



**Laboratorio 4 - Principio de Inversión de dependencias, Contenedores Livianos,
Inyección de dependencias, Componentes y conectores.**

Andrés Felipe Arias Ajiaco

Cesar David Amaya Gómez

Johan Sebastián Gracia Martínez

Sebastián David Blanco Rodríguez

Universidad Escuela Colombiana de ingeniería Julio Gravito

Arquitecturas de software

Ing. Javier Iván Toquica Barrera

23 de febrero de 2024

Introducción

El laboratorio propone dos partes distintas que involucran el manejo de componentes, conectores, Inversión de dependencias, contenedores livianos e inyección de dependencias.

En la primera parte, se plantea un ejercicio práctico cuyo objetivo principal reside en la aprehensión profunda del manejo de las diversas etiquetas presentes en Spring Boot. Dichas etiquetas se encuentran directamente relacionadas con la implementación de la inyección de dependencias, la gestión de contenedores livianos y la aplicación del principio de inversión de dependencias. Estos tres elementos constituyen pilares fundamentales en el desarrollo de aplicaciones robustas, escalables y con un elevado grado de mantenibilidad utilizando el marco de trabajo Spring Boot.

En la segunda parte, se presenta un ejercicio de mayor complejidad. El objetivo principal consiste en desarrollar un programa utilizando Spring Boot, poniendo en práctica los conocimientos y habilidades desarrolladas en la primera parte.

La aplicación a desarrollar tendrá como finalidad la gestión de planos arquitectónicos para una prestigiosa compañía de diseño. A continuación, se detallan las funcionalidades que deberá implementar la aplicación:

- Búsqueda: El programa debe permitir realizar búsquedas de planos arquitectónicos por diversos criterios, como por ejemplo, nombre del plano y autor.



UNIVERSIDAD

- Visualización: El programa debe permitir visualizar los puntos de los planos arquitectónicos de forma individual o en conjunto.
- Filtros: El programa debe permitir pasar los planos por filtros que modifican los puntos de cada plano según el filtro.

Desarrollo del Laboratorio

- Parte 1 :

Taller – Principio de Inversión de dependencias, Contenedores Livianos e Inyección de dependencias.

1. Haciendo uso de la configuración de Spring basada en anotaciones marque con las anotaciones `@Autowired` y `@Service` las dependencias que deben inyectarse, y los 'beans' candidatos a ser inyectados -respectivamente:
- GrammarChecker será un bean, que tiene como dependencia algo de tipo 'SpellChecker'.

```
no usages = 1 method  
  
@Service  
public class GrammarChecker {  
  
    @Autowired  
    SpellChecker sc;  
  
    no usages  
    String x;  
}
```

Le ponemos la etiqueta `@service` a GrammarChecker para definir que será un Bean y le inyectamos el SpellChecker con la etiqueta `@Autowired`.

- EnglishSpellChecker y SpanishSpellChecker son los dos posibles candidatos a ser inyectados. Se debe seleccionar uno, u otro, más NO ambos (habría conflicto de resolución de dependencias). Por ahora haga que se use EnglishSpellChecker.

```
@Service
public class EnglishSpellChecker implements SpellChecker {

    1 usage  ± Hector
    @Override
    > public String checkSpell(String text) { return "Checked with english checker:"+text; }
}
```

Al usar la etiqueta de `@Service` en `EnglishSpellChecker` podemos decir que este es el `SpellChecker` que se usará.

- Haga un programa de prueba, donde se cree una instancia de `GrammarChecker` mediante Spring, y se haga uso de la misma:

```
INFORMACIÓN: Loading XML bean definitions from class path resource [applicationContext.xml]
Spell checking output:Checked with english checker:la la la Plagiarism checking output: Not available yet
```

De esta manera comprobamos que al momento de correr el main del programa se está usando `EnglishSpellChecker`.

- Modifique la configuración con anotaciones para que el Bean '`GrammarChecker`' ahora haga uso de la clase `SpanishSpellChecker` (para que a `GrammarChecker` se le inyecte `EnglishSpellChecker` en lugar de `SpanishSpellChecker`. Verifique el nuevo resultado.

```
@Service
public class SpanishSpellChecker implements SpellChecker {

    1 usage  ± Hector
    @Override
    public String checkSpell(String text) {
        return "revisando (" + text + ") con el verificador de sintaxis del español";
    }
}
```

```
INFORMACIÓN: Loading XML bean definitions from class path resource [applicationContext.xml]
Spell checking output:revisando (la la la ) con el verificador de sintaxis del españolPlagiarism checking output: Not available yet
```

De esta forma observamos que al quitarle la etiqueta `@Service` a `EnglishSpellChecker` y poniéndosela a `SpanishSpellChecker` Spring Boot identifica que este es el que se va a inyectar.

