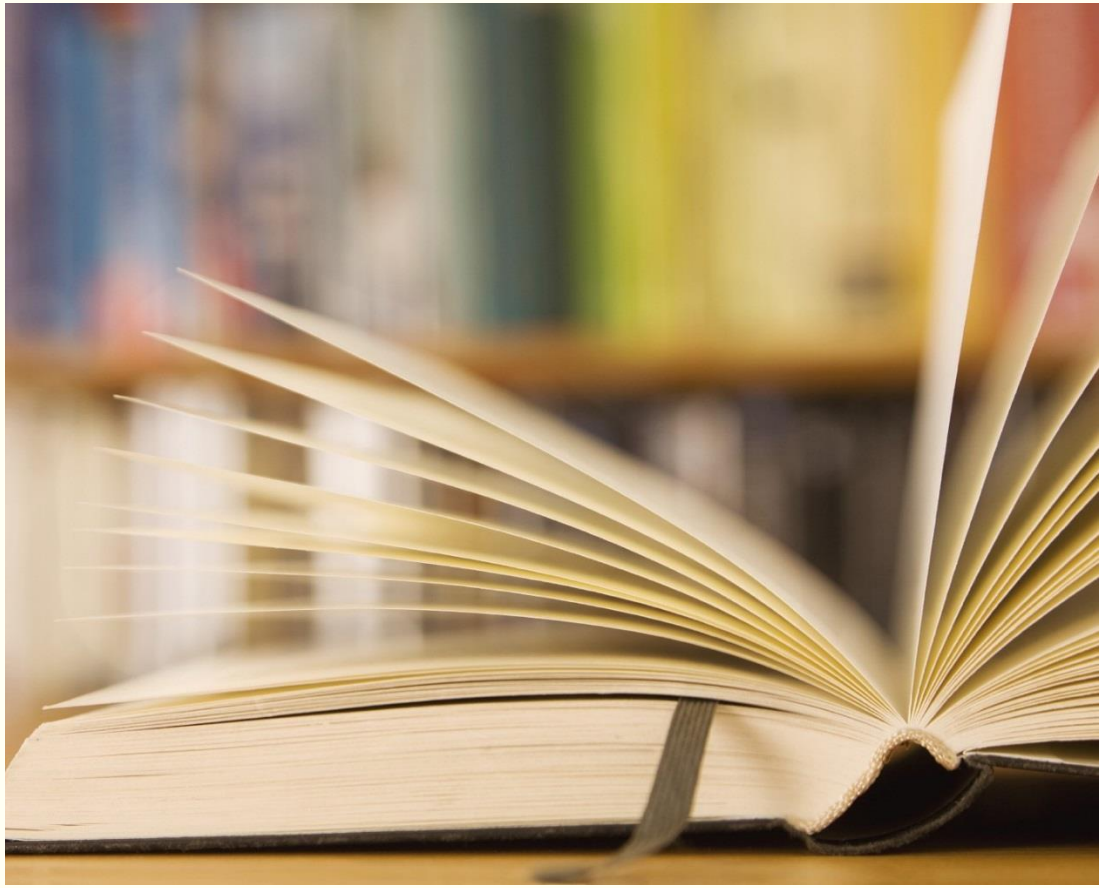
ENTERPRISE ARCHITECTURE

Najeeb Najeeb, PhD



LESSON 07 SPRING - CORE

What is Spring?

- Umbrella open-source framework.
- The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.
- A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments..

Benefits of Using Spring

- Popular (?)
- Promotes Loosely Coupled Architecture
- Eliminate Cross-Cutting Concerns
- Easily Testable Applications
- Light Weight and Feature Rich (?)
- Spring is an eco-system, a lot of projects
  - https://spring.io/projects
  - Start with Spring – Core
    - Dependency Injection
    - Inversion of Control
    - Aspect Oriented Programming
  - Templating

Setup Java & Maven

- Install and add Java to environment variables
  - JAVA_HOME = location of your jdk
  - PATH add %JAVA_HOME%/bin
- Download Maven
  - https://maven.apache.org/download.cgi
  - Unzip in folder
- Add Maven to Environment Variables
  - M2_HOME= location of your maven folder
  - M2= %M2_HOME%\bin
  - PATH add M2
- Test things
  - java -version
  - mvn -version

SEPARATION OF CONCERNS

P2C – Tightly Coupled

Main

```
public static void main() {
        Game game= new Game(new Car());
        game.play();
}

public class Game() {
        private Car car;
        public Game(Car car) {
                this.car= car;
        }
        public void play() {
                car.move();
        }
}

// Code change to switch to Bike
```

Car, Bike

```
class Car implements Vehicle {
        public void move() {
                System.out.println("moving at 50
mph");
        }
}

class Bike implements Vehicle {
        public void move() {
                System.out.println("moving at 10
mph");
        }
}
```

P2I – Loosely Coupled

Main

```
public static void main() {
	Game game= new Game(new Car());
	game.play();
}

public class Game() {
	private Vehicle vehicle;
	public Game(Vehicle vehicle) {
		this.vehicle= vehicle;
	}
	public void play() {
		vehicle.move();
	}
}

// less code change to switch to Bike
```

Car, Bike, Vehicle

```
interface Vehicle {
	public void move();
}

class Car implements Vehicle {
	public void move() {
		System.out.println("moving at 50
mph");
	}
}

class Bike implements Vehicle {
	public void move() {
		System.out.println("moving at 10
mph");
	}
}
```

P2I + Factory

Main

```
public static void main() {
	Game game= new Game(VehicleFactory.getVehicle());
	game.play();
}

public class Game() {
	private Vehicle vehicle;
	public Game(Vehicle vehicle){
		this.vehicle= vehicle;
	}
	public void play(){
		vehicle.move();
	}
}

//  switch to Bike with no code change
```

Car, Bike, Vehicle

```
interface Vehicle {
	public void move();
}

class Car implements Vehicle {
	public void move() {
		System.out.println("moving at 50 mph");
	}
}

class Bike implements Vehicle {
	public void move() {
		System.out.println("moving at 10 mph");
	}
}

class VehicleFactory{
	public static Vehicle getVehicle(){
		return new Car();
	}
}
```

SIMPLE SPRING DI APPLICATION

Spring

Vehicle, Car, Bike

```
interface Vehicle {
	public void move();
}

class Car implements Vehicle {
	public void move() {
		System.out.println("moving at 50
mph");
	}
}

class Bike implements Vehicle {
	public void move() {
		System.out.println("moving at 10
mph");
	}
}
```

pom.xml

```
<dependencies>
		<dependency>
			<groupId>org.springframework</
groupId>
			<artifactId>spring-
context</artifactId>
			<version>5.3.15</version>
		</dependency>
</dependencies>
```

Spring

Main

```
public static void main() {
	ApplicationContext springContext= new
ClassPathXmlApplicationContext("config.xml");
	Game game= (Game) springContext.getBean("game");
	Game game= springContext.getBean(Game.class);
	game.play();
}

public class Game() {
	private Vehicle vehicle;
	public Game(Vehicle vehicle){
		this.vehicle= vehicle;
	}
	public void play(){
		vehicle.move();
	}
}

// No Game or Vehicle is created in our code
```

config.xml

```
<bean id="vehicle"
 class="edu.miu.cs544.najeeb.vehicle.Car"></bean>

<bean id="game"
 class="edu.miu.cs544.najeeb.game.Game">
	<constructor-arg ref="vehicle" /></bean>
```

SPRING CONFIGURATION

Xml Configuration

Main

```
ApplicationContext springContext= new
ClassPathXmlApplicationContext("config.x
ml");
```

Config.xml

```
<bean id="vehicle"
class="edu.miu.cs544.najeeb.vehicle.Car
"></bean>

<bean id="game"
class="edu.miu.cs544.najeeb.game.Gam
e">
      <constructor-arg ref="vehicle"
/></bean>
```

