Spring Framework

UA.DETI.IES - 2021/22

José Luís Oliveira

Main topics

- Spring 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.
  - Help developers to focus on implementing business code.
  - They make more effective the 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 sophisticated) dependency injection container into an eco-system of frameworks that cover a broad set of use cases in application development.

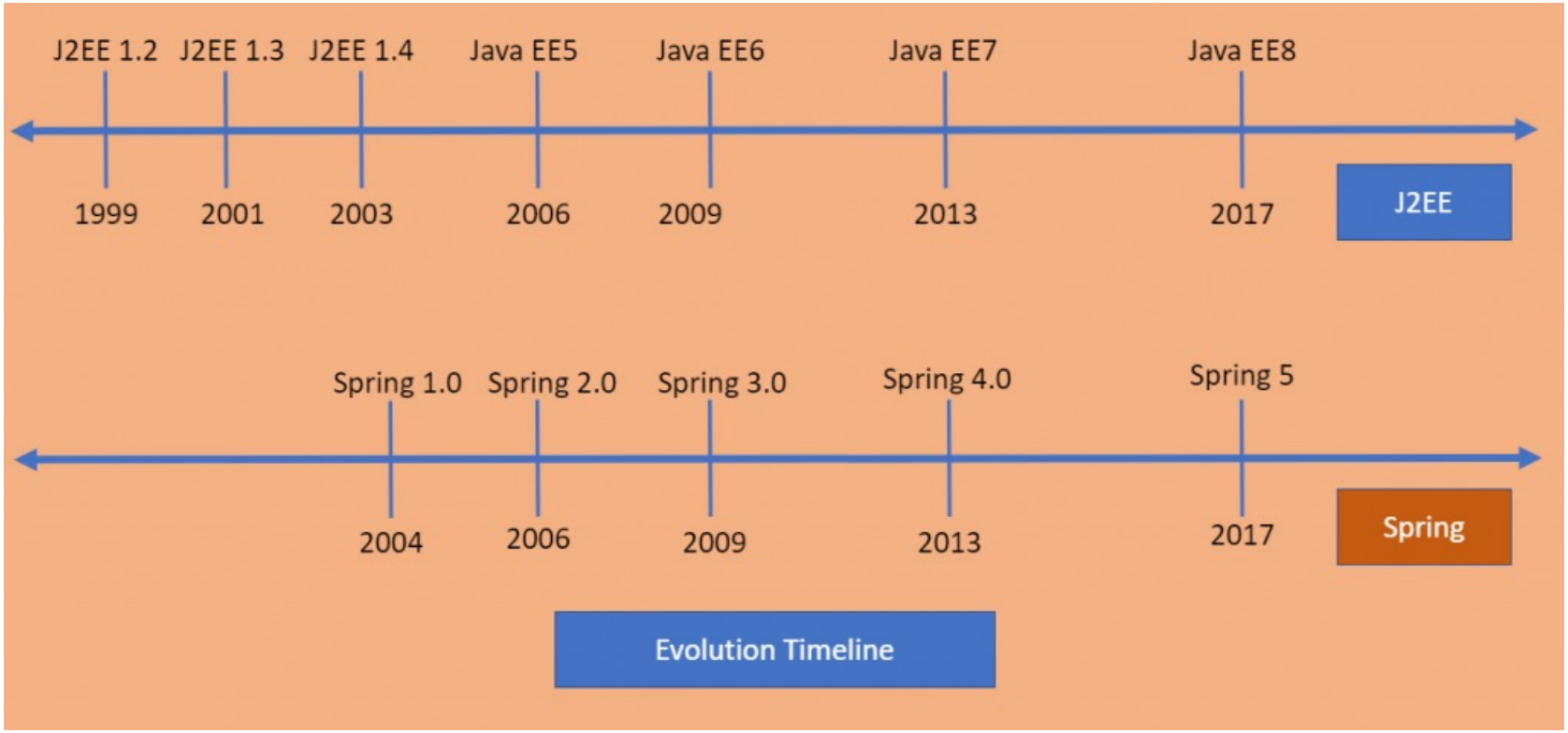
J2EE, Java EE, Jacarta EE – history..

- In the first version of Java, Java enterprise extensions were simply a part of the **core JDK**.
- 1999 – as part of Java 2, these extensions were broken out of the standard binaries
  - **J2EE**, or Java 2 Platform Enterprise Edition, was born. It would keep that name until 2006.
- 2006 – with Java 5, it was renamed to **Java EE** or Java Platform Enterprise Edition.
- 2017 – Oracle give away the rights to the Eclipse Foundation
  - But.. the language was/is still owned by Oracle.
  - Eclipse Foundation legally had to choose a new name: **Jakarta EE** (2018).

https://www.baeldung.com/java-enterprise-evolution

Jakarta EE and Spring

- **Jakarta Enterprise Edition (JEE)** – since 2018
  - formerly **Java Enterprise Edition (JEE)**



Jakarta EE and Spring

- The Spring programming model does not embrace the Java EE platform specification.
- But, it integrates selected specifications from EE:
  - Servlet API (JSR 340)
  - WebSocket API (JSR 356)
  - Concurrency Utilities (JSR 236)
  - JSON Binding API (JSR 367)
  - Bean Validation (JSR 303)
  - JPA (JSR 338)
  - JMS (JSR 914)
  - Spring also supports the Dependency Injection (JSR 330) and Common Annotations (JSR 250) specifications
    - But it provides specific mechanisms for this.

JSR: Java Specification Request.

