Spring Framework

UA.DETI.IES - 2019/20

UNIVERSIDADE DE AVEIRO

Main topics

- String Framework

- Architecture
- Annotations
- Inversion of Control (IoC)
- Dependency Injection (DI)
- Beans
- Aspect Oriented Programming (AOP)

Server-side Frameworks

- Micro Frameworks
  - focused on routing HTTP request to a callback, commonly used to implement HTTP APIs.
  - Flask (Python), Express.js (Node.js), Spark (Java)

- Full-Stack Frameworks
  - feature-full frameworks that includes routing, templating, data access and mapping, plus many more packages.
  - Django, ASP.NET, Spring, ..

- Component Frameworks
  - collections of specialized and single-purpose libraries that can be used together to make a a micro- of full-stack framework.
  - Angular (google), React (facebook), Vue,..

Frameworks

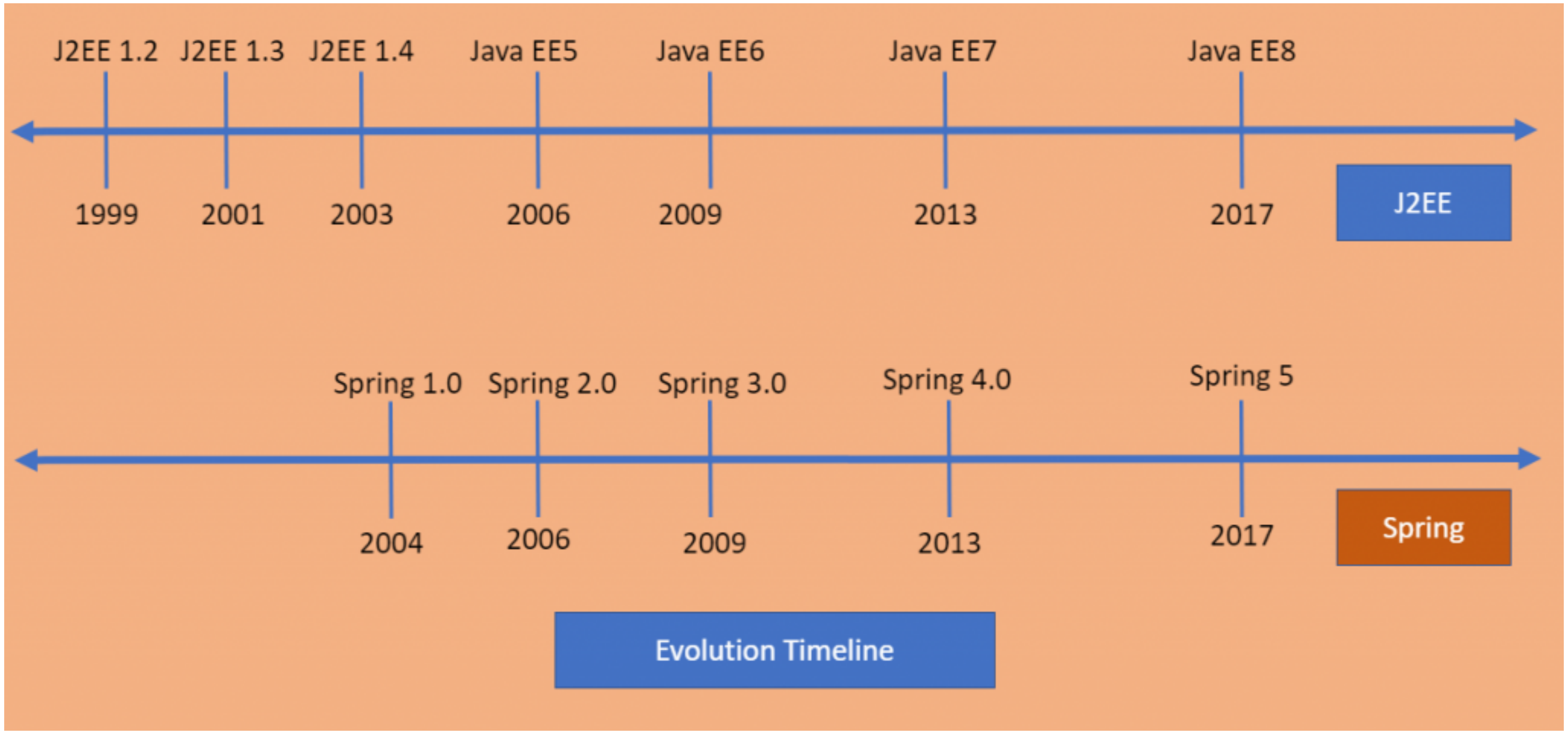
- A framework is something that provides a standard way to do certain things.
  - If we consider the task of constructing a car, a framework can be thought of as the frame of a car. In other words, the structure or skeleton that helps in building the car.
- Frameworks are ubiquitous in todays software development world.
  - They help developers to focus on implementing business code and and abstract and implement technical aspects aspects of programming.
  - They effectively form an instance of the concept of practical reuse in software engineering.
- A major challenge in developing and using frameworks is the (de)coupling between the software component being developed and the framework.