Java Class

Main

```
ApplicationContext springContext= new
AnnotationConfigApplicationContext(Sprin
gConfig.class);
```

SpringConfig.java

SpringConfig.java

```
@Configuration
public class SpringConfig {
    @Bean
    public Vehicle vehicle() {
        return new Car();
    }
    @Bean
    public Game game() {
        return new game(vehicle());
    }
}
```

Multi-Config Files

Main

```
ApplicationContext springContext=
new ClassPathXmlApplicationContext("con
fig.xml");
```

Xml

config.xml

```
<import source="config2.xml">
<bean id="game" class="edu.miu.cs544.na
jeeb.game.Game">
        <constructor-
arg ref="vehicle" /></bean>
```

config2.xml

```
<bean
   id="vehicle" class="edu.miu.cs544.najee
   b.vehicle.Car"></bean>
```

Multi-Config Files

Main

```
ApplicationContext springContext= new
AnnotationConfigApplicationContext(SpringConfig.cl
ass);
```

Java

SpringConfig.java

```
@Configuration
@Import(SpringConfig2.class)
public class SpringConfig{
    @Bean
    public Game game(Vehicle vehicle){
        return new game(vehicle);
    }
}
```

SpringConfig2.java

```
@Configuration
  public class SpringConfig2{
      @Bean
      public Vehicle vehicle(){
          return new Car();
      }
  }
```

Mixed Configuration

Main

```
ApplicationContext springContext= new
AnnotationConfigApplicationContext(Spring
Config.class);
```

Configuration

SpringConfig.java

```
@Configuration
@ImportResource("classpath:config2.xml")
public class SpringConfig {
    @Bean
    public Game game(Vehicle vehicle) {
        return new game(vehicle);
    }
}
```

config2.xml

```
<bean id="vehicle" class="edu.miu.cs544.naj
  eeb.vehicle.Car"></bean>
```

Mixed Configuration

Main

```
ApplicationContext springContext= new ClassPathXmlApplicationContext("config.xml");
```

Configuration

config.xml

```
xmlns:context="http://www.springframework.org/schema/context"
...
xsi:schemaLocation= ...
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd

<context:annotation-config />
        <bean name="springconfig"
    class="edu.miu.cs544.najeeb.config.SpringConfig2" />
        <bean id="game" class="edu.miu.cs544.najeeb.game.Game">
            <constructor-arg ref="vehicle" />
        </bean>
```

SpringConfig2.java

```
@Configuration
    public class SpringConfig2 {
        @Bean
        public Vehicle  vehicle() {
            return new Car();
        }
    }
```

Bean Factory

- Application Context
- Bean Factory
- Light weight factory without all the context features. Can only work with xml configuration.

```
DefaultListableBeanFactory   springFactory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader xmlReader = new XmlBeanDefinitionReader(springFactory);
xmlReader.loadBeanDefinitions(new ClassPathResource("config3.xml"));

Game game= springFactory.getBean(Game.class);
```

BEAN INITIALIZATION

Spring

Code

```
main
	ApplicationContext springContext= new
ClassPathXmlApplicationContext("config.xml");
	Game game=
springContext.getBean(Game.class);
	game.play();
}

Car
public class  Car() {
	...
	public Car(int year, String make, String model, int
millage) {
		this.year = year;
		this.make = make;
		this.model = model;
		this.millage = millage;
	}
	//  toString
}
```

config.xml

```
<bean id="vehicle"
class="edu.miu.cs544.najeeb.vehicle.Car
">
		<constructor-arg type="int"
value="2020" />
		<constructor-arg type="String"
value="Toyota" />
		<constructor-arg type="String"
value="Prius" />
		<constructor-arg type="int"
value="55000" />
</bean>
```

Spring

Code

```
main
	ApplicationContext springContext= new
AnnotationConfigApplicationContext(SpringConfig.cl
ass);
	Game game=
springContext.getBean(Game.class);
	game.play();
}

Car
public class Car() {
	...
	public Car(int year, String make, String model, int
millage) {
		this.year = year;
		this.make = make;
		this.model = model;
		this.millage = millage;
	}
	// toString
}
```

SpringConfig.java

```
@Bean
  public Vehicle vehicle() {
		return new Car(2020, "Toyota",
  "Prius", 66000);
  }

@Bean(name = "game")
  public Game myGame(Vehicle vehicle) {
		return new Game(vehicle);
  }
```

Benefits

- We do not know what object is being created and not knowing it's state.
- This can be applied to connections to other libraries and their configurations.
  - Using an external library and setting it up is done outside of your application code.

Setter Injection

Code

```
main
	ApplicationContext springContext= new ClassPathXmlApplicationContext("config.xml");
	Game game= springContext.getBean(Game.class);
	game.play();
}

Bike
public class Bike() {
	private int tireSize;
	private String brand;
	...
	// getters & setters
	// toString
}
```

config.xml

```
<bean id="vehicle" class="edu.miu.cs544.najeeb.vehicle.Bike">
	<property name="tireSize" value="16" />
	<property name="brand" value="BMX" />
</bean>
```

Setter Injection

Code

```
main
	AnnotationConfigApplicationContext(Sp
ringConfig.class);
	Game game=
springContext.getBean(Game.class);
	game.play();
}

Bike
public class Bike() {
	private int tireSize;
	private String brand;
	…
	// getters & setters
	// toString
}
```

SpringConfig.java

```
@Bean
public Vehicle vehicle() {
	Bike bike= new Bike();
	bike.setTireSize(18);
	bike.setBrand("BMX");
	return bike;
}
```

Constructor VS Setter

Constructor

Must provide all the parameters

Used for Immutable objects

Setter

Can work without setting all parameters

Cannot be used for Immutable objects

Mixed initialization

- Using both constructor and setter initialization
- Bike class

```
private int tireSize;
private String brand;
public Bike(String brand) {
        this.brand = brand;
}
//  setters & getters
```

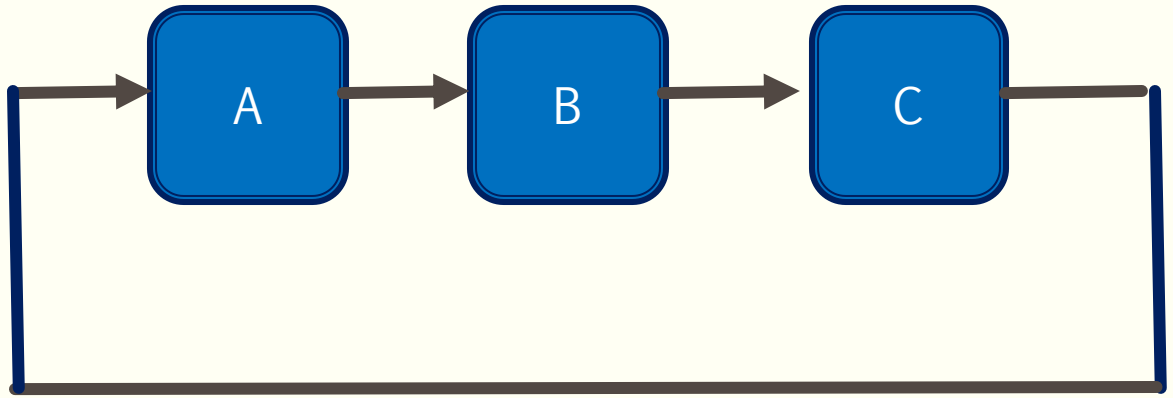
- SpringConfig.xml

```
<bean id="vehicle" class="edu.miu.cs544.najeeb.vehicle.Bike">
        <constructor-arg type="String" value="BMX" />
        <property name="tireSize" value="17" />
</bean>
```

