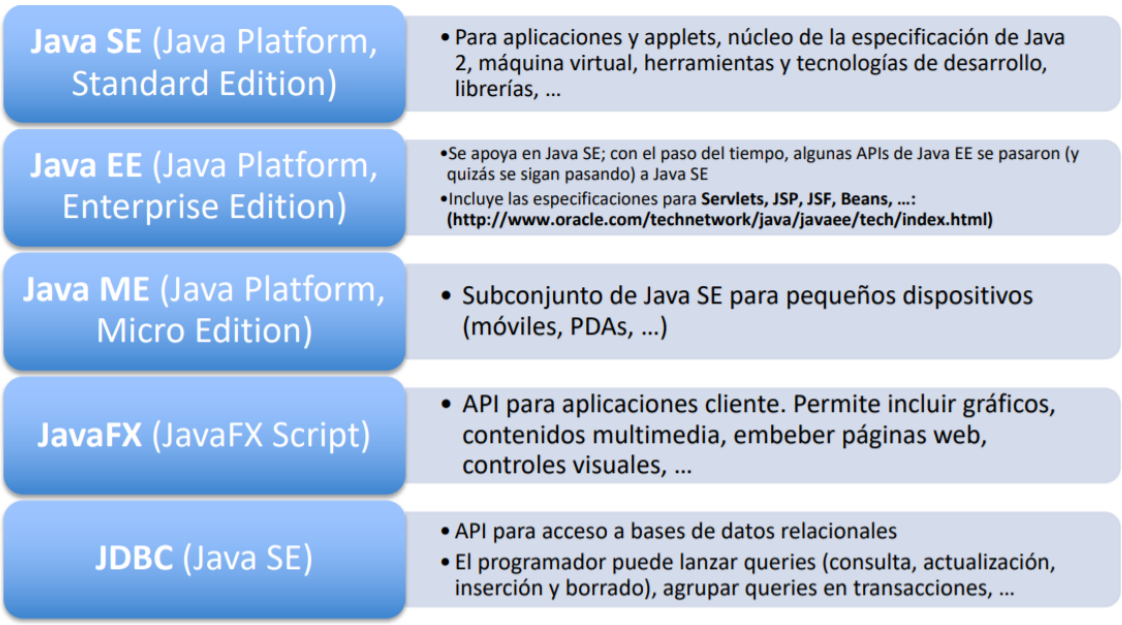
SDI SpringBoot

Guillermo Astorga Manzanal

Teoría Itroducción

Introducción: Plataformas Java



Introducción: JEE

Definición: Java EE = Java Enterprise Edition

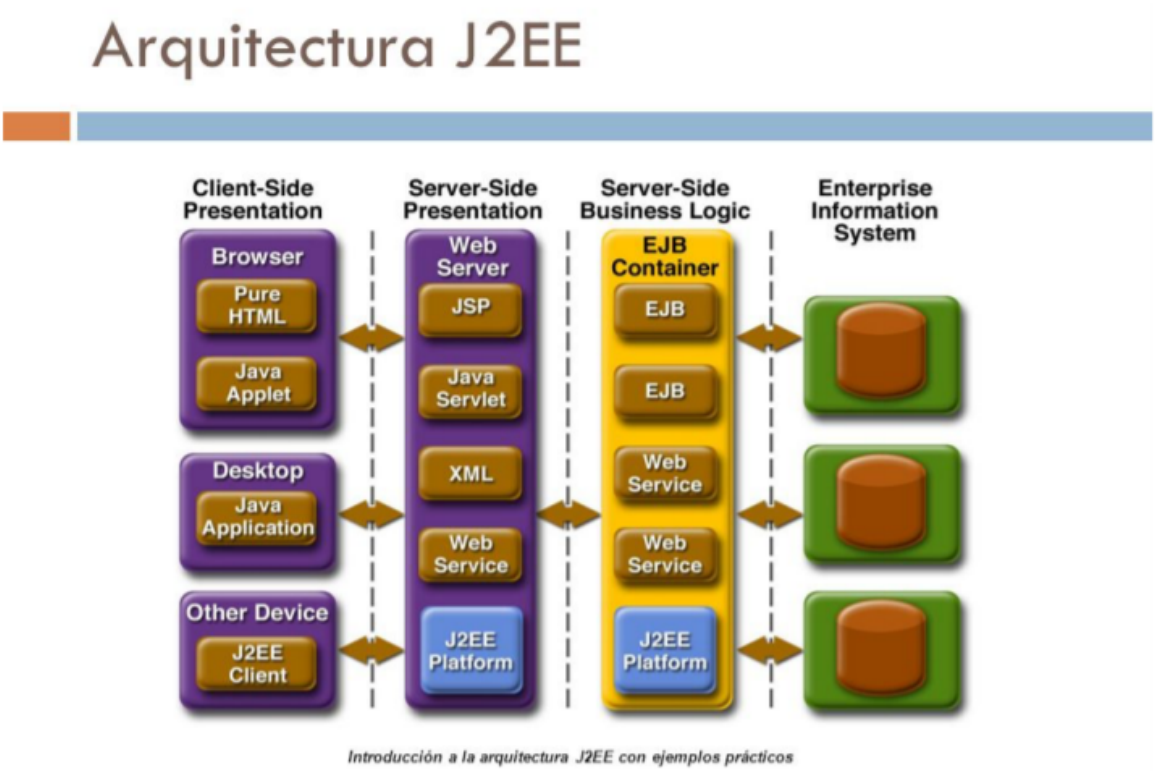
Servidores de aplicaciones

Programa que provee la infraestructura necesaria para aplicaciones web empresariales

- Los programadores van a poder dedicarse casi en exclusiva a implementar la lógica del dominio
- Servicios como seguridad, persistencia, transacciones, etc. son proporcionados por el propio servidor de aplicaciones
- Pieza clave para cualquier empresa de comercio electrónico

Es una capa intermedia (middleware) que se sitúa entre el servidor web y las aplicaciones y bases de datos subyacentes.

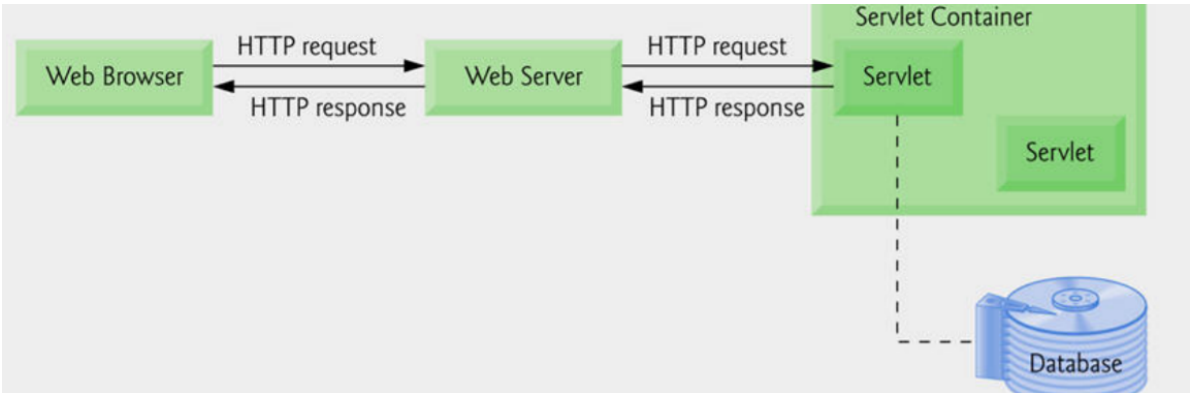
Servidores de aplicaciones



Qué es un servlet

Es una clase Java que hereda de la clase JEE HTTpServlet y que:

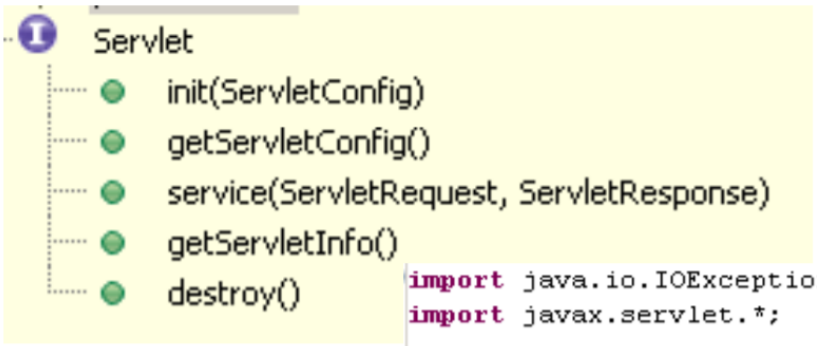
- Acepta peticiones de cualquier método HTTP (get, post, put, delete, head, trace, ...)
- Responde también usando el protocolo HTTP
- Se ejecuta dentro de un contenedor de Servlets que a su vez está dentro de un servidor de aplicaciones JEE



Contenedor de servlets/Web container

- Un contenedor define un ambiente estandarizado de ejecución que provee servicios específicos a los servlets. Por ejemplo, dan servicio a las peticiones de los clientes, realizando un procesamiento y devolviendo el resultado
- Los servlets tienen que cumplir un contrato con el contenedor para obtener sus servicios. Los contratos son interfaces Java.Por ejemplo, la interfaz Servlet.

Servlet API

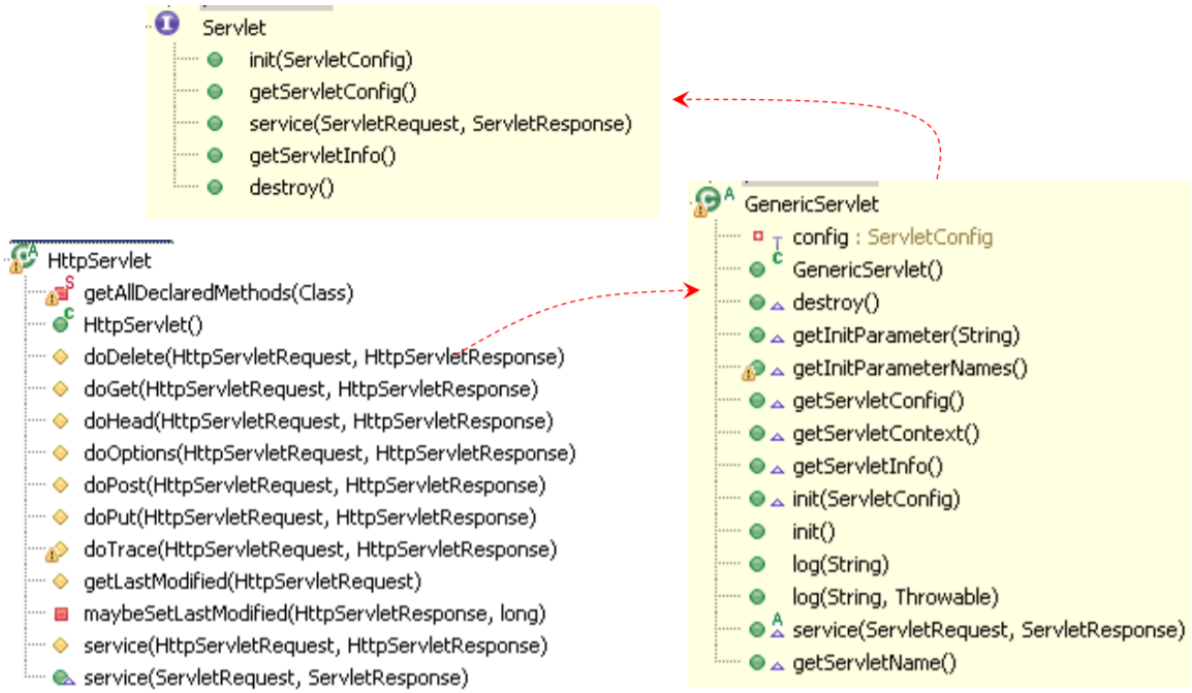



```
import java.io.IOException;
import javax.servlet.*;

public class MyServlet extends GenericServlet {

    public void service (
        ServletRequest   request,
        ServletResponse  response)
        throws ServletException, IOException {
        //...
    }
```

Métodos y herencia en Servlet



Ciclo de vida

- **INICIALIZACIÓN**: Una única llamada al metodo “init” por parte del contenedor de servlets public void init(ServletConfig config) throws ServletException. Se pueden recoger unos parametros concretos con “getInitParameter” de “ServletConfig”. Estos parámetros se especifican en el descriptor de despliegue de la aplicación: web.xml
- **PETICIONES**: Primera petición a init se ejecuta en un thread que invoca a service. El resto de peticiones se invocan en un nuevo hilo mapeado sobre service
- **DESTRUCCIÓN**: Cuando todas las llamadas desde el cliente cesen o un temporizador del servidor así lo indique. Se deben liberar recursos retenidos desde init() public void destroy()

Políticas de acceso concurrente (threading)

Los servlets están diseñados para soportar múltiples accesos simultáneos por defecto. ¡Ojo! El problema puede surgir cuando se hace uso de un recurso compartido.

