Deep Dive Into

Spring Framework 5

— Nandakumar Purohit

Spring Framework

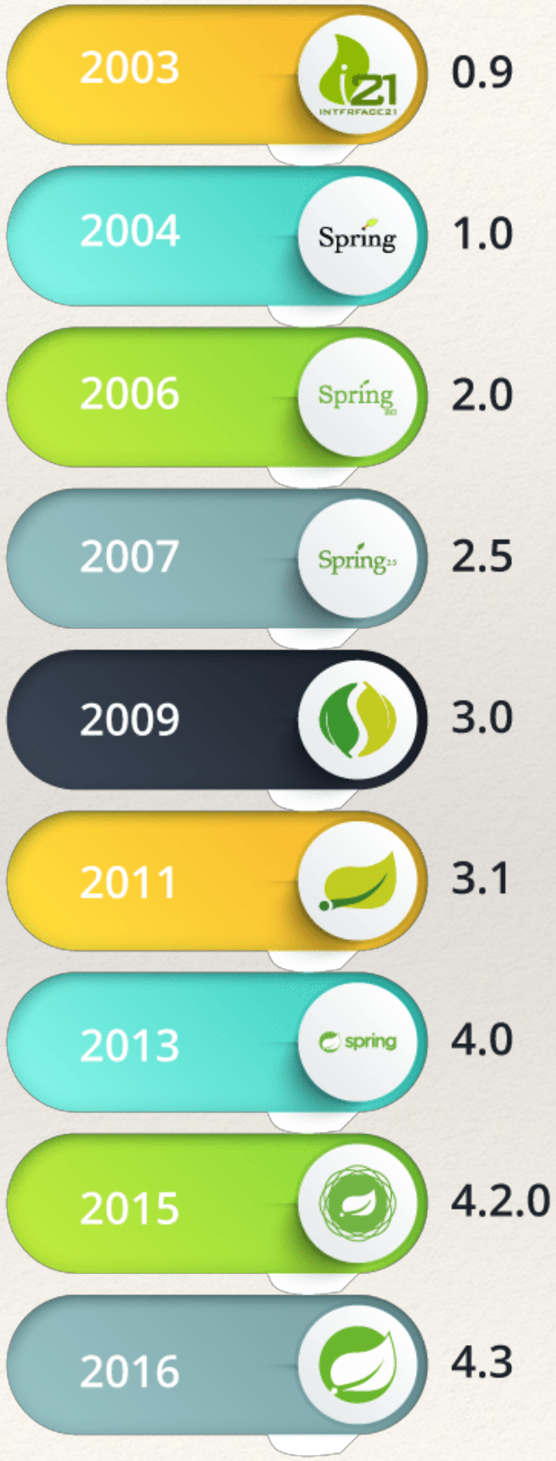
Agenda

Fundamentals
Architecture
IoC Container & Lifecycle
Spring XML Configuration
Java Configuration using Annotations
Bean Scopes
Properties
Spring EL

AOP & AspectJ
Introduction
Aspects
Advices
Pointcuts & Wildcards
Join Points
Custom Advices

Spring Framework

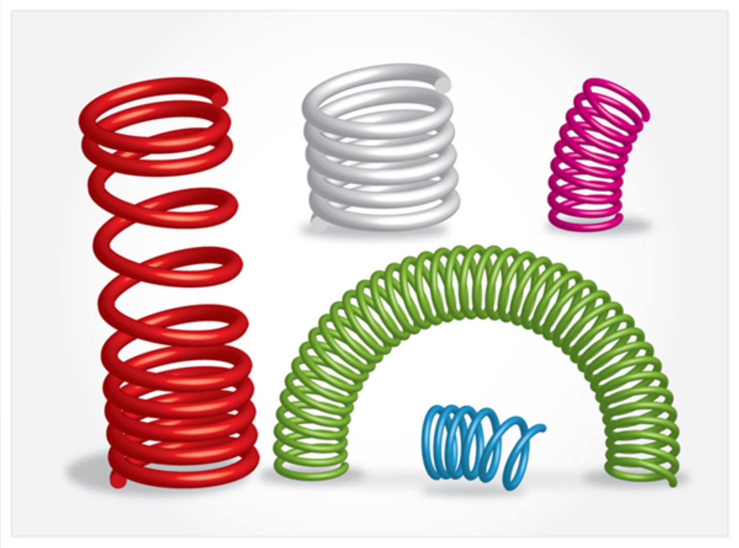
History - The Beginning



- In October 2002, Rod Johnson wrote a book titled Expert One-on-One J2EE Design and Development
- This book covered the state of Java enterprise application development at the time and pointed out a number of major deficiencies with Java EE and EJB component framework.
- In the book he proposed a simpler solution based on ordinary java classes (POJO – plain old java objects) and dependency injection
- In the book, he showed how a high quality, scalable online seat reservation application can be built without using EJB.

