

# Sistemas Distribuidos e Internet

Tema 4

*Spring Boot 3*

# Índice

- Sesión
- Datos y acciones sensibles
- Repositorios
- Paginación
- Transacciones
- Logging
- Subida de ficheros

# Sesión > Introducción

- La aplicación crea una sesión para cada nuevo cliente
  - El primer acceso del cliente/navegador genera una sesión
- Es propia a cada cliente, se identifica con una ID única
- La aplicación envía la ID de al cliente en una cookie
  - Después, el cliente envía esa ID en todas las peticiones

## ▼ Response Headers [view source](#)

**Cache-Control:** no-cache, no-store, max-age=0, must-revalidate

**Content-Language:** es-ES

**Set-Cookie:** JSESSIONID=178FB17BC75030FA0DD8036FFF7E905E; Path=/; HttpOnly

- La aplicación puede almacenar/recuperar datos de las sesiones
  - Estos datos se almacenan en la aplicación no en el cliente
  - Datos de carácter temporal, no es una base de datos

# Sesión > Introducción

- Se **destruyen automáticamente** tras un tiempo de **inactividad (expiración)**
    - Ej: más de 3 minutos sin recibir peticiones con esa ID de sesión
  - Pueden destruirse explícitamente por código
  - Tienen una **configuración por defecto** que define su funcionamiento: *tiempo de vida, encriptación, política de generación de id, etc.*
    - Puede ser modificado
  - Algunos usos comunes de la sesión
    - Carrito de la compra: almacenar temporalmente productos
    - Últimos artículos visitados
    - Almacenar el usuario autenticado (sistemas de autorización manuales)
      - Almacena la ID del usuario autenticado correctamente
      - En toda las peticiones se comprueba si hay usuario autenticado en sesión
- \* *Spring Security utiliza internamente la sesión*

# Sesión > HttpSession

- El **Bean HttpSession** permite acceder a la sesión
  - Puede ser directamente inyectado (no requiere instanciar ni configuración)

```
@Autowired  
private HttpSession httpSession;
```

- **HttpSession** funciona como una tabla hash
  - Gestiona atributos por clave (string) -> valor (object)
    - Admite cualquier tipo de objeto
  - **setAttribute (clave, valor):** almacenar un atributo
  - **getAttribute (clave):** obtiene el atributo
  - **removeAttribute (clave):** elimina el registro
  - Otros

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

# Sesión > Thymeleaf

- Las plantillas **Thymeleaf** pueden acceder a la sesión
  - **Indirectamente**, con los atributos del modelo
    - Enviados desde el controlador a la plantilla
    - Ej: el controlador recupera un atributo de sesión y lo envía a la plantilla con una clave
  - **Directamente** utilizando el objeto **`{session}`**
    - Algunos métodos comunes
    - **`{session.isEmpty()}`** Comprueba si hay algún atributo en sesión
    - **`{session.containsKey('carrito')}`** comprueba si el atributo carrito existe
    - **`{session.carrito}`** obtiene el valor del atributo carrito