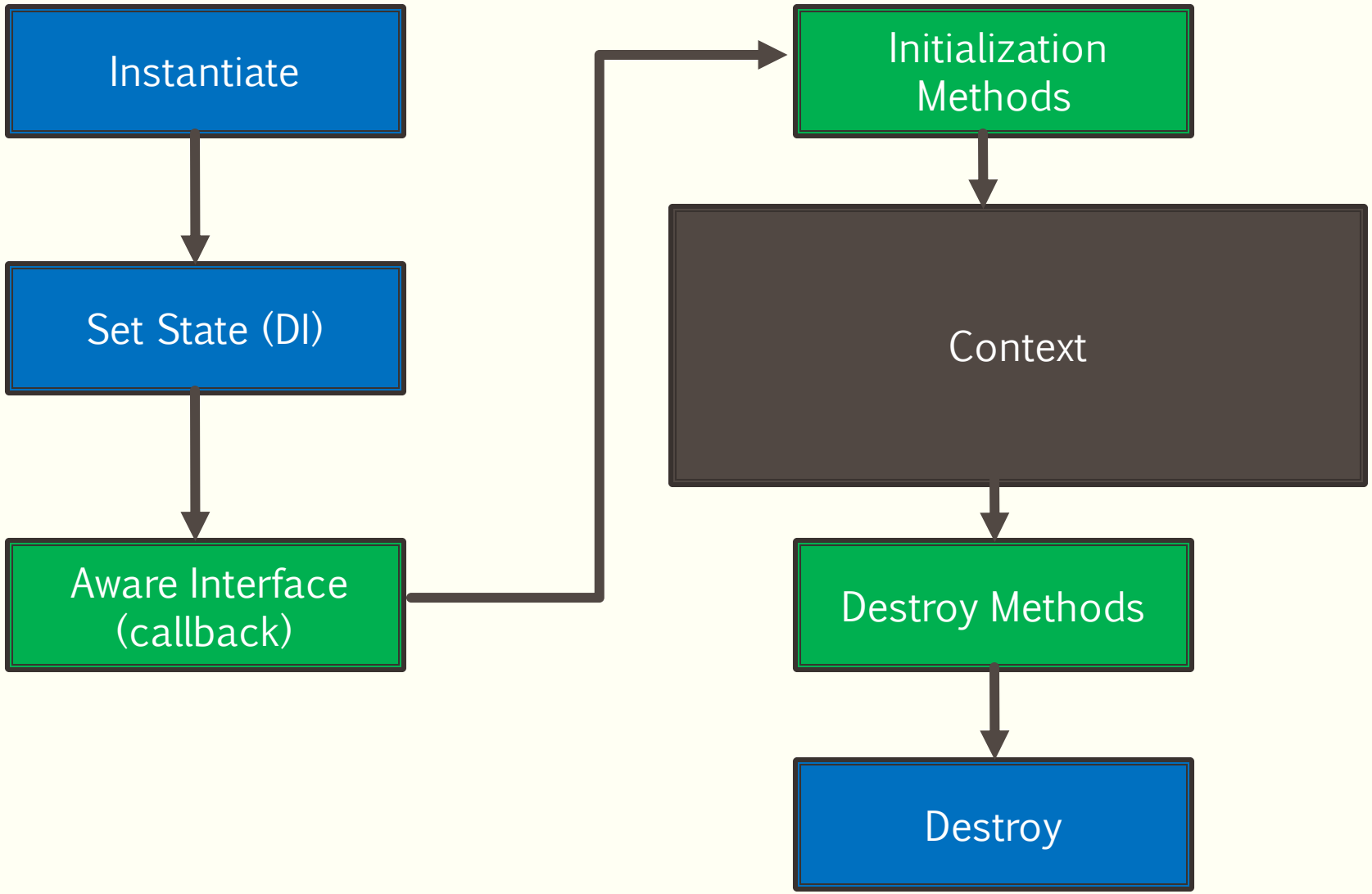
BEAN LIFECYCLE

DI Check & Egg Problem

- Assume Constructor DI
- How can you create an instance?
- A constructor needs a B
- B constructor needs a C
- C constructor needs an A
- How can this problem be solved?



Bean Lifecycle



Initialization Methods

- Implement interface InitializingBean

```
@Override
public void afterPropertiesSet() throws Exception {      }
```

- Coupled to Spring :(
- Configure in config.xml
  - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" init-method="init" >
- Java JavaConfig.java
  - @Bean(initMethod="init")
- JEE provides a Specification JSR-250
  - @PostConstruct

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Destroy Methods

- Implement interface DisposableBean

```
@Override
public void destroy() throws Exception {       }
```

- Coupled to Spring :(
- Configure in config.xml
  - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" destroy-method="destroy">
- Java JavaConfig.java
  - @Bean(destroyMethod= "destroy")
- JEE provides a Specification JSR-250
  - @PreDestroy

BeanPostProcessor

Steps

- Create a bean
- Implement interface BeanPostProcessor
- Override default implementations of
  - public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException
  - public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException
- Make sure the methods return bean

Configuration

- Xml
  - <bean id="postProcessor" class="edu.miu.cs544.najeeb.addons.MyBeanPostProcessor" ></bean>
- JavaCode
  - @Bean
    Public MyBeanPostProcessor myBeanPostProcessor( ) {
    return new MyBeanPostProcessor();
    }

Define init and destroy for all beans

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
        default-init-method="init" default-destroy-method="destroy">
```

Initialization sequence

Execution order

- Dependency dependent
- Order of declaration when no dependencies

config

```
<bean id="game"
   class="edu.miu.cs544.najeeb.game.Gam
   e">

      <constructor-arg ref="vehicle" />

</bean>

<bean id="vehicle"
class="edu.miu.cs544.najeeb.vehicle.Car">
</bean>
```

AWARE INTERFACE

Bean Name Aware

Bean

```
public class Book implements
BeanNameAware {
        private String beanName;
        @Override
        public void setBeanName(String s) {
                this.beanName= s;
        }
}
```

Spring performs the Injection

Coupled to Spring

Configuration

- Xml
  - <bean id="book" class="edu.miu.cs544.najeeb.awareInterfaces.Book">
- Java

```
@Bean(name="MyBook")
        public Book book() {
                return new Book();
        }
```

Resource Loader Aware

Bean

```
public class Book implements
ResourceLoaderAware {
        private ResourceLoader resourceLoader;
        @Override
        public
void setResourceLoader(ResourceLoader
resourceLoader) {
                this.resourceLoader= resourceLoader;
        }
}
```

Spring performs the Injection

Coupled to Spring

Configuration

- Xml
  - <bean id="book" class="edu.miu.cs544.najeeb.awareInterfaces.Book" >
- Java

```
@Bean(name="MyBook")
        public Book book(){
                return new Book();
        }
```

Application Context Aware

Bean

```
public class Book implements
ApplicaitonContextAware {
        private ApplicationContext
applicationContext;
        @Override
        public
void setApplicationContext(ApplicationContext
applicationContext) throws BeansException {
                this.applicationContext=
applicationContext;
        }
}
```

Spring performs the Injection

Coupled to Spring

Configuration

- Xml
  - <bean id="book" class="edu.miu.cs544.najeeb.awareInterfaces.Book" >
- Java

```
@Bean(name="MyBook")
        public Book book(){
                return new Book();
        }
```

Aware Interfaces

- Aware Interface is the parent of all the Spring Aware Interfaces
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/Aware.html

BEAN SCOPE

Singleton

- Reduce Creation Cost
  - Time
  - Memory
- Stateless Class
- Equivalent to Stateless EJB
- Example: DB_Connection
- Global Variables ?? :(
- Lazy VS Eager
- Unique per JVM

Spring Singleton Bean

- Default Types of Beans
- Your job is to make them thread safe if they have attributes
- Unique per Spring Container per Bean Type
- Xml
  - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" scope="singleton">
- Java
  - @Scope(value="singleton")
- Loading
  - Eager
    - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" lazy-init="false">
  - Lazy
    - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" lazy-init="true">
- Applying loading strategy to all beans
  - Use default-lazy-init in beans tag
  - @Lazy on class level

Spring Prototype Bean

- New instance created for each bean request
- Do not use Singleton when you have state
- Xml
  - <bean id="game" class="edu.miu.cs544.najeeb.game.Game" scope="prototype">
- Java
  - @Scope(value= "prototype")

Prototype Disclaimer

- This is still the case even in Spring
- In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean. The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.
- In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java `new` operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see Lifecycle Callbacks.)

Create Your Own Destroy

- Implement DisposableBean in the prototype bean
- Create a Singleton bean to manage the destruction
- Configure the Singleton bean with its own destroy method
- <bean id="prototypeDestroy" class="edu.miu.cs544.najeeb.vehicle.PrototypeDestroy" destroy-method="destroy" ></bean>

PrototypeDestroy

