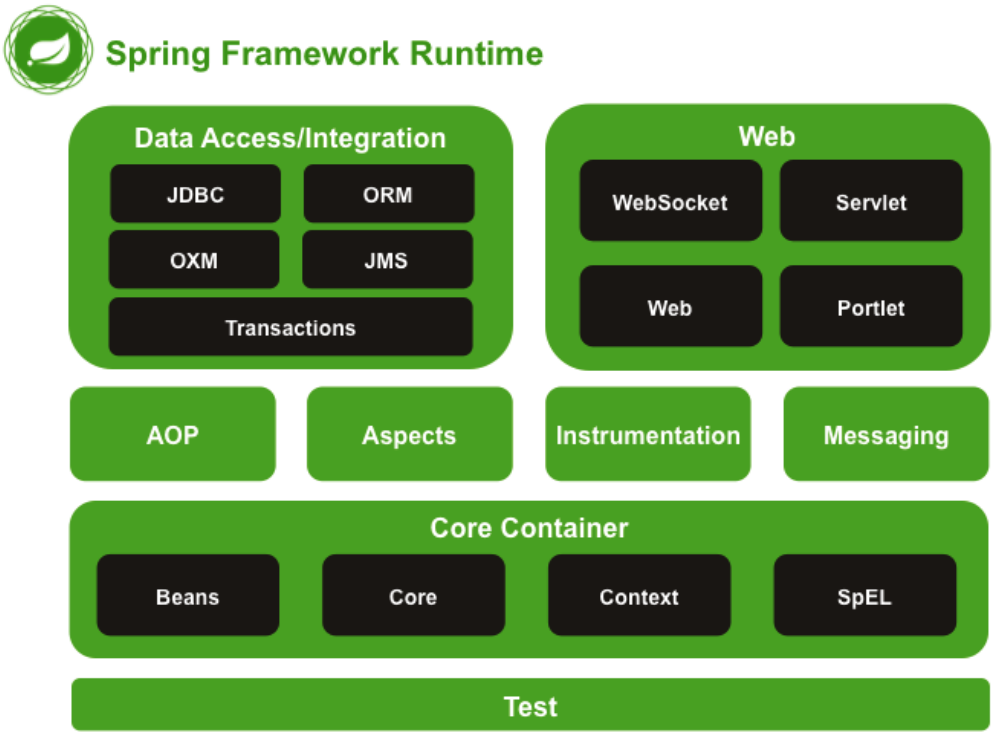
GUSTAVO CORONEL
DESARROLLA SOFTWARE

DESARROLLO WEB

CON SPRING BOOT



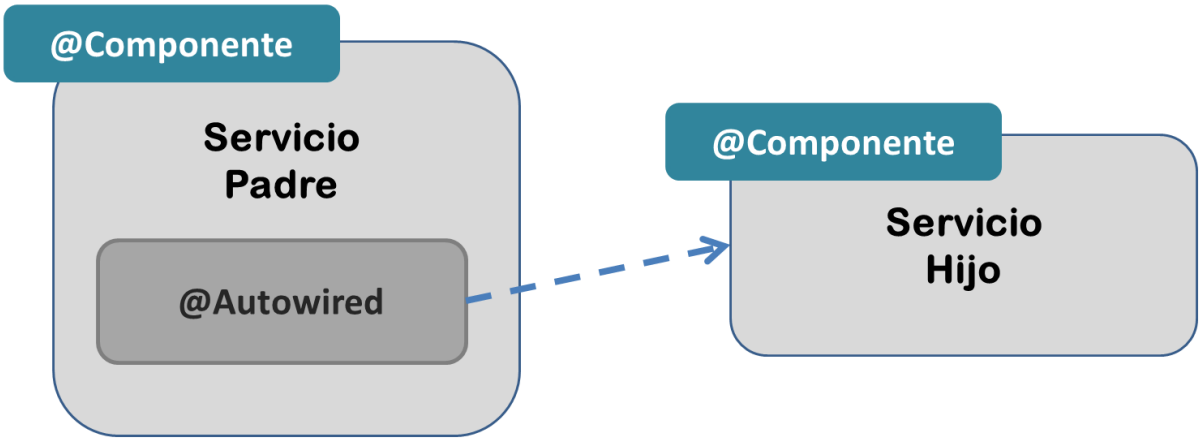
UNIDAD 03
ANOTACIONES

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
gcoronelc@gmail.com
I N S T R U C T O R