# Sesión > Beans

- Por defecto los **Beans** (*@Bean, @Component, @Service, etc*) se gestionan como **singleton**
  - Una instancia que se inyecta en todas las solicitudes
- Anotación **@SessionScope** una instancia distinta para cada sesión
  - Ej: servicio que gestiona los productos **para cada sesión**

```
@SessionScope
@Service
public class CarritoService {

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

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

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

# Datos y acciones sensibles > Introducción

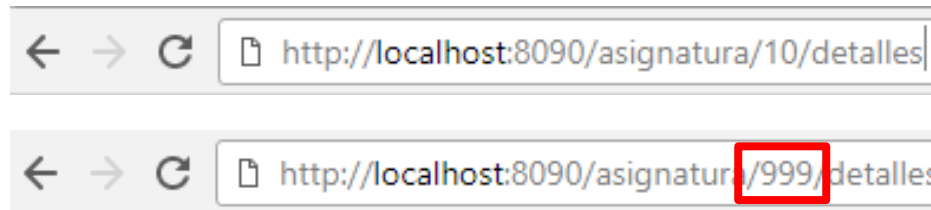
- Las aplicaciones web suelen ser **multiusuario** y de **acceso publico**
  - Multiplica la posibilidad de que sea atacada
  - Multiplica la posibilidad de encontrar las vulnerabilidades
- Un gran problema es la **exposición de datos sensibles**
  - Vulnerabilidad Top 3 según el OWASP
- **No se protege correctamente** el acceso a **datos** o **acciones** sensibles
  - Un usuario accede a una URL a la que no debería
    - Que la URL no tenga enlace no significa que no sea accesible
  - Problema muy dependiente de la implementación
  - Muy grave en la gestión de recursos/entidades y acciones



# Datos y acciones sensibles > Introducción

## ■ 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;  
}
```

# Datos y acciones sensibles > Implementación

- Las URLs con **datos/acciones sensibles** solo pueden ser utilizadas por usuarios concretos
  - Ver detalles, modificar, borrar, otras acciones...
- Identificar todos los datos sensibles en toda la aplicación
- Incluir las **validaciones necesarias** en los servicios
  - Ej, para obtener la asignatura comprobar la ID del 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;  
  
}
```

# Datos y acciones sensibles > DecisionManager

- La protección de URLs se también se puede realizar en la configuración de **Spring Security**
- Especialmente útil cuando las peticiones no pasan por ningún controlador , por ejemplo a recursos de la carpeta static
  - Fotos, ficheros, etc.
- WebSecurityConfigurerAdapter declara diferentes políticas de acceso
  - **Authenticated()** y **hasAuthority(role)** permiten el acceso a usuarios autenticados o con roles
  - **accessDecisionManager(manager)** crear políticas de acceso configurables (ejecutan lógica de negocio), por ejemplo:
    - /streaming.avi solo acceso durante X segundos cada Ip
    - /fotos/12.png solo acceso al usuario con ID 12
    - Otros

# Datos y acciones sensibles > AccessDecision

- Las comprobaciones se realiza en una clase que implementa **AccessDecisionVoter<FilterInvocation>** (Interfaz)
  - El retorno de **vote(authentication,filter,attributes)** determina si permite el acceso. Tipos de retorno:
    - **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;  
        }  
    }  
}
```

# Datos y acciones sensibles > AccessDecision

- La interfaz **AccessDecisionVoter** requiere sobrescribir los métodos **supports()**
  - Ambos deben retornar **true**
- Para aplicar los **AccessDecisionVoter** en la configuración de seguridad:
  - Se agrupan los **AccessDecisionVoter(1-N)** un objeto **AccessDecisionManager**  
Ej, creamos un método para generar el **AccessDecisionManager**
  - Se incluye la política **.accessDecisionManager(manager)** para un conjunto de URLs
    - El **decisionManager** se agrega después de una política básica (Ej, **permitAll()**)

```
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")
```