Java frameworks

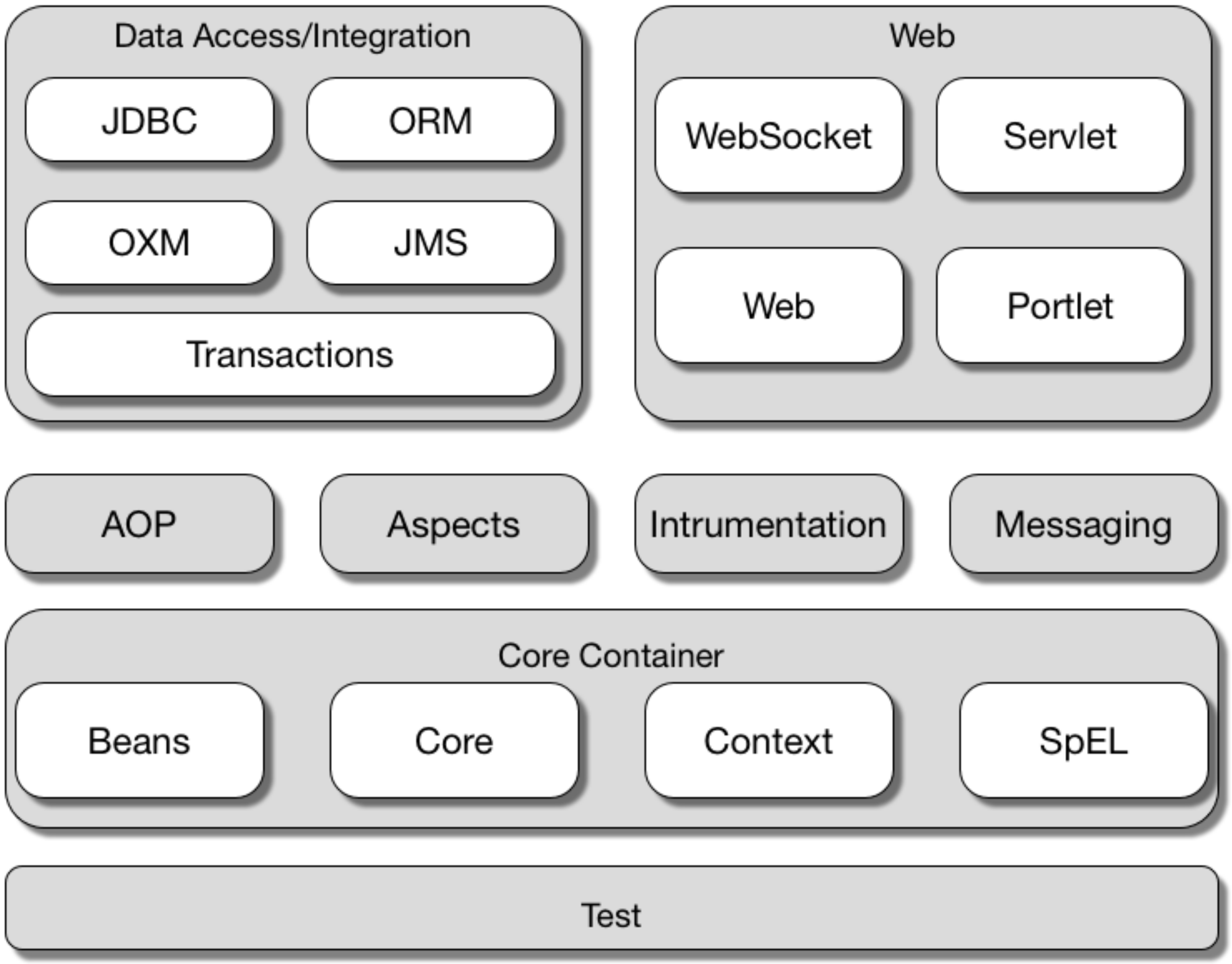
- Spring was not the first framework trying to solve the problems around building enterprise Java applications.
- J2EE or Java Enterprise Edition that preceded Spring tried to do the same thing.
  - On paper, J2EE was meant to become the standard way of building Java enterprise applications.
  - The major guiding principle behind J2EE was to provide clean separation between various parts of an application.
- Spring Framework is a Java application framework started in the early 2000s.
  - Since then it has grown from a already pretty sophisticated dependency injection container into an eco-system of frameworks that cover a broad set of use cases in application development.

JEE and Spring

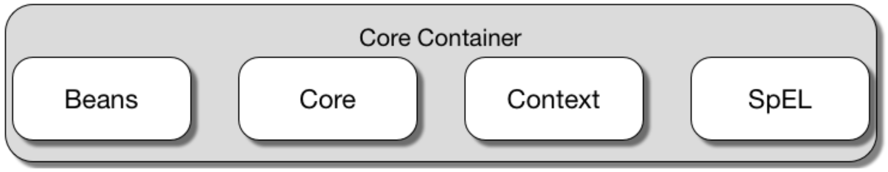
- **Jakarta Enterprise Edition (JEE)**
  - formerly **Java Enterprise Edition (JEE)**



Spring Framework Architecture



Core Container modules



- **Spring Core**
  - provides implementation for features like IoC (Inversion of Control) and Dependency Injection with singleton design pattern.
- **Spring Bean**
  - This module provides implementation for the factory design pattern through BeanFactory.
- **Spring Context**
  - This module is built on the solid base provided by the Core and the Beans modules and is a medium to access any object defined and configured.
- **Spring Expression Languages (SpEL)**
  - This module is an extension to expression language supported by Java server pages. It provides a powerful expression language for querying and manipulating an object graph, at runtime.