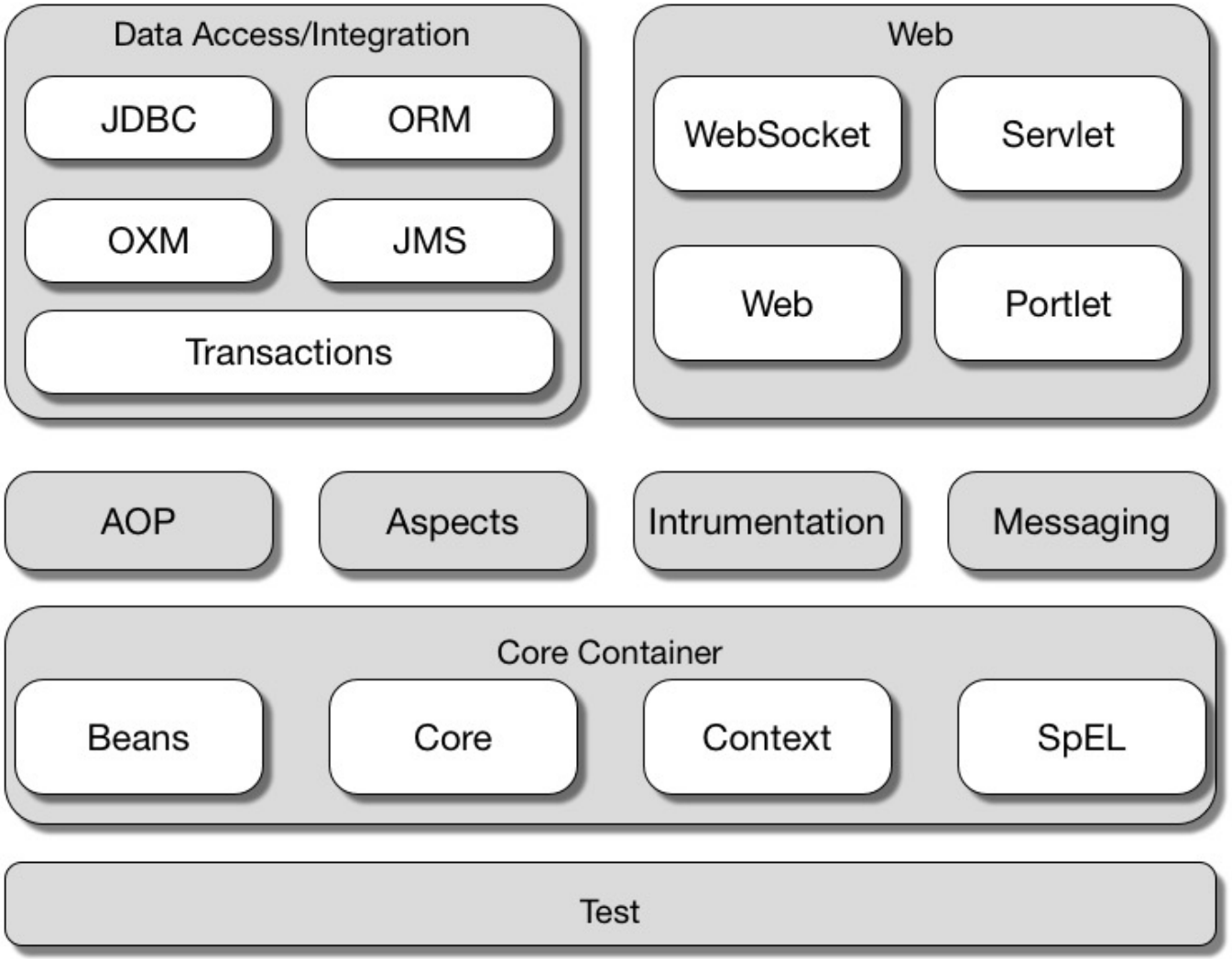
The JSR was a bit like the *interface* for an EE feature.

Spring framework

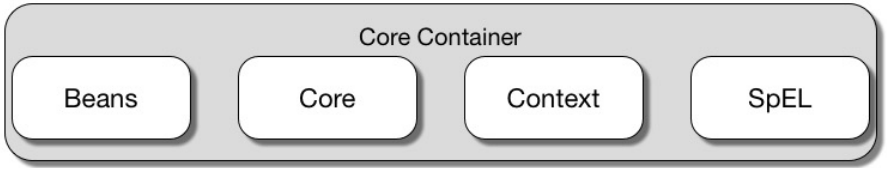
- Development tools

- Beyond the Spring Framework
  - there are other projects such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others.
    - *See spring.io/projects for the complete list of Spring projects.*
  - Each project has its own source code repository, issue tracker, and release cadence

Spring Framework Architecture



Core Container modules



- **Core (spring-core)**
  - The core of the framework it provides Inversion of Control and dependency injection with singleton design pattern.
- **Beans (spring-beans)**
  - This module provides implementation for the factory design pattern through BeanFactory (and others).
- **Context (spring-context)**
  - Builds on Core and Beans and provides a medium to access defined objects and provides support third-party interactions such as caching, mailing, and template engines.
  - ApplicationContext interface is the core part of the Context module.
- **Spring Expression Languages – SpEL (spring-expression)**
  - Enables users to use the Spring Expression Language to query and manipulate the object graph at runtime.

recap.. 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, javadoc, 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:
  - we defined a class
    - e.g., class Person

```
public class Person {
   private String id;
   private String name;
   // ...
}
```

  - a class may define dependencies (dependency injection)
    - e.g., Person *has-a* Address

```
...
private Address address;
...
```

- But … we do not create the objects!
  - We just define how they should be created by the IoC container.