Spring Framework

What is Spring?



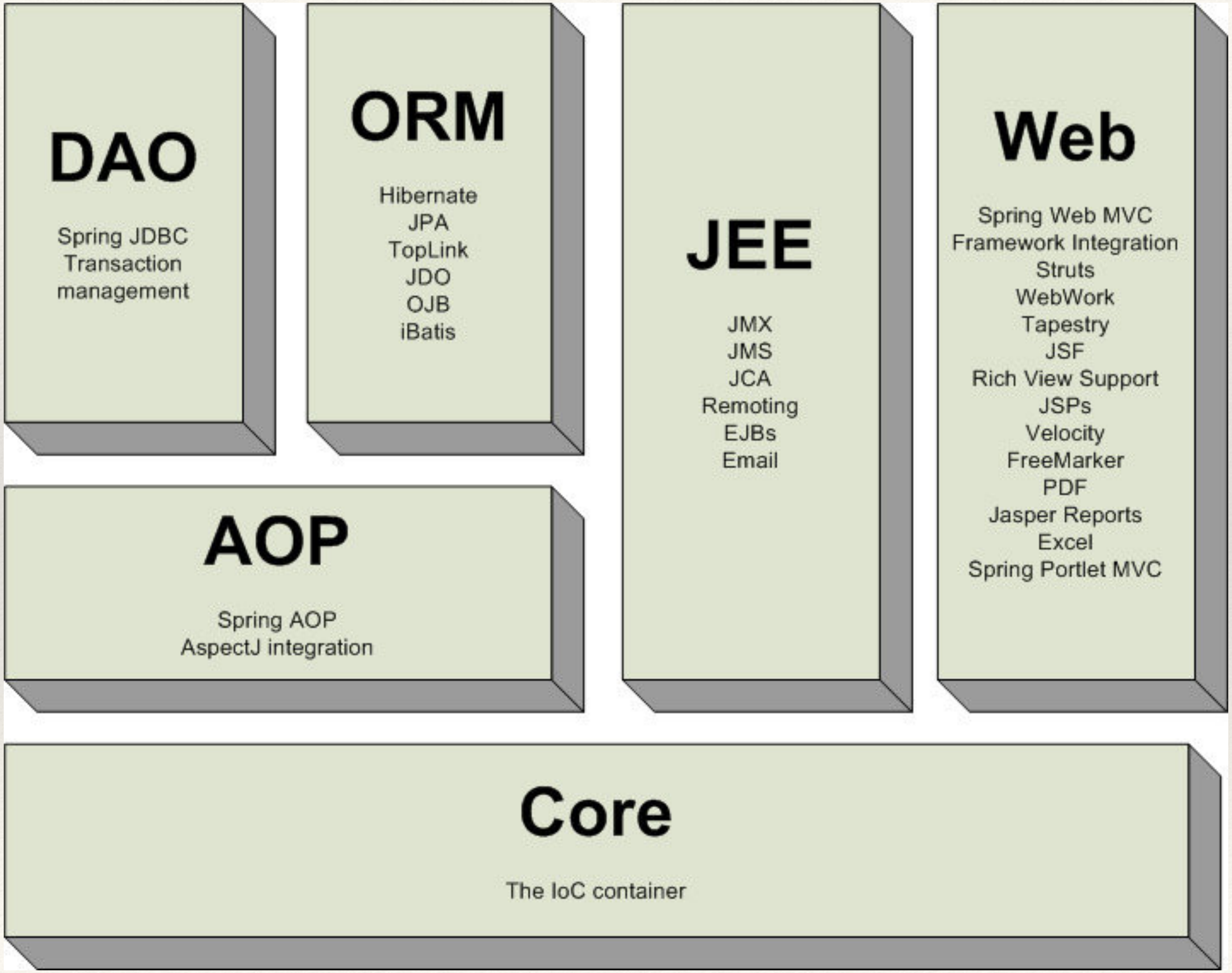
spring

by Pivotal

- IoC Container
- Reduce or replace configuration
- The original concept was how to work better with EJBs
- Enterprise Development without Application Server
- Tomcat is a Standard Java Development Container
- Completely POJO-based
- Is for better, cleaner code with POJO & Interface driven
- Used Best Practices

Spring Framework

The Architecture



DAO

Spring JDBC
Transaction
management

ORM

Hibernate
JPA
TopLink
JDO
OJB
iBatis

JEE

JMX
JMS
JCA
Remoting
EJBs
Email

Web

Spring Web MVC
Framework Integration
Struts
WebWork
Tapestry
JSF
Rich View Support
JSPs
Velocity
FreeMarker
PDF
Jasper Reports
Excel
Spring Portlet MVC

