

# Desarrollo MVC con Spring y Maven

Realizado por A.Garay (Dpto. de Informática)

# Resumen de contenidos

- Spring: conceptos
- Maven: instalación y funcionamiento
- Conceptos subyacentes
  - Inyección de dependencias (DI)
  - Inversión de control (IoC)
- Spring MVC

# Spring

- Uno de los frameworks MVC más utilizados para desarrollos web con JAVA, es SPRING.
- SPRING no es un framework MVC propiamente dicho.
  - Es un framework de DI / IoC, que tiene funcionalidades / extensiones para MVC, AOP, administración remota y [otras funcionalidades](#)

# Spring: instalación (introducción)

- Para poder usar Spring, tenemos que descargar unas cuantas librerías implementadas con ficheros “jar” y hacerlas accesibles a nuestro proyecto.
- Cuando un proyecto necesita de muchos subsistemas externos: persistencia, logging, seguridad, acceso a BBDD, inyección de dependencias, es bastante laborioso descargar, instalar, copiar todos los “jar” necesarios. Además las versiones utilizadas pueden ser incompatibles entre ellas.
- Por todas estas razones, en la mayoría de proyectos utilizaremos un gestor de proyectos como Maven, para ayudarnos a crear la estructura del mismo y gestionar todas las “dependencias” de otros subsistemas.

# Spring: instalación (con Maven)

- (Desde ECLIPSE EE) File → New → Other → Maven Project
- Create a simple project (skip archetype)
- Editar el fichero pom.xml → Dependencies
- Añadir la dependencia
  - `org.springframework`
  - `spring-context`
  - `5.3.3.RELEASE` (buscar [aquí](#) la versión más actual)

# DI: concepto

- Dependencia: objeto A que necesita a otro B (incluido entre sus atributos) para hacer un trabajo.
- Para que al acceder a los servicios de B desde A no se provoque una excepción de tipo *nullPointer*, alguien debe crear el objeto B previamente.
- Normalmente lo hará el propio objeto A, sin embargo en muchas ocasiones es más conveniente ceder esta responsabilidad a una factoría o a un framework de DI, con el objetivo de flexibilizar nuestro sistema.
- PROBLEMA: “Si yo cambio un “new” en un código, tengo que recompilar para que vuelva a funcionar, y eso no es bueno”
- SOLUCIÓN: Si hay un sistema externo que decide qué objetos “inyecto” en determinados atributos se puede cambiar el comportamiento del sistema en “caliente”, y además lleva a diseños más desacoplados y por tanto más robustos y más “testeables”

# (1/6) DI: El problema

```
public class A {  
  
    private B b; // DEPENDENCIA  
    public A() {  
        }  
    public void fa() {  
        b.fb(); // ERROR en EJECUCIÓN nullPointer  
    }  
}
```

```
public class B { public void fb() { } }
```

```
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        a.fa();  
    }  
}
```

## (2/6) DI: La solución (patrón creador)

```
public class A {  
  
    private B b; // DEPENDENCIA  
    public A() { this.b = new B(); }  
    public void fa() {  
        b.fb();  
    }  
}
```

```
public class B { public void fb() { } }
```

```
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        a.fa();    }  
}
```



# (3/6) DI: La solución (DI con Spring)

@Component

```
public class A {  
    @Autowired  
    private B b; // DEPENDENCIA  
    public A() {  
    }  
    public void fa() {  
        b.fb();  
    }  
}
```

@Component

```
public class B { public void fb() { } }
```

@ComponentScan

```
public class Main {  
    public static void main(String[] args) {  
        ApplicationContext context = new AnnotationConfigApplicationContext(Main.class);  
        A a=(A)context.getBean("a");  
        a.fa();  
    }  
}
```

## (4/6) DI: Añadiendo flexibilidad (Adapter)

```
public class A {  
    private IB b; // DEPENDENCIA  
    public A() { if (cond1) {  
        this.b = new B1();  
    } else if (cond2) {  
        this.b = new B2();  
    }  
}  
    public void fa() { b.fb();}  
}
```

```
public interface IB { public void fb(); }  
  
public class B1 implements IB {  
    public void fb() { //Implementación 1 }  
}  
  
public class B2 implements IB {  
    public void fb() { //Implementación 2 }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        a.fa();  
    }  
}
```