CONTENIDO

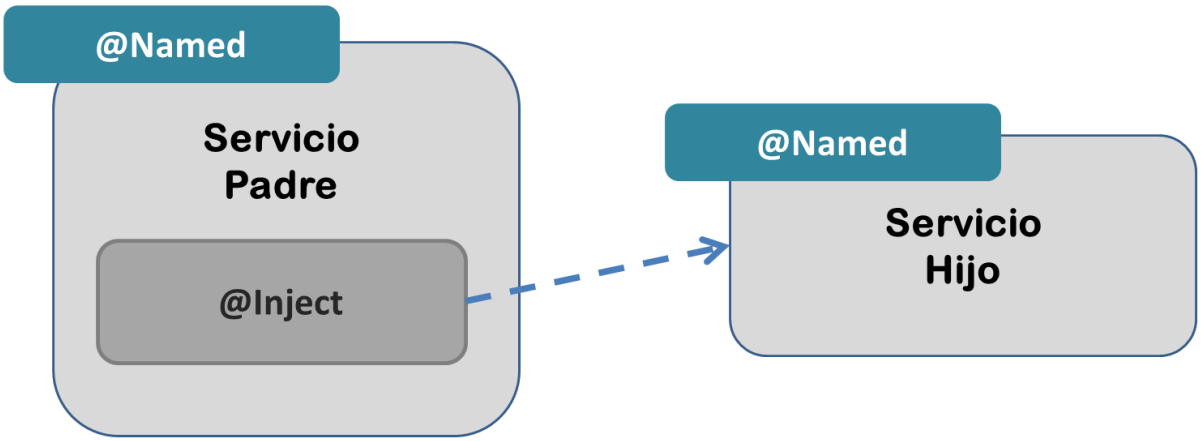
INTRODUCCIÓN

Contexto



Usar anotaciones dentro de las clases permite que la configuración y la implementación estén en un único sitio.

Anotaciones Estándares de Java EE



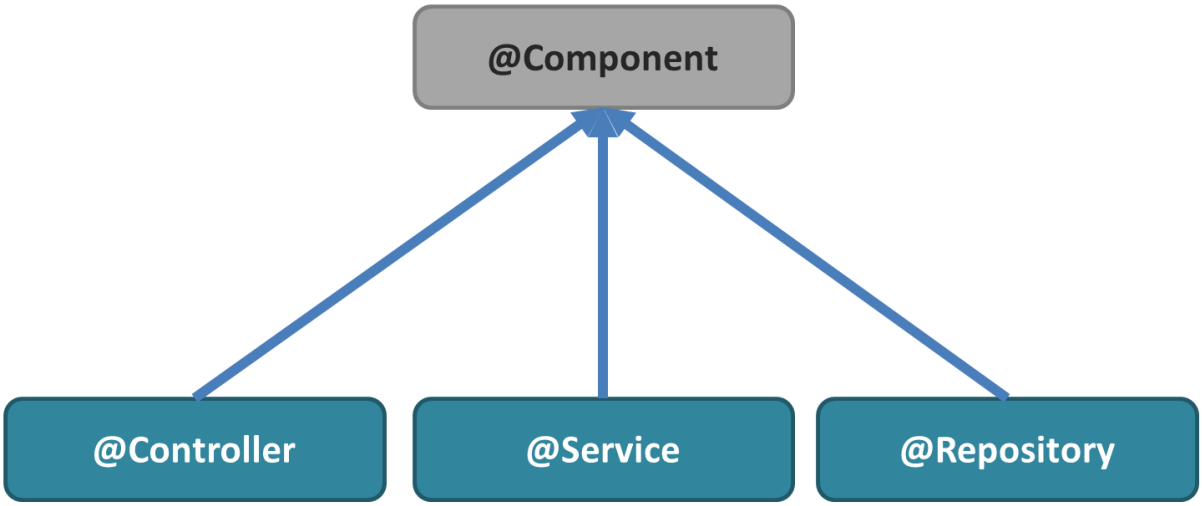
La competencia entre los estándares de Java EE y el Spring Framework es cada vez más dura ya que las similitudes entre ambos son muchas. Elegir uno u otro depende de muchas cosas. Spring también soporte las anotaciones de Java EE.