# Repositorios > @Query

- Las aplicaciones suelen requerir **consultas detalladas fuera del CRUD ej:** (búsquedas, filtrados, ordenadores, etc.)
- La anotación **@Query** permite especificar consultas propias en un repositorio
  - Abstrae la conexión con la base de datos y tratamiento de los datos
- Para definir una nueva consulta:
  - Crear la **signatura de un método** en el repositorio
  - Añadir la anotación **@Query(<JPQL>)** en el método
    - JPQL lenguaje de consultas de JPA basado en SQL
    - Pará acceder a los parámetros se utiliza: ?1 , ?2, ?3 etc.
    - [https://en.wikibooks.org/wiki/Java\\_Persistence/JPQL](https://en.wikibooks.org/wiki/Java_Persistence/JPQL)
  - Ejemplo:

```
@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**
  - Ej: Amazon no muestra una vista con 1.000.000 artículos, muestra 20 artículos y un sistema de paginación



- Cargar muchos elementos en una misma página:
  - Es **costoso para el servidor**
    - Recupera y envía miles de recursos que **en muchos casos no serán realmente consultados**
  - Perjudica la **experiencia de usuario**

# Paginación > Introducción

- Muchas aplicaciones utilizan sistemas de paginación
  - Tanto en las vistas como en la lógica de negocio
- La paginación puede **implementarse manualmente** o utilizando **elementos de un framework**
- Spring incluye un **clases para implementar mecanismos de paginación**
  - Con configuración por defecto (simplificar su uso)
  - **Altamente configurable** en caso de requerir un funcionamiento muy específico



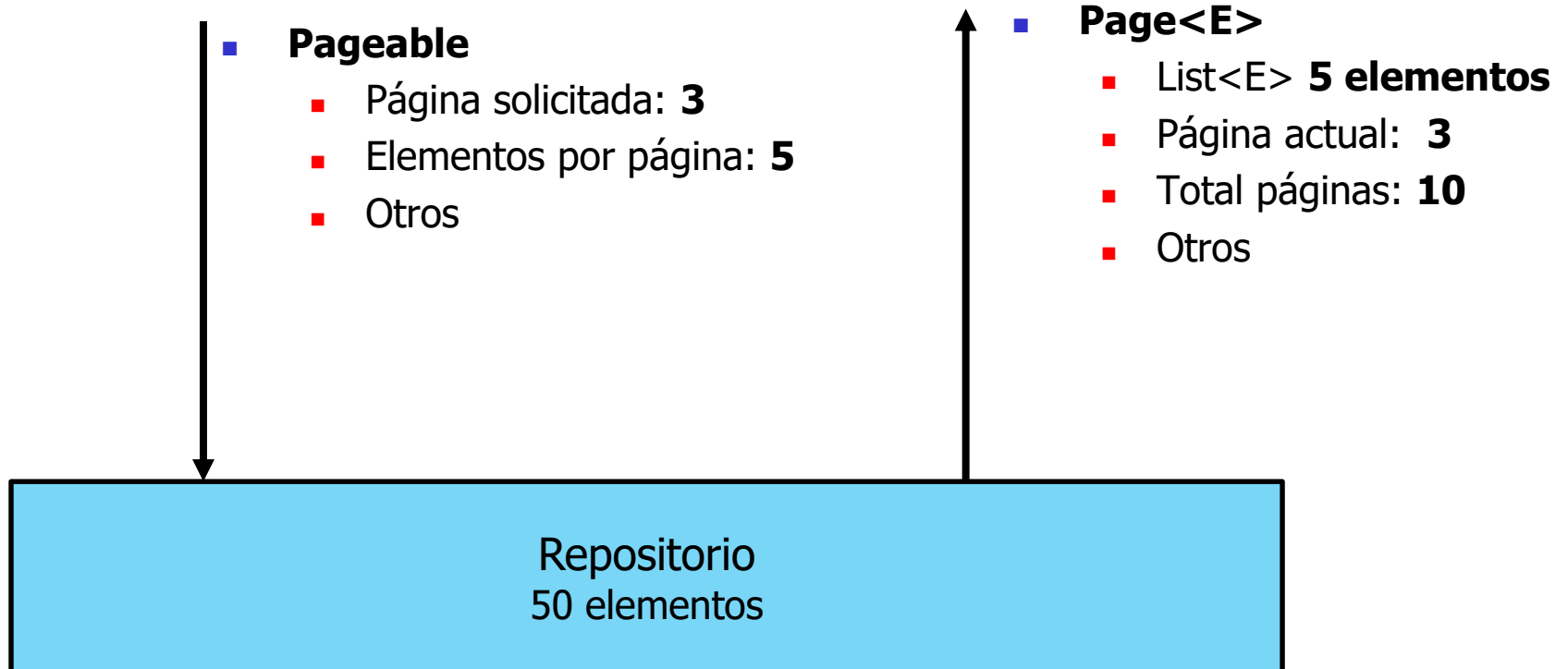
# Paginación > Page<E>

- **Page<E>** objeto del sistema de paginación