Métodos doGet y doPost

Son llamados desde el método **service**(). Reciben interfaces instanciada:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, java.io.IOException {
. . .
}
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, java.io.IOException {
. . .
}
```

Registro de un Servlet

Dos opciones:

Opción1: Registro de un Servlet en el descriptor de despliegue web.xml

Insertamos en web.xml la declaración del servlet y del servletmapping:

```
<servlet>
    <servlet-name>HolaMundo</servlet-name>
    <servlet-class>uo.sdi.servlet.HolaMundoServlet</servlet-class>
</servlet>
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>HolaMundo</servlet-name>
    <url-pattern>/HolaMundoCordial</url-pattern>
</servlet-mapping>
```

Opción2: Registro de un Servlet usando anotaciones (@WebServlet)

Insertamos antes de la clase **Servlet** la anotación con el nombre del Servelt y el URLde mapeo:

```
@WebServlet(name = "HolaMundo", urlPatterns = { "/HolaMundoCordial" })
public class HolaMundoServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    /**
    * @see HttpServlet#HttpServlet()
    */
    public HolaMundoServlet() {
        super();
        // TODO Auto-generated constructor stub
}
```

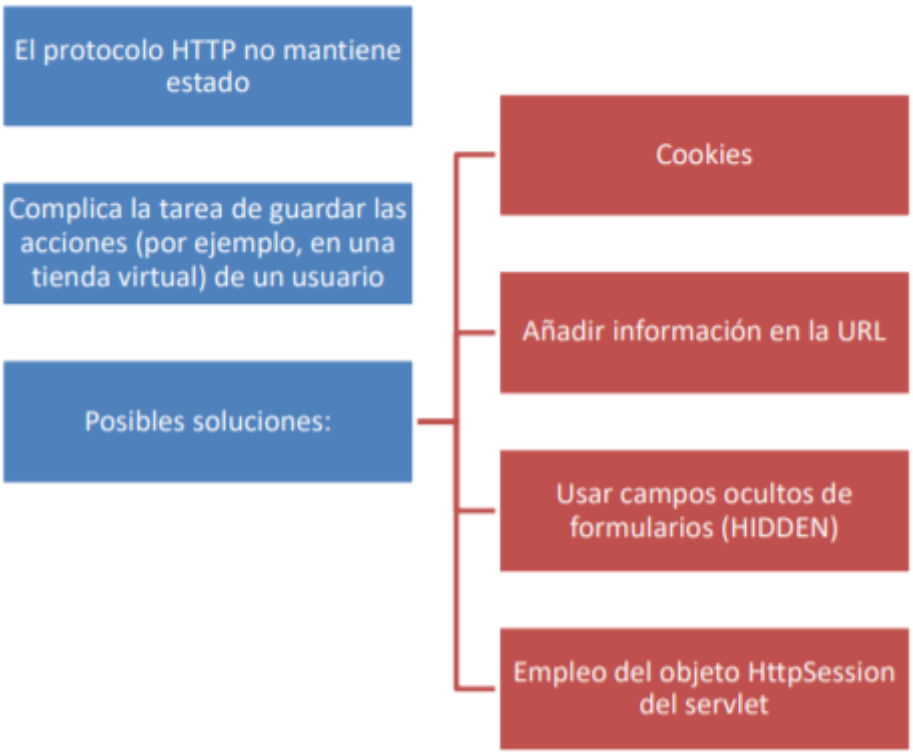
HttpServletRequest: Recogiendo información de usuario

Los parámetros nos llegan en la request, que es tipo "**HttpServletRequest**" con el método **getParameter**(). Estos parámetros on los parámetros de la QueryString o de los campos del formulario.

Devuelve:

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

Gestión de la Sesión. Mantenimiento del estado de la sesión.



Seguimiento de sesión

Los servlets proporcionan una solución técnica – La API HttpSession. Es una interfaz de alto nivel construida sobre las cookies y la reescritura de URLs, pero transparente para el desarrollador. Permite almacenar objetos

Para buscar el objeto HttpSession asociado a una petición HTTP: Se usa el método "getSession()" de "HttpServletRequest" que devuelve null si o 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 a la etiqueta nombre_variable o null en su defecto.
- **setAttribute**("nombre_variable", referencia). Coloca el objeto referenciado por referencia en la sesión del usuario bajo el nombre nombre_variable. A partir de este momento, el objeto puede ser recuperado por este mismo usuario en sucesivas peticiones.
- **getAttributesNames**() • Retorna una 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 (navegador) nunca ha visto la sesión. False para sesión preexistente

- **getCreationTime**(): Devuelve la hora, en milisegundos desde 1970, en la que se creo la sesión
- **getLastAccessedTime**(): La hora en que la sesión fue enviada por última vez al cliente

Caducidad de la sesión : Peculiaridad de las aplicaciones web: no sabemos cuándo se desconecta el usuario del servidor. Automáticamente el servidor web invalida la sesión tras un periodo de tiempo (p.e., 30') sin peticiones o manualmente usando el método "invalidate". Los elementos almacenados no se liberan hasta que no salta el timeout o session.invalidate().

Contexto de la aplicación

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

Para colocar o recuperar objetos del contexto

- Añadir un atributo
    - Se usa el método "setAttribute" de "ServletContext"
    - El control sobre varios servlets manipulando un mismo atributo es responsabilidad del desarrollador
- Recuperar un atributo
    - Se usa el método "getAttribute" de "ServletContext"
    - Hay que convertir el objeto que devuelve al tipo requerido (retorna un Object)

Qué es JSP

Una tecnología para crear páginas Web dinámicas. Contienen código HTML normal junto a elementos especiales de JSP. Estas están construidas sobre servlets.

Beneficio

Incluir mucha lógica de programación en una página Web no es mucho mejor que generar el HTML por programa. Pero JSP proporciona acciones (action elements) que son como etiquetas HTML pero que representan código reutilizable. Además, se puede invocar a otras clases Java del servidor, a componentes (Javabeans o EJB).

Separación de presentación y lógica

En definitiva, lo que permite JSP (bien utilizado) es una mayor separación entre la presentación de la página y la lógica de la aplicación, que iría aparte.

Elementos JSP

Tres tipos de elementos en JSP:

- Scripting • Permiten insertar código java que será ejecutado en el momento de la petición.
- Directivas
    - Permiten especificar información acerca de la página que permanece constante para todas las peticiones
        - Requisitos de buffering
        - Página de error para redirección, etc.
    - Acciones

    - Permiten ejecutar determinadas acciones sobre información que se requiere en el momento de la petición de la JSP
        - Acciones estándar
        - Acciones propietarias (Tag libs)

Elementos de “scripting”

- `<% ... %>` **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
- `<%-- ... --%>` **Comentario**. Comentario ignorado cuando se traduce la página JSP en un servlet. (comentario en el HTML)

Objetos predefinidos

El código java incrustado en JSP tiene acceso a los mismos objetos predefinidos que tenían los servlets:

- request
- response
- pageContext
- session
- application
- out
- config
- page

Directivas

Las directivas son mensajes para el contenedor de JSP:

- `<%@ page ... %>` Permite importar clases Java, especificar el tipo de la respuesta (“text/html” por omisión), etc.
- `<%@ include ... %>` Permite incluir otros ficheros antes de que la página sea traducida a un servlet
- `<%@ taglib ... %>` Declara una biblioteca de etiquetas con acciones personalizadas para ser utilizadas en la página

Acciones

Usan construcciones de sintaxis XML para controlar el comportamiento del motor de servlets. Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, etc.

Tipos: Estándar, A medida y JSTL

- `<jsp:useBean>` Permite usar un JavaBean desde la página
- `<jsp:getProperty>` Obtiene el valor de una propiedad de un componente JavaBean y lo añade a la respuesta
- `<jsp:setProperty>` Establece el valor de una propiedad de un JavaBean
- `<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 o
- `<jsp:plugin>` Genera el HTML necesario para ejecutar un applet

Acción useBean

Permite cargar y utilizar un JavaBean en la página JSP y así utilizar la reusabilidad de las clases Java:

```
<jsp:useBean id="name" class="package.class" />
```

Ahora podemos modificar sus propiedades mediante `<jsp:setProperty>`, o usando un scriptlet y llamando a un método del objeto referenciado por id

- **id**: Da un nombre a la variable que referenciará el bean. Se usará un objeto bean anterior en lugar de instanciar uno nuevo si se puede encontrar uno con el mismo id y ámbito (scope)
- **class**: Designa el nombre cualificado completo del bean
- **scope**: Indica el contexto en el que el bean debería estar disponible. Hay cuatro posibles valores: page, request, session, y application
- **type**: Especifica el tipo de la variable a la que se referirá el objeto
- **beanName**: Da el nombre del bean, como lo suministraríamos en el método instantiate de Beans. Está permitido suministrar un type y un beanName, y omitir el atributo class

Encadenamiento de Servlets/JSPs

Desde un Servlet concatenar otro Servlet: `RequestDispatcher/forward/include`

Desde un JSP: `<jsp:forward>` o `<jsp:include>`

Cómo reenviar peticiones

Para reenviar peticiones/incluir un contenido externo se emplea: `RequestDispatcherR getServletContext().getRequestDispatcher(URL)`