Para hacer uso de las anotaciones de JavaEE es necesario agregar la siguiente dependencia:

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

COMPONENTES

Anotaciones de Beans



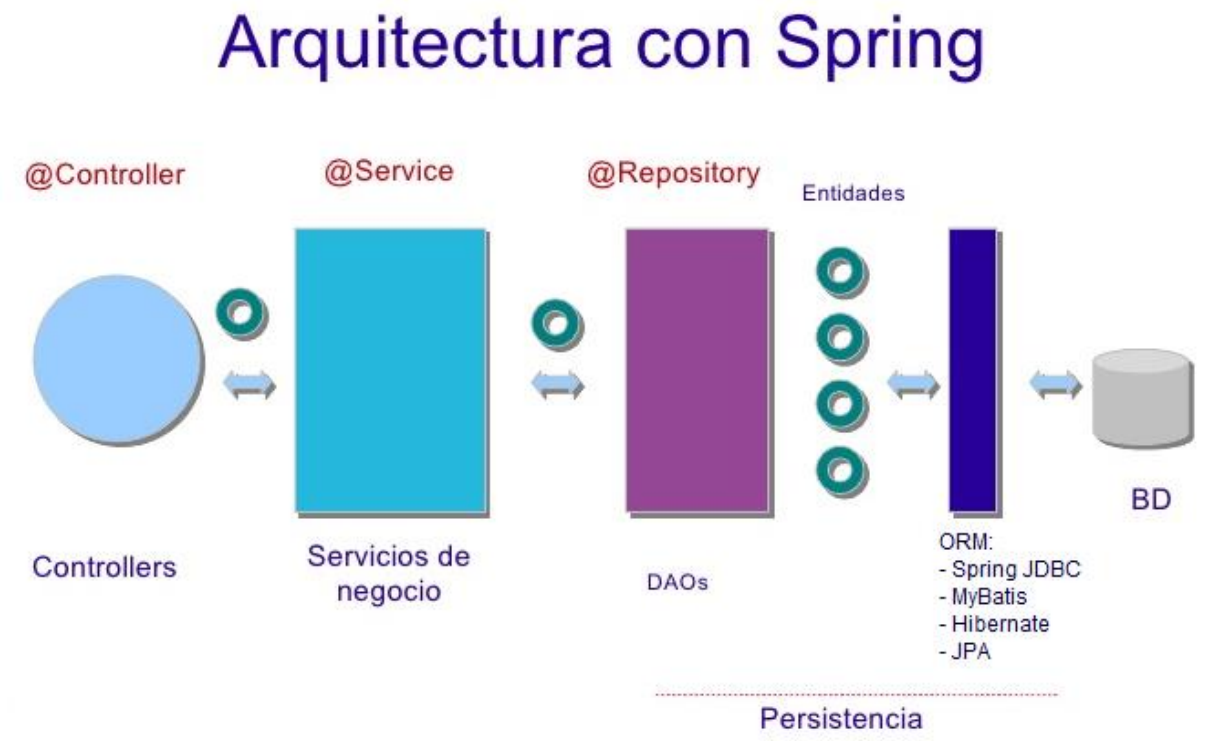
Las anotaciones se utilizan para que Spring detecte las clases importantes dentro del propio código fuente de Java.

Tienes cuatro anotaciones para definir tus componentes:

- **@Component:** Para que definas componentes genéricos.
- **@Controller:** Para que definas tus componentes controladores.
- **@Service:** Para que definas tus componentes de negocio.
- **@Repository:** Para que definas tus componentes de persistencia.

Las anotaciones Controller, Service y Repository heredan de Component.

Arquitectura



La figura anterior, te muestra la arquitectura que debes utilizar con Spring para el desarrollo de aplicaciones, como puedes observar, la capa de persistencia la puedes trabajar con Spring JDBC o utilizar algún Framework ORM.

@Component

Esta anotación te sirve para añadir un estereotipo de forma genérica a una clase. Un "estereotipo" es una manera de clasificar las clases a un alto nivel. Esta información normalmente de tipo semántica, pero puede utilizarse para caracterizar una clase al punto de establecer como debe comportarse cuando se produzca una excepción, por ejemplo.

```
import org.springframework.stereotype.Component;

@Component
public class VentaService {
   . . .
   . . .
}
```

La anotación @Component es genérica y no se espera que la utilice realmente, lo esperado es que usen algunas de las otras anotaciones derivadas de ella como @Repository, @Service y @Controller, para los componentes de persistencia, servicios y controlador respectivamente.

@Scope

Por defecto el alcance de un bean es sigleton, lo que indica que solo existirá una sola instancie de la clase, o que es lo mismo, solo se creará un solo objeto.

Esta anotación debes utilizarla cuando quieras modificar el alcance por defecto de un bean. A continuación, tienes un ejemplo ilustrativo.

```
import org.springframework.stereotype.Component;

@Component
@Scope("session")
public class VentaService {
   . . .
   . . .
}
```

Los valores asociados con esta anotación son: singleton, prototype, request y session.

DEPENDENCIAS

@Autowired

Esta anotación hace que el framework aplique el autodescubrimiento e inyección automática de dependencias.