  - **Colección** de datos similar a una List
  - Además incluye información sobre la paginación
    - Página a la que corresponden los registros, páginas totales, etc.
- El uso de la paginación afecta a los **repositorios**
  - Afectara a los **métodos que retornan colecciones**
  - Se usará **Page<E>** para retornar los datos obtenidos
    - En lugar de otras colecciones (List)
  - Deben recibir un parámetro adicional **Pageable**
    - Encapsula información sobre la pagina solicitada
    - N° de página solicitada, cantidad elementos que debe contener, etc.

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

# Paginación > Page<E>

## ■ Concepto



# Paginación > Page<E>

- Se puede utilizar **Page<E>** 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()** obtiene la **List<E>** original
  - Es un buen enfoque utilizar **Page<E>** en todas las capas, hasta llegar al **controlador**
    - **Page<E>** contiene información adicional útil para las vistas
      - **getNumber()** N° de página a la que pertenecen los registros
      - **getTotalPages()** N° de páginas totales
    - Esta información es útil para crear componentes de navegación entre páginas en las vistas

# Paginación > Page<E>

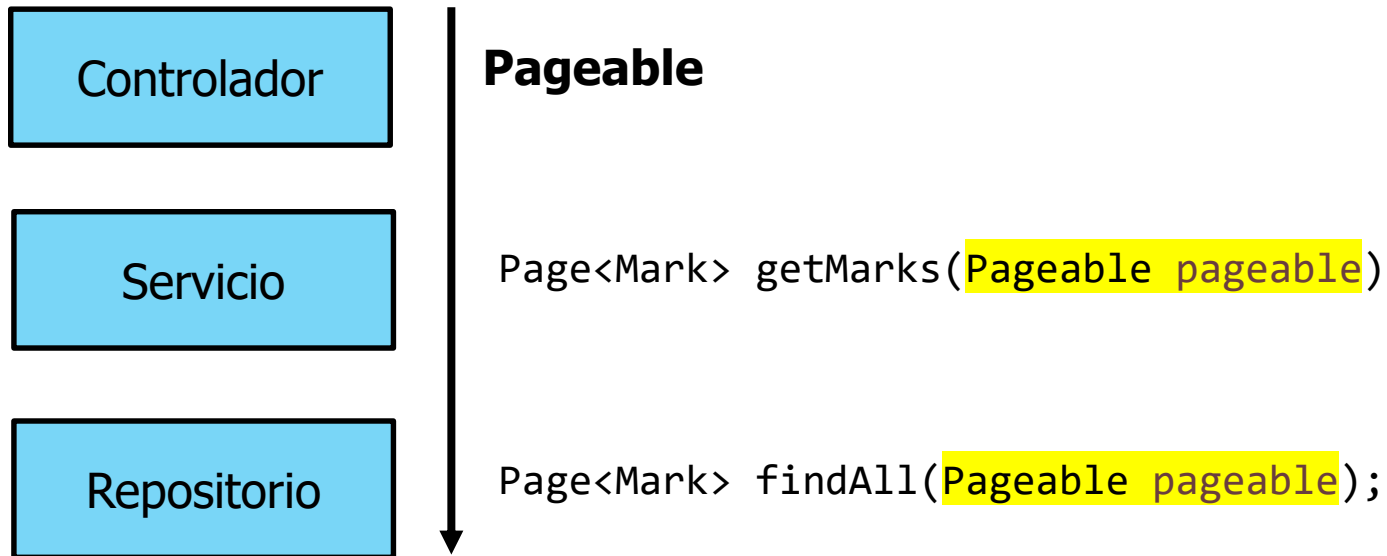
- El parámetro **Pageable** debe ser enviado desde los **controladores** hasta los **repositorios**
  1. \*Los **Controladores** que manejan listados con paginación
    - Deben recibir un nuevo parámetro **Pageable**
      - Encapsula información sobre la página solicitada
      - Debe llegar hasta el repositorio, para saber 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