- Parte 2 :

Componentes y conectores - Parte I.

1. Configure la aplicación para que funcione bajo un esquema de inyección de dependencias, tal como se muestra en el diagrama anterior.

Lo anterior requiere:

- Agregar las dependencias de Spring.
- Agregar la configuración de Spring.
- Configurar la aplicación -mediante anotaciones- para que el esquema de persistencia sea inyectado al momento de ser creado el bean 'BlueprintServices'.

```
@Service
public class BlueprintsServices {

    @Autowired
    @Qualifier(value = "inMemoryBluePrintPersistence")
    BlueprintsPersistence bpp;
```

Colocamos la etiqueta `@Service` para definir `BluePrintServices`

```
@Component
@Qualifier("inMemoryBluePrintPersistence")
public class InMemoryBlueprintPersistence implements BlueprintsPersistence{
```

Colocamos la etiqueta de `@Component` en `InMemoryBluePrintPersistence` para cargarla en el contenedor de spring Boot.

Además agregamos la etiqueta de `@Qualifier` para identificar qué tipo de `BluePrintPersistence` es.

UNIVERSIDAD

2. Complete las operaciones `getBlueprint()` y `getBlueprintsByAuthor()`.
Implemente todo lo requerido de las capas inferiores (por ahora, el esquema de persistencia disponible 'InMemoryBlueprintPersistence') agregando las pruebas correspondientes en 'InMemoryPersistenceTest'.

```
2 usages new *
public void addNewBlueprint(Blueprint bp) throws BlueprintPersistenceException {
    bpp.saveBlueprint(bp);
}

1 usage new *
public Set<Blueprint> getAllBlueprints() throws BlueprintNotFoundException {
    return bpp.getBlueprint();
}
```

Modificamos los métodos de `addNewBlueprint` y `getAllBlueprints`.

```
public List<Blueprint> getBlueprint() throws BlueprintNotFoundException;
no usages 1 implementation new *
public List<Blueprint> getBlueprintByAuthor(String author) throws BlueprintNotFoundException;
```

Creamos los métodos en `BlueprintServices` :

- `getBlueprint()`
- `getBlueprintsByAuthor()`

```
2 usages new *
@Override
public Set<Blueprint> getBlueprint() throws BlueprintNotFoundException {
    Set<Blueprint> blueprintList = new HashSet<>();
    for(Tuple<String,String> key: blueprints.keySet()){
        blueprintList.add(blueprints.get(key));
    }
    return blueprintList;
}

1 usage new *
@Override
public Set<Blueprint> getBlueprintByAuthor(String author) throws BlueprintNotFoundException {
    Set<Blueprint> blueprintList = new HashSet<>();
    for(Tuple<String,String> key: blueprints.keySet()){
        if(blueprints.get(key).getAuthor().equals(author)){
            blueprintList.add(blueprints.get(key));
        }
    }
    return blueprintList;
}
```

Definimos el cuerpo de cada método en la clase `InMemoryBlueprintPersistence`

- Haga un programa en el que cree (mediante Spring) una instancia de BlueprintServices, y rectifique la funcionalidad del mismo: registrar planos, consultar planos, registrar planos específicos, etc.

```
@SpringBootApplication
public class BlueprintApplication implements CommandLineRunner {
    @Autowired
    BlueprintServices bs;

    public void main(String[] args){
        SpringApplication.run(BlueprintApplication.class, args);
    }

    @Override
    public void run(String... args) throws BlueprintPersistenceException, BlueprintNotFoundException {
        Point[] points = new Point[]{new Point(0, 2), new Point(2, 4), new Point(2, 0), new Point(4, 2)};
        int planos = 5;
        String author2 = "Andres Arias";
        String author1 = "Sebastian Blanco";
        for (int i = 0; i < planos; i++) {
            bs.addNewBlueprint(new Blueprint(author1, "plano " + i, points));
            bs.addNewBlueprint(new Blueprint(author2, "plano " + i, points));
        }

        System.out.println("-----PLANOS-----");
        System.out.println(bs.getAllBlueprints());
        System.out.println("-----PLANOS POR AUTOR-----");
        System.out.println("-----PLANOS DE ANDRES-----");
        System.out.println(bs.getBlueprintsByAuthor(author2));
        System.out.println("-----PLANOS DE SEBASTIAN-----");
        System.out.println(bs.getBlueprintsByAuthor(author1));
    }
}
```

Creamos la clase BlueprintApplication para simular el programa de gestión de planos arquitectónicos haciendo uso de la etiqueta @SpringBootApplication.