Lo vas a usar frecuentemente sobre setters, de forma que Spring automáticamente busca el bean que mejor se adapta al tipo del parámetro:

```
@Autowired
public void setMotor(Motor motor) {
   this.motor = motor;
}
```

Pero también puedes utilizar esta anotación directamente sobre las propiedades, sobre el constructor, o sobre un método, con cualquier nombre y con cualquier cantidad y tipo de parámetros:

```
// Sobre una propiedad
@Autowired
private Motor motor;

// Sobre un constructor
@Autowired
public Coche(Motor motor) {
   this.motor = motor;
}

// Sobre un método
@Autowired
public void montaje(Motor motor, Volante volante)       {
   this.motor = motor;
   this.volante = volante;
}
```

Incluso lo puedes utilizar para obtener todos los beans de un mismo tipo declarados en la configuración y que se almacenen en algún tipo de colección:

```
@Autowired
private Pieza[] piezas;

@Autowired
private List<Pieza> piezas;

@Autowired
private Map<String, Pieza> piezas;
```

Si la dependencia no se puede resolver se dispara una excepción. No obstante, este comportamiento lo puede modificar utilizando el parámetro required, tal como lo observas a continuación:

```
@Autowired(required=false)
private Copiloto copiloto;
```

ANOTACIONES DE JAVA EE

Configuración

Para que puedas utilizar estas anotaciones es necesario añadir la librería respectiva.

Es necesario que añadas la dependencia en el fichero pom.xml:

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

@Named

Esta anotación es similar a @Component.

```
import javax.inject.Named;

@Named
public class Motor {
   . . .
   . . .
}
```

@Inject

Esta anotación tiene el mismo comportamiento que la anterior @Autowired, aunque carece del parámetro required.

```
import javax.inject.Named;
import javax.inject.Inject;

@Named
public class Coche {

   @ Inject
   private Motor motor;

}
```

@Resource

Esta anotación la utilizaras para eliminar ambigüedades a la hora de inyectar dependencias automáticamente, sobre todo si se aplica la anotación **@Autowired** en propiedades que tienen como tipo una interface, ya que distintas clases pueden implementar una misma interface.

```
   @Autowired
   @Resource(name="volante")
   private Pieza volante;

   @Autowired
   @Resource(name="retrovisor")
   private Pieza retrovisor;
```

Si omites el valor del parámetro **name** intentará resolver la dependencia buscando un bean que tenga el mismo nombre de la propiedad a la que se ha aplicado la anotación. Y si no encuentra un bean intentará resolverlo por tipo.

@Qualifier

Esta anotación tiene un comportamiento similar a la anterior @Resource, pero utiliza los roles de los beans en vez de sus identificadores para resolver las dependencias.

Su uso más común es aplicarla usando el valor de alguna característica semántica de los beans definidos en la configuración. La "característica semántica" es simplemente el valor de la etiqueta **qualifier** que tiene un bean dentro del fichero XML, y que se usa para indicar el "rol" que tiene un bean dentro de la aplicación.

Supongamos que tienes la siguiente definición de beans, donde los beans representan piezas de un coche que se clasifican en función de su posición:

```
public interface Pieza {
   . . .
}

@Component
@Qualifier("frontal")
public class Parachoque implements Pieza {
   . . .
}

@Component
@Qualifier("maletera")
public class Maletera implements Pieza {
   . . .
}
```

Utilizando la anotación **@Qualifier** se puede eliminar la ambigüedad a la hora de resolver las dependencias de forma automática, incluso cuando se aplican a colecciones:

```
  @Autowired
  @Qualifier("posterior")
  private Pieza maletera;

  @Autowired
  @Qualifier("frontal")
  private List<Pieza> morro;
```

@PostConstruct

Esta anotación la debes utilizar cuando necesitas ejecutar un método de un bean después de que ha sido instanciado por Spring y resuelto todas sus dependencias.

```
import javax.inject.Named;
import javax.inject.PostConstruct;

@Named
public class Coche {

   @PostConstruct
   public void initBean() {
      System.out.println("Bean iniciado.");
   }

}
```

Esta misma funcionalidad la consigues implementando la interface **InitializingBean**.

@PreDestroy

Esta anotación la debes utilizar para configurar un método que se ejecuta antes de que sea destruido del contexto de Spring.

```
import javax.inject.Named;
import javax.inject.PreDestroy;

@Named
public class Coche {

   @PreDestroy
   public void resetBean() {
      System.out.println("Bean listo para ser destruido.");
   }

}
```

Esta misma funcionalidad la consigues implementando la interfaz **DisposableBean**.