Java Annotations

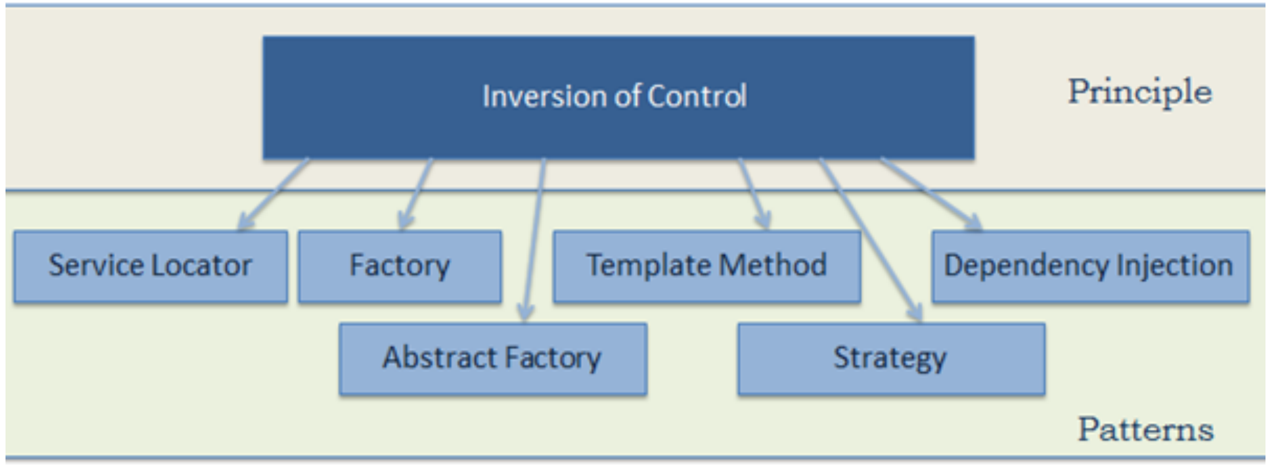
- Annotations provide metadata about a program or about its components.

  `@Override, @SuppressWarning, @Deprecated, ..`

- They can be applied on declarations of:
  - classes, fields, methods, and other program elements.
- Common usage:
  - Information for the compiler - to detect errors or suppress warnings.
  - Compile-time and deployment-time processing - to generate code, XML files, configs.
  - Runtime processing – some annotations are available to be examined at runtime
- Annotations are largely used in Spring Framework

Inversion of Control (IoC)

- IoC is a process in which an object defines its dependencies without creating them.
- In object-oriented programming, there are several techniques to implement inversion of control.
  - using a factory pattern
  - using a service locator pattern
  - using **dependency injection**



https://www.tutorialsteacher.com/ioc/inversion-of-control

Dependency Injection (IoC / DI)

- Dependency Injection a design pattern that **removes the dependency from the code**.
  - We may specify classes' dependencies through **annotations** or from external source, such as **XML** file.
  - We do not create the objects, but just define how they should be created by the IoC container.

- Dependencies can be injected in two ways
  - By **constructor** – The <constructor-arg> subelement of <bean> is used for constructor injection

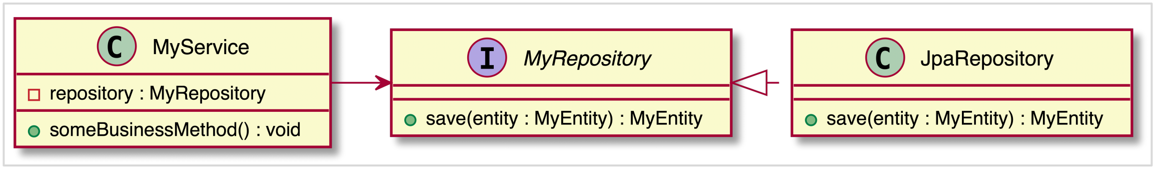
    ```
    <constructor-arg value="101" type="int"></constructor-arg>
    ```

  - By **setter** method – The <property> subelement of <bean> is used for setter injection

    ```
    <property name="id" value="101"></property>
    ```

UNIVERSIDADE DE AVEIRO

Example


C MyService
repository : MyRepository
someBusinessMethod() : void
I MyRepository
save(entity : MyEntity) : MyEntity
C JpaRepository
save(entity : MyEntity) : MyEntity

```
interface MyRepository {
    MyEntity save(MyEntity entity);
  }

class JpaRepository implements MyRepository {
    @Override
    MyEntity save(MyEntity entity) { ... }
}

class MyService {
    private final MyRepository repository;
    MyService(MyRepositoryrepository){
      this.repository = repository;
    }
    void someBusinessMethod(...) {
      this.repository.save(new MyEntity());
    }
}

// Manual dependency injection
MyRepository repository = new MyRepository(...);
MyServiceservice = new MyService(repository);
```

MyService expresses a dependency to MyRepository instead of creating it itself.

The manual setup code creates as instance of MyRepository to the service instance to be created.

Example (with Spring DI)

```
@Component
class MyService { ... }

@Component
class JpaRepository { ... }

// Spring-driven dependency injection
ApplicationContext context =
    new AnnotationConfigApplicationContext(Config.class);
MyService service = context.getBean(MyService.class);
```

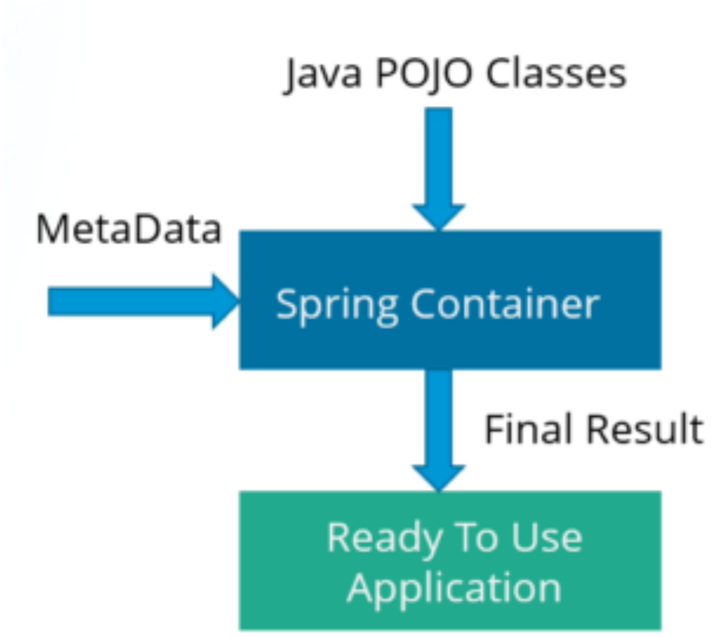
Classes are annotated with framework specific annotations so that it can discover the components of an application that it's supposed to handle.