Inversion of Control (IoC) – example

Old way

Person.java

```
public class Person {
   private String id;
   private String name;
   // ...
}
```

beans.xml

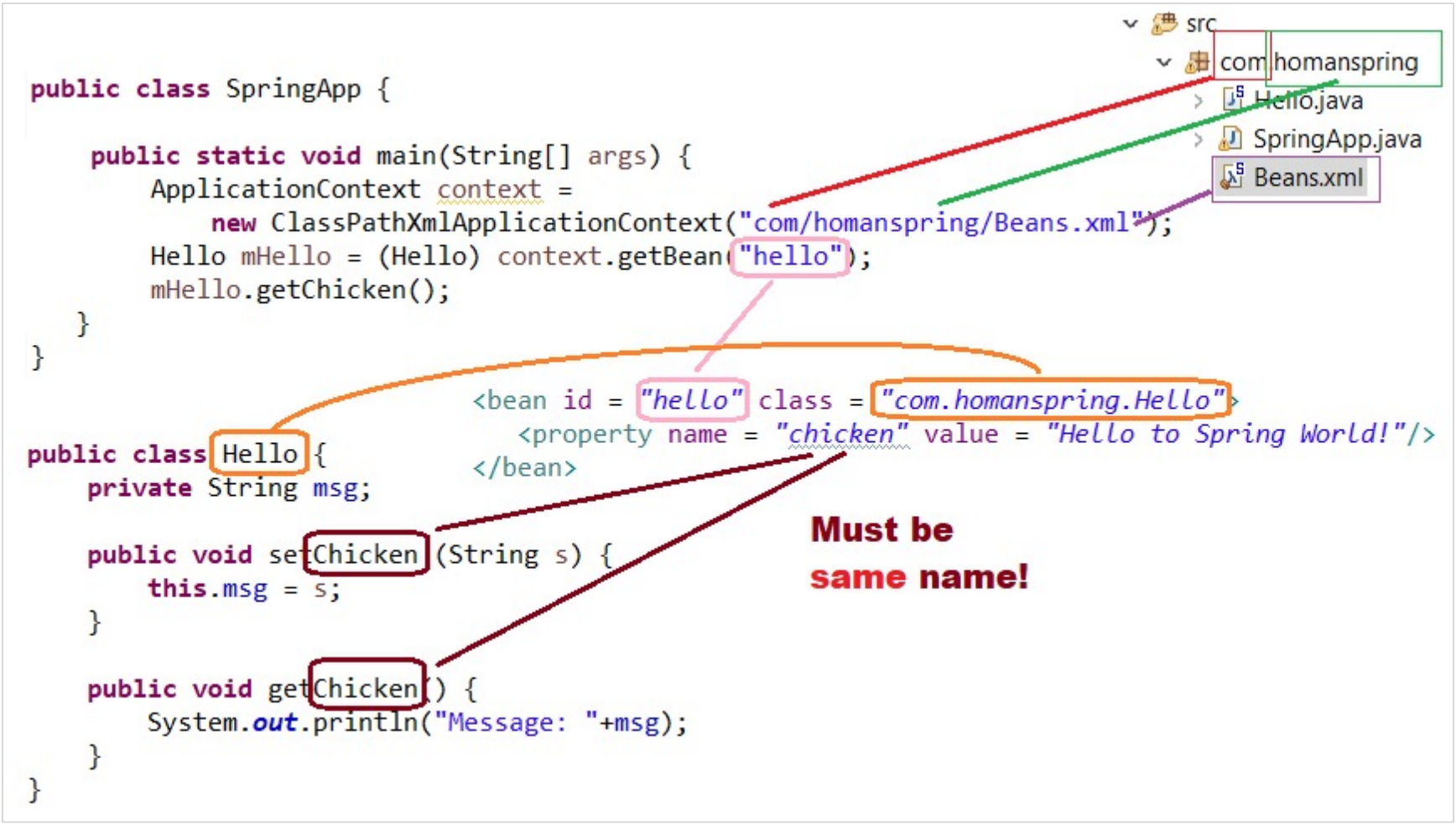
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
<!--  other stuffs ... -->
<beans ...>
   <bean id="person1" class="ua.ies.Person">
     <property name="id" value="12345"/>
     <property name="name" value="Leonor"/>
   </bean>
</beans>
```

Main.java

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    Person p1 = context.getBean("person1", Person.class);
    System.out.println(p1.getId() + ", " + p1.getName()));
}
```

IoC through XML definition is tricky!

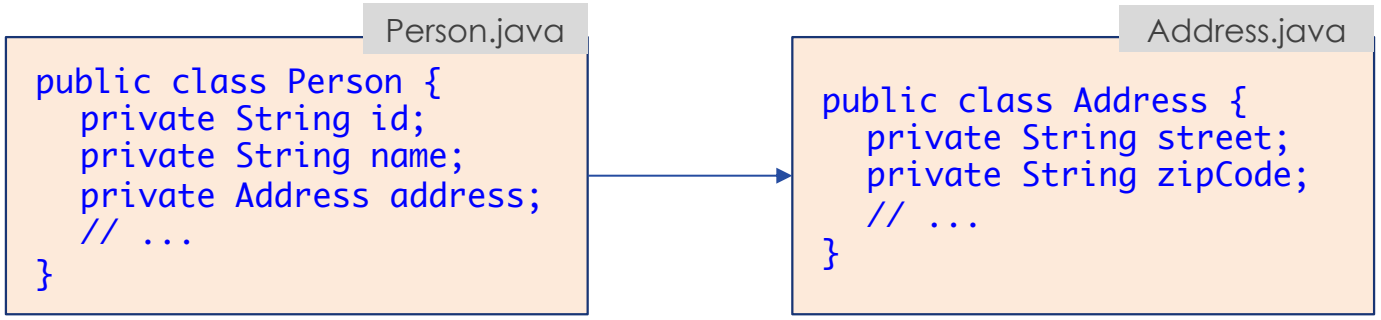
Old way



https://homanhuang.medium.com/java-spring-on-windows-10-part-1-spring-framework-b017ee39aed5

Inversion of Control (IoC) – example

- How, if we have dependencies?