# (5/6) DI: Delegando responsabilidades de creación compleja de objetos a una **Factory Singleton**

```
public class A {  
  
    private IB b; // DEPENDENCIA  
    public void fa() { b.fb();}  
    public A() {  
        this.b = BFactory.getBF().getB();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        a.fa();  
    }  
}
```

```
public class IB { public void fb(); }  
public class B1 implements IB { public void fb() { //1 } }  
public class B2 implements IB { public void fb() { //2 } }
```

```
public class BFactory {  
    private static BFactory bf; // SINGLETON  
    private IB b; // FACTORY  
    private BFactory() {} // SINGLETON  
    public static BFactory getBF() { // SINGLETON  
        if (bf==null) {bf = new BFactory(); return bf; }  
    }  
    public IB getB() { // FACTORY  
        Properties p = new Properties(); p.load(new FileInputStream("conf.properties"));  
        if (this.b==null) {this.b=(IB)Class.forName(p.getProperty("impl")).newInstance();}  
        return this.b; }  
}
```

conf.properties

impl = ruta.a.B1

Puedo cambiar la implementación de b desde un fichero de texto sin tener que recompilar

## (6/6) DI: Inyectando subsistemas desde SPRING

```
public class A {  
  
    @Autowired  
    private IB b; // DEPENDENCIA  
    public void fa() { b.fb();}  
  
}
```

```
public class IB { public void fb(); }  
public class B1 implements IB { public void fb() { //1 } }  
public class B2 implements IB { public void fb() { //2 } }
```

beans.xml

```
<?xml version = "1.0" encoding = "UTF-8"?> ... <context:annotation-config />  
  
<bean id="b" class="ruta.a.B1"></bean>  
  
<bean id="a" class="ruta.a.A"></bean>
```

Puedo cambiar la implementación de b desde un fichero de texto XML sin tener que recompilar

```
public class Main {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
        A a=(A)context.getBean("a");  
        a.fa();    }  
}
```

# IoC: inversión de control. Concepto

- Para que un framework de DI como Spring, pueda hacer su “magia”, es necesario que tenga lugar una “inversión del control” de la ejecución del programa.
- La diferencia entre el caso 2 y el 3, es que en el 3 no se hace “new” de los objetos A ni B por ningún lado.
- Esto no es magia. Alguien lo debe hacer. En realidad lo hace Spring (a través de su objeto context) es tomar el control de la ejecución de mi programa y decidir qué líneas de código se ejecutarán a continuación

# Proyecto Spring básico (1/2)

- Guía rápida [aquí](#)
- Crear un nuevo [proyecto Maven simple](#) (no usar arquetipos)
- Activar spring en nuestro proyecto Maven
  - Incluir la dependencia ( [org.springframework / spring-context / 5.0.3 RELEASE](#) )
- Clase “Main.class”
  - Anotar con [@ComponentScan](#) para que autoescaneé (desde el paquete de la clase anotada hacia “abajo” recursivamente, o bien desde el que especifiquemos) y genere un singleton por cada clase anotada con [@Component](#) o cualquiera de sus “variedades”
- Método “main(...)”
  - Crear el contexto Spring con
    - [ApplicationContext context = new AnnotationConfigApplicationContext\(Main.class\)](#)
- Clases que queremos que sean instanciadas por Spring
  - Anotar la clase con [@Component](#)
  - Alternativamente con [@Service](#), [@Repository](#), [@Controller](#) (son componentes, pero más específicos)