The detection is triggered by the framework, pointing it to the code written by the user.

The framework API is then used to access the components.

Spring IoC Container

- Spring IoC is the heart of the Spring Framework.
- Each object delegates the job of constructing to an IoC container.
- The IoC container receives metadata from either
  - an XML file,
  - Java annotations, or
  - Java code
- and produces a fully configured and executable system or application.



Spring IoC Container

- The **main tasks** performed by the IoC container are:
  - Instantiating the bean
  - Wiring the beans together
  - Configuring the beans
  - Managing the bean's entire life-cycle

- Two types of Spring IoC containers:
  - the **BeanFactory** provides the configuration framework and basic functionality, and
  - the **ApplicationContext** adds more enterprise-specific functionality.

ApplicationContext

- Interface org.springframework.context.**ApplicationContext**.
  - It supports the features supported by Bean Factory but also provides some additional functionalities.
  - It follows eager-initialization technique which means instance of beans are created as soon as you create the instance of Application context.

- The Spring framework provides several implementations of the ApplicationContext interface
  - ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications,
  - WebApplicationContext for web applications.

```
ApplicationContext context
  = new ClassPathXmlApplicationContext("applicationContext.xml");
```

What is a bean?

- Spring documentation definition
  - "the objects that form the backbone of an application and that are managed by the Spring IoC container are called beans. A bean is instantiated, assembled, and managed by the IoC container."
- Spring is responsible for creating bean objects. But first, we need to tell the framework which objects it should create.
  - **Bean definitions** tell Spring which classes the framework should use as beans.
  - Bean definitions are like recipes. They also describe the properties of a bean.

https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-introduction

Defining a Spring bean

- There are **three different ways** to define a Spring bean:
  - annotating a class with the *@Component* annotation (or its derivatives, *@Service*, *@Repository*, *@Controller*)
  - writing a bean factory method annotated with the *@Bean* annotation in a custom Java configuration class
  - declaring a bean definition in an XML configuration file
- In modern projects, we use only component annotations and bean factory methods.
  - If it comes to the XML configuration, nowadays Spring allows it mainly for the backward compatibility.
- The **choice between bean definition methods** is mainly dictated by access to the source code of a class that we want to use as a Spring bean.

Spring bean as @Component

- If we own the source code, we'll usually use the @Component annotation directly on a class.

```
@Component
class MySpringBeanClass {
    //...
}
```

  - At runtime, Spring finds all classes annotated with @Component or its derivatives and uses them as bean definitions.
  - The process of finding annotated classes is called **component scanning**.

Using @Bean for factory methods

- For classes we don't own, we have to create factory methods with the @Bean annotation in a custom bean configuration class.
  - If we don't want to make a class dependent on Spring, we can also use this option for classes you own.

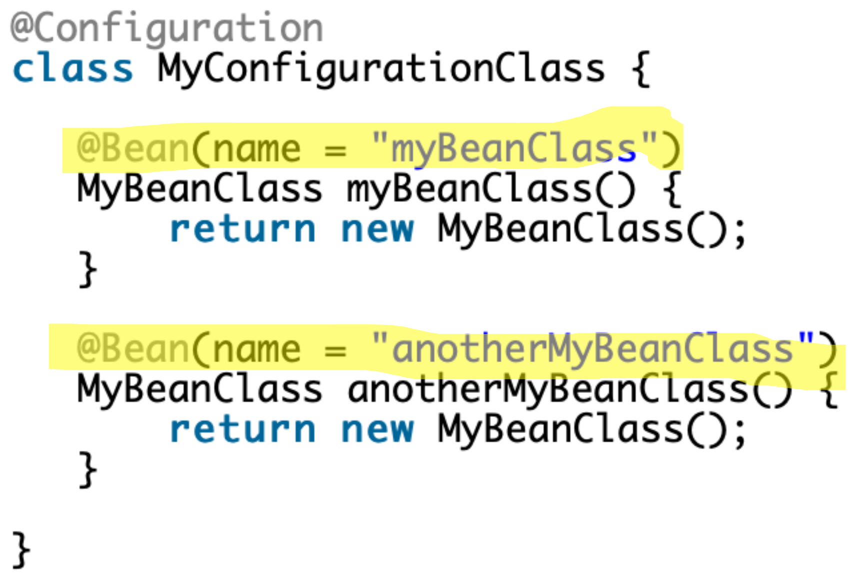
```
@Configuration
class MyConfigurationClass {

   @Bean
   public NotMyClass notMyClass() {
       return new NotMyClass();
   }
}
```

- The @Configuration annotation also comes from Spring. The annotation marks the class as a container of @Bean definitions.
  - We may write multiple factory methods inside a single configuration class.

UNIVERSIDADE DE AVEIRO

Spring bean properties

- Properties give details about how Spring should create objects.
- The list of Spring bean properties includes:
  - class
  - name - must be unique across the whole application
  - dependencies
  - scope
  - initialization mode
  - initialization callback
  - destruction callback



```
@Configuration
class MyConfigurationClass {

   @Bean(name = "myBeanClass")
   MyBeanClass myBeanClass() {
       return new MyBeanClass();
   }

   @Bean(name = "anotherMyBeanClass")
   MyBeanClass anotherMyBeanClass() {
       return new MyBeanClass();
   }

}
```

Bean dependencies

- For a class with only one constructor, marked with @Component Spring uses the list of constructor parameters as a list of mandatory dependencies.

```
@Component
class BeanWithDependency {
   private final MyBeanClass beanClass;
   BeanWithDependency(MyBeanClass beanClass) {
      this.beanClass = beanClass;
   }
}
```

- If a bean class defines multiple constructors, one must be marked with @Autowired.
  - This way Spring knows which constructor contains the list of bean dependencies.
  - Before Spring 4.3, @Autowired annotation was mandatory on every constructors.