- .. the IoC container needs to create address before person

Dependency Injection

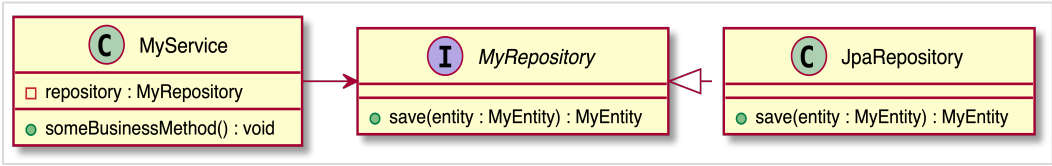
- A design pattern that **removes the dependency from the code**.
  - We may specify classes' dependencies through **annotations** or from external source, such as **XML** file.


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

- Dependencies can be injected in two ways
  - By **constructor**
  - By **setter** method

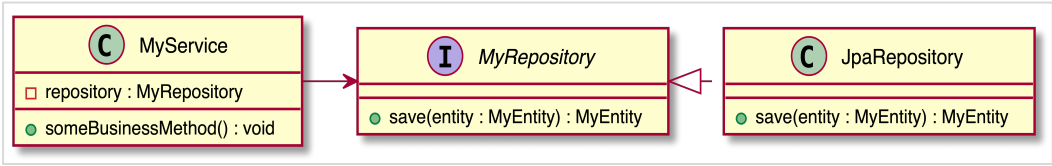
Example (POJO)

(Plain Old Java Object)




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

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

class MyService {
    private final MyRepository repository;
    MyService(MyRepository repository){
      this.repository = repository;
    }
    void setRespository(MyRepository newRepository) {
      this.repository = newRepository;
   }
}

class Main(String ...) {
   MyRepository repository = new JpaRepository(...);
   MyService service = new MyService(repository);
}
```

Example (POJO)




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

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

class MyService {
    private final MyRepository repository;
    MyService(MyRepository repository){
      this.repository = repository;
    }
    void setRespository(MyRepository newRepository) {
      this.repository = newRepository;
    }
}
// Manual dependency injection
class Main(String ...) {
    MyRepository repository = new JpaRepository(...);
    MyService service = new MyService(repository);
}
```

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

- by constructor

- by setter method

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);
...
```

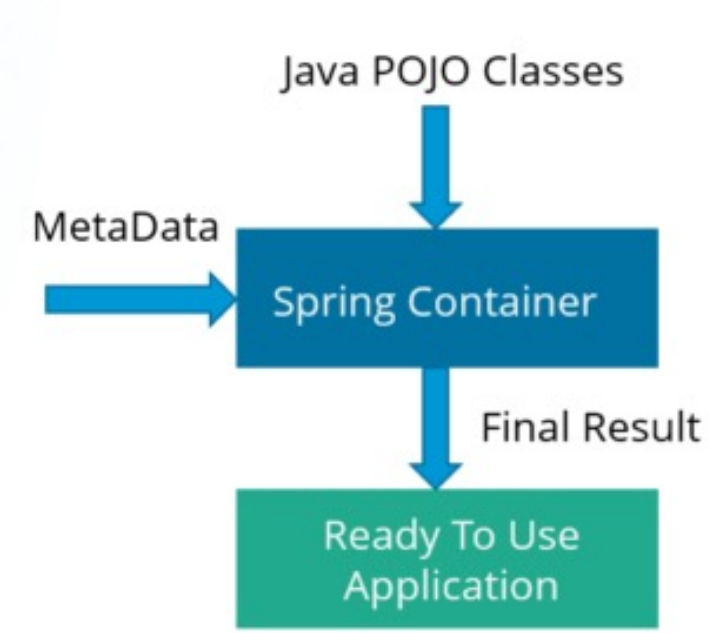
Classes are annotated with framework specific annotations so that it can discover the components of an application that it's supposed to handle.

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 (interfaces):
  - the **BeanFactory** is the simplest container, providing the configuration framework and basic functionality.
  - the **ApplicationContext** built on top of the BeanFactory interface. It adds more enterprise-specific functionality.

ApplicationContext

- org.springframework.context.**ApplicationContext**.
  - It follows eager-initialization technique
    - instance of beans are created with the ApplicationContext.
  - Spring framework provides several implementations

All Known Implementing Classes:

AbstractApplicationContext, AbstractRefreshableApplicationContext, AbstractRefreshableConfigApplicationContext, AbstractRefreshableWebApplicationContext, AbstractXmlApplicationContext, AnnotationConfigApplicationContext, AnnotationConfigWebApplicationContext, ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, GenericApplicationContext, GenericGroovyApplicationContext, GenericWebApplicationContext, GenericXmlApplicationContext, GroovyWebApplicationContext, ResourceAdapterApplicationContext, StaticApplicationContext, StaticWebApplicationContext, XmlWebApplicationContext

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

ApplicationContext context =
   new AnnotationConfigApplicationContext(Config.class);
```

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."

- Some key characteristics:
  - All fields should be private
  - all fields must have setters and getters
  - there should be a no-arg constructor
  - fields are accessed exclusively by the constructor or the getter/setter methods

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

What is a bean?

- 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.

Defining a Spring bean

- There are **three different ways** to define a Spring bean:
  - declaring a bean definition in an XML configuration file
  - 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
- In modern projects, we use only the last two, component annotations and bean factory methods.
  - Spring still allows the XML configuration 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 must create factory methods with the @Bean annotation in a bean configuration class.
  - If we don't want to make a class dependent on Spring, we can also use this option for classes we own.

```
@Configuration
class MyConfigurationClass {

   @Bean
   public static 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.

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;
   }
}
```

- But 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.

The @Autowired annotation

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

- Spring uses the bean's name as a default qualifier value.
  - It will inspect the container and look for a bean with the exact name as the property to autowire it.

- We can use autowiring on constructors, properties, and setters.