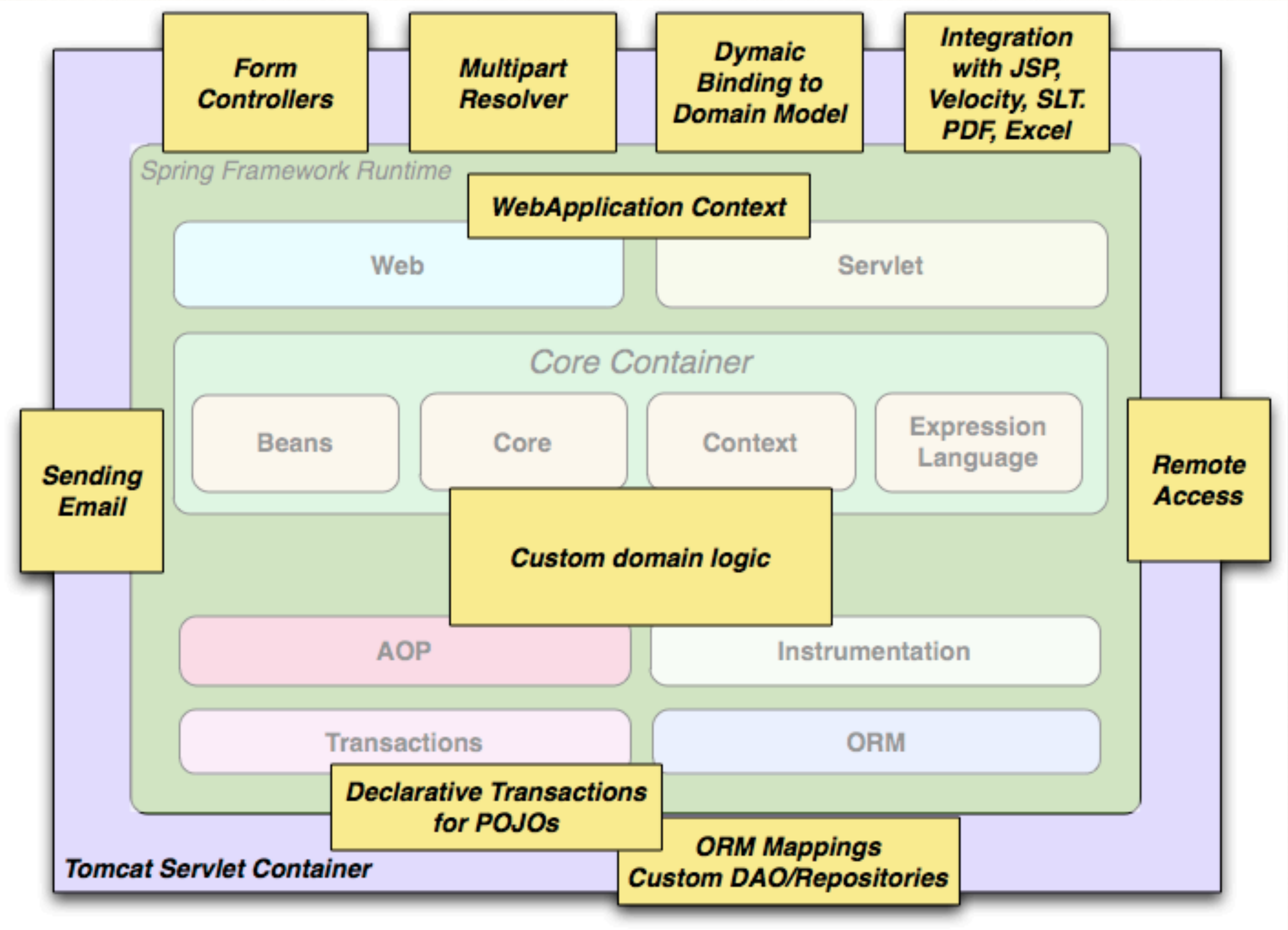
AOP

Spring AOP
AspectJ integration

Core

The IoC container

Spring Framework - Architecture



Spring Framework

The Problem

- Testability
- Maintainability
- Scalability
- Complexity

Spring Framework

Business Focus

```
public Car getById(String id) {
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        String sql = "select * from CAR where ID = ?";
        con = DriverManager.getConnection("localhost:3306/cars");
        stmt = con.prepareStatement(sql);
        stmt.setString(1, id);
        rs = stmt.executeQuery();
        if(rs.next()) {
            Car car = new Car();
            car.setMake(rs.getString(1));
            return car;
}
else {

            return null;
        }
    } catch (SQLException e) { e.printStackTrace();}
    finally {
        try {
            if(rs != null && !rs.isClosed()) {
                rs.close();
            }
        } catch (Exception e) {}
    }
    return null;
}
```

Spring Framework

The Solution

- Configuration
- Focus
- Testing
- Annotation or XML Based

Spring Framework

How Simplified?

JAVA