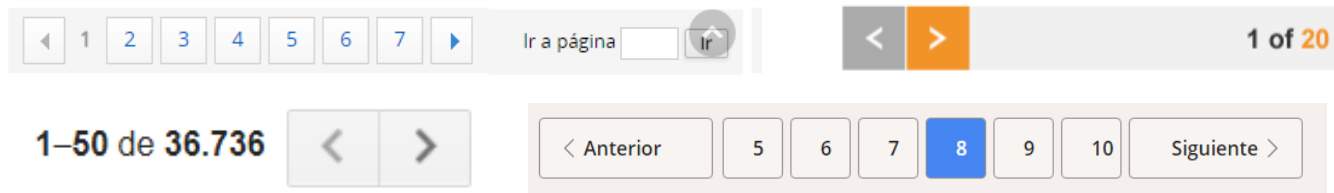
# Paginación > Page<E>

- Reciben el parámetro **Pageable**:
  - Los **servicios** invocados desde **controladores** que manejan paginación
  - Los **repositorios** invocados desde **servicios** que manejan la paginación



# Paginación > Controles

- La vista debe ofrecer un sistema de **navegación entre páginas**
  - Numerosas alternativas



- Se recomienda incluir al menos:
  - Notificación clara de la **página actual**
  - Acceso a las páginas **cercanas**
    - Ej: anterior y siguiente (si es que las hay)
  - Acceso a la **primera** y **última** página



# Paginación > Controles

- El sistema de navegación se implementa en un **fragmento**
  - Reusable en todos los listados con paginación de la aplicación
- La colección **Page<E>** puede ser enviada a la vista, dispone de todo lo necesario para la navegación entre páginas
- Un ejemplo de implementación
  - El controlador envía **Page<Mark>** a la plantilla con clave “**page**”
  - La plantilla utiliza Thymeleaf para incluir enlaces a:
    - La **primera página** (page=0) **siempre**
    - La **anterior a la actual** (page=getNumber-1) **si es que existe** (>=0)
    - La **actual** (page=getNumber) **siempre**
    - La **siguiente a la actual** (page=getNumber+1) **si es que existe** (<=getTotalPages)
    - La **ultima** (page=getTotalPages) **siempre**

# Paginación > Controles

- Un ejemplo de implementación
  - **page=0:** para Spring la primera página es la 0
  - Para el texto en la vista (usuario) comenzamos contando en 1 (no en el 0).

Para **getNumber = 3**  
**?page=3**  
**Página 4 para el usuario**