```
public class PrototypeDestroy implements
BeanPostProcessor, BeanFactoryAware,
DisposableBean {
	private BeanFactory beanFactory;
	private final List<Object> prototypeBeans=
new ArrayList<>();
	public PrototypeDestroy() {
		System.out.println("PrototypeDestroy
created");
	}
	public Object
postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException{
		return bean;
	}
	public Object
postProcessAfterInitialization(Object bean, String
beanName) throws BeansException{
		if (beanFactory.isPrototype(beanName)) {
			synchronized (prototypeBeans) {
				prototypeBeans.add(bean);
			}
		}
		return bean;
	}
	@Override
	public
void setBeanFactory(BeanFactory beanFactory)
throws BeansException {
		this.beanFactory= beanFactory;
	}
	@Override
	public void destroy() throws Exception {
		synchronized (prototypeBeans) {
			for (Object bean : prototypeBeans) {
				if
(bean instanceof DisposableBean) {
					DisposableBean disposable
Bean= (DisposableBean) bean;
					disposableBean.destroy();
				}
			}
			prototypeBeans.clear();
		}
	}
}
```

EJB BEANS

EJB Beans

- Singleton Beans
  - Very similar to Spring Singleton Bean but it is one per JVM
- Sateles Beans
  - Slightly better performance (Pool is used)
- Stateful Beans
  - Can have state
  - Bean per EJB session (one per user)
  - Singleton per user
- Much richer than Spring :) (but Spring could be much more efficient)
- Is it too much? Do we really need all these option?
  - Many developers and systems don't seem to need all this.
- Way more complex, hence used to be complex and slow
  - EJB 3.0 much lighter and faster (got rid of much legacy code)

METHOD INJECTION

DI Prototype in Singleton

- DI a prototype bean in a singleton bean
  - Make Game singleton
  - Make Vehicle prototype
  - Now make Game return the vehicle
  - If you use constructor DI, then you get the same vehicle each time :(
- Solution
  - Get rid of DI, and return a new Vehicle each time
  - You write return new Car(); :(
- Use ApplicationContextAware
  - Tightly coupled to Spring

```
// First case

public Vehicle getVehicle() {
        return vehicle;
    }

// Solution 1 P2C, Tight Coupling

public Vehicle getVehicle() {
        return new Car();
    }

// Solution 2

public Vehicle getVehicle() {
        return applicationContext.getBean(
Car.class);
    }
```

Method DI

- Make the get method abstract
- Make the Singleton abstract
- XML Configure the abstract method as a lookup-method

```
<bean id="vehicle"
class="edu.miu.cs544.najeeb.vehicle.Car"
scope="prototype" />
<bean id="game"
class="edu.miu.cs544.najeeb.game.Game"
scope="singleton">
    <lookup-method name="getVehicle"
bean="vehicle" />
</bean>
```

- Spring implements the abstract method to return the prototype bean

Java Configuration

```
    @Bean
    @Scope(value= "prototype")
    public Vehicle vehicle() {
        return new Car();
    }
    @Bean(name = "game")
    @Scope(value = "singleton")
    public Game myGame(Vehicle vehicle)
{
        return new Game() {
            @Override
            public Vehicle getVehicle() {
                return vehicle();
            }
        };
    }
```

AUTOWIRING

ByName

Game

```
public class Game {
        private Car car;
        private Bike bike;
        public Game() {       }
        public void setCar(Car car) {
                this.car = car;
        }
        public void setBike(Bike bike) {
                this.bike = bike;
        }
        public void play() {
                System.out.println("Game Started");
                car.move();
                bike.move();
                System.out.println("Game Ended");
        }
  }
```

XML

```
<bean id="car"
class="edu.miu.cs544.najeeb.vehicle.Car"></bean
>


 <bean id="bike"
class="edu.miu.cs544.najeeb.vehicle.Bike"></bean
>


<bean id="game"
class="edu.miu.cs544.najeeb.game.Game"
autowire="byName">
</bean>
```

ByType

Game

```
public class Game {
        private Car car;
        private Bike bike;
        public Game() {       }
        public void setCar(Car car) {
                this.car = car;
        }
        public void setBike(Bike bike) {
                this.bike = bike;
        }
        public void play() {
                System.out.println("Game Started");
                car.move();
                bike.move();
                System.out.println("Game Ended");
        }
  }
```

XML

```
<bean id="car"
class="edu.miu.cs544.najeeb.vehicle.Car"></bean
>


 <bean id="myBike"
class="edu.miu.cs544.najeeb.vehicle.Bike"></bean
>


<bean id="game"
class="edu.miu.cs544.najeeb.game.Game"
autowire="byType">
</bean>
```

JavaConfig (Deprecated)

ByName

```
@Configuration
public class SpringConfig {
	@Bean
	public Car car() {
		return new Car();
	}
	@Bean
	public Bike bike() {
		return new Bike();
	}
	@Bean(autowire = Autowire.BY_NAME)
	public Game game() {
		return new Game();
	}
}
```

ByType

```
@Configuration
  public class SpringConfig {
	@Bean
	public Car car() {
		return new Car();
	}
	@Bean
	public Bike bike() {
		return new Bike();
	}
	@Bean(autowire = Autowire.BY_TYPE)
	public Game game() {
		return new Game();
	}
  }
```

Annotations

XML

```
...
xmlns:context="http://www.springframework.org/schema/context"
...
xsi:schemaLocation="...
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
"

...

<context:annotation-config />
...
```

Game

```
public class Game {

    @Autowired
    private Car car;
    @Autowired
    private Bike bike;

...

//  Identify beans by type
```

Annotations

XML

```
...
<bean id="bike1"
class="edu.miu.cs544.najeeb.vehicle.Bike"
></bean>
<bean id="bike2"
class="edu.miu.cs544.najeeb.vehicle.Bike"
></bean>
```

Game

```
public class Game {

    @Autowired
    private Car car;
    @Autowired
    private Bike bike1;

...

//  Uses the qualifier
```

Explicit Qualifier

XML

```
...
<bean id="bike1"
class="edu.miu.cs544.najeeb.vehicle.Bike"
></bean>
<bean id="bike2"
class="edu.miu.cs544.najeeb.vehicle.Bike"
></bean>
```

Game

```
public class Game {

    @Autowired
    private Car car;
    @Autowired
    @Qualifier("bike1")
    private Bike bike;

...

//  Uses the qualifier
```

Java Config

Java Config

```
@Configuration
public class SpringConfig {
	@Bean
	public Car car() {
		return new Car();
	}
	@Bean
	public Bike bike() {
		return new Bike();
	}
	@Bean
	public Game game() {
		return new Game();
	}
}
```

Game

```
public class Game {

   @Autowired
   private Car car;
   @Autowired
   private Bike bike;

...