```
String url = “/presentaciones/presentacion1.jsp”;
RequestDispatcher
despachador=getServletContext().getRequestDispatcher(url);
//Para pasar el control total usar forward
despachador.forward(request, response);
//forward genera las excepciones ServletException y IOException
```

Paso de información procesada a Servlet/JSP

- Criterio: El servlet procesará los datos y se los pasará a la/s página/s JSP cuando:
    - Los datos son complejos de procesar (el servlet es más adecuado para ello).
    - Son varias las páginas JSP las que pueden recibir los mismos datos (el servlet los procesa de forma más eficiente).
- Formas de pasar datos a un JSP desde un Servlet:
    - Almacenándolos en HttpServletRequest.
    - En el URL de la página objetivo.
    - Pasándolos con un Bean.
- Paso de datos en HttpServletRequest
    - En el servlet: request.setAttribute(“clave1”, valor1);

    - En JSP: request.getAttribute(“clave1”);
- Pasar datos en el URL
    - En el servlet: `address =“/ruta/recurso.jsp?clave1=valor1” RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(address); dispatcher.forward(request, response);`
    - En JSP: el nuevo parámetro se agrega al principio de los datos consultados. `request.getParameter(“clave1”);`

Cómo interpretar los URLs relativos en la página objetivo

- Un servlet puede reenviar peticiones a lugares arbitrarios en mismo servidor mediante forward(). Diferencias con sendRedirect():
    - sendRedirect ()
        - Necesita que el cliente se vuelva a conectar al nuevo recurso
        - No conserva todos los datos de la conexión
        - Trae consigo un URL final distinto
    - forward()
        - Se maneja internamente en servidor
        - Conserva los datos de la conexión
        - Conserva el URL relativo del servlet.

Cómo incluir contenido estático o dinámico

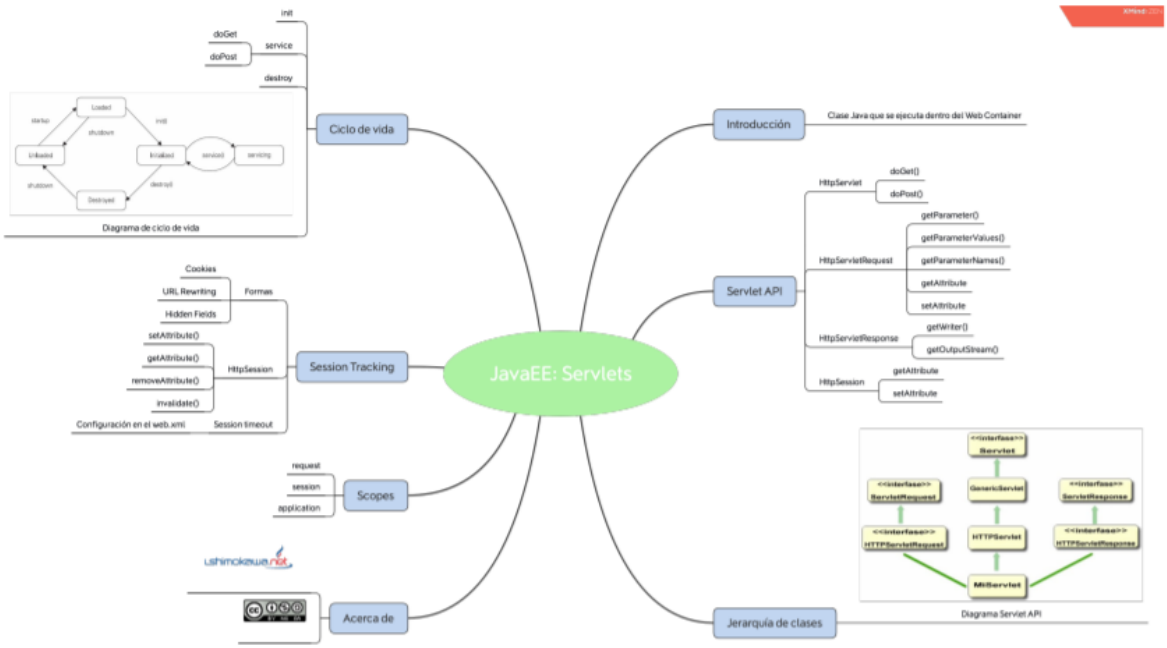
Si un servlet usa el método RequestDispatcher. forward() no podrá enviar ningún resultado al cliente, tendrá que dejarlo todo a la página objetivo. Para mezclar resultados del propio servlet con la página JSP o HTML se deberá emplear RequestDispatcher.include():

Las diferencias con una redirección consiste en:

- Se pueden enviar datos al navegador antes de hacer la llamada. (out.println(“…”);)
- El control se devuelve al servlet cuando finaliza include.
- Las páginas JSP, servlets, HTML incluidas por el servlet no deben establecer encabezados HTTP Respuesta

include() se comporta igual que forward propagando toda la información de la petición original (GET o POST, parámetros de formulario, …) y además establece.

MindMap JEE



Introducción a patrones

Patrones

Un patrón es la repetición de técnicas y las mejores prácticas que funcionan en cualquier dominio.

1. **Arquitectónicos**: Relacionados con el diseño a gran escala: Patrón de Capas.
2. **Diseño**: Relacionados con el diseños de frameworks y objetos a pequeña escala: Patrón Fachada.
3. **Estilos**: Soluciones a bajo nivel orientadas a la implementación o al lenguaje: Patrón Singleton.

Los patrones arquitectónicos se clasifican en:

1. Domain Logic Pattern
2. Mapping to Relational Databases
3. WebPresentation Patterns: MVC, PageController

Los patrones de diseño se clasifican en:

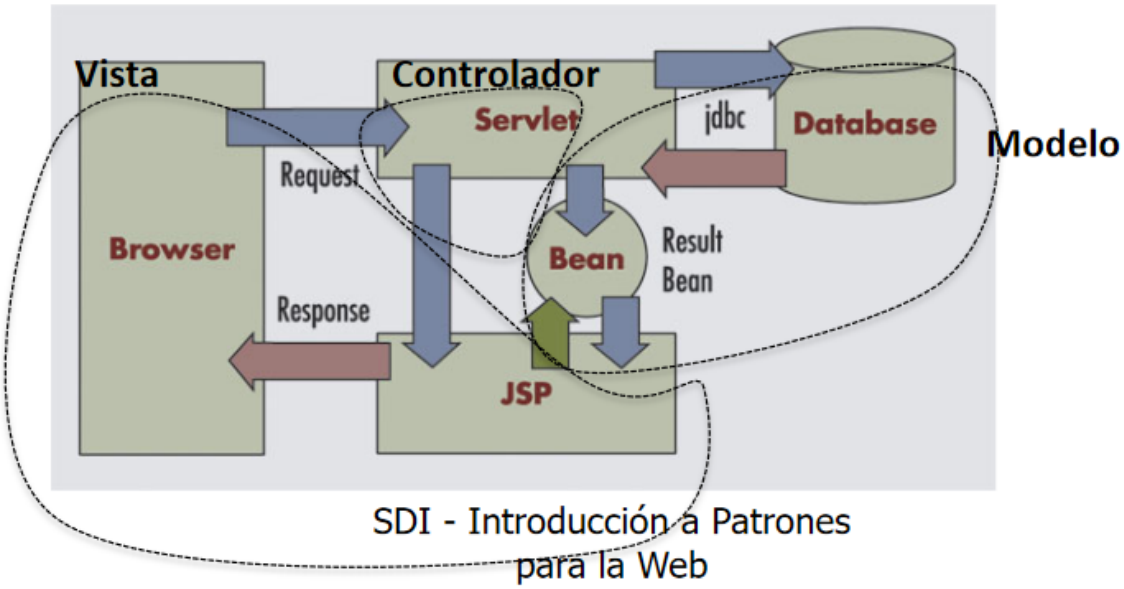
1. Creacionales (Factory, Prototype,..)
2. Estructurales(Facade, Adapter,..)
3. Compotamiento(Command, Interpreter)

Modelos de desarrolo de Aplicaciones Web en JEE

Patrón MVC

Roles del patrón MVC

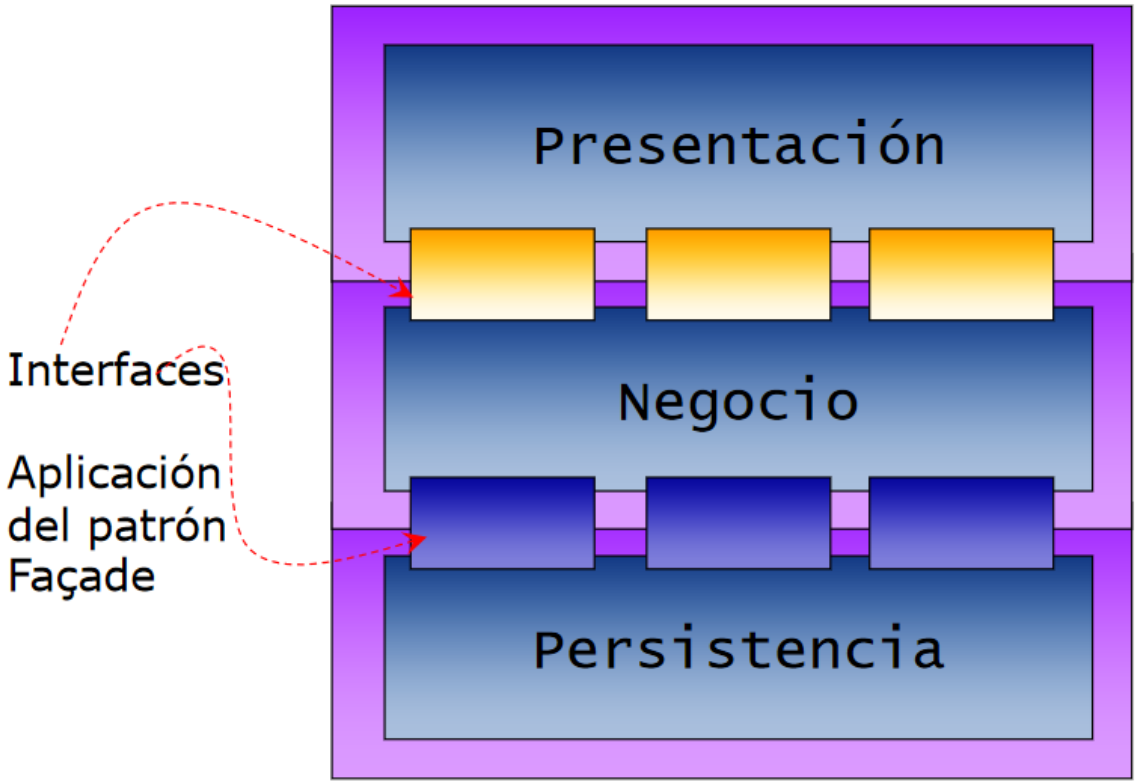
1. Controlador: Navegación-> Servlet
2. Modelo (Negocio y Datos)->Servlet/Beans
3. Presentaciñon: JSPs



Patrón N-Capas

Divididas entre Presentación, Negocio y Persistencia; distribuyen las responsabilidades del software. Entre ellas se implementan interfaces que hacen de Fachada de cada una de las capas.

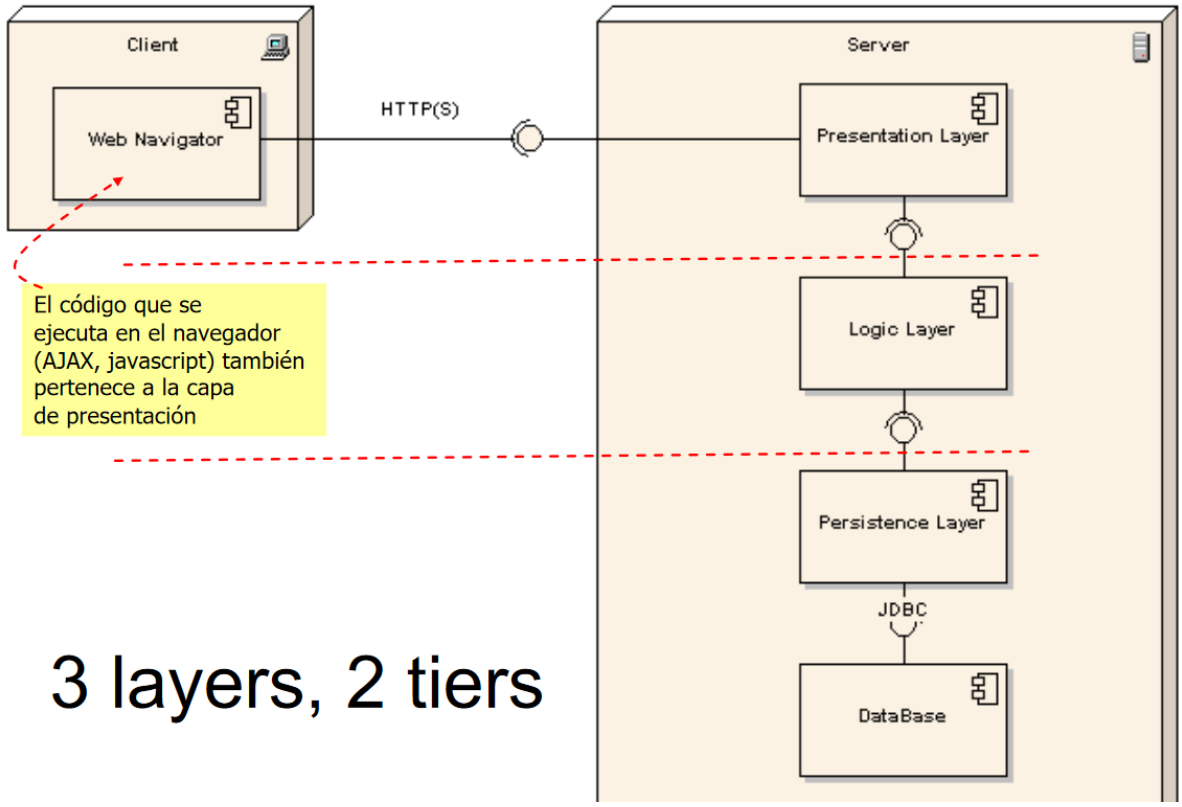
1. **Prensentación**: JSPs, HTML y lógica de presentación
2. **Lógica de negocio**, procesos neg.
3. **Persistencia**: Componente de acceso a datos.



Layers y Tiers

- **Layer**: Capa arquitectónica de la aplicación software: Presentación, Negocio y Lógica.
- **Tier**: Capa física de la arquitectura de despliegue de la aplicación software

Las Layers son desplegadas sobre las Tiers.



Client es la capa Tier y Server contiene la capa Layer. Cada división es la separación entre una de las capas.

Arquitectura en capas: Patrones

Presentación	Negocio	Persistencia
MVC	Fachada Factoría	DAO DTO Factoría Active Record

Patrón Fachada

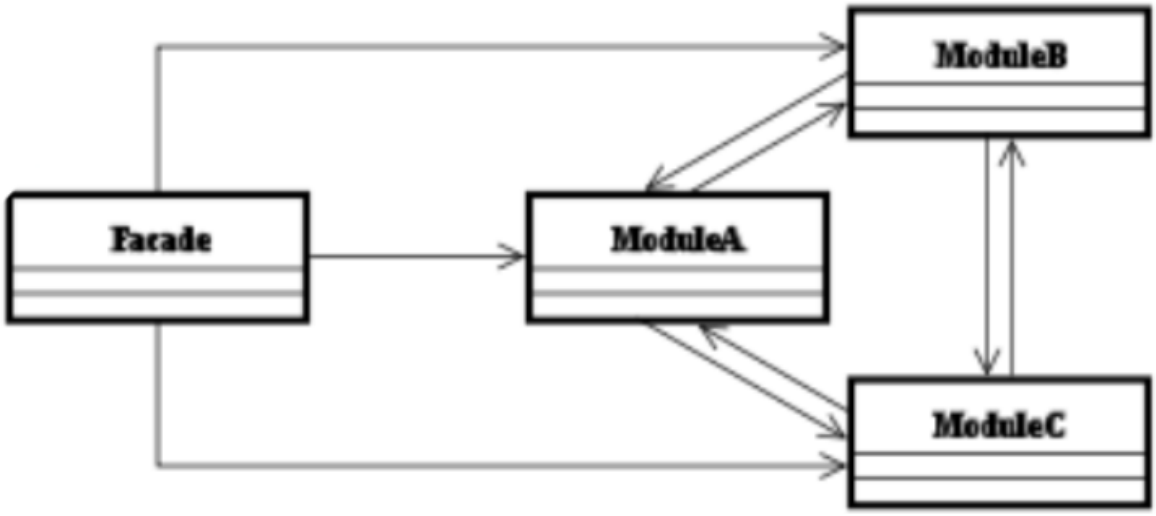
Definición de Interfaz único y simplificado de los servicios más generales de un subsistema

Casos de Uso:

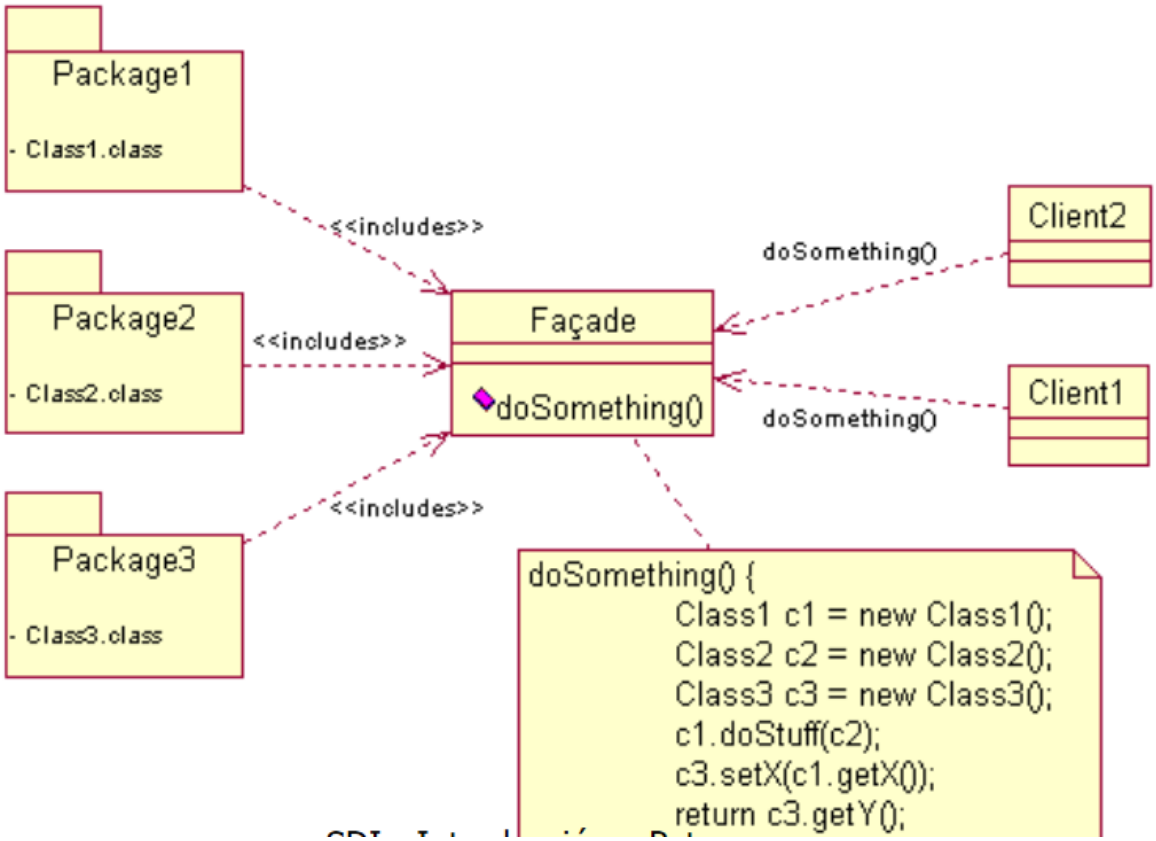
1. Se busca una Interfaz simple para un sistema complejo.
2. Hay muchas dependencias entre clientes y clases que implementan una abstracción.
3. Se desea obtener una división de capas de nuestros subsistemas

Como se Usa:

- Reducción del acoplamiento cliente subsistema (alternativa a la herencia).
- Clases de subsistema públicas o privadas. (No todos los lenguajes lo soportan)

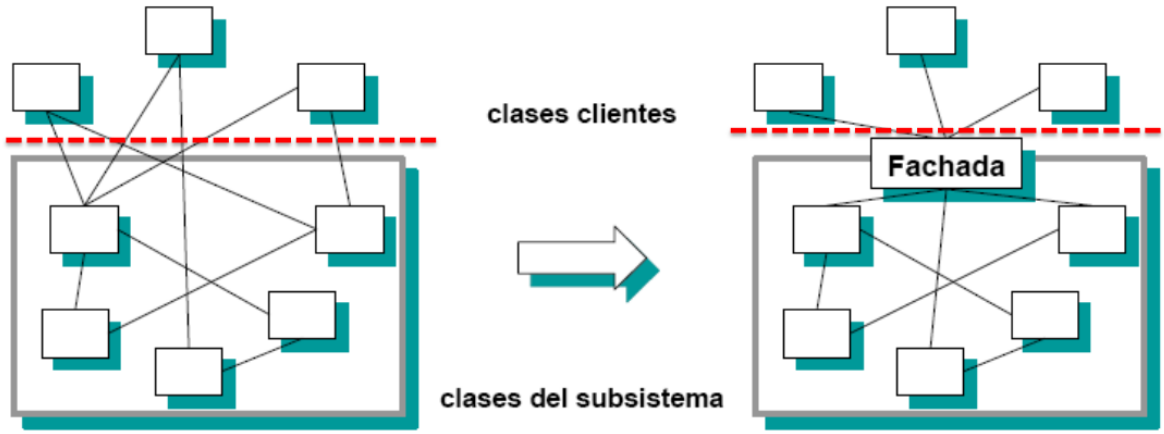


Capa lógica (facade)



Client1 y Client2 implementan Fachada, lo que permite trabajar en el mismo ámbito (doSomething()) de distintas maneras con las distintas clases.

Desacoplamiento de capas



Las interacciones de las elementos de la derecha están entrelazadas, lo que provoca una sostenibilidad del sistema baja. Para ello refactorizamos la estrucutra y hacemos que los distintos agentes del sistema se comuniquen a través de Fachada.

Capa de negocio (Factory)

Una factoría es un objeto encargado de la creación de otros objetos.

Se utiliza en casos en los que hacerse con un objeto implica más complejidad que crearlo sin más.

- Crear la clase del objeto dinámicamente
- Obtenerlo de un Pool de objetos
- Realizar una configuración compleja del mismo

- ETC

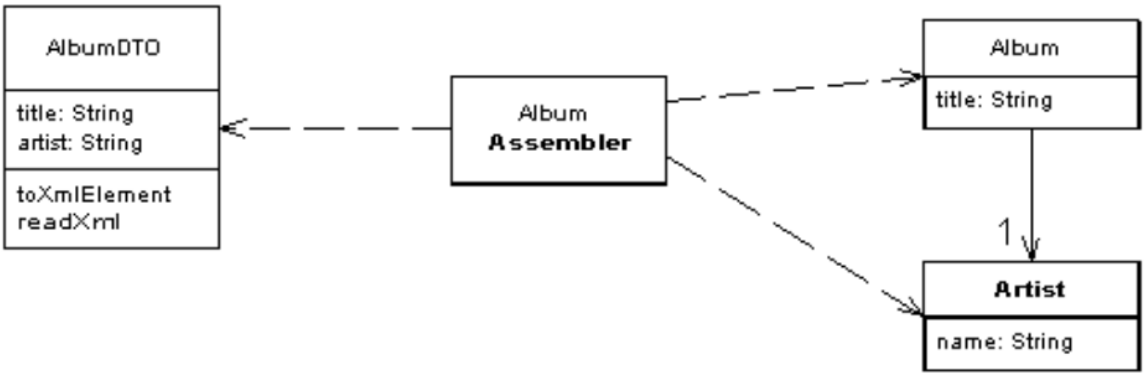
Hay una capa abstracción entre el cliente y el objeto que está utilizando, puesto que el cliente solo conoce la interfaz del objeto que está utilizando.

Patrón Data Transfer Object (DTO, Capa de persistencia)

Utilizado en sistemas para transferir datos entre ellos

- Reducir el numero de llamadas a un método.
- Su único comportamiento lo constituyen los getters y setters del mismo.

Se utilizan a menudo en combinación con objetos DAO (persistencia) para obtener los datos de una base de datos.



DAO (Data Acces Object)

Proporciona una interfaz única de acceso a los datos, de forma independiente a donde estén almacenados.

Independiza (Separa) la lógica de negocio del acceso a datos.

Ofrece operaciones CRUD (create, read, update, delete) para cada objeto persistente en el dominio