Sobrescribimos el método run para mostrar el comportamiento de : registrar y consultar los planos ya sea en general o por autor.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath/>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.3.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
</dependencies>

</project>
```


Agregamos las dependencias necesarias para poder correr la aplicación de Spring Boot.

```

/Users/andresfelig/Library/Java/JavaVirtualMachines/corretto-17.0.8.1/Contents/Home/bin/java ...

      _   _          _ __| |__ 
     / \   \         / //_/ \| | |
    / ___\___\       /_//_\_| |_|
   /_____\___\_____/_//_\_| |_|
   / ____ \___\ _____/_//_\_| |_|
  /_____\___\_____/_//_\_| |_|
=====|=====|=====/./././

:: Spring Boot ::                (v2.0.1.RELEASE)


2024-02-21 01:09:00.998 INFO 6426 --- [           main] e.e.a.blueprints.BluePrintApplication : Starting BluePrintApplication on 192.168.88.25 with PID 6426 (star
2024-02-21 01:09:01.803 INFO 6426 --- [           main] e.e.a.blueprints.BluePrintApplication : No active profile set, falling back to default profiles: default
2024-02-21 01:09:02.487 INFO 6426 --- [           main] e.e.a.blueprints.BluePrintApplication : Started BluePrintApplication in 2.279 seconds (JVM running for 3.3

-----PLANOS-----
[Blueprint{author=andresname_, name=bpname_}, Blueprint{author=Andres Arias, name=piano 4}, Blueprint{author=Sebastian Blanco, name=piano 4}, Blueprint{author=Andre
-----PLANOS POR AUTOR-----
-----PLANOS DE ANDRES-----
[Blueprint{author=Andres Arias, name=piano 4}, Blueprint{author=Andres Arias, name=piano 0}, Blueprint{author=Andres Arias, name=piano 1}, Blueprint{author=Andres Ari
-----PLANOS DE SEBASTIAN-----
[Blueprint{author=Sebastian Blanco, name=piano 4}, Blueprint{author=Sebastian Blanco, name=piano 3}, Blueprint{author=Sebastian Blanco, name=piano 2}, Blueprint{autho
```

Corremos la aplicación y observamos que obtenemos las salidas esperadas.

- Se quiere que las operaciones de consulta de planos realicen un proceso de filtrado, antes de retornar los planos consultados. Dichos filtros lo que buscan es reducir el tamaño de los planos, removiendo datos redundantes o simplemente submuestrando, antes de retornarlos. Ajuste la aplicación (agregando las abstracciones e implementaciones que considere) para que a la clase `BlueprintServices` se le inyecte uno de dos posibles 'filtros' (o eventuales futuros filtros). No se contempla el uso de más de uno a la vez:

```
package edu.eci.arsw.blueprints.persistence;

import java.util.Set;

import org.springframework.stereotype.Service;
import edu.eci.arsw.blueprints.model.Blueprint;

4 usages 2 implementations
public interface Filters {
    3 usages 2 implementations
    public void filterBlueprint(Blueprint bp) throws BlueprintNotFoundException;
    1 usage 2 implementations
    public void filterBlueprints(Set<Blueprint> bps) throws BlueprintNotFoundException;
}
```

Implementamos una interfaz llamada `Filters` la cual cuenta con dos métodos:

- Uno que recibe un solo mapa para filtrarlo
- Otro que recibe un conjunto de mapas para filtrarlos

UNIVERSIDAD

- (A) Filtrado de redundancias: suprime del plano los puntos consecutivos que sean repetidos.

```
@Component
@Qualifier("RedundancyFilter")
public class RedundancyFilter implements Filters {

    1 usage
    public void removePoints(Blueprint bp, Point point) {
        List<Point> points = new ArrayList<Point>(bp.getPoints());
        for (int i = 0; i <= points.size() - 1; i++) {
            if (point.equals(points.get(i))) {
                points.remove(i);
            }
        }
        points.add(point);
        bp.setPoints(points);
    }

    3 usages
    @Override
    public void filterBlueprint(Blueprint bp) throws BlueprintNotFoundException {
        for (Point point : bp.getPoints()) {
            removePoints(bp, point);
        }
    }

    1 usage
    @Override
    public void filterBlueprints(Set<Blueprint> blueprints) throws BlueprintNotFoundException {
        for (Blueprint blueprint : blueprints) {
            filterBlueprint(blueprint);
        }
    }
}
```

Implementamos uno de los filtros, y desarrollamos el cuerpo de cada método según su comportamiento.

- (B) Filtrado de submuestreo: suprime 1 de cada 2 puntos del plano, de manera intercalada.