The @Autowired annotation

- It is used to wire a bean to another one without instantiating the former one.
  - By marking a bean as @Autowired, Spring expects it to be available when constructing the other dependencies.

```
public class ProductController {

    @Autowired
    private ProductService productService;
   // instead of ...
   // private ProductService productService = new ProductService();
    public void someMethod() {
         List<Product> productList = productService.getProductList();
    }
}
```

Some Spring annotations

- @ComponentScan
  - A crucial annotation which specifies which packages contain classes that are annotated. It is always used alongside the @Configuration annotation.

```
@Configuration
@ComponentScan(basePackage = "com.stackabuse")
public class SomeApplication {
    // some code
}
```

- @Required
  - is used on setter methods and constructors, to tell Spring that these fields are required to initialize the bean.
- @Scope
  - is applied on bean-level and defines its visibility/life cycle.
- @Value
  - It has quite a few use-cases in Spring. More on SpEL..

Spring Expression Language (SpEL)

- SpEL can be used for querying and manipulating an object graph at runtime.
  - SpEL is available via XML or annotations and is evaluated during the bean creation time.
- There are several operators available in the language:

Type	Operators
Arithmetic	+, -, *, /, %, ^, div, mod
Relational	<, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge
Logical	and, or, not, &&, ||, !
Conditional	?:
Regex	matches

SpEL examples

- Context 1: Create *employee.properties* file in the resources directory.

```
employee.names=Petey Cruiser,Anna Sthesia,Paul Molive,Buck Kinnear
employee.type=contract,fulltime,external
employee.age={one:'26', two : '34', three : '32', four: '25'}
```

- Context 2: Create a class EmployeeConfig as follows:

```
@Configuration
@PropertySource (name = "employeeProperties",
                value = "employee.properties")
@ConfigurationProperties
public class EmployeeConfig {
}
```

- SpEL: The @Value annotation can be used for injecting values into fields in Spring-managed beans,
  - it can be applied at the field, constructor, or method parameter level.

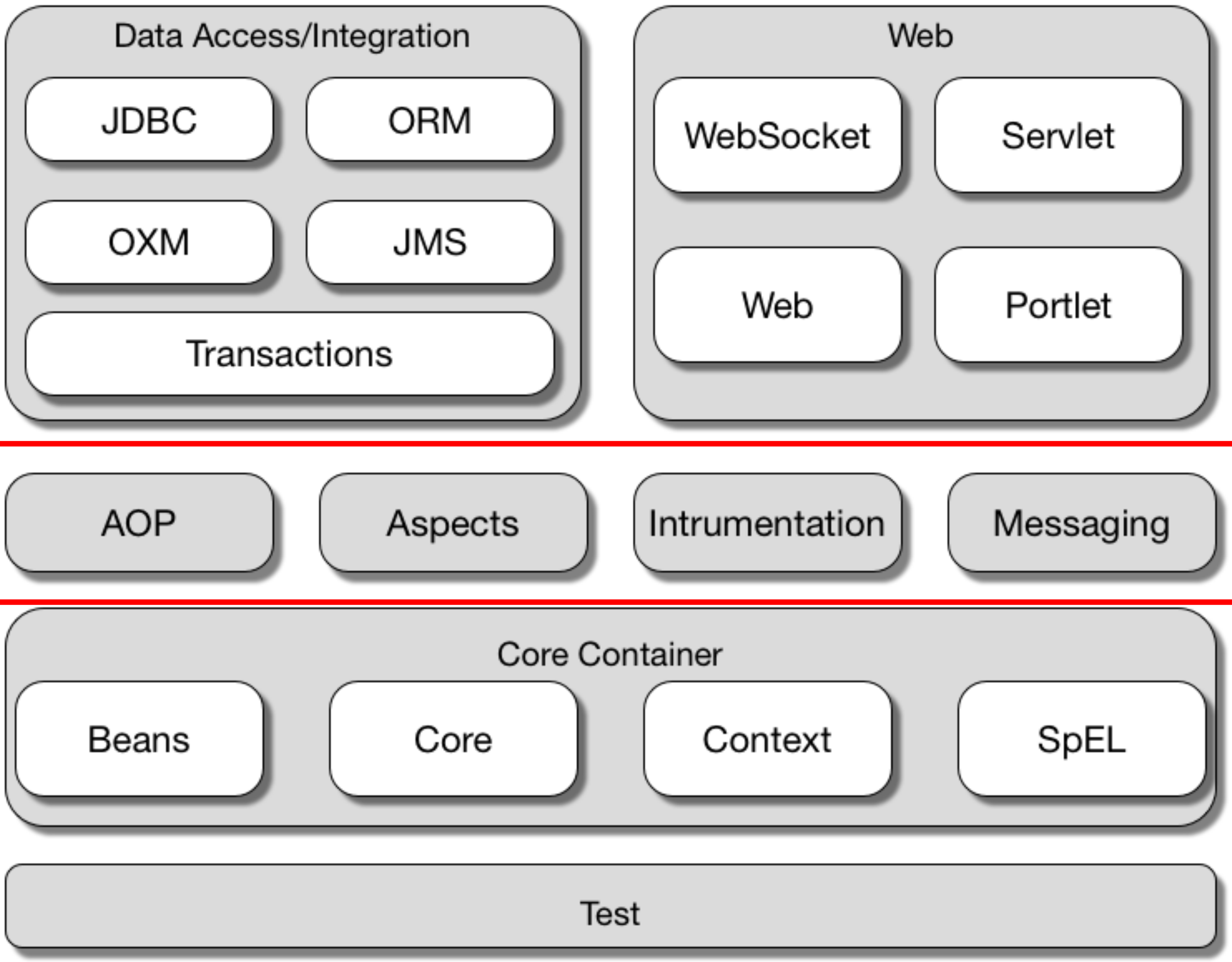
```
@Value ("#{'${employee.names}'.split(',')}")
private List<String> employeeNames;
```

```
[Petey Cruiser, Anna Sthesia, Paul Molive, Buck Kinnear]
```