// Uses the qualifier
```

Constructor Autowire

Game

```
public class Game {
    private Car car;
    private Bike bike;
    public Game(){}
    public Game(Car car) {
        this.car= car;
    }
    public Game(Car car, Bike bike) {
        this.car= car;
        this.bike= bike;
    }

...
```

xml

```
<bean id="car"
    class="edu.miu.cs544.najeeb.vehicle.Car
    "></bean>

<bean id="bike"
    class="edu.miu.cs544.najeeb.vehicle.Bik
    e"></bean>

<bean id="game"
    class="edu.miu.cs544.najeeb.game.Gam
    e" autowire="constructor"></bean>
```

Constructor Autowire

Game

```
public class Game {
    private Car car;
    private Bike bike;
    public Game(){}
    public Game(Car car) {
        this.car= car;
    }
    @Autowired
    public Game(Car car, Bike bike) {
        this.car= car;
        this.bike= bike;
    }

...
```

xml

```
<bean id="car"
    class="edu.miu.cs544.najeeb.vehicle.Car
    "></bean>

<bean id="bike"
    class="edu.miu.cs544.najeeb.vehicle.Bik
    e"></bean>

<bean id="game"
    class="edu.miu.cs544.najeeb.game.Gam
    e" ></bean>
```

Constructor Autowire

Game

```
public class Game {
    private Car car;
    private Bike bike;
    public Game(){}
    public Game(Car car) {
        this.car= car;
    }
    @Autowired
    public Game(Car car,
@Qualifier(value="bike1") Bike bike) {
        this.car= car;
        this.bike= bike;
    }

...
```

xml

```
<bean id="car"
    class="edu.miu.cs544.najeeb.vehicle.Car
    "></bean>

<bean id="bike1"
    class="edu.miu.cs544.najeeb.vehicle.Bik
    e"></bean>

<bean
    id="bike2" class="edu.miu.cs544.najeeb.
    vehicle.Bike"></bean>

<bean id="game"
    class="edu.miu.cs544.najeeb.game.Gam
    e" ></bean>
```

Constructor Autowire

Game

```
public class Game {
    private Car car;
    private Bike bike;

    public Game(Car car,
@Qualifier(value="bike1") Bike bike) {
        this.car= car;
        this.bike= bike;
    }

...
```

xml

```
<bean id="car"
    class="edu.miu.cs544.najeeb.vehicle.Car
    "></bean>

<bean id="bike1"
    class="edu.miu.cs544.najeeb.vehicle.Bik
    e"></bean>

<bean
    id="bike2" class="edu.miu.cs544.najeeb.
    vehicle.Bike"></bean>

<bean id="game"
    class="edu.miu.cs544.najeeb.game.Gam
    e" ></bean>
```

Exclude Beans From Autowired

Game

```
...
@Autowired(required = false)
private Car car;
...
if (car != null) {
    car.move();
...
```

XML

```
<bean id="car"
class="edu.miu.cs544.najeeb.vehicle.Car"
autowire-candidate="false"></bean>
```

Exclude Beans From Autowired

Game

```
...
@Autowired(required = false)
private Car car;
...
if (car != null) {
    car.move();
...
```

Java Config

```
@Bean(autowireCandidate = false)
public Car car() {return new Car();}
```

Define Autowire Candidates

<beans …

default-autowire-candidates="b*, c*"

…

<bean id="car" class="edu.miu.cs544.najeeb.vehicle.Car" autowire-candidate="false"></bean>

- This will still exclude car but include all other beans starting with "c"

default-autowire-candidates="b*"

<bean id="car" class="edu.miu.cs544.najeeb.vehicle.Car"></bean>

- This will not include any bean that starts with "c"

Autowiring

Good

- Save typing
- Adding additional beans in your Java code does not require changing configuration xml

Bad

- Cannot use autowiring for String and primitive types for parameters.
- Not a good idea to mix autowing and explicit configuration (mixing will result in confusion in the future)
- Explicit configuration will override autowining.
- Tools use explicit configuration (not many use autowired, why?)

PROFILES

Create Two Configuration Environments

SpringConfigDev.java

```
@Configuration
@Profile({"development", "default"})
public class SpringConfig {
	@Bean
	public DbService dbService() {
		DbService dbService= new
DbService();
		dbService.setDb_url("developmen
t_db_url");
		return dbService;
	}
}
```

SpringConfigProd.java

```
@Configuration
@Profile("production")
public class SpringConfig {
	@Bean
	public DbService dbService() {
		DbService dbService=
new DbService();
		dbService.setDb_url("production_
db_url");
		return dbService;
	}
}
```

Create Two Configuration Environments

DbService

```
public class DbService {
	private String db_url;
	public DbService() {
	}
	public void connect() {
		System.out.println("Connected to "+db_url);
	}
    //getters & setters
}
```

Main

```
AnnotationConfigApplicationContext springContext= new AnnotationConfigApplicationContext();

springContext.getEnvironment().setActiveProfiles("development");

springContext.scan("edu.miu.cs544.najeeb.config");

springContext.refresh();

DbService dbService= springContext.getBean(DbService.class);

dbService.connect();
```

Configure the environment to come from outside

Get From IDE

Add VM Options

-Dspring.profiles.active=development

Main

```
AnnotationConfigApplicationContext springContext= new AnnotationConfigApplicationContext();

springContext.scan("edu.miu.cs544.najeeb.config");

springContext.refresh();

DbService dbService= springContext.getBean(DbService.class);

dbService.connect();
```

Config Files Interface

SpringConfigInterface.java

```
public interface SpringConfigInterface {
        public DbService dbService();
}
```

SpringConfigProd.java

```
@Configuration
@Profile("production")
public class SpringConfig implements
SpringConfigInterface{
        @Bean
        public DbService dbService() {
                DbService dbService=
new DbService();
                dbService.setDb_url("production_
db_url");
                return dbService;
        }
}
```

Single Configuration File

SpringConfig.java

```
@Configuration
public class SpringConfig{
	@Bean("dbService");
	@Profile({"development","default"})
	public DbService dbServiceDev(){
		DbService dbService= new DbServiceDev();
		dbService.setDb_url("development_db_url");
		return dbService;
	}
	@Bean("dbService")
	@Profile("production")
	public DbService dbServiceProd(){
		DbService dbService= new DbServiceProd();
		dbService.setDb_url("production_db_url");
		return dbService;
	}

}
```

Main

```
ApplicationContext springContext= new
AnnotationConfigApplicationContext(SpringConfig.cl
ass);

DbService dbService=
   springContext.getBean(DbService.class);

dbService.connect();
```

Profiles in XML

configDev.xml

```
<beans profile="development, default"
xmlns ...

    <bean id="dbService"
class="edu.miu.cs544.najeeb.game.DbServ
ice" >

            <property name="db_url"
value="development_db" />

    </bean>
```

configProd.xml

```
<beans profile="production" xmlns ...

    <bean
id="dbService" class="edu.miu.cs544.naje
eb.game.DbService">

            <property
name="db_url" value="development_db"
/>

    </bean>
```

Profiles in XML

```
GenericXmlApplicationContext springContext= new GenericXmlApplicationContext();
springContext.load("config*.xml");
springContext.refresh();