The @Autowired annotation

- on **Constructors**

```
public class ProductController {
    private ProductService productService;
    @Autowired
    public ProductController(ProductService service) {
        productService = service;
    }
}
```

- on **Properties**

```
public class ProductController {
   @Autowired
   private ProductService productService;
   ...
}
```

- on **Setters**

```
public class ProductController {
    private ProductService productService;
    @Autowired
    public void setProductService(ProductService service) {
        productService = service;
    }
}
```

Some other annotations

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

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

- **@Required**
  - is used on setter methods to tell Spring that these fields are required to initialize the bean. Otherwise an exception of type BeanInitializationException is thrown.
- **@Value**
  - As the @Autowired annotation tells Spring to inject object into another when it loads your application context, we can also use @Value annotation to inject values from a property file into a bean's attribute. More on SpEL…

Component scanning – Example

```
@Configuration
@ComponentScan
public class SpringComponentScanApp {
    private static ApplicationContext applicationContext;

    @Bean
    public ExampleBean exampleBean() {  return new ExampleBean();    }

    public static void main(String[] args) {
        context =
            new AnnotationConfigApplicationContext(SpringComponentScanApp.class);

        for (String beanName : context.getBeanDefinitionNames()) {
            System.out.println(beanName);
        }
    }
}
```

```
springComponentScanApp
cat
dog
exampleBean
```

```
@Component
public class Cat {}
```

```
@Component
public class Dog {}
```

Spring Expression Language (SpEL)

- SpEL can be used for manipulating and querying 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:
  - Arithmetic +, -, *, /, %, ^, div, mod
  - Relational <, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge
  - Logical and, or, not, &&, | |, !
  - Conditional ?:
  - Regex matches

SpEL examples

- 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'}
```

- 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.

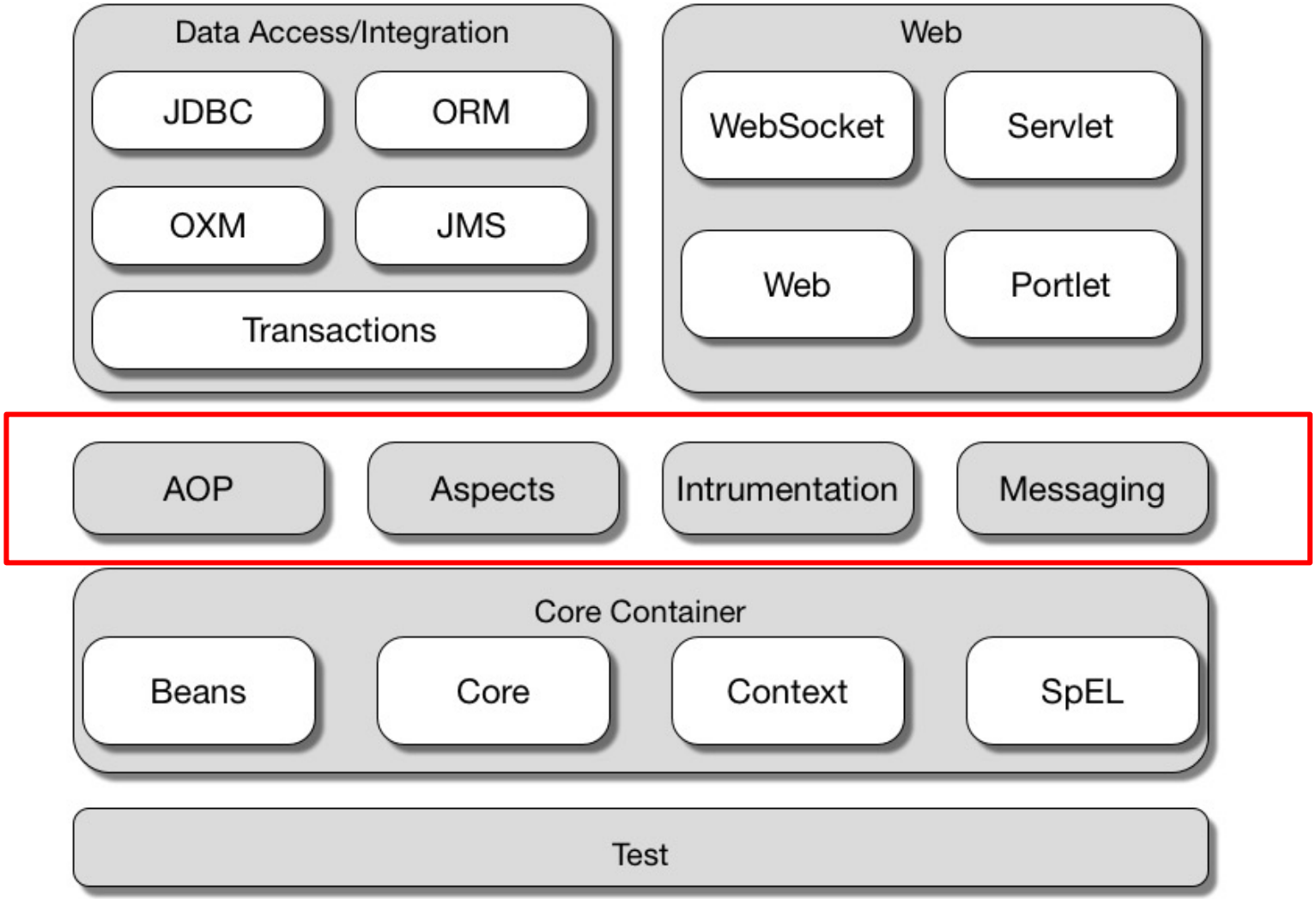
```
@Value ("#{'${employee.names}'.split(',')}")
private List<String> employeeNames;
```