SpEL examples

```
@Value ("#{'${employee.names}'.split(',')[0]}")
private String firstEmployeeName;

@Value ("#{systemProperties['java.home']}")
private String javaHome;

@Value ("#{systemProperties['user.dir']}")
private String userDir;

@Value("#{someBean.someProperty != null ?
someBean.someProperty : 'default'}")
private String ternary;
```

Spring Framework Architecture



Aspect Oriented Programming (AOP)

- Why AOP? An example:

- We want to keep a log, or send a notification, after(or before) calling a class method (or several classes methods)

```
public class A {
   public void m1() { .. }
   public void m2() { .. }
   public void m3() { .. }
   public void m4() { .. }
   public void m5() { .. }
}
```

- How?

Spring AOP

- AOP is a programming approach that allows global properties of a program to determine how it is compiled into an executable program.
  - AOP compliments OOPs in the sense that it also provides modularity. But here, the key unit of modularity is an aspect rather than a class.
- AOP breaks down the logic of program into distinct parts called *concerns*. This increases modularity by **cross-cutting concerns**.
  - A **cross-cutting concern** is a concern that affects the whole application and is centralized in one location in code like transaction management, authentication, logging, security etc.
- AOP can be considered as a **dynamic decorator** design pattern.
  - The decorator pattern allows additional behavior to be added to an existing class by wrapping the original class and duplicating its interface and then delegating to the original.

Core AOP Concepts

- **Aspect:** a modularization of a concern that cuts across multiple classes (e.g., transaction, logger) is known as the aspect.
  - It can be normal class configured through XML configuration or through regular classes annotated with @Aspect.
- **Join point**: a point represents a method execution during the execution of a program, such as the execution of a method or the handling of an exception.
- **Advice:** action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.
- **Pointcut:** a predicate / expressions that are matched at join points to determine whether advice needs to be executed or not.

AOP Examples

- Defines a pointcut named 'anyOldTransfer' that matchs the execution of any method named 'transfer':

```
@Pointcut("execution(* transfer(..))") // the pointcut expression
private void anyOldTransfer() {} // the pointcut signature
```

- The following example shows three pointcut expressions:
  - matches if a method execution join point represents the execution of any public method

    ```
    @Pointcut("execution(public * *(..))")
    private void anyPublicOperation() {}
    ```

  - matches if a method execution is in the trading module

    ```
    @Pointcut("within(com.xyz.someapp.trading..*)")
    private void inTrading() {}
    ```

  - matches if a method execution represents any public method in the trading module

    ```
    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
    ```

AOP Examples

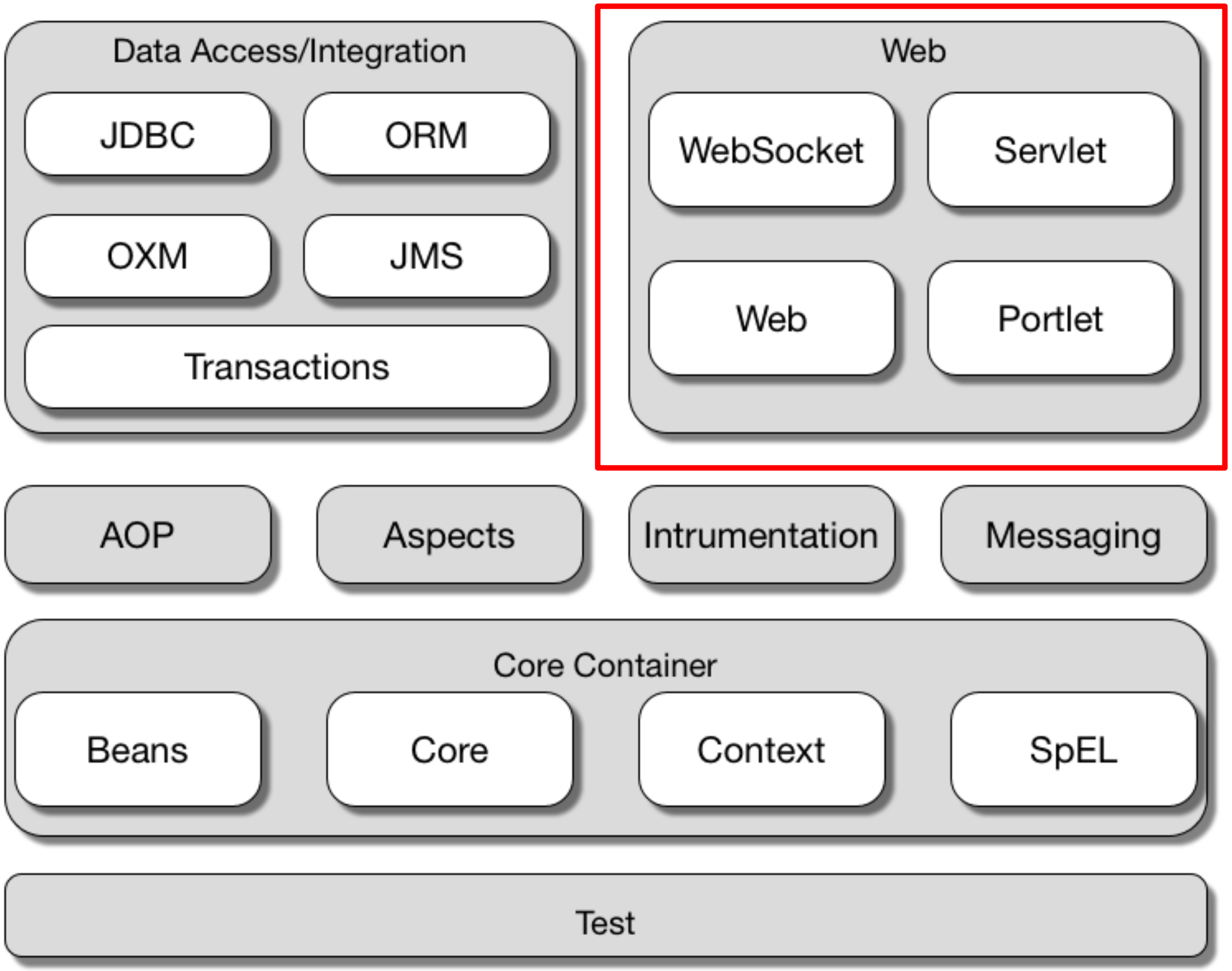
```
@Component
class MyService {
  private final MyRepository repository;
  // …

  @PreAuthorize("hasRole('ADMIN')")
  void someBusinessMethod(…) {
    this.repository.save(new MyEntity());
  }
}

@Component
class JpaRepository implements MyRepository {

  @Transactional
  MyEntity save(MyEntity entity) { … }
}
```