DbService dbService= springContext.getBean(DbService.class);
dbService.connect();
```

Single XML for Profile

config.xml

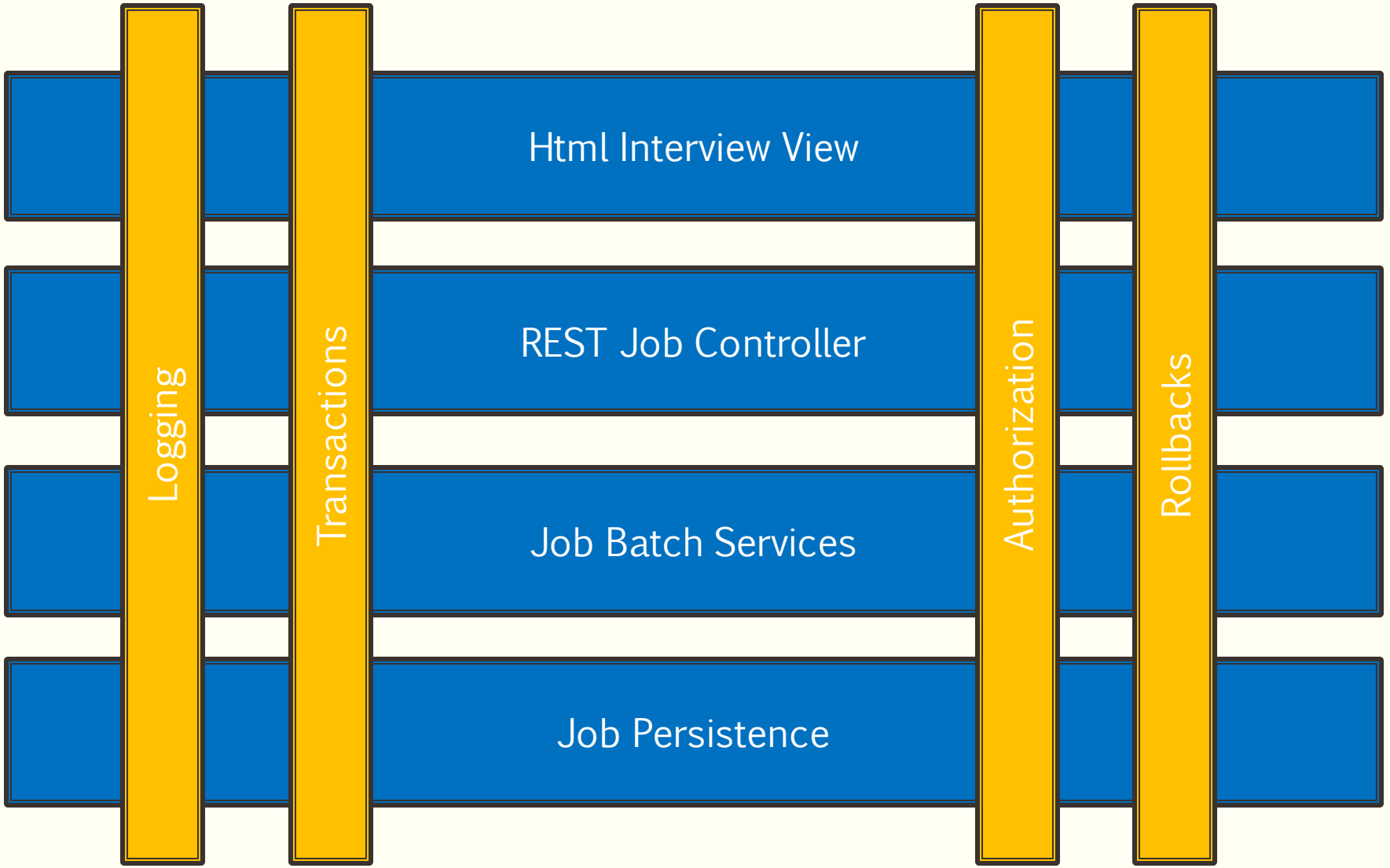
```
<beans xmlns ...

    <beans profile="development, default">
        <bean id="dbService"
class="edu.miu.cs544.najeeb.game.DbService">
            <property name="db_url"
value="development_db" />
        </bean>
    </beans>

    <beans profile="production">
        <bean
id="dbService" class="edu.miu.cs544.najeeb.game
.DbService">
            <property
name="db_url" value="production_db" />
        </bean>
    </beans>
```

Main

```
GenericXmlApplicationContext springContext=
new GenericXmlApplicationContext("config.xml");


DbService dbService= springContext.getBean(DbS
ervice.class);
dbService.connect();
```

AOP

Cross Cutting Concerns



AOP

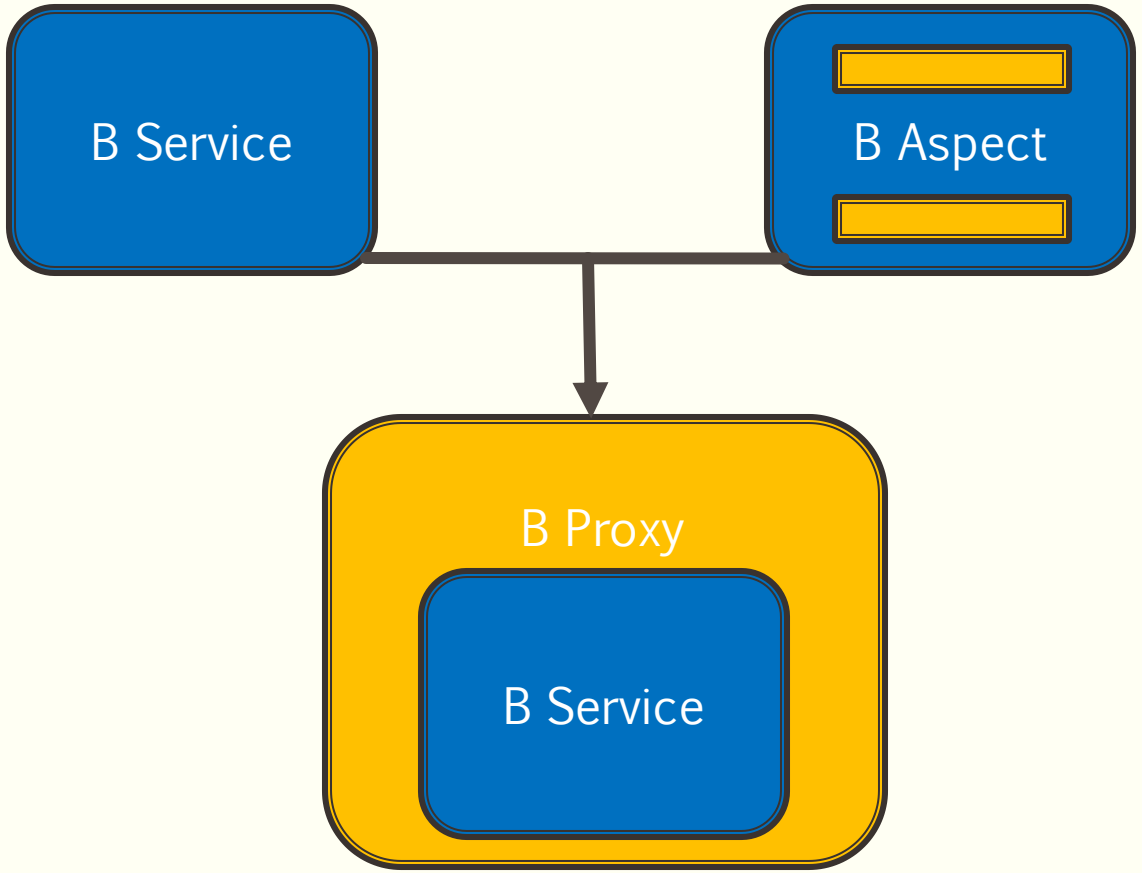
- Achieve SoC
- Enable Single Responsibility
- Decoupling

AOP Concepts

- Joinpoint
  - A location where the advice is applied
- Pointcut
  - Collection of one or more Joinpoint(s)
- Advice
  - The cross-cutting concern implementation
- Aspect
  - What Advice is executed at which Pointcut
- Weaving
  - Applying Advice code to Joinpoint to enable execution of Aspects

SPRING AOP

Spring Weaving Process



Define the Advice Class and Implement

Implement MethodBeforeAdvice

```
@Override
  public void before(Method method,
Object[]  args, Object target) throws
Throwable {
        System.out.println("Starting : " +
target.getClass() + " : " +
method.getName());

}
```

Implement AfterReturningAdvice

```
@Override
public void afterReturning(Object
returnValue, Method method, Object[]
args, Object target) throws Throwable {
        System.out.println("Ended : " +
target.getClass() + " : " +
method.getName());