```
@Component
@Qualifier("SubsamplingFilter")
public class SubsamplingFilter implements Filters {

    1 usage
    public void subsampling(Blueprint bp) throws BlueprintNotFoundException {
        List<Point> points = new ArrayList<Point>(bp.getPoints());
        List<Point> pointsFilter = new ArrayList<Point>();
        for (int i = 0; i < points.size(); i++) {
            if (i % 2 == 0) {
                pointsFilter.add(points.get(i));
            }
        }
        bp.setPoints(pointsFilter);
    }

    3 usages
    @Override
    public void filterBlueprint(Blueprint bp) throws BlueprintNotFoundException {
        subsampling(bp);
    }

    1 usage
    @Override
    public void filterBlueprints(Set<Blueprint> bps) throws BlueprintNotFoundException {
        for (Blueprint blueprint : bps) {
            filterBlueprint(blueprint);
        }
    }
}
```

Implementamos el filtro faltante, y desarrollamos el cuerpo de cada método según su comportamiento.

```
1 usage
@Service
public class FilterServices {
    @Autowired
    @Qualifier("subsamplingFilter")
    Filters filter;

    1 usage
    public void filterBlueprint(Blueprint bp) throws BlueprintNotFoundException {
        filter.filterBlueprint(bp);
    }

    1 usage
    public void filterBlueprints(Set<Blueprint> bps)
        throws BlueprintNotFoundException, BlueprintPersistenceException {
        filter.filterBlueprints(bps);
    }
}
```

Creamos la clase FilterServices la cual es la que se va a inyectar en BlueprintServices, y es la encargada de los servicios de la interfaz Filters.

```
25 @Override
26 public void run(String... args) throws Exception{
27     Point[] points = new Point[]{new Point(0, 2), new Point(2, 4), new Point(2, 0), new Point(4, 2)};
28     int planos = 5;
29     String author1 = "Andres Arias";
30     String author2 = "Sebastian Blanco";
31     for (int i = 0; i < planos; i++) {
32         bs.addNewBlueprint(new Blueprint(author1, name: "plano " + i, points));
33         bs.addNewBlueprint(new Blueprint(author2, name: "plano " + i, points));
34     }
35     //System.out.println("-----PLANOS-----");
36 }

Run BluePrintApplication
Console
2024-02-21 02:32:41.968 INFO 7395 --- [main] e.e.a.blueprints.BluePrintApplication : Starting BluePrintApplication on 192.168.88.25 with PID 7395 (star
2024-02-21 02:32:41.975 INFO 7395 --- [main] e.e.a.blueprints.BluePrintApplication : No active profile set, falling back to default profiles: default
2024-02-21 02:32:42.967 INFO 7395 --- [main] e.e.a.blueprints.BluePrintApplication : Started BluePrintApplication in 1.549 seconds (JVM running for 2.6
-----PLANOS POR AUTOR-----
Blueprint{author:Andres Arias, name:plano 0,Puntos(0, 2), (2, 0)}
Blueprint{author:Andres Arias, name:plano 1,Puntos(0, 2), (2, 0)}
Blueprint{author:Andres Arias, name:plano 2,Puntos(0, 2), (2, 0)}
Blueprint{author:Andres Arias, name:plano 3,Puntos(0, 2), (2, 0)}
Blueprint{author:Andres Arias, name:plano 4,Puntos(0, 2), (2, 0)}
```

En este caso se usó el filtro de submuestreo: suprime 1 de cada 2 puntos del plano, de manera intercalada.

Teníamos los puntos: $\{(0,2),(2,4),(2,0),(4,2)\}$

Y su salida por cada plano fue : $\{(0,2),(2,0)\}$, de esta forma identificamos que el filtro realiza su función correctamente.

```
@Override
public void run(String... args) throws Exception {
    Point[] points = new Point[]{new Point(0, 2), new Point(2, 4), new Point(2, 0), new Point(4, 2)};
    int planos = 1;
    String author1 = "Andres Arias";
    String author2 = "Sebastian Blanco";
    for (int i = 0; i < planos; i++) {
        bs.addNewBlueprint(new Blueprint(author1, name: "plano " + i, points));
        bs.addNewBlueprint(new Blueprint(author2, name: "plano " + i, points));
    }

    //System.out.println("-----PLANOS-----");
    //System.out.println(bs.getAllBlueprints());
}