```
public interface GenericDao<T> {
    void save(T t);
    T update(T t);
    void delete(T t);

    T findById(Long id);
    Collection<T> findAll();
}
```

Métodos CRUD básicos

Métodos CRUD específicos para cada entidad del modelo

```
public interface StudentDao extends GenericDao<Student>{
    Collection<Student> findByNameSurname(String name, String surname);
    Collection<Student> findByEnrollmentId(Long enrollmentId);
    Collection<Student> findMatesByNameSurnameCourse(String name,
            String surname, String course);
    Set<Student> findUnenrolledAfterDate(Date date);
}
```

Posibles alternativas a la Implementación de JEE (Persistencia)

- Clases java manejando SQL (DAO con JDBC).
- Framework de persistencia de mapeo (JPA).
- Conjunto de conectores a otros sistemas BackEnd.
- La combinación de las anteriormente citadas tambien es una alternativa.
- Generación de código JDBC.

Repaso general de la web

Introducción a SD

Un SD es aquel en el que los componentes hardware y software ubicados en los computadores en red se comunican y coordinan sus acciones intercambiando mensajes

Consecuencias derivadas de la definición:

- Concurrencia
- Inexistencia de un reloj global (coordinación)
- Fallos independientes

Un SD se construye principalmente para compartir recursos y para alojar aplicaciones inherentemente distribuidas (de su propia naturaleza). Internet es el mejor ejemplo de un SD

Paradigmas de la computación distribuida

- Paso de mensajes (sockets emisor/receptor)
- Cliente/servidor (FTP/HTTP, roles marcados)
- Igual a Igual (mismos roles, Mens. Inst., Transfearchivos...)
- MOM (Message-Oriented-Middleware). JMS, MQS, MSMQ.
- RPC (RemoteProcedureCalling)
- Invocación métodos remotos (RMI, CORBA

Todas las implementaciones de estos paradigmas son cerradas (conllevan dificultad de añadir servicios al no tener una definición pública), utilizaban protocolos propietarios para el intercambio de información y dificultan la implementación en entornos heterogéneos.

Esto se soluciona utilizando protocolos y especificaciones abiertas, como puede ser HTTP

WWW

Sistema de distribución de información basado en hipertexto o hipermedios enlazados y accesibles a través de Internet. Funciona, por tanto, sobre internet.

Cuatro componentes básicos:

- HTTp
- HTML
- Un servidor web
- Un navegador web

HTTP

Ejemplo de diálogo HTTP

- Petición

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: nombre-cliente
[Línea en blanco]`
```

- Contestación

```
HTTP/1.1 200 OK
 Date: Fri, 31 Dec2003 23:59:59 GMT
 Content-Type: text/html
Content-Length: 1221
<html>
<body>
 <h1>Página principal de tu host</h1>
(Contenido) . . .
</body>
</html>
```

Métodos de petición

```
HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
```

Códigos de respuesta

- 1xx: respuestas informativas
- 2xx: petición correcta
- 3xx: redirecciones
- 4xx: errores del cliente
- 5xx: errores del servidor

URL

Es una secuencia de caracteres que sigue un formato estándar y se utiliza para nombrar recursos en Internet para su localización o identificación. Formato:

`esquema://máquina:puerto/directorio/archivo`

Cadena de consulta

Las url pueden incluir una cadena de consulta, utilizadas para refinar (especificar) la consulta sobre una información en concreto. Se separa de la url con ? y tienen formato:

`Parametro= valor`

`http://directo.uniovi.es/catalogo/DetalleCentroDpto.asp**?departamento=34&centro=66**`

Servidores Web

Se refiere al hardware (ordenador) o software (programa) que ayuda a entregar contenido web que puede ser accedido en Internet: Apache, Internet InformationServer, ...

Originalmente los servidores solo eran capaces de gestionar páginas estáticas. Hoy en día sabemos que las páginas pueden ser dinamicas y actualizables en tiempo real.

Mantenimiento de la sesión

HTTP es un protocolo sin estado: El servidor no mantiene información o estado sobre cada usuario a lo largo de múltiples peticiones.