```
<!-- Primera -->
<li><a th:href="@{'?page=0'}">Primera</a></li>

<!-- Anterior (si la hay) -->
<li th:if='${page.getNumber()-1} >= 0'>
  <a th:href="@{'?page='+${page.getNumber()-1} }" th:text="${page.getNumber()}"></a>
</li>

<!-- Actual -->
<li class="page-item active" >
  <a th:href="@{'?page='+${page.getNumber() } }" th:text="${page.getNumber()+1}"></a>
</li>

<!-- Siguiente (si la hay) -->
<li th:if='${page.getNumber()+1} <= page.getTotalPages()-1'>
  <a th:href="@{'?page='+${page.getNumber()+1} }" th:text="${page.getNumber()+2}"></a>
</li>
```

Diagram illustrating the pagination controls and their corresponding user-visible text:

- Anterior (si la hay):** URL: /marks?page=2, Texto: 3
- Actual:** URL: /marks?page=3, Texto: 4
- Siguiente (si la hay):** URL: /marks?page=4, Texto: 5



# Paginación > Controles

- Un ejemplo de implementación

```
<!-- Última -->
<li>
    <a th:href="@{'?page='+${page.getTotalPages()-1}}"> Última</a>
</li>
```

- Se podrían no mostrar cuando dos enlaces se corresponde con la misma página, Ej:
  - La **primera/ultima** coinciden con la **actual**
  - La **primera/ultima** coinciden con la **anterior** o la **siguiente**
  - **No esta claro si replicar enlaces perjudica la experiencia de usuario**

# Paginación > Configuración

- La **configuración** del sistema de paginación es **modificable**
- Una modificación común es dar otros valores por omisión a los parámetros **page** y **size**
- Se define un objeto **ArgumentResolver** que será insertado en el **Pageable** que reciben los controladores

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

Configuración  
PageableHandlerMethodArgumentResolver



- El nuevo objeto definirá valores **page** y **size** que serán usados en caso de omisión

# Paginación > Configuración

- Para modificar la **configuración** hay que:
  - Partir de una clase **@Configuration**
    - La clase debe implementar **WebMvcConfigurerAdapter** (igual que en la internacionalización)
  - Sobrescribir el método **addArgumentResolvers** de la clase de configuración para añadir un objeto **resolver**
    - **addArgumentsResolvers(resolver)**
  - Crear un **resolver: PageableHandlerMethodArgumentResolver**
    - Contiene toda la información del objeto **Pageable** (Los **Pageable** son recibidos por el controlador)
    - **resolver.setFallbackPageable(PageRequest(page,size))** permite especificar valores para **page** y **size**

# Paginación > Configuración

## ■ Implementación

```
@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);
    }
}
```

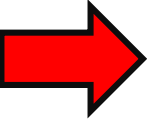
# Transacciones > Introducción

- Los errores y excepciones pueden afectar a la **consistencia de la aplicación**
  - Un método podría ejecutar parte de la lógica y otra parte no
- Las operaciones de lógica pueden ejecutarse como **transacciones potencialmente reversibles**
  - En caso de error vuelve al estado previo
- La anotación **@Transactional** declara el uso de transacciones en un **método o componente**
  - Al incluirla en un **componente** todos sus métodos son **transaccionales**
  - En caso de excepción trata de realizar un “rollback” de las acciones relativas a los repositorios
  - No todos los motores de persistencia lo soportan (Hsqldb sí)

# Transacciones > Introducción

- Ejemplo, la función transferencia modifica dos cuentas en base de datos
  - Se ejecuta **reducirSaldo()** y se produce una **excepción**, antes de ejecutar **aumentarSaldo()**
  - Los datos quedan en un estado **inconsistente**

```
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);  
    }  
}
```



# Transacciones > Introducción

- Al incluir **@Transactional**, si se produce una **unChecked exception** (derivada de RuntimeException) el **repositorio** trata de volver al estado anterior
- Los objetos en memoria no vuelve a su estado previo (datoMemoria)
  - Se podrían aplicar otras técnicas (try catch...)

```
@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);
}
```

- El funcionamiento por defecto de **@Transactional** puede ser ampliamente configurado
  - Mecanismos de propagación, que de excepciones admiten rollback, etc.

```
@Transactional(rollbackFor = Exception.class)
```

**TODAS** las Exception producen rollback

# Logging > Introducción

- **Logging** consiste en guardar información sobre eventos relativas a la aplicación
  - Errores, trazas, accesos, acciones de los usuarios, etc.
- Permite almacenar **información sobre errores**
  - Condiciones y momento en que se ha producido
- Útil para procesos de **auditoria**
  - Registrar los accesos de los usuarios a recursos
- Spring ofrece muchas dependencias para **logging**
  - **org.slf4j.Logger** integrado
  - Muchas otras dependencias externas



# Logging > Logger

- Obtener un objeto **Logger** en las clases que vayan a registrar **log**
  - Se utiliza la factoría **LoggerFactory.getLogger(clase)**
    - Al guardar log se referencia a la **clase lo produjo**