BlueprintApplication
Actuator
[INFO] 7414 --- [main] e.e.a.Blueprints.BluePrintApplication : Starting BlueprintApplication on 192.168.88.25 with PID 7414 (sta
[INFO] 7414 --- [main] e.e.a.Blueprints.BluePrintApplication : No active profile set, falling back to default profiles: default
[INFO] 7414 --- [main] e.e.a.Blueprints.BluePrintApplication : Started BlueprintApplication in 1.229 seconds (JVM running for 2.6
-----PLANOS POR AUTOR-----
-----PLANOS DE ANDRES-----
Blueprint(author=Andres Arias, name=plano 3)Puntos(0, 2), (2, 4), (2, 0), (4, 2)
Blueprint(author=Andres Arias, name=plano 4)Puntos(0, 2), (2, 4), (2, 0), (4, 2)
Blueprint(author=Andres Arias, name=plano 0)Puntos(0, 2), (2, 4), (2, 0), (4, 2)
Blueprint(author=Andres Arias, name=plano 1)Puntos(0, 2), (2, 4), (2, 0), (4, 2)
Blueprint(author=Andres Arias, name=plano 2)Puntos(0, 2), (2, 4), (2, 0), (4, 2)
-----PLANOS DE SEBASTIAN-----
[Blueprint(author=Sebastian Blanco, name=plano 4), Blueprint(author=Sebastian Blanco, name=plano 3), Blueprint(author=Sebastian Blanco, name=plano 2), Blueprint(auth
Process finished with exit code 0
```

En este caso se uso el Filtro de redundancias: suprime del plano los puntos consecutivos que sean repetidos.

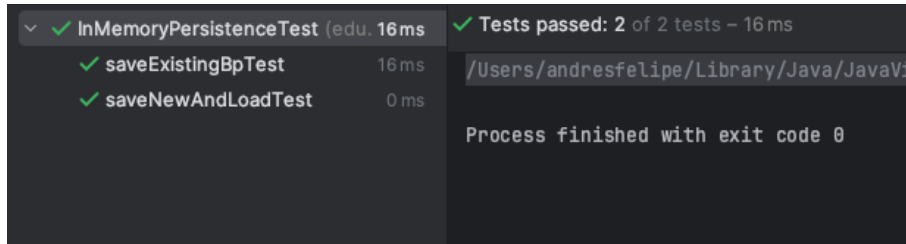
Teníamos los puntos: $\{(0,2),(2,4),(2,0),(4,2)\}$

Y su salida por cada plano fue : $\{(0,2),(2,4),(2,0),(4,2)\}$ de esta forma identificamos que el filtro realiza su función correctamente, ya que no habían puntos repetidos.

5. Agregue las pruebas correspondientes a cada uno de estos filtros, y pruebe su funcionamiento en el programa de prueba, comprobando que sólo cambiando la posición de las anotaciones -sin cambiar nada más-, el programa retorne los planos filtrados de la manera (A) o de la manera (B).

```
✓ shouldRedundancyFilterElimin 9 ms
✓ shouldSubsamplingFilterElimin 2 ms
✓ shouldRedundancyFilterElimin 0 ms
✓ shouldSubsamplingFilterElimin 2 ms
/Users/andresfelipe/Library/Java/JavaVirtu
Process finished with exit code 0
```

Corremos los test de filters



```
✓ InMemoryPersistenceTest (edu. 16 ms) ✓ Tests passed: 2 of 2 tests - 16 ms
  ✓ saveExistingBpTest 16 ms
  ✓ saveNewAndLoadTest 0 ms
/Users/andresfelipe/Library/Java/JavaVi
Process finished with exit code 0
```

Corremos los test de InMemoryBlueprintPersistence

Conclusiones

- La aplicación del Principio de Inversión de Dependencias y la Inyección de dependencias permite crear código más flexible y desacoplado. Esto facilita la reutilización de componentes, la prueba unitaria y la evolución del software a lo largo del tiempo.
- Los Componentes y Conectores bien diseñados encapsulan la funcionalidad y las dependencias, lo que permite un desarrollo modular y facilita el mantenimiento del código.
- Los Framework como Spring Boot permiten crear aplicaciones escalables y fáciles de mantener. Se encargan de la gestión del ciclo de vida de los componentes, la configuración y el despliegue.

Bibliografía

- (1) Documenting Software architectures: Sección 3.2 (vistas)
 - (2) Documenting Software architectures: 4.3 (call-return)
 - (2) Documenting Software architectures: Secciones 4.3.1, 4.3.2, 4.3.3 (casos)
- Para el laboratorio: <http://www.drdoobbs.com/web-development/soa-web-services-and-restful-systems/199902676?pgno=1> (sólo la página 1).