Se necesitan pues alternativas de software que implemente esta funcionalidad.

SpringBoot 1

Spring

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

SpringBoot

Spring Bootno es un frameworkes una forma fácil de desarrollar aplicaciones Spring pero facilitando los aspectos duros de Spring (configuración, generación de código, servidor embebido, ...)

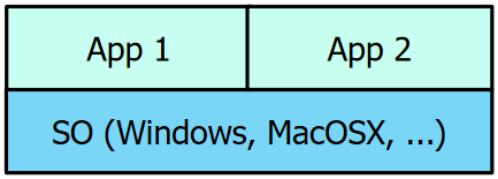
Conceptos

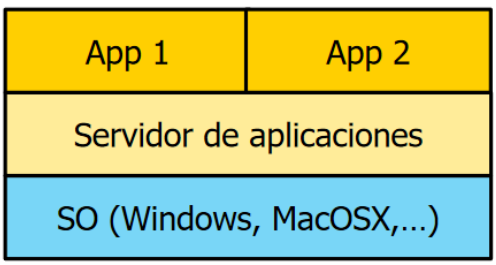
Framework Spring

- Aplicaciones basadas en el patrón MVC
- Soporte completo al desarrollo de aplicaciones de empresariales basadas en POJOs
- Sistema de inyección de dependencias basado en el IoC container(InversionofControl): Menor consumo de recursos que los EJB
- Gran cantidad de módulos con funcionalidad reutilizable
- Traducción de excepciones especificas a genéricas (EjJDBC, Hibernate, etc.)

Standalone

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





Maven: POM

El fichero POM.xml nos permite definir :

- Las propiedades de la aplicación.
- Configuración por defecto de SpringBoot
- Dependencias

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

- Intrucciones para construir el ejecutable (JAR)

Main

Las aplicaciones stand-alone deben definir una clase de inicio, que implementa un método main e invoca a `SpringApplication.run();` . Se recomienda utilizar la anotacion @SpringBootAplication en la clase principal.

Ejemplo

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

@SpringBootAplication

Esta anotacioón agrupa a otras tres:

- Configuration: indica que esta clase puede definir elementos de configuración
- EnableAutoConfiguration: habilita la autoconfiguración
- ComponentScan: se debe escanear la aplicación en busca de componentes implementados (controladores, servicios, repositorios, etc.)

Aplication.properties

Permite modificar las propiedades por defecto / definir nuevas, aunque eta función tambien se puede implementar en clases, anotaciones específicas y en application.yml

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:hsql://localhost:9001
spring.datasource.username=SA
spring.datasource.passwaord=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

Elementos principales

Todos los componentes se procesan internamente como Beans. Aquellos registrados contienen una instancia de un objeto. Pueden ser inyectados en cualquier parte de la aplicación.

Existe la declaración explícita @**Bean**, principalmente para crear manualmente una instancia de una clase y registrarla como Bean.

Componentes

- **Componente**: componente genérico sin un propósito especifico
- **Controladores**: definen las peticionesque van a ser recibidas por la aplicación. Suelen utilizar los serviciosy 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 negocion
    - Encriptación, seguridad, internacionalización, etc

Controladores, Servicios, Repositoriosy Configuración son todos estereotipos de Componente, añaden funcionalidad adicional (algunos), cada uno de ellos tiene un propósito claramente definido y pueden ser procesados por herramientas específicas.

Los componentes son instanciados y resgitrados como Beans.

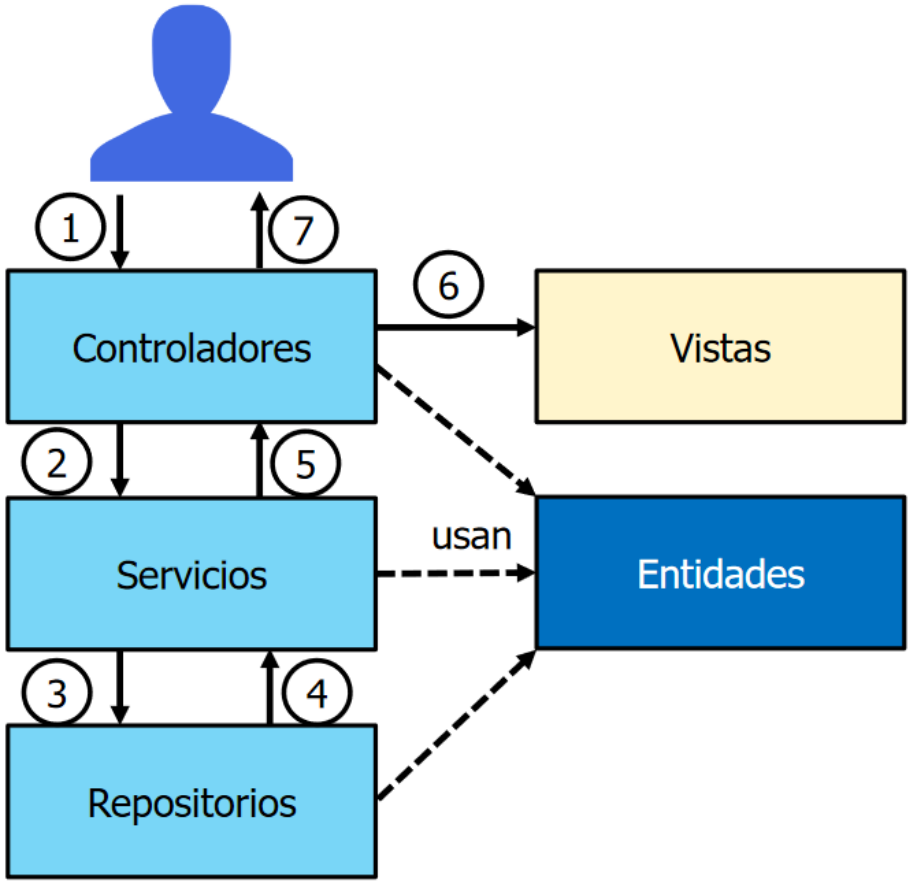
Entidades

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.

Ejemplo de una arquitectura



Controladores

@Controller indica que la clase es un componente de tipo Controlador: Recibe peticiones. Es un component, pero se recomienda utilizar esta etiqueta. Suele incluir etiquetas **@RequestMapping** en la definición de sus métodos.

En este componente se escanea y registra en el RequestMappingHandlerMapping de Spring. Cuando el DispatcherServlet recibe una petición busca la URL en el HandlerMapping. Si hay coincidencia con la petición HTTP delega al controlador, que devuelve una vista.

RequestMapping

Indica que debe responder a una petición especificada en anotación URL

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

Admite atributos:

- **value**: especifica la URL
- **method**: especifica el tipo de petición HTTP

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

Peticiones y parámetros

RequestParam

Las peticiones pueden contener parámetros. Para obtenerlos los incluimos en la dirección (URL con ?parámetro=valor) y colocamos @**RequestParam** delante del parámetro. Es válido para parámetros GET(URL) y POST(Cuerpo de la petición).

Debe especificar el tipo de dato (String, double,..). Si estos parámetros no se encuentran o bien no son del tipo especificado se generará una Excepción. Se puede cambiar esta propiedad, definida por defecto, añadiendo el parámetro `required=(true/false)`, así como dar valores por defecto con `value=(valor)`

`http://localhost:8090/coche/detalles?id=4`

```
@RequestMapping("/coche/detalles")
    public String getDetalles(@RequestParam(value ="coche1",required=false) Long
id){
        Stringfrase =" Detalles del coche : "+id;
   }
```

Ejemplo de una petición GET con parámetros
`http://localhost:8090/coche?id=4`

PathParam

Si la petición es de la forma : `http://localhost:8090/coche/audi/4/` la anotacion será @**PathVariable** y se deberá definir:

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

ModelAttribute

Esta anotación construye automáticamente un objeto en base a parámetros recibidos.

La clase definida como @**ModelAttribute** puede definir:

- Contructor 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, y el resto quedan sin valor.

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

Respuestas

Los @**Controller** retornan la ruta para localizar una plantilla. El motor de plantillas se encarga de generar la vista (PE: thymeleaf)

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

ResponseBody

@**ResponseBody** hace que la respuesta sea un objeto en lugar de una plantilla. Se pone en la definición del método

ResponseController

@**ResponseController** se define en la cabeza del controlador y añade de forma transparente @**ResponseBody** en todos lo métodos.

Thymeleaf

Permiten componer respuestas de forma dinámica y ágil, pensado para la definición de vistas. Pensando para los formatos HTML, XML,.. y aplicable a más formatos.

Soporta:

- Acceso a atributos del modelo (objetos enviados desde el controlador)
- Acceso a requesty otros elementos http
- Definición de lógica en la plantilla: iteraciones, condiciones, variables, etc.
- Spring WebFlox(eventos AJAX)

Se debe declarar la dependencia en POM.xml y las plantillas se almacenan por defecto en /templates.

Los controladores pueden enviar un modelo de datos a la plantilla. La plantilla tiene acceso a los atributos del modelo. El modelo contiene atributos, puestos mediante:

`model.addAttribute(clave,objeto);`

Acciones comunes con estos atributos:

- Insertar sus valores en el HTML
- Usarlos en estructuras de control y utilidades de Thymeleaf(condiciones, bucles, etc.)
- Insertar sus valores en el JavaScript

En la plantilla accedemos con `<p th:text="${tienda}"></p>` habiendo definido en el modelo el atributo: `model.addAttribute("tienda","Mi Tienda");` . Genera: `<p>Mi Tienda</p>`

Además:

- Se puede modificar cualquier atributo HTML
- Admite uso de literales, operaciones y utilidades
- Si el atributo es un objetose accede a sus datos y métodos con el operador ‘ . ’

Atributos en estructuras de control.

Recorre la lista creando los componentes de la vista

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

Permite expresiones condicionales:

```
<p th:if="${producto.nuevo== true}">Nuevo </p>
```

Los atributos del modelo pueden insertarse en JavaScript

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

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

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

En parámetros emdebidos en la URL `/detalles/334`

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

Incluir y reemplazar

Ofrece la posibilidad de componer plantillas a partir de otras para evitar repetir partes comunes (head, navbar, footer, listas, ...); lo que mejora la arquitectura y el mantenimiento

Otros

Permite acceder a objetos de la aplicación:

- #**locale** propiedades de localización

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

- #**httpServletRequestla** petición

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

- #**httpSessional** objeto sesión

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

La gran funcionalidad del core puede ser extendida con dependencias. PE, para aceder a los objetos de Spring-Security debemos importar la dependencia thymeleaf-extras-springsecurity4.

Servicios

Introducción

Los servicios son los componentes que contienen la lógica de negocio. Suelen ser utilizados desde los controladores o desde otros servicios. Son un estereotipo de un componente. Esto indica que pertenecen a la capa de servicios/ lógica de negocio.

Tienen un sistema de autoconfiguración y autodetección basado en Beans, son registrados al inciar la aplicación.

Autowired

La anotación @**Autowired** permite inyectar una dependencia, sin necesidad de ninguna configuración adicional. Se asocia a los atributos. Funciona como alternativa a instanciar manualmente el objeto. String instancia los componentes, y cuando los necesita los inyecta.