```
[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;
```

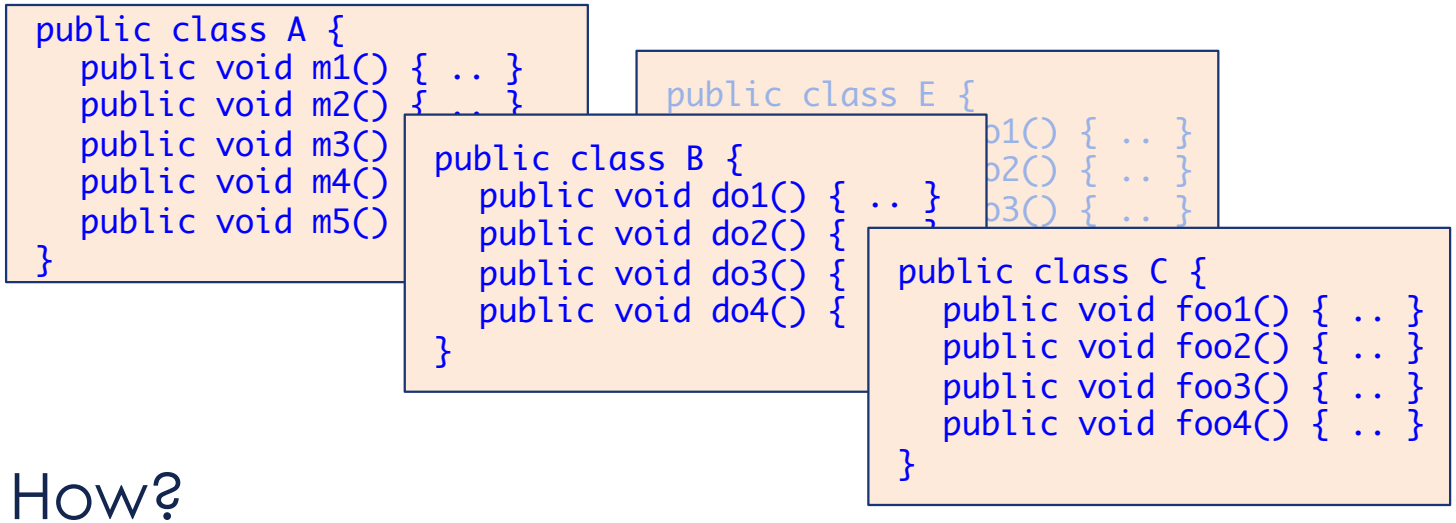
Spring Framework Architecture



Aspect Oriented Programming (AOP)

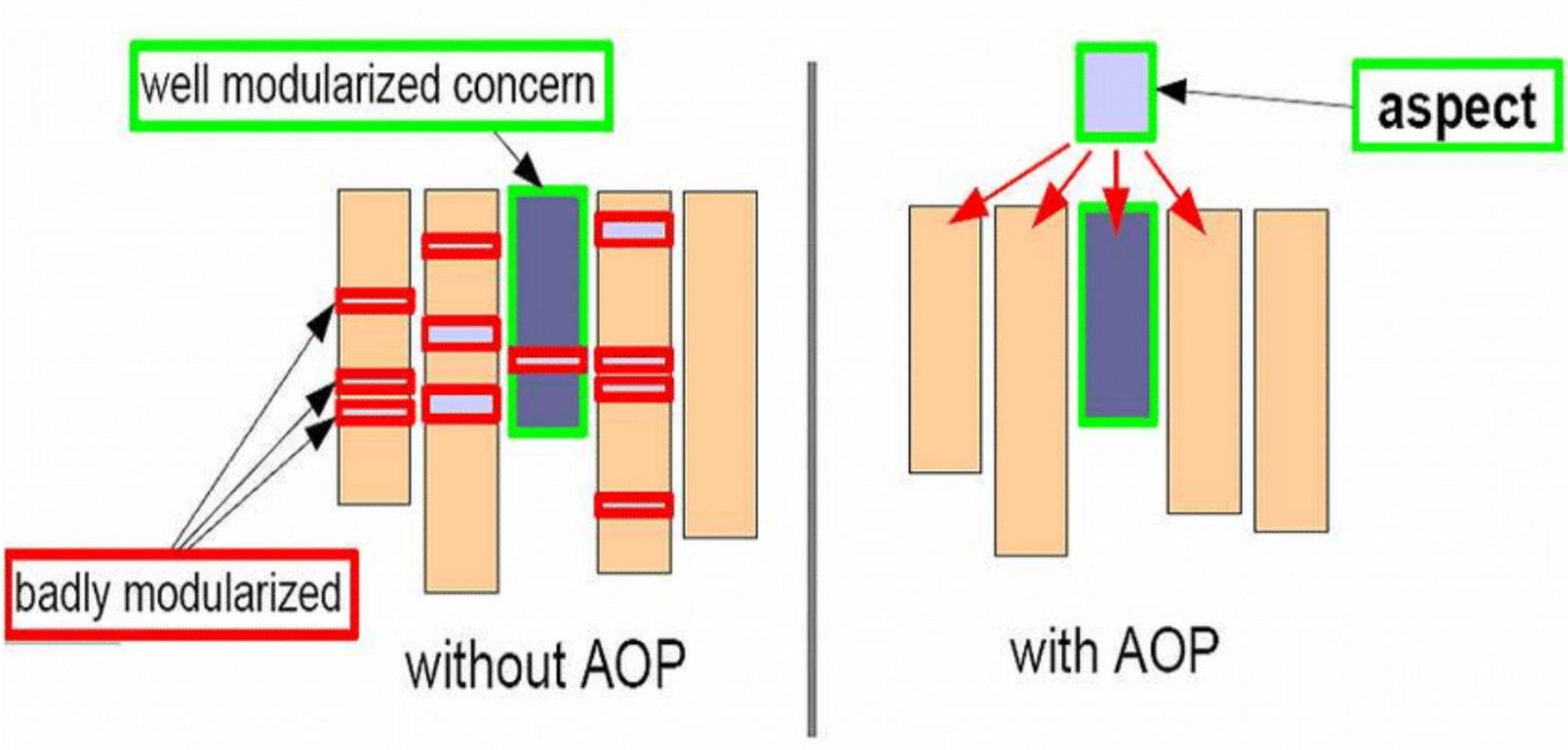
- Why AOP? An example:

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



- How?

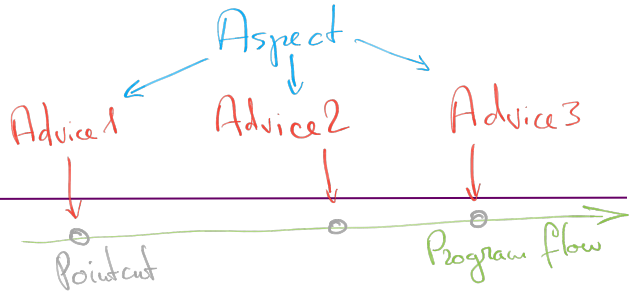
Aspect Oriented Programming (AOP)



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).
  - It can be a normal **class**, configured through XML configuration or through the annotation **@Aspect**.
- **Join point**: is a point during the execution of a program, e.g., a method or the handling of an exception.
- **Advice:** action taken by an aspect at a particular join point.
  - @Before, @After, @Around, @AfterReturning, @AfterThrowing.
- **Pointcut:** expressions that are matched at join points to determine whether advice needs to be executed or not.

AOP Examples - pointcut

- 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