# Proyecto Spring básico (2/2)

- Obtener un bean explícitamente desde el contexto. Recordar hacer casting
  - context.getBean("nombreClaseEnMinuscula")
    - Si anotamos la clase a instanciar con @Component(value="otroNombre"), se podría utilizar éste
- Permitir que Spring "inyecte" el bean cuando se necesite
  - Anotar con @Autowired un atributo o un setter de ese atributo
  - Poner @Autowired(required=false), si queremos que pueda quedar a null la dependencia
- Singleton vs. múltiples instancias.
  - Anotar con @Scope("prototype") aquellos componentes que no queramos que sean singletons
- Inyectar un valor a un atributo desde un fichero de properties externo
  - Anotar la clase en la que inyectar el value con @PropertySource("classpath:fichero.properties")
  - "fichero.properties" debe estar en "src/main/resources"
  - Escribir "categoria.dato = valor" en el "fichero.properties"
  - Anotar el atributo cuyo valor queremos inyectar con
    - @Value("\${categoria.dato}")

# Spring boot

- Es un nuevo tipo de proyecto Spring MVC, que utilizaremos en lugar del visto anteriormente.
- Su gran ventaja es que incorpora el código del equivalente a Tomcat, por tanto...
- **¡¡ No es necesaria la instalación de un servidor Tomcat!!.**
- Un proyecto SpringBoot es independiente, y al ejecutarse levanta un servicio en el puerto 8080



# Spring MVC (1/6)

1. Instalar Eclipse STS (usaremos “Spring boot”, en lugar de “Tomcat”)
2. New → Spring starter project

Name	MiWeb		
<input checked="" type="checkbox"/> Use default location			
Location	C:\Users\Alberto\workspaceSTS\MiWeb		
Type:	Maven	Packaging:	Jar
Java Version:	1.8	Language:	Java
Group	org.agaray.spring		
Artifact	miWeb		
Version	0.0.1-SNAPSHOT		
Description	Demo project for Spring Boot		
Package	org.agaray.spring.miweb		

# Spring MVC (2/6)

## 3. Escoger dependencias

- Web (Spring web)
- Template engines (Thymeleaf)
- SQL (Spring data JPA, MySQL Driver)

## 4. Editar el fichero “application.properties”

```
spring.datasource.url = jdbc:mysql://localhost:3306/test
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
spring.datasource.username = root
spring.datasource.password =
spring.jpa.database-platform = org.hibernate.dialect.MariaDBDialect
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.output.ansi.enabled=always
```

# Spring MVC (3/6)

5. Crear bajo “entities” nuestros beans de negocio anotados como es habitual con JPA - hibernate

- **@Entity** en la clase
- **@Id** y **@GeneratedValue(strategy=GenerationType.IDENTITY)** en el atributo *id:Long*
- **@Column(unique=true)** en los atributos cuyos valores queramos que sean únicos en la BD (p.ej. un NIF, o una matrícula de coche)
- **@ManyToOne**, **@OneToMany**, **@ManyToMany** ó **@OneToOne** en los atributos dependientes

# Spring MVC (4/6)

## 6. Crear bajo el paquete “repository” nuestros componentes DAO (equivalentes a los “model” en CodeIgniter)

- Un buen nombre para ellos sería **BeanRepository**
- Implementarlos como interfaces que heredan de **JpaRepository<Bean, Long>** y anotadas con **@Repository**
- Usar después (desde los controladores) métodos útiles estándar como:
  - **save(bean)**
  - **findAll()**
  - **findAll(beanEjemplo)**
  - **getById(id)**

# Spring MVC (5/6)

## 7. Crear bajo “controller” los <Bean>Controller

- anotados como `@Controller`
- Crear tantos atributos `@Autowired private BeanRepository repoBean;` como necesitemos
- Crear un método `public String metodo(...)` por cada operación de bean GET/POST
  - Anotar con `@GetMapping("/url")` O `@PostMapping(value = "/otra/url")`
  - Pasarle como parámetro `ModelMap model`, para pasar datos a la vista
  - Pasarle como parámetros tantos `@RequestParam("name_param") Tipo name_param` como “name’s” de “forms” provengan de la vista GET
- Construir los modelos a enviar a la vista, dentro de los métodos con `model.put("k",v)`
- Desplegar vistas con `return “ruta/a/vista”` (ruta desde la carpeta “templates”)

# Spring MVC (6/6)

## 8. Crear bajo “`/src/main/resources/`templates” las vistas

- Organizarlas por carpetas, según se considere (por beans, por roles, etc.)