Funcionamiento Básico

Los servicios implementan método de lógica de negocio. Suelen acceder a los Repositorios para obtener los datos.

```
@Service
public class CochesService {
    @Autowired
    CochesRepository cochesRepository; //Inyección del Repositorio

    public List<Coche> getCoches(){
        List<Coche> coches = cochesRepository.findAll(); //Uso del Repositorio
        return coches;
    }
    public void agregarRevision (Long idCoche, String revision){
        Coche coche = cochesRepository.findAll(id); //Uso del Repositorio
        coche.agregarRevision(revision);
        ...
}
```

Y desde un controlador

```
@Controller publicc lass CocheController{
    @Autowired
    CochesService cochesService;//Inyección del servicio

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

Inyección

La inyección de dependencias es una forma de inversión de control. Esto provoca que despreocupemos y liberemos parte de la carga de trabajo del Desarrollador, simplificación de los test unitarios

Procesamiento en Spring

Spring tiene un contenedor de dependencias (TheIoCContainer-InversionofControl). Este se encarga de escanear el código en busca de los componentes y cuando una clase solicita el **Bean** lo inyecta.

PostConstruct

@**postConstruct** Permite especificar que un método se ejecutará una vez construido el componente.

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

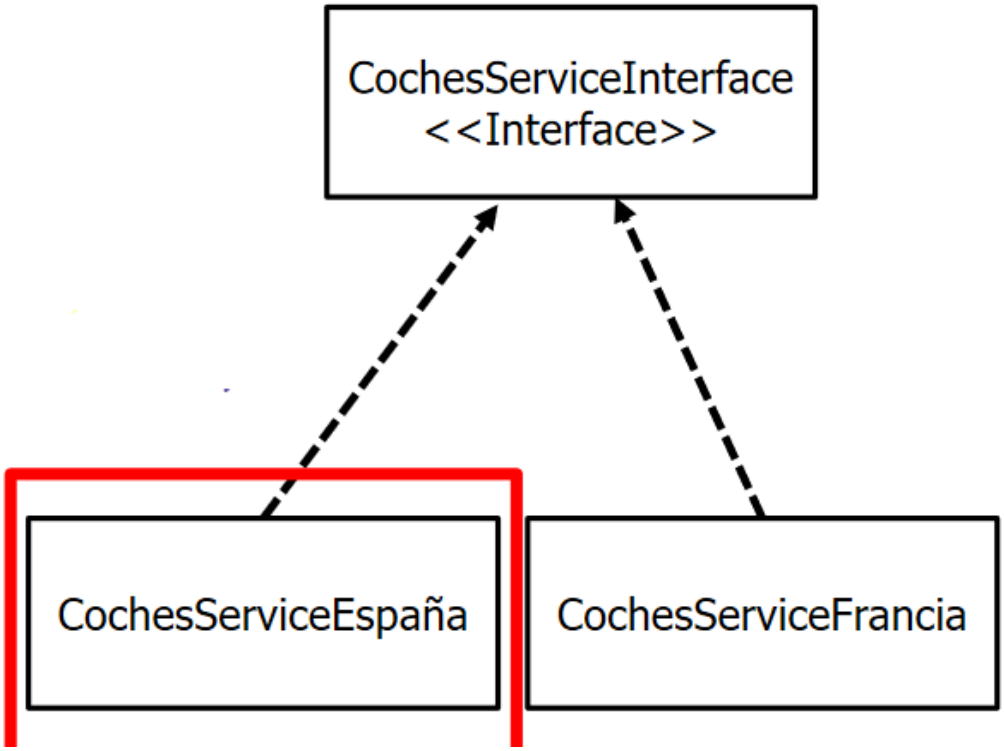
PreDestroy

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

Interfaces

Para inyectar una implementación de una Interfaz se combina @**Autowired** con @**Qualifier**

```
@Controller
public class CocheController{
    @Autowired
    @Qualifier("CochesServiceEspaña")
    CochesServiceInterfacecochesService;
    ...
}
```



Ámbito

Por defecto los servicios tienen un ámbito (Scope) singleton, es decir, la msima 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

PE: lógica que almacena las IDs de los productos agregados al carrito de la compra para cada sesión.

```
@SessionScope
@Service
public class CarritoService{...}
```

Repositorios

Introducción

Los repositorios son componentesque acceden a bases de datos. Estos suelen ser usados en la capa de servicios.

La anotación @**Repository** indica que una clase es un componente de tipo repositorio, y que por lo tanto pertenece a la capa de acceso a datos / persistencia. Tienen por defecto habilitado **PersistenceExceptionTranslationPostProcessor**, que traduce cualquier error en proceso de persistencia a **DataAccessException**.

El repositorio puede utilizar multitud de APIs/ librerías para acceder a las bases de datos, como puede ser JPA.

DataSource y JPA

Configuración del datasourcen en **application.properties**.

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

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

DataSource

Requiere la instalación del driver, el cual se instala mediante una dependencia en POM.xml. Dependiendo del driver, este puede ya estar incluido. Para el caso da arriba seria **org.hsqldb/ hsqldb**

Funcionamiento básico

Una vez configurado el datasource incluida la dependencia org.hsqldb/ hsqldb implementamos los componentes @**Repository**. Además de las operaciones CRUD (Incluidas en CrudRepository), podremos declarar nuevas operaciones dentro de esta.

Además podemos obtener los objetos haciendo búsquedas por sus atributos: find(All)By[Nombre Del Atributo]; así como consultas con métodos con consultas especificas @**Query** y lenguaje JPQL

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

Se inyectan en los servicios de la siguiente manera:

```
@Service
public class CochesService {

    @Autowired
    CochesRepository cochesRepository;//Inyección al Repositorio

    public List<Coche> getCoches(){
        List<Coche> coches = cochesRepository.findAll(); //Uso del Repositorio
        return coches;
    }
    public void agregarRevision (Long idCoche, String revision){
        Coche coche = cochesRepository.findAll(id); //Uso del Repositorio
        coche.agregarRevision(revision);
    }
    ...
}
```

Definición de Entidades

Anotaciones más comunes:

- @**Entity** indica que una clase es una idem
- @**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

PE: Definimos la entidad 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;
    }
    ...
}
```

SpringBoot 2

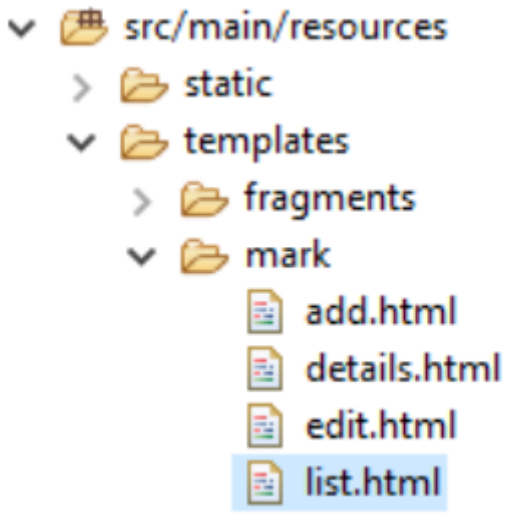
Fragmentos

Sí, otra vez

Introducción

El uso por defectode Thymeleaftconsiste en retornar una vista correspondiente a una plantilla.
PE: la plantilla mark/list es devuelta por getList()

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



Pero para ganar fluidez y eficiencia, a veces conviene devolver un fragmento/parte de una vista. Estos fragmentos se sustituyen/insetan en otra vista.

Th:fragment

El atributo th:fragment delimita un fragmento en una plantilla. Estos no tienen ni un mínimo ni un máximo de ocurrencias en una plantilla, la cantidad depende del enfoque y el desarrollo de la aplicación. Se le suele dar el id para referenciarlo desde JScript.

Controladores

Los controladores referencian los fragmentscon: `<ruta plantilla> :: <nombre fragmento>` .
PE: el siguiente controlador solo retorna el fragmento tableMarks:

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

Cliente

Además se suele invocar y cargar desde un script cliente (JScript).

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

Configuración

Introducción

SpringBoot incluye una configuración por defecto que puede ser modificada o ampliada. Aún así, la funcionalidad relativa a la configuración es genérica y no tiene que ver con la lógica de negocio.

Configuración

Las clases base de configuración del framework (@**Configuración**)se pueden utilizar de diferentes formas, lo más común.

PE: clase base WebSecurityConfigureAdapter.

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
    . . .
    Usar el objeto HttpSecurity para configurar las autorizaciones
    . . .
}
```

Bean

En muchos casos las clases de configuración instancian objetos como Beans “básicos”. Se declaran con la anotación @**Bean**. A diferencia de los componentes, se debe instanciar el objeto.

Pueden ser inyectados al igual que los componentes, con la anotación @**Autowired**

Definen:

- Funcionalidad necesaria para la propia configuración
- Funcionalidad común que será utilizada en otras partes de la aplicación

Internacionalización

Introducción

Consisten en adaptar una aplicación a diferentes idiomas o regiones, traduciendo todos los contenidos y estándares a los propios de estos. Tiene una gran influencia en el desarrollo de la aplicación.

Configuración

Incluir internacionalización requiere ampliar la configuración de la aplicación y añadir una clase de @**Configuration** que extienda de WebMvcConfigurerAdapter. Se configura agregando interceptores, los cuales procesan las peticiones antes de que lleguen al controlador.

Para implementar un Interceptor heredamos la clase de **HandlerInterceptorAdapter**, la configuramos y posterior mente agregamos ese interceptor a la clase que extienda de **WebMvcConfigurerAdapter**.

Interceptor

LocaleChangeInterceptor es un interceptor, activo en todas las URLs del sitio, implementado en el framework que permite definir un parámetro para realizar cambios de “Localización" . Si la petición contiene el parámetro se realiza el cambio de idioma.

Se crea un Bean con una instancia de LocaleChangeInterceptor, se sobrescribe el método addInterceptors y se registra el Bean Interceptor.

LocaleResolver es un objeto del framework permite hacer cambios automáticos de idioma. Para habilitar la funcionalidad se debe registrar una instancia de LocalResolver como Bean.

Mensajes

Las cadenas de texto internacionalizadas se definen en ficheros de propiedades. Cada cadena tiene una clave y un valor

```
welcome.message=Welcome to homepage
```

Se suelen usar varios ficheros de propiedades, uno por cada localización. Cada fichero define un valor para cada clave. Y desde thymeleaf usamos las claves para obtener los mensajes.

```
<div class="container" style="text-align: center">
    <h2 th:text="#{welcome.message}"></h2>
</div>
```

Cambio de idioma

Para cambiar de idioma se define un parámetro en la instancia de LocaleChangeInterceptor. El sistema selecciona los mensajes del fichero correspondiente al idioma actual. Y se define en la URL: para ?lang=en se usa el fichero “message_en”.

`http://miaplicacionweb.com/index?lang=en`

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

Spring Securtity

Autenticación

Es el proceso para validar la identidad de un usuario. Los procesos más comunes verifican sí existe coincidencia para un identificador único de usuario y una contraseña. El módulo **spring-boot-starter-security** incluye soporte para procesos de autenticación y autorización. Se debe añadir esa dependencia. Los datos a guardar son un identificador único, una contraseña y, aunque es opcional, muchas aplicaciones almacenan tambien un sistema de roles.

Encriptación

El servicio que guarda los usuarios debe encriptar la contraseña. Esto se puede realiza con el objeto **BCryptPasswordEncoder** de SpringSecurity. En concreto es la siguiente línea, que devuelve la contraseña codificada.

```
bCryptPasswordEncoder.encode(user.getPassword()));
```

El Bean de BCryptPasswordEncoder ser instanciado como tal, dado que no es un componente. Por lo tanto lo declararemos en la misma clase donde lo vayamos a usar.

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

Configuración

Se debe especificar la configuración de Spring Security, la clase debe ser hija de WebSecurityConfigurerAdapter

Incluimos las anotaciones de clase:

- @Configuration: clase de configuración
- @EnableWebSecurity: activa Spring Web Security en la aplicación

Autorización

Sobreescribimos el método configure(HttpSecurity http) , que recibe un objeto **HttpSecurity** como parámetro. Este nos permite configurar el sistema de autorización y otros aspectos de seguridad como:

- Definición de autorizaciones, URLS a las que pueden acceder disitntos roles de usuario o estados del mismo.
- Formulario de autenticación / login
- Sistema “logout”, dejar de estar autenticado
- Configuración de seguridad CSRF

La definicion de autorizaciones viene dada por la función raíz **authorizeRequests()** y dentro de esta se anidan otras funciones:

- **permitAll**() : cualquier petición puede acceder.
- **authenticated**(): cualquier usuasrio autenticado puede acceder.
- **hasAuthority**(“Nombres de Roles”): para acceder el usuario autenticado debe tener el Role especificado

Se basa en un orden de prioridad, las anidadas primero "son más significativas" .

FormLogin

FormLogin() no se incluye en **authorizeRequest**(). Los accesos que no hayan sido autorizado se redirijen a **FormLogin**(). Dentro de este se anidan:

- **loginPage**(“URL”): URL del formulario de login
- **Tipo de autenticación** (permitAll(), autenticated() ...)
- **defaultSucessUrl** (“URL”): URL que se carga después de la autenticación valida
- **failureUrl**(“URL"): URL de carga en caso de fallo en la autenticación.
- **failureHandler**(authenticationFailureHandler()): Manejador que captura el evento de fallo en la autenticación

El sistema **logout**() permite a los usuarios cerrar sesión, por defecto en la URL /logout. Aunque se puede especificar con **logoutSucessUrl**(“URL”) que redirige a la URL después de cerrar sesión.

CSRF

Configuración de seguridad CSRF por defecto activada. Para desactivarla se puede especificar `and().csrf().disable();` Previene de la falsificación de petición en sitios cruzados. Es un **Exploit** basado en que un cliente envía peticiones no intencionadas y de carácter malicioso a un sitio web en el que confía.

Se debe incluir tokens CSRF en: Login, logout, todos los formularios (otras acciones sensibles).

UserDetails

La información del usuario relativa a la autenticación se almacena en la base de datos de forma estándar, con un identificador único, una contraseña y un role. UserDetails es necesario dado que Spring Security no procesa usuarios de la lógica, sino objetos de este tipo.

UserDetailsService

Es el encargado de crear los **UserDetails**, sobreescribiendo la función `loadUserByUsername(username)` . Este obtiene el username y carga el objeto correspondiente del **Repositorio**. Después crea la colleción **GrantedAuthority** y después se crea el objeto UserDetails y se devuelve.

Implementación:

```
@Override
    public UserDetails loadUserByUsername(String dni) throws
UsernameNotFoundException{
    // 1 Usuario en la aplicación
    User user = usersRepository.findByDni(dni);
    // 2 Configurar Autoridades / EJ,TODOS EL ROLE_ESTUDIANTE
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
    grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_ESTUDIANTE"));
    // 3 Construir y retornar el "Userdetails"
    return new org.springframework.security.core.userdetails.User(
    user.getDni(), user.getPassword(), grantedAuthorities);
}
```

Controladores

Estos procesan consultas relativas a la autentificación. El esquema según Spring sería:

- GET /signup: mostrar la vista que solicita los datos de registro de un usuario
- POST /signup: registrar un usuario en la aplicación
- GET /login: mostrar la vista que solicita los datos de autenticación

No se define un POST de /login dado que ya está implementado en SpringSecurity.

SpringSecurity necesita usar un formulario de autenticación el cual se compone de : POST al /login, `input username` y `input password`.

Logica de negocio

El **SecurityContextHolder** da acceso al usuario autenticado desde toda la aplicación, que proporciona el ususario logeado para, por ejemplo, la lógica de negocio.

Thymeleaf

Permite, como en otras ocasiones, acceso a los objetos de Spring desde la plantillas. Utilizando `sec:authentication` podemos acceder al nombre del usuario autentificado:

```
<p>
Usuario
    <span sec:authentication="principal.username"></span>
</p>
```

Utilizando `sec:authorize` hace que, añadido a una etiqueta, esta se muestre si el usuario está auntenticado.