```
public class Employee {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String nm) {
        this.name=nm;
    }
    public void throwException(){
        throw new RuntimeException("Dummy Exception");
    }
}

@Aspect
public class EmployeeAspect {
    @Before("execution(public String getName())")
    public void getNameAdvice(){
        System.out.println("Executing Advice on getName()");
    }
    @Before("execution(* ies.*.get*())")
    public void getAllAdvice(){
        System.out.println("Service method getter called");
    }
}
```

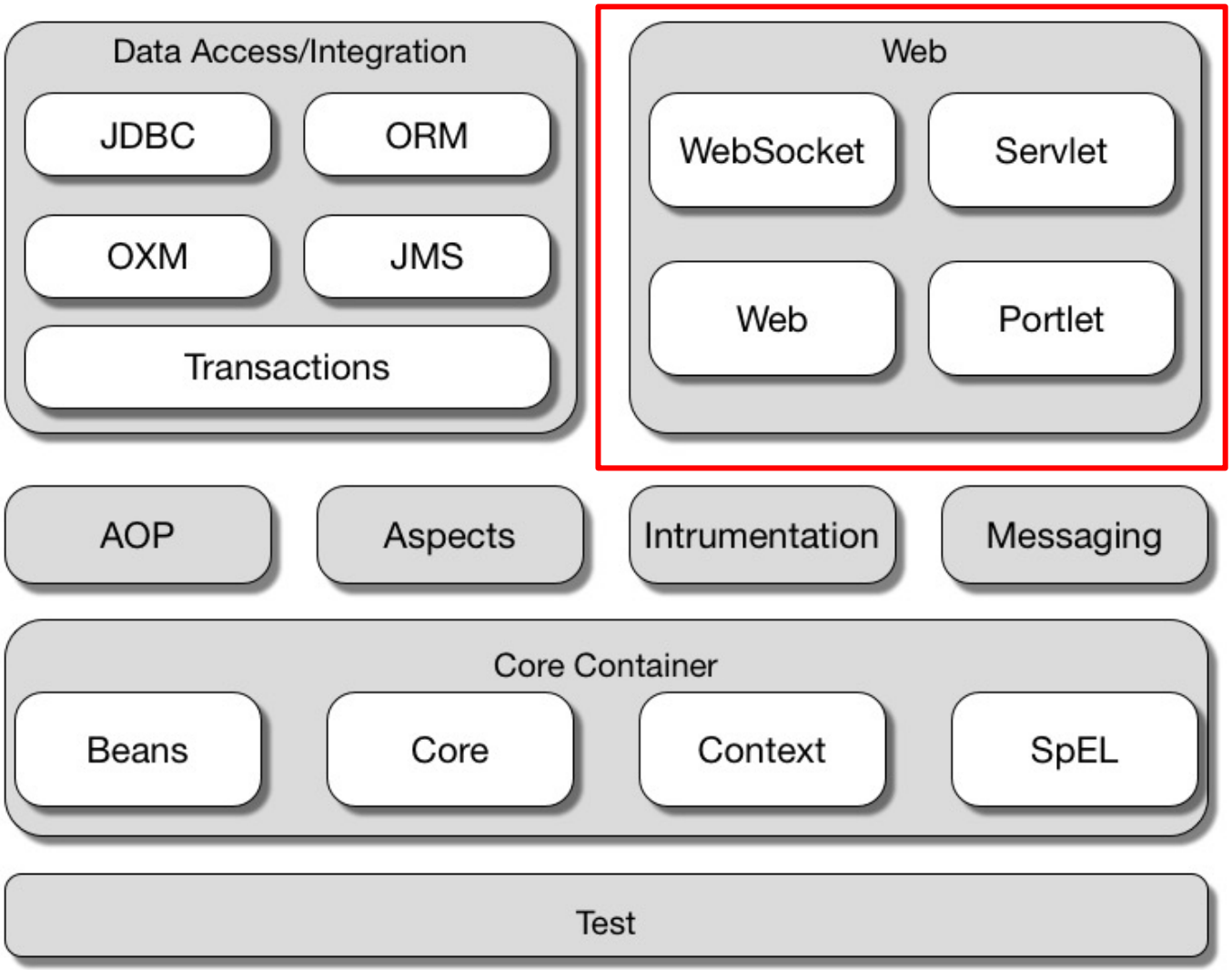
AOP Examples