## 9. Uso de thymeleaf

- Encabezar con `<html xmlns:th="http://www.thymeleaf.org">`
- Usar `<span th:text="${var_model}"></span>` para introducir datos provenientes del modelo (var\_model puede ser un objeto. Usar `var_model.atributo` si necesario)
- Usar `<tr th:each="elem : ${coleccion}">`  
`<td th:text="${elem.atributo}"></td></tr>` para recorrer colecciones

## 10. Ejecutar el proyecto

- Run as → Spring boot app

# Spring MVC (Apéndice 1/7)

- Gestionando **SESIONES**

- Basta con pasarle al método del controlador que necesite acceso a una sesión, un objeto tipo `HttpSession` (si no existe lo creará e inyectará)
  - `public String accion(HttpSession s) { ... }`
- A partir de ahí, utilizaremos, como siempre los datos de sesión mediante **`setAttribute("nombre",valor)`**, **`getAttribute("nombre")`**, **`removeAttribute("nombre")`**
- También desde `thymeleaf`, podremos acceder a atributos de sesión fácilmente:
  - `<span th:text="${session.nombre}"/>`

# Spring MVC (Apéndice 2/7)

- Trabajando con **assets estáticos**
  - Ubicarlos en subcarpetas bajo la carpeta “resources/static” creada por defecto en un Spring Boot starter project
  - P.ej, si se quiere utilizar bootstrap offline, es el lugar ideal para dejar los “css” y “js” necesarios
  - Aludir a dichos recursos desde HTML utilizando ruta absoluta (desde static )
    - ``



# Spring MVC (Apéndice 3/7)

- **Enmarcando con Thymeleaf**
  - Utilizar `<div th:replace="~{ /ruta/a/vista }"></div>`
  - Sustituye el div actual por el contenido del archivo HTML aludido (¡ojo, la extensión debe ser `html`, si no, hay que especificarla)
  - Se podría incluir incluso sólo un fragmento de dicho archivo utilizando el operador `::`
  - `"/ruta/a/vista"` se entiende que está referida desde la carpeta `"templates"`

# Spring MVC (Apéndice 4/7)

- **Filtrando y ordenando repositories** (parte 1)
  - Para filtrar, añadir al repository métodos del estilo:
    - `public List<Pojo> findBy{atributo} (Tipo {atributo});`
    - `public Pojo getBy{atributo} (Tipo {atributo});`
    - `public List<Pojo> findBy{Atributo}And{OtroAt}(Tipo {atributo}, OtroTipo {otroAt});`
  - Para ordenar:
    - `public List<Pojo> findAllOrderBy{Atributo}[Asc|Desc]{OtroAtributo}[Asc|Desc]();`
  - También se pueden mezclar orden y filtros:
    - `List<Persona> findByApellidoOrderByNombreAsc(String apellido);`
  - Para más modificadores, ver la siguiente diapositiva

# Spring MVC (Apéndice 5/7)

## ● Filtrando repositories (parte 2)

- Añadir (en el cuerpo del Repository) métodos del tipo...

@Query("select de solo un bean")

o bien

@Query("select de varios beans")

public Bean miTitulo(misDatos);

public List<Bean> miTitulo(misDatos);

- Otros modificadores (ejemplos abajo)

- IgnoreCase
- Not (distinto a)
- Like / Containing / StartingWith
- BeanAtributo (anidamientos, ej: findByPaisNombre(String nombre) en PersonaRepository)
- LessThan / GreaterThan / Between
- And / Or
- OrderBy

# Spring MVC(Apéndice 6/7)

- Añadiendo métodos personalizados a un repository **[Bean]Repository**
  - Crear una nueva interface: **[Bean]RepositoryCustom**
  - Implementar dicha interfaz: **[Bean]RepositoryImpl**
    - Ver ejemplo en los comentarios de esta diapositiva
  - Hacer que **[Bean]Repository** herede ahora de **JpaRepository<[Bean], Long>** y de **[Bean]RepositoryCustom**
  - Más información en <https://dzone.com/articles/add-custom-functionality-to-a-spring-data-reposito>