AOP Examples

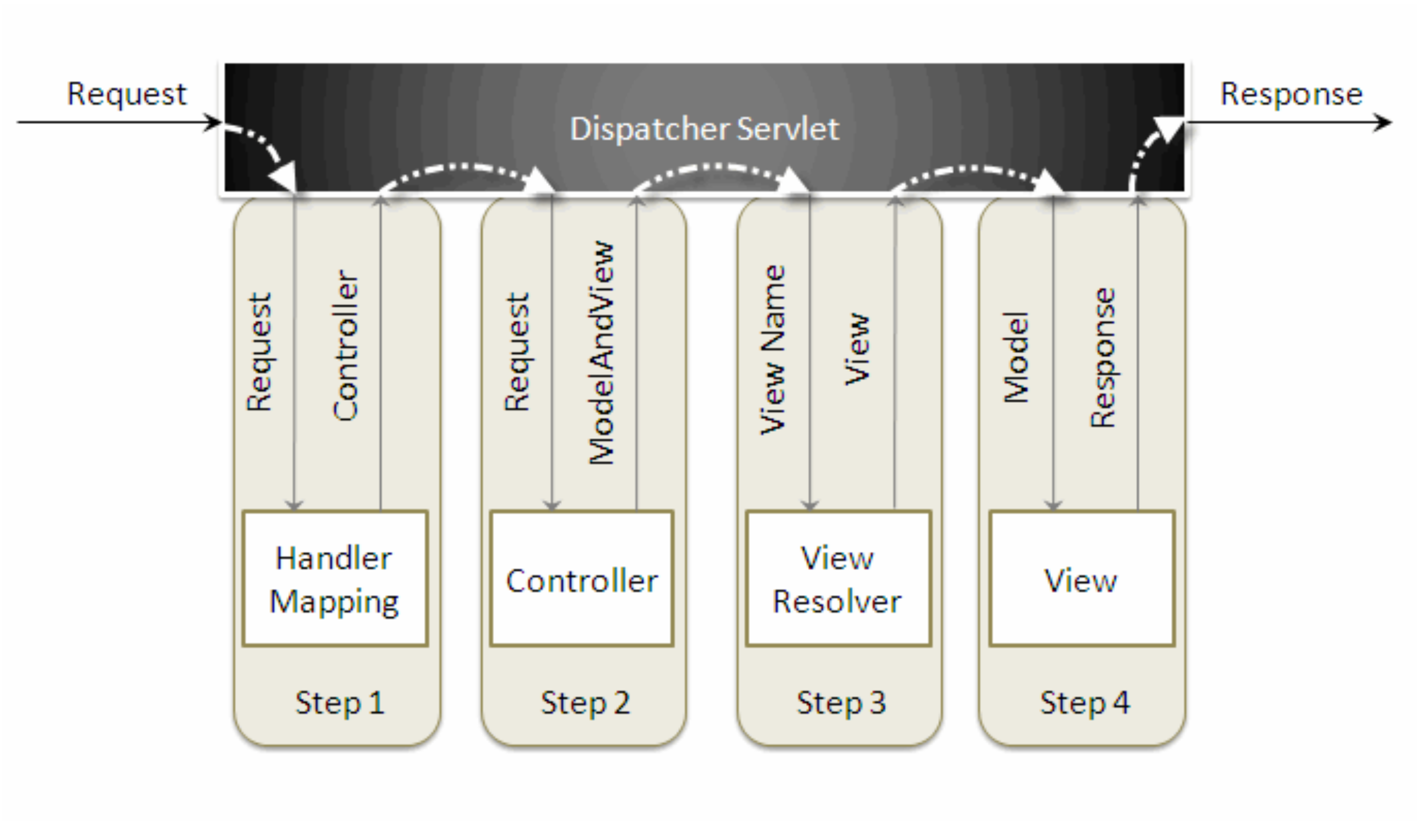
```
package foo
import org.aspectj.lang.*;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;
@Aspect
public class ProfilingAspect {
    @Around("methodsToBeProfiled()")    // action taken at the join point.
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
         try {
             sw.start(pjp.getSignature().getName());
             return pjp.proceed();
         } finally {
             sw.stop();
             System.out.println(sw.prettyPrint());
         }
     }
     @Pointcut("execution(public * foo..*.*(..))")
     public void methodsToBeProfiled(){}
}
```