```
public class AccesosService {
```

```
    private final Logger log = LoggerFactory.getLogger(this.getClass());
```

- Métodos para incluir información de **diferentes niveles**
  - **Debug, info, trace, warn, error** : clasificación del nivel de log

```
public void setBloquear(Usuario usuario){  
    log.debug("Log de depuración");  
    log.info("Información");  
    log.error("Log de error");  
}
```

- Permiten formateado de cadenas con parámetros

```
log.trace("{} ejecuta setBoquear {}", usuario.nombre, usuario.permiso);
```

# Logging > Configuración

- Spring y otros componentes (ej hybernate) pueden generar log automático
  - Ese log queda registrado
  - 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
```

- El log siempre aparece en la **consola**
  - **Fecha, nivel, clase, mensaje**
  - Se puede configurar su salida a un fichero de texto

```
2018-01-11 21:20:47.374 INFO 7104 --- [nio-8090-exec-1] com.uniovi.services.AccesosService : Información
2018-01-11 21:20:47.374 ERROR 7104 --- [nio-8090-exec-1] com.uniovi.services.AccesosService : Log de error
```

target	Carpeta de archivos	
.classpath	Archivo CLASSPA...	2 KB
.gitignore	Documento de tex...	1 KB
.project	Archivo PROJECT	2 KB
miaplicacion.log	Documento de tex...	21 KB

logging.file = miaplicacion.log



# Subida de ficheros > Introducción

- La subida de ficheros es un proceso común en unas aplicaciones web
- Los formularios HTML permiten la subida de ficheros
  - Configurando **method="POST", enctype="multipart/form-data"**
  - Incluyendo un campo de tipo **type="file"**

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

- \*Valores para **enctype**
  - **application/x-www-form-urlencoded (Omisión /defecto)**. Codifica los caracteres de todos los campos (espacios a "+"), caracteres especiales convertidos a ASCII HEX.
  - **multipart/form-data**: no usa conversión a caracteres, es para subir ficheros, se basa en MIME data streams [\[RFC2045\]](#).
  - **text/plain**: espacios convertidos a "+", no codifica caracteres

# Subida de ficheros > Procesamiento

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

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

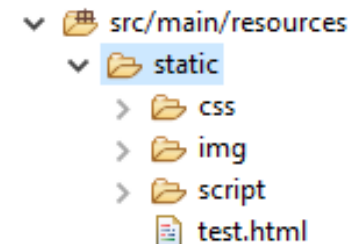
# Subida de ficheros > Procesamiento

- El **controlador** puede procesar el fichero de forma **estándar (Java)**
  - El fichero se podría guardar en el **servidor** o en una **base de datos**
- Ej: guardarlo en el servidor: usando un **InputStream** y la clase de utilidad **File**
  - **File.copy(inputStream, Path, modo)**. Los paths relativos comienzan en la ruta de la aplicación.  
Modo: StandardCopyOption.REPLACE\_EXISTING, otros

```
try {
    String fileName = informe.getOriginalFilename();
    InputStream is = informe.getInputStream();
    Files.copy(is, Paths.get("src/main/resources/static/informes/" + fileName),
        StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    e.printStackTrace();
    return "redirect:/error";
}
return "redirect:/exito";
```

# Subida de ficheros > Configuración

- El path donde se guarda el fichero es importante por ejemplo:
  - Directorio de acceso web **src/main/static/\***
  - Directorios privados en el servidor de la aplicación
- El nombre con el que se salva el fichero suele ser especificado por la lógica de negocio
  - Evita conflictos de nombres
  - Nombres o rutas que permitan asociar el fichero a un usuario asociar (ej ID del usuario)
- En algunos casos es necesario hacer **comprobaciones de autorización** para acceder a ficheros de **directorio de acceso web /static/** . Ej
  - **/static/informes** solo pueden acceder usuarios registrados.
  - **/static/fotos/31** solo puede acceder el usuario 31



# Subida de ficheros > Configuración

- Desde **application.properties** se modifica el tamaño máximo admitido

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