# Spring MVC(Apéndice 7/7)

- Manejo de errores

- Crear una clase “manejadora” anotada con `@ControllerAdvice`
- Crear un método en esa clase por cada tipo de exception (p.ej. `MiException`) que queramos manejar, anotada con `@ExceptionHandler(MiException.class)`.
- Este método debe devolver (desplegar) la vista de error que queramos
- Para acceder a más información basta con pasarle como parámetro la excepción que ha saltado

**Patrones de diseño básicos (GRASP y GoF)**

# Patrones de diseño

- Son soluciones comunes y robustas a problemáticas habituales de programación.
- Técnicamente se definen como un par (**problema**, **solución**), al que le asignamos un nombre concreto.
  - p.ej Patrón **CREADOR** (¿Quién debe crear un objeto?, El que se lo quede)
- Las problemáticas resueltas son muy variadas y pueden ir desde un problema muy concreto a algo muy genérico.

# “Familias” de patrones GRASP y GoF (/)

- GRASP (General Responsibility Assignment Patterns)
  - Contiene entre otros, 5 de los patrones más básicos de la POO. Son problemas muy genéricos, por tanto más difíciles de entender, pero de alguna forma los más básicos que marcarán nuestra habilidad y buen hacer como programadores.
- GoF (Gang of Four)
  - Colección heterogénea de patrones genéricos de programación pero para casuísticas más concretas que los GRASP



# Patrones GRASP (/)

- **EXPERTO:** ¿Qué clase es la más apropiada para ejercer una determinada responsabilidad?
  - Aquélla que tenga toda la información necesaria para realizar esa tarea y más “a mano”.
- **CREADOR:** ¿Qué clase debería crear un objeto de una clase determinada?
  - Aquélla que se la vaya “a quedar”, es decir que mantenga el sistema con el acoplamiento más bajo.
- **BAJO ACOPLAMIENTO:** ¿Qué regla general aplicaremos para mantener un sistema escalable?
  - Asociar las clases entre sí de manera que mantengan el acoplamiento al mínimo nivel.
    - **Acoplamiento:** medida del número de clases que se relacionan (tienen atributos) de otras clases.
- **ALTA COHESIÓN:** ¿Qué responsabilidades se deben asignar en general a una clase?
  - Aquéllas que mantengan la cohesión interna lo más alta posible.
    - **Cohesión:** medida del grado de relación semántica que tiene un método con otro y con el cometido teórico de la clase.
- **CONTROLADOR:** ¿Quién debería recibir en primera instancia los eventos del sistema?
  - Una clase de “invención pura” llamada controlador que hará de bypass (o portal) de todos los casos de uso del sistema.
  - Si hubiera muchos CdU podría ser útil definir varios controladores en función de algún criterio, p.ej. por rol de usuario

# Patrones GoF

- Hay tres tipos de patrones GoF:
  - Creacionales: Creación de instancias
    - Factory, Singleton, MVC, ...
  - Estructurales: Composición de clases y objetos
    - Adapter, Facade, Composite, ...
  - Comportamiento: Interacción, algoritmos encapsulados.
    - Observer, Strategy

# Ejemplo práctico de adaptación de subsistemas

- Patrón adapter

- Establece una interfaz común de acceso a un subsistema, independiente de éste.
- Permite al sistema cliente crecer sin depender del subsistema que necesita para hacer su trabajo
  - [Diagrama](#)
-

# Referencias

- Más información en

[https://docs.google.com/presentation/d/1FVaoPNPyeQj3DxzwH8FxmHOfRITkmJy\\_\\_bSXb-gtAvA/edit#slide=id.p18](https://docs.google.com/presentation/d/1FVaoPNPyeQj3DxzwH8FxmHOfRITkmJy__bSXb-gtAvA/edit#slide=id.p18)