```
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(){}
}
```

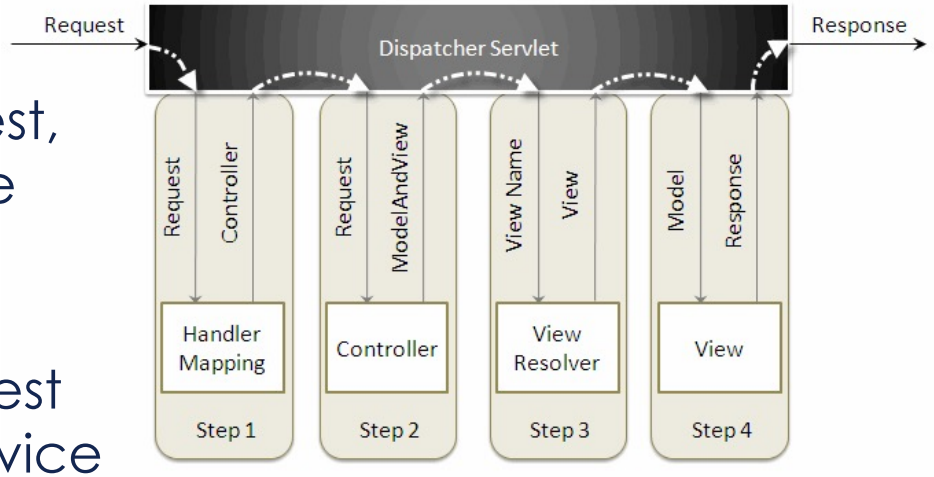
AOP Examples

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

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

@Component
class JpaRepository implements MyRepository {

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

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)

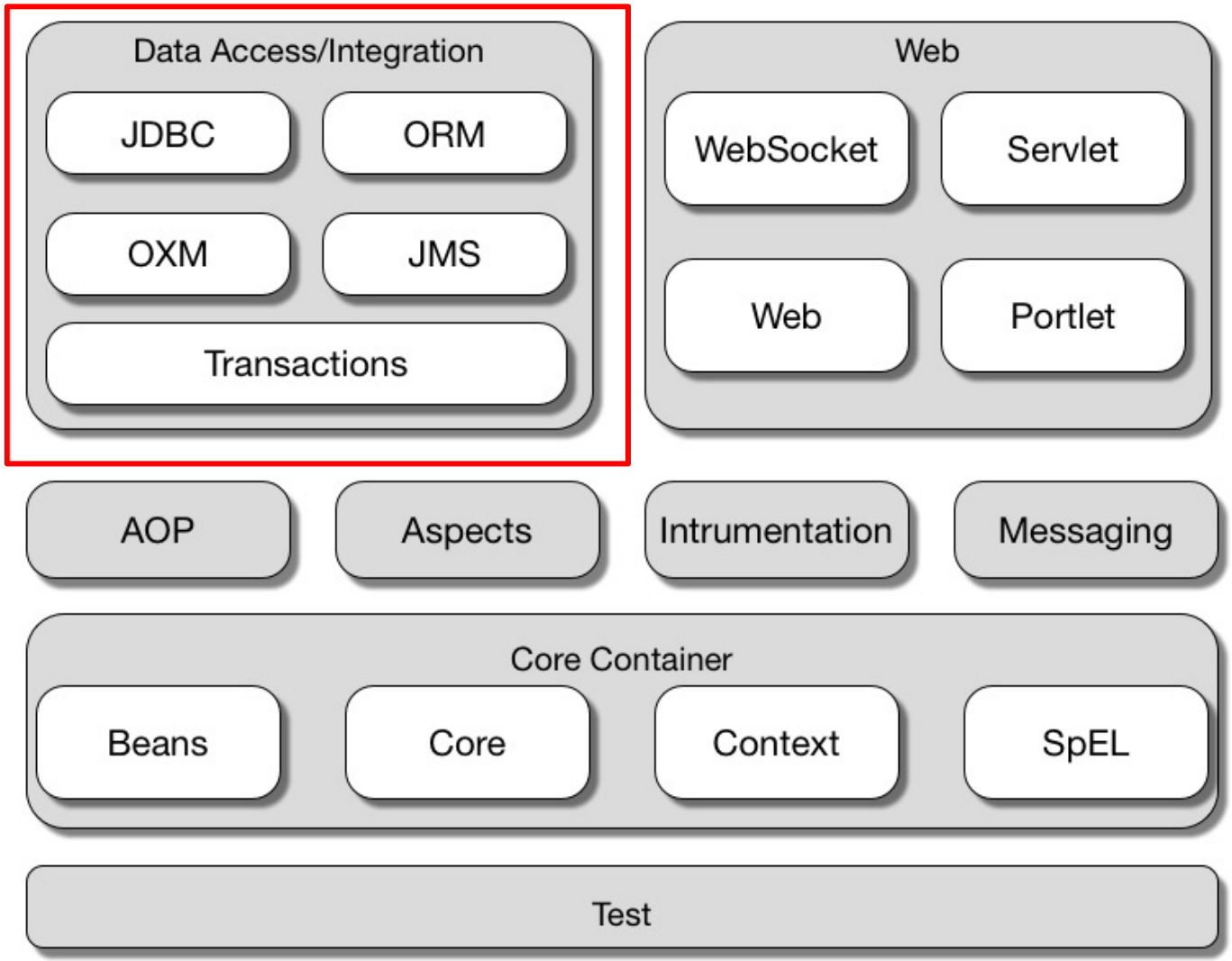


- After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
- The Controller takes the request and calls the appropriate service methods based on used GET or POST method.
  - The service method will set model data based on defined business logic 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")  // a kind of path..
public class FuncionariosController {
    @Autowired
    private FuncionarioService funcionarioService;
    @Autowired
    private Funcionarios funcionarios;

    @ResponseBody
    @GetMapping("/todos")  // curl -i http://localhost:8080/iesapp/funcionarios/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

- *… and many others*