```
<li sec:authorize="!isAuthenticated()">
    <a href="/login" th:text="#{login.message}">
    Identifícate
    </a>
</li>
```

Fuerza bruta

Consiste en probar una gran cantidad de contraseñas para un usuario. Para evitarlo debemos detectar cuando un usuario se intenta identificar muchas veces in éxito. La implementeación de la prevención de la Fuerza bruta se puede implementar de forma manual o bien con Spring Security con componentes como **AuthenticationFailureBadCredentialsEvent**

Repositorios

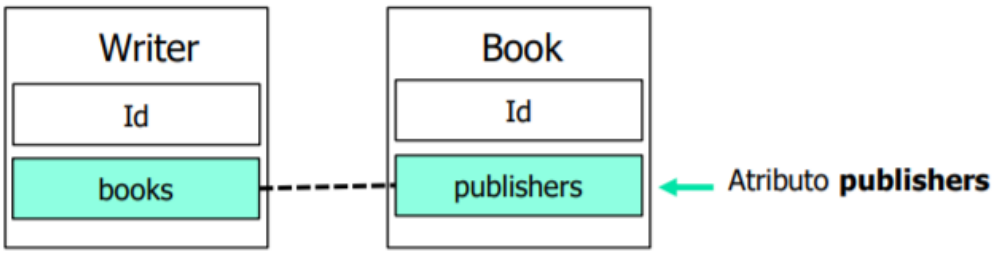
Mapeo de Entidades usando las anotaciones de java.persistence.

OneToMany y ManyToOne

@**OneToMany** indica que desde la entidad en la que se pone la etiqueta se van a mapeaer muchos objetos (0 o mas) de otra entidad. Atributo **mappedBy** = user indica que esta relación fue construida añadiendo un atributo user en la clase Destino. Atributo **cascade** = CascadeType.ALL, cuando la entidad es guardada, eliminada o actualizada, su entidad relacionada también lo es.

@**ManyToOne** indica que desde la entidad que se pone la etiqueta se va mapear un solo objeto. La anotación @JoinColumn, especifica que esta columna es una asociación de entidades; y esta permite especificar con un atrobuto name.

```
@ManyToMany(mappedBy = "publishers")
public Set<Book> books;
```



Y el esquema de datos sería:

Datos de Prueba

Definición de un servicio para insertar datos: por ejemplo creamos un servicio InsertSampleDataService, que nos insertará datos en las tablas de datos, para tener un sistema de pruebas consistente. Para ello definimos un método como @PostConstruct que se ejecutará después de crear la clase.

Validación de datos

Los datos introducidos por los usuarios deben ser validados. Consiste en comprobar que los datos son adecuados: formato, longitud, cumplimiento de reglas de lógica de negocio, etc. Se suele informar al usuario cuando introduce datos incorrectos. Hay dos tipos, validación en el cliente y validación en el servidor.

En la validación en el cliente los datos se validan antes de ser enviados al servidor, y acosutmbran a ser de formato, longitud,..

La validación en el servidor se realiza cuando se hace una petición al mismo.

Este se añade en el controlador correspondiente de la siguiente manera:

```
@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]);
    usersService.addUser(user);
    securityService.autoLogin(user.getDni(), user.getPasswordConfirm());
    return "redirect:home"; //Se accede la vista de operaciones
autorizadas/privadas
}
```

Validator

Esta interfz nos permite crear nuestros propios validadores. Se recomienda asignar un validar a cada formulario / proceso. Son gestionados por un **Bean** . Basta con sobreescribir la función validate y introducir en ella las validaciones que nos interesen.

```
@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 (usersService.getUserByDni(user.getDni()) != null) {
    errors.rejectValue("dni", "El DNI ya esta siendo usado");
}
```

Mostrar errores

Para mostrar errores en la vista utilizamos thmeleaf

```
<input type="text" class="form-control"
name="dni" placeholder="99999999Y" />
<span th:if="${#fields.hasErrors('dni')}"
th:errors="*{dni}">
```

ValidationUtils

Es la clase estatica de utilizdades de validación. PE:

```
@Override
public void validate(Object target, Errors errors) {
    User user = (User) target;
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "dni", "Error.empty");
}
```

SpringBoot 3

Sesión

Introducción

La aplicación crea una nueva sesión para cada nuevo cliente, es decir, cada navegador que accede genera una sesión. Esta esta identificada por un ID único, enviada del servidor al cliente en un cookie. Los datos de las sesiones son de caracter temporal, almacenados en la aplicación no en el cliente, y pueden ser recuperados por la misma; aunque se destruyen automáticamente tras x tiempo. También se pueden eliminar a través de código. Ejemplos de estos datos pueden ser: el carrito de la compra, los últimos artículos visitados, almacenaje del usuario autenticado.

HttpSession

Funciona como una tabla hash. Gestiona los atributos a través de una clave, asignada a un valor.

```
// set
httpSession.setAttribute("carrito",listaProductos);
// get
Set<Producto> listaProductos =
(Set<Producto>) httpSession.getAttribute("carrito");
```

Thymeleaf

Las plantillas thymeleaf también pueden acceder a la sesión, bien indirectamente o directamente.

Se dice que accede indirectamente cuando accede con los datos del modelo enviados desde el controlador a la plantilla. O bien directamente, utilizando objetos `{$session}` , pudiendo acceder a funciones : `${session.isEmpty()}` o a atributos : `${session.carrito}`

Beans

Los beans por defecto son gestionados a través del patrón singleton, comparten la misma instancia en todas partes.

Con la anotación @**SessionScope** creamos una sesión própia del ámbito de la aplicación en donde la hayamos especificado.

Datos y acciones sensibles

Introducción

Las aplicaciones web suelen ser multiusuario y de acceso publico. Esto multiplica la posibilidad de ser atacada y de encontrar vulnerabilidades. Uno de los peligros mas grande es la exposición de datos sensibles. Por ello hay que evitar que se pueda acceder a URL concretas,... es decir, proteger el acceso a datos.

Implementación

Se debe identificar qué usuarios pueden acceder a cada URL de la aplicación, e incluir las comprobaciones de acceso necesarias.

DecisionManager

Las URL también se pueden proteger por **Spring Security**, especialmente cuando las peticiones no pasan a través de ningún controlador. PE: la carpeta static. Como ya hemos dicho, **WebSecurityConfigurerAdapter** oferce diferentes políticas de acesso

AccessDecision

Las comprobaciones de acceso se realiza en una clase que implementa **AccessDecisionVoter** (Interfaz). Esta requiere sobrescribir los métodos **supports**(). El retorno de vote(authentication,filter,attributes) determina si habrá acceso con los siguientes códigos:

- **ACCESS_DENIED**: no permitido
- **ACCESS_GRANTED**: permitido
- **ACCESS_ABSTAIN**: abstención (útil en casos en los que hay más de un **AccessDecisionVoter**)

```
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;
        }
    }
}
```

Posteriormente incluiremos `accessDecisionManager(accessDecisionManager())` en el método configure() de WebSecurityConfig, además del siguiente método en la misma clase.

```
public AccessDecisionManager accessDecisionManager() {
    List<AccessDecisionVoter<? extends Object>> decisionVoters
    = Arrays.asList(new EjemploVoter(), new RoleVoter(), new
AuthenticatedVoter());
    return new UnanimousBased(decisionVoters);
}
```

Repositorios

Query

@**Query** permite incluir consultas propias (además de las CRUD por defecto) en un repositorio. Para incluir una nueva query en un repositorio debemos declarar la signatura del metodo y la anotación @Query justo encima:

```
@Query("SELECT n FROM Nota n WHERE n.descripcion LIKE ?1 AND n.usuario = ?2 ")
List<Nota> notaPorDescripcionYUsuario(String descripcion, Usuario usuario);
```

Paginación

Introducción

No se deben manejar colecciones con muchos recursos/entidades. Cargar muchos elementos en una vista es costoso (además que es muy posible que no todos esos elementos sean realmente consultados) y perjudica la experiencia de usuario.

Mucha aplicaciones usan un sistema de paginación, tanto en las vistas como en la lógica de negocio. Esta puede implementare bien manualmente o bien utilizando un framework. **Spring** incluye un clases para implementar mecanismos de paginación altamente configurables.

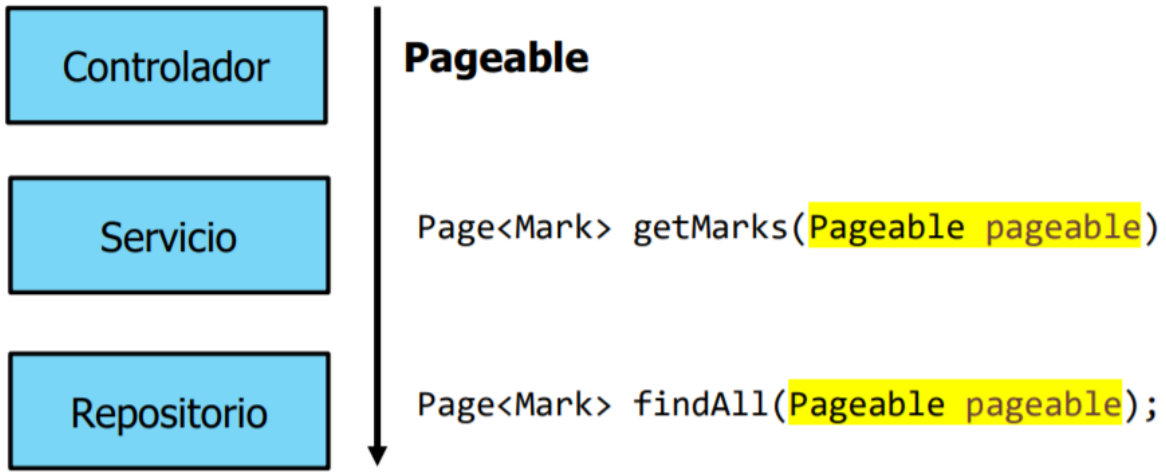
Page

Page es el objeto del sistema de paginación. Es una colección de datos similar a un List, pero además incluye datos sobre la paginación. Este uso de la paginación afecta a los **Repositorios**, ya todos los métodos que devuelvan colecciones; dado que se deberá devolver Page para la paginación.

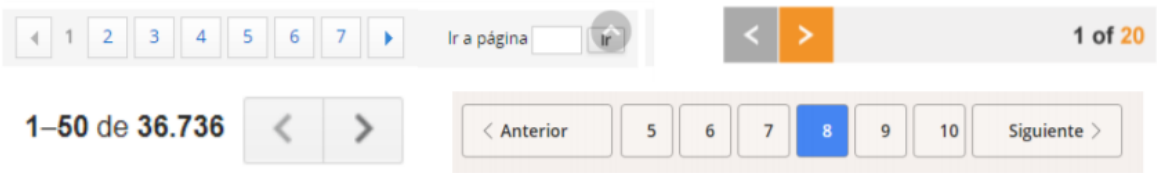
```
Page<Mark> findAllByUser(Pageable pageable, User user);
Page<Mark> findAll(Pageable pageable);
```

Se puede usar tambien en los **Controladores** y los **Servicios**. Es un buen enfoque utilizar Page en todas las capas, hasta llegar al controlador. El parámetro **Pageable** debe ser enviado desde los controladores hasta los **Repositorios**

La lista (interna) se puede obtener haciendo `.getContent()` al objeto Page.



Se recomienda incluir al menos: notificación clara de la página actual, acceso a las páginas cercanas y acceso a la primera y ultima página. En la vista debe ofrecer navegación por las página:



El sistema de navegación se implementa en un fragmento reusable en todos los listados con paginación de la aplicación. Solo se necesita que la vista reciba un objeto Page para crear un sistema de paginación.

Configuración

Es posible modificar la configuración de la paginación. Una modificación común es especificar otros valores por omisión a los parámetros page y size. Se debe hacer a partir de una clase **@Configuration**