}
```

Trigger Proxy Generation

XML

```
<bean id="emailService"
class="edu.miu.cs544.najeeb.services.EmailService" ></bean>
bean id="databaseService"
class="edu.miu.cs544.najeeb.services.DatabaseService" ></bean>
<bean id="logAspect"
class="edu.miu.cs544.najeeb.aspects.LogBeforeAndAfter"
></bean>
<bean id="emailServiceProxyy"
class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target" ref="emailService" />
   <property name="interceptorNames" >
         <list>
               <value>logAspect</value>
         </list>
   </property>
</bean>
```

Java Config

```
@Configuration
public class SpringConfig {
      @Bean
      public EmailService emailService() throws Exception {
            return new EmailService();
      }
      @Bean
      public DatabaseService databaseService() throws Exception {
            return new DatabaseService();
      }
      @Bean
      public LogBeforeAndAfter logAspect() throws Exception {
            return new LogBeforeAndAfter();
      }
      @Bean
      public ProxyFactoryBean emailServiceProxy() throws
Exception {
            String[] interceptorNames = {"logAspect"};
            ProxyFactoryBean proxyFactoryBean= new
ProxyFactoryBean();
            proxyFactoryBean.setTarget(emailService());
            proxyFactoryBean.setInterceptorNames(interceptorName
s);
            return proxyFactoryBean;
      }
}
```

Usage Code

Main

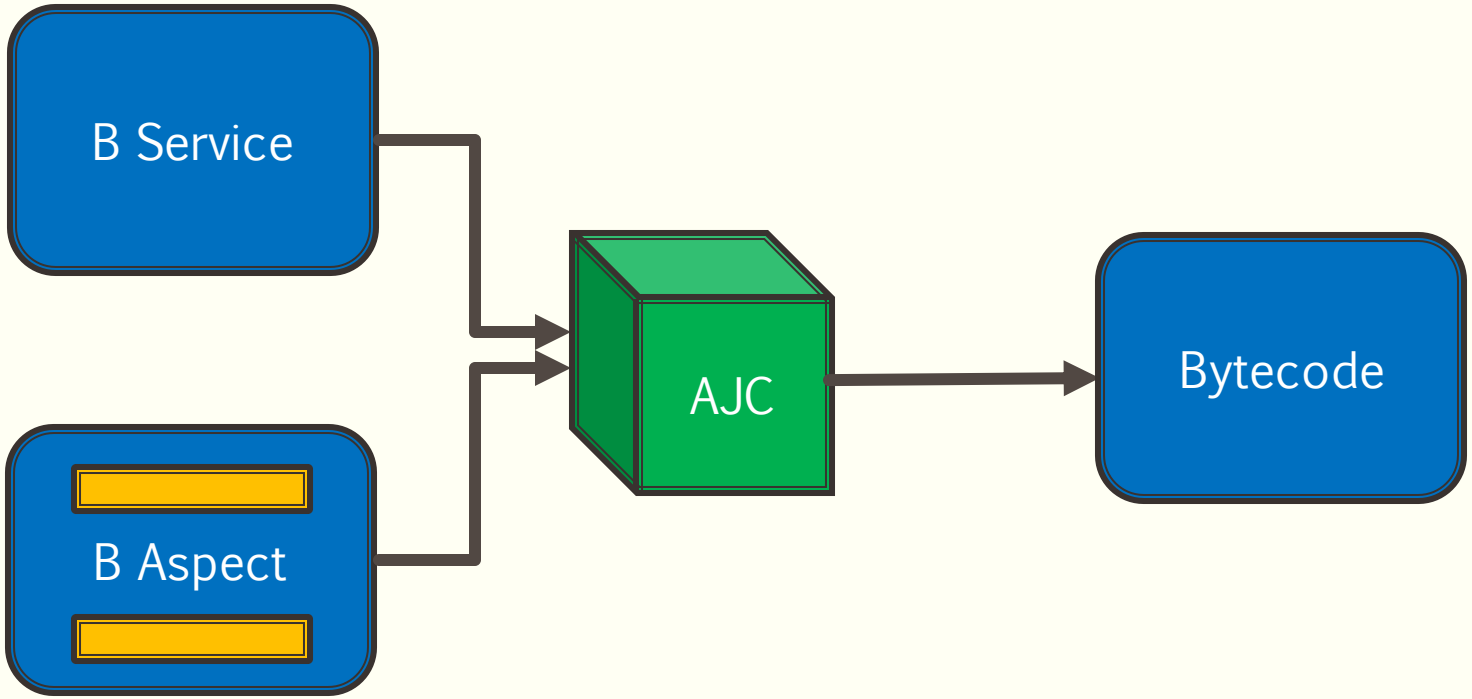
```
EmailService emailService= (EmailService) springContext.getBean("emailServiceProxy");

emailService.sendEmail();
```

What did we create?

- Advice
- Aspect
- Joinpoint ?
- Weaving ??

ASPECTJ

AspectJ Weaving



Setup

pom.xml

```
<dependency>
	<groupId>aspectj</groupId>
	<artifactId>aspectjweaver</artifactId>
	<version>1.5.4</version>
</dependency>
<dependency>
	<groupId>aspectj</groupId>
	<artifactId>aspectjrt</artifactId>
	<version>1.5.4</version>
</dependency>
```

Aspect

```
@Aspect
public class Logger {
	@Before("execution(* edu.miu.cs544.najeeb.services.EmailService.*())")
	public void before(JoinPoint joinPoint) {
		System.out.println("Before :
"+joinPoint.getSignature().getDeclaringTypeName()+" :
"+joinPoint.getSignature().getName());
	}
	@After("execution(* edu.miu.cs544.najeeb.services.EmailService.*(..))")
	public void after(JoinPoint joinPoint) {
		System.out.println("After :
"+joinPoint.getSignature().getDeclaringTypeName()+" :
"+joinPoint.getSignature().getName());
	}
}
```

Setup

XML

```
<beans            xmlns:aop="http://www.springframework.org
    /schema/aop"
      xsi:schemaLocation="...
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-
    aop-3.0.xsd">

<contex:annotation-config/>

<aop:aspectj-autoproxy />

<bean id="emailService"
    class="edu.miu.cs544.najeeb.services.EmailService"
    ></bean>

<bean id="databaseService"
    class="edu.miu.cs544.najeeb.services.DatabaseService"
    ></bean>

<bean id="logAspect"
    class="edu.miu.cs544.najeeb.aspects.Logger"></bean>
```

Java Config

```
@Configuration
@EnableAspectJAutoProxy
public class SpringConfig{
        @Bean
        public EmailService emailService() throws Exception{
                return new EmailService();
        }
        @Bean
        public DatabaseService databaseService() throws
Exception{
                return new DatabaseService();
        }
        @Bean
        public Logger logger(){
                return new Logger();
        }
}
```

Pointcut Expression Language

- Narrow down JoinPoints
- @Before("execution(public * *.*.*(..))")
- execution visibility returnType.package.class.method(args)
- Visibility (method access modifier)
  - Optional
  - Cannot be *
  - Can be
    - private
    - public
    - protected
- Return Type (method return type)
  - Not optional
  - Can be *
- package.class.method
  - Not optional
  - Can be * or *.* or *.*.* (I think package is no longer allowed to be *)
- Args
  - Can be ..

Advice Types

- @Before
  - The advice is executed before the Pointcut
- @After
  - The advice is executed after the Pointcut
- @AfterReturning
  - The advice is executed only after successful execution of the Pointcut
- @AfterThrowing
  - The advice is executed only after the Pointcut throws an exception
- @Around
  - The advice wraps the Pointcut and will run before and after the Pointcut
  - The advice can disable the execution of the Pointcut

@After vs @AfterReturning

Service

```
public void sendEmail() {
        System.out.println("Sending email");
        if (Math.random() > 0.5)
            throw new NullPointerException();
    }
```

Advice

```
@After("execution(*
edu.miu.cs544.najeeb.services.EmailService.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("After :
"+joinPoint.getSignature().getDeclaringTypeName(
)+" : "+joinPoint.getSignature().getName());
    }

