 - **size**: número de registros en la página (por defecto 20)
 - <http://localhost:8090/mark/list?page=2&size=1>
 - Ambos son opcionales, si no figuran toman el valor por defecto

⚡ Controles:

- La vista debe ofrecer navegación por las páginas.
- Se recomienda incluir al menos: notificación de página actual, acceso a más cercanas y acceso a primera y última página.
- Sistema de navegación se implementa en un fragmento.
- Si la vista recibe el objeto Page<E> dispone de todo lo necesario para crear un sistema de navegación.

⚡ Configuración:

- Una modificación común de la paginación es especificar otros valores por omisión a los parámetros page y size.
- Se define un objeto que será insertado en el pageable que reciben los controladores.
- El nuevo objeto definirá valores page y size que serán usados en caso de omisión.

```
@Configuration  
public class CustomConfiguration extends WebMvcConfigurerAdapter{  
  
    @Override  
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers){  
  
        PageableHandlerMethodArgumentResolver resolver =  
            new PageableHandlerMethodArgumentResolver();  
  
        resolver.setFallbackPageable(new PageRequest(0, 1));  
        argumentResolvers.add(resolver);  
        super.addArgumentResolvers(argumentResolvers);  
    }  
}
```

Transacciones:

Anotación @Transactional declara el uso de transacciones en un método o componente. Todos sus métodos son transaccionales.

Ejemplo transferencia: se ejecuta reducirSaldo() y se produce una excepción antes de ejecutar aumentarSaldo(). Los datos quedan en estado inconsistente. Si incluimos @Transactional si se produce una excepción, el repositorio vuelve al estado anterior:

```
@Transactional  
public void transferencia(Long emisor, Long receptor, float cantidad){  
    datoMemoria = true;  
    cuentasRepository.reducirSaldo(emisor, cantidad);  
    int a = 4 / 0; // Excepción!!! Producida  
    cuentasRepository.aumentarSaldo(receptor, cantidad);  
}
```

Logging:

Consiste en guardar información sobre eventos relativos a la aplicación. Permite almacenar información sobre errores.

Logger:

- Obtener objeto Logger en las clases que vayan a registrar log.
- Se utilizará la factoría LoggerFactory.getLogger(clase).

```
public class AccesosService {  
  
    private final Logger log = LoggerFactory.getLogger(this.getClass());  
  
}
```

Configuración:

- Configurable en application.properties.

Subida de ficheros:

Los formularios permiten subir ficheros configurando:

```
<form method="POST" action="/informe" enctype="multipart/form-data">  
    <input type="file" name="informe" accept=".docx,.doc,.pdf" />  
    <input type="submit" value="Enviar" />  
</form>
```

El controlador puede procesar el fichero de forma estándar (Java). El fichero se podría guardar en un servidor o en la BBDD.