```
@Configuration
public class CustomConfiguration extends WebMvcConfigurerAdapter{
    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
argumentResolvers){
        PageableHandlerMethodArgumentResolver resolver =
        new PageableHandlerMethodArgumentResolver();
        resolver.setFallbackPageable(new PageRequest(0, 5));
        argumentResolvers.add(resolver);
        super.addArgumentResolvers(argumentResolvers);
    }
}
```

Transacciones y Logging

Introducción

La anotación **@Transactional** declara el uso de transacciones en un método o componente. Los errores y excepciones pueden afectar a la consistencia de la aplicación, por tanto las operaciones de lógica pueden ejecutarse como transacciones potencialmente reversibles.

Ejemplo excepción e incosistencia:

```
public class TransferenciasService {
    @Autowired
    private CuentasRepository cuentasRepository;
    private boolean datoMemoria = false;
    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);
    }
}
```

Al incluirla en un **Componente** todos sus métodos son transaccionales, y en caso de excepción trata de realizar un “rollback” de las acciones relativas a los repositorios. Ejemplo:

```
@Transactional
public void transferencia(Long emisor, Long receptor,float cantidad){
    datoMemoria = true;// Los datos en memoria no vuelven a su estado anterior
!!
    cuentasRepository.reducirSaldo(emisor,cantidad);
    int a = 4 / 0; // Excepción!!! Producida. Rollback
    cuentasRepository.aumentarSaldo(receptor,cantidad);
}
```

El funcionamiento por defecto de @Transactional puede ser ampliamente configurado:
`@Transactional(rollbackFor = Exception.class)`

Logging

Introducción

Logging consiste en guardar información sobre eventos relativas a la aplicación: Errores, trazas, accesos, acciones de los usuarios, etc. Es útil para procesos de auditoria: registrar los accesos de los usuarios a recursos. Spring ofrece muchas dependencia para Loggin. PE: **org.slf4j.Logger** está integrado

Los objeto Logger se obtienen a través de factoria de Loggers.

Configuración

Spring y otros componentes como hybernate generan log. Este log queda registrado, almacenado en la raíz del proyecto y el nivel es configurable application.properties:

```
# Solo mensajes de error en Spring Web
logging.level.org.springframework.web = ERROR
# Solo mensajes de error en hibernate
logging.level.org.hibernate = ERROR
```

Subida de ficheros

Introducción

La subida de ficheros es un proceso común en unas aplicaciones web. 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>
```

Procesamiento

El controlador debe recibir el parámetro de tipo file (informe) ■ Se puede utilizar el tipo MultipartFile , su metodo isEmpty() indica si hay fichero o no.

```
@PostMapping("/informe")
public String subirInforme(@RequestParam("informe") MultipartFile informe) {...}
```

El controlador puede procesar el fichero de forma estándar (Java). El fichero se podría guardar en el servidor o en una base de datos.

Si lo almacenamos en el servidor, el path donde se guarda es: `src/main/static/*` y el fichero se puede renombrar para evitar problemas de conflictos.

Desde application.properties se modifica el tamaño máximo admitido:

```
spring.http.multipart.max-file-size=10MB
spring.http.multipart.max-request-size=10MB
```

Sellenium Testing

Qué es?

Es la prueba de una aplicación web para la detección de posible fallos antes de que sea desplegada en su entorno de producción. Test posibles:

- Test funcional
- Test de usabilidad
- Test de interface
- Test de compatibilidad
- Test de rendimiento
- Test de seguridad

Tipos de tests

- **Funcional**: Prueba de todos los enlaces web, conexiones a bases de datos, envío y recepción de datos de formularios, cookies, ...
- **Usabilidad**: Es el proceso mediante el que se miden las características de la interacción computador-humano, por lo que las debilidades de esta interacción deben identificarse para corregirse.
- **Interface**: Se refiere a las conexiones entre el servidor de aplicaciones y el servidor web, y el servidor de aplicaciones y el Servidor de base de datos.
- **Compatibilidad**: Navegadores, SSOO, Disp. Móviles e Impresión.

Características

- Rendimiento:
    - Pruebas de carga: un volumen de usuarios/conexiones, datos gestionados, conexiones a la BD, alta carga en páginas concretas
    - Pruebas de estrés: Exponer al sistema a valores límite de demanda de recursos y ver como responde. Los puntos críticos suelen ser campos de entrada, y areas de registro y

login.
- Seguridad:
    - Uso de URLs internos directos sin identificarse.
    - Acceder a URLs para otro rol diferente al que se está identificado.
    - Ver la reacción a valores incorrectos en los formularios de login.
    - Acceso a directorios de recursos de descarga sin acceder a los enlaces de descarga.
    - Test CAPTCHA para scripts de login automático.
    - Test SSL, aviso cuando se accede a un URL https desde URLs http (no segura) y viceversa.
    - Todas las transacciones, mensaje de error y avisos de seguridad deben quedar reflejados en los archivos de log.

Componentes de Selenium

- Selenium IDE. Extensión de Firefox que permite grabar, editar y depurar pruebas. Permite exportar las pruebas grabadas a código Selenase (nativo de Selenium) o bien código Java, Ruby, Python y C# basado en Selenium API Client.
- Selenium API Client. API para interactuar con Selenium desde código cliente.
    - Selenium 3 presenta una nueva API basada en WebDriver.
- Selenium WebDriver. Componente de Selenium 3 API Cliente. Controlador del navegador que permite enviar comandos al propio navegador para realice acciones como si de un usuario real se tratase.
- Selenium Grid. Servidor que permite usar instancias de navegador ejecutándose en máquinas remotas.

API Selenium

WebDriver: Es la clase de la API de Selenium que encapsula la interacción con el navegador.

```
WebDriver driver = FirefoxDriver();
driver.get("http://www.google.com");
```

Para localizar un elemento mediante Webdriver utilizamos findElement or findElements, que nos devilverá List .

Categorias de búsqueda:

- By.id
    - `WebElement element = driver.findElement(By.id("coolestWidgetEvah"));`
- By.className
    - `List cheeses = driver.findElements(By.className("cheese"));`
- By.tagName, By.name, By.LinkText, By.partialLinkText, By.cssSelector, By.xpath

Como Obtener el valor de un campo input

```
WebElement dni = driver.findElement(By.id("dni"));
dni.getText();
```

Como rellenar un campo input

```
WebElement dni = driver.findElement(By.id("dni"));
dni.click();
dni.clear();
dni.sendKeys(dnip);
```

Como clickar un botón Submit o un enlace

```
driver.findElement(By.id("submit")).click();
```

El método navigate: Ir a una página: `driver.navigate().to("http://www.example.com");` – Ir adelante y atrás: `driver.navigate().forward();` `driver.navigate().back();`

Esperas explícitas e implícitas

Con Selenium se combinan WebDriverWait con ExpectedCondition

Espera explícita: consiste en una espera por una condición antes de continuar la ejecución

```
WebDriver driver = new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement = (new WebDriverWait(driver,
10))
.until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicElement")));
```

Espera implícita: le dice a WebDriver que sondee el árbol DOM cada cierto tiempo para encontrar un elemento sino está disponible

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

Sellenium para SpringBoot

Se puede dividir en dos proyectos (uno para Selenium y otro para SpringBoot) o bien combinarlos en el mismo.

Pasos el diseño de pruebas con Selenium

- Ser sistemático etiquetando los atributos de las vistas, Ids, style, ... (analizar el proyecto Notaneitor final).
- Crear un PageObject por vista o conjuntos de vistas con misma interacción.
- Diseñar y definir los casos de test:
    - Casos válidos
    - Casos inválidos

Esquema de una suite de Pruebas

```
//Ordenamos las pruebas por el nombre del método
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class NotaneitorTests {
	//En Windows (Debe ser la versión 46.0 y desactivar las actualizacioens automáticas)):
	//String PathFirefox = "C:\\Path\\FirefoxPortable.exe";
	//En MACOSX (Debe ser la versión 46.0 y desactivar las actualizacioens automáticas):
	static String PathFirefox = "/Applications/Firefox.app/Contents/MacOS/firefox-bin";
	//Común a Windows y a MACOSX
	static WebDriver driver = getDriver(PathFirefox);
	static String URL = "http://localhost:8090";

	public static WebDriver getDriver(String PathFirefox) {
		//Firefox (Versión 46.0) sin geckodriver para Selenium 2.x.
		System.setProperty("webdriver.firefox.bin",PathFirefox);
		WebDriver driver = new FirefoxDriver();
		return driver;
	@Before
	public void setUp(){
		driver.navigate().to(URL);
	}
	@After
	public void tearDown(){
		driver.manage().deleteAllCookies();
	}
```

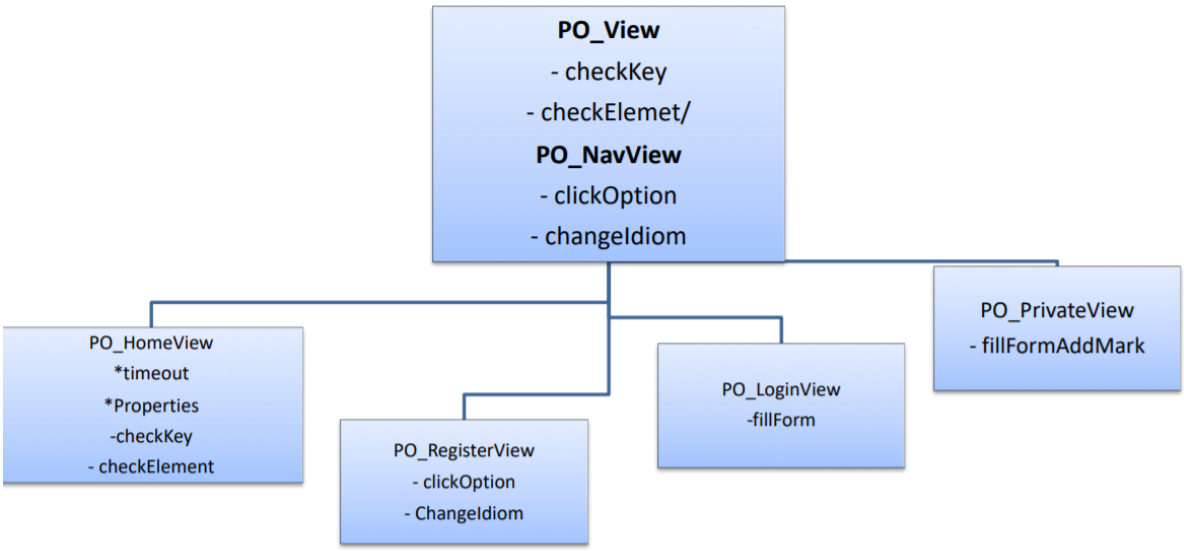
```
@BeforeClass
	static public void begin() {
	//COnfiguramos las pruebas.
	//Fijamos el timeout en cada opción de carga de una vista. 2 segundos.
	PO_View.setTimeout(2);

	}
	@AfterClass
	static public void end() {
	//Cerramos el navegador al finalizar las pruebas
		driver.quit();
	}
```

Esquema de un caso de una prueba:

```
@Test
public void metodo_prueba {
    //Paso1. Solicitud de página
    Driver.get(URL)
    //Paso2 .Esperamos carga de pagina.
    elementos = SeleniumUtils.EsperaCargaPagina(driver, tipo_elemento, cadena,
timeout);
    //Paso3. Interacción con la pagina … Pinchar, rellenar, ….
    elementos.get(0).click(); // Por ejemplo
    //Paso4. Esperar por la respuesta a la interacción
    elementos = SeleniumUtils.EsperaCargaPagina(driver, tipo_elemento, cadena,
timeout);
    //Paso5. Assert de comprobacion.
    Assertrue(“No se obtuvo el resultado esperado”, condicion_basada_elementos);
    //Empezar en Paso 3. de Nuevo si es necesario según la prueba.
}
```

Jerarquía de los Page_Objects:


PO_View
- checkKey
- checkElemet/
PO_NavView
- clickOption
- changeIdiom
PO_HomeView
*timeout
*Properties
-checkKey
- checkElement
PO_RegisterView
- clickOption
- ChangeIdiom
PO_LoginView
-fillForm
PO_PrivateView
- fillFormAddMark