UNIVERSIDADE DE AVEIRO

Spring Web


Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Intrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

Spring MVC (Model-View-Controller)



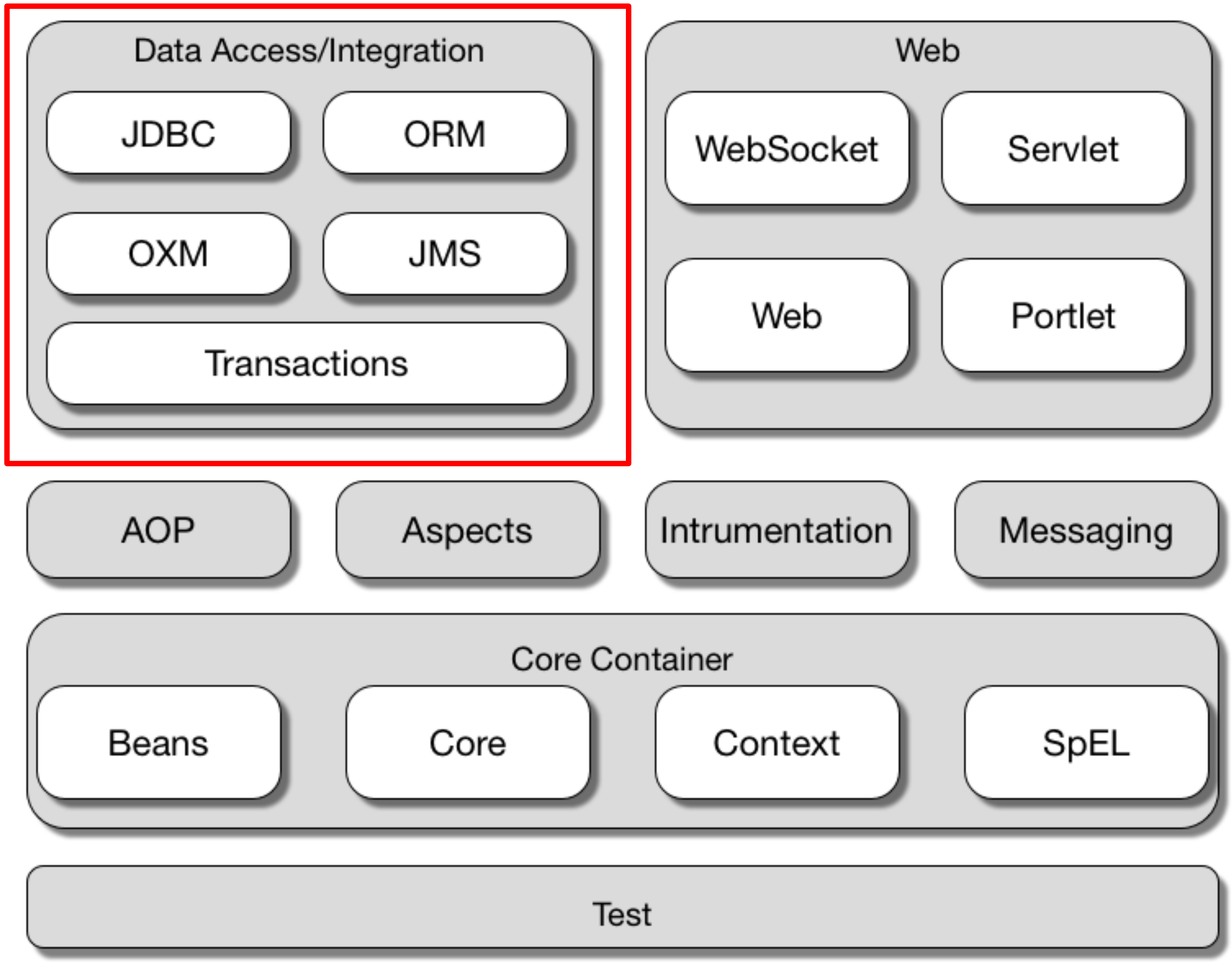
MVC flow

- After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
- The Controller takes the request and calls the appropriate service methods based on used GET or POST method.
  - The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
- The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.
- Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

Controller example

```
@Controller
@RequestMapping("/funcionarios")
public class FuncionariosController {
    @Autowired
    private FuncionarioService funcionarioService;
    @Autowired
    private Funcionarios funcionarios;

    @ResponseBody
    @GetMapping("/todos")
    public List<Funcionario> todos() {
         return funcionarios.findAll();
    }
    @GetMapping("/lista")
    public ModelAndView listar() {
        ModelAndView modelAndView = new ModelAndView("funcionario-lista.jsp");
        modelAndView.addObject("funcionarios", funcionarios.findAll());
        return modelAndView;
    }

// ...
}
```

Spring Framework Architecture



Spring Data Access modules

- **JDBC:** This module provides JDBC abstraction layer which eliminates the need of repetitive and unnecessary exception handling overhead.
- **ORM:** ORM stands for **O**bject **R**elational **M**apping. This module provides consistency/ portability to our code regardless of data access technologies based on object oriented mapping concept.
- **OXM:** OXM stands for **O**bject **X**ML **M**appers. It is used to convert the objects into XML format and vice versa. The Spring OXM provides an uniform API to access any of these OXM frameworks.
- **JMS:** JMS stands for **J**ava **M**essaging **S**ervice. This module contains features for producing and consuming messages among various clients.
- **Transaction:** This module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs. All the enterprise level transaction implementation concepts can be implemented in Spring by using this module.

References

- https://docs.spring.io/spring/docs/current/spring-framework-reference/
- https://www.baeldung.com/spring-tutorial
- https://www.edureka.co/blog/spring-tutorial/