@AfterReturning("execution(*
edu.miu.cs544.najeeb.services.EmailService.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("After :
"+joinPoint.getSignature().getDeclaringTypeName(
)+" : "+joinPoint.getSignature().getName());
    }
```

@After vs @AfterThrowing

Service

```
public void sendEmail() {
        System.out.println("Sending email");
        if (Math.random() > 0.5)
            throw new NullPointerException();
    }
```

Advice

```
@After("execution(*
edu.miu.cs544.najeeb.services.EmailService.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("After :
"+joinPoint.getSignature().getDeclaringTypeName(
)+" : "+joinPoint.getSignature().getName());
    }

@AfterThrowing("execution(*
edu.miu.cs544.najeeb.services.EmailService.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("After :
"+joinPoint.getSignature().getDeclaringTypeName(
)+" : "+joinPoint.getSignature().getName());
    }
```

@Around

Service

```
public void sendEmail() {
        System.out.println("Sending email");
        if (Math.random() > 0.5)
            throw new NullPointerException();
    }
```

Advice

```
@Around("execution(*
edu.miu.cs544.najeeb.services.EmailService.*(..))")
  public void around(ProceedingJoinPoint
proceedingJoinPoint) throws Throwable{
    System.out.println("Before  execution :
"+proceedingJoinPoint.getSignature().getDeclaring
TypeName()+" :
"+proceedingJoinPoint.getSignature().getName());
    if (Math.random() > 0.5) {
        proceedingJoinPoint.proceed();
    }
    System.out.println("After  execution :
"+proceedingJoinPoint.getSignature().getDeclaring
TypeName()+" :
"+proceedingJoinPoint.getSignature().getName());
}
```

Aspect Execution Order

- Add precedence using @Order
- Aspects are executed in incrementing order
- Logger
  - @Aspect
  - @Order(1)
  - public class LoggerAspect …
- Transaction
  - @Aspect
  - @Order(2)
  - pubilc class TransactionAspect
- These are executed is LIFO (Last In First Out)
  - @Before 1, 2, 3, …, n-1, n
  - @After n, n-1, …, 3, 2, 1

Advice Execution Order

- There is no control that Spring provides
- If we need to run advice that runs on the same PointCuts in a certain order, how can we do that?

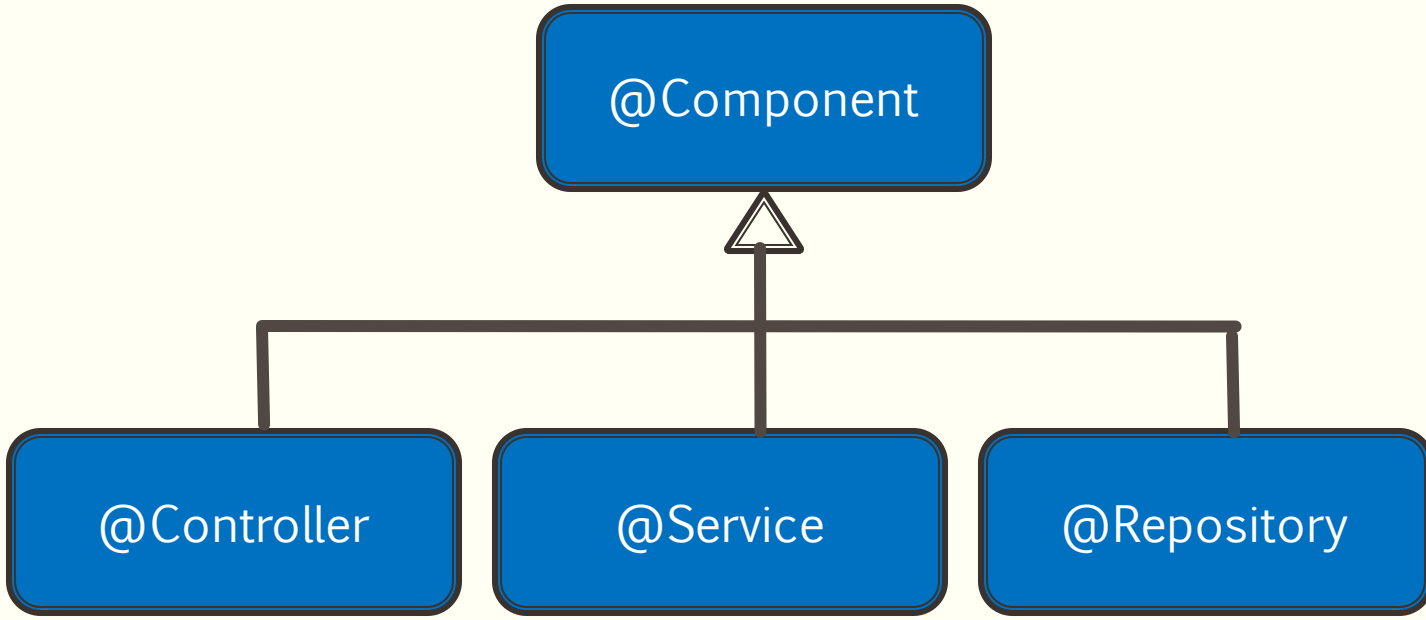
Advice Attributes

- Access return value of @AfterReturning advice
  - Use @AfterReturning(pointcut="execution(Boolean edu.miu.cs544.najeeb.*.*(..)", returning= "success")
  - public void after(JoinPoint joinPoint, Boolean success)
- Access the exception object thrown in @AfterThrowing
  - Use @AfterThrowing(pointcut="execution(* edu.miu.cs544.najeeb.*.*(..)", throwing= "exception")
  - public void after(JoinPoint joinPoint, Exception exception)

COMPONENT

Using Components

- XML
  - <contex:component-scan base-package="edu.miu.cs544.najeeb" />
- Java
  - @ComponentScan(basePackages = "edu.miu.cs544.najeeb")



POINTCUT EXPRESSION LANGUAGE

Using @Pointcut

```
@Pointcut("execution(* edu.miu.cs544.najeeb.services.EmailService.*())")
Public void allMethodsInEmailService(){}

@Before("allMethodsInEmailService()")
public void before(JoinPoint joinPoint) {
            System.out.println("Before :
"+joinPoint.getSignature().getDeclaringTypeName()+" :
"+joinPoint.getSignature().getName());
      }
```

Designators

execution()

args()
@Before("execution(* edu.miu.cs544.najeeb*.AccountService.deposit(..)) &&
args(clientId, amount,...)")
public void before(JoinPoint joinPoint, String clientId, float amount) {
	System.out.println("client "+clientId+" depositing "+amount);
}

args()
@Before("execution(* edu.miu.cs544.najeeb*.AccountService.deposit(..))
&& args(String, float,...)")
public void before(JoinPoint joinPoint, String clientId, float amount) {
	System.out.println("client "+clientId+" depositing "+amount);
}

AOP Comparison

Spring AOP

- Weaving occurs during run time. May result in performance cost (runtime overhead).
- Proxy based, so can only be executed on methods (cannot be applied to constructors).
- Can only be applied to beans.
- Not applied to internal method calls

AspectJ

- Weaving occurs more during compile time and less during runtime. Less impact on performance.
- More control over JoinPoints
- Uses annotation
- Check that only what you want to be weaved is weaved.
- Extra build time overhead.

Main Point

- Spring is an umbrella open-source frameworks project. Spring-core is an IoC bean container. Spring has very rich configuration options to satisfy several development needs. Spring-core enables my application to eliminate creating objects and managing dependencies in code, thanks to Spring-core DI options. Spring IoC container enables developers to regain control during the bean lifecycle (by using aware interfaces, init, and destroy).
- Spring has its own AOP, but at the same time support the very popular AspectJ AOP. AOP results in being able to achieve SoC and single responsibility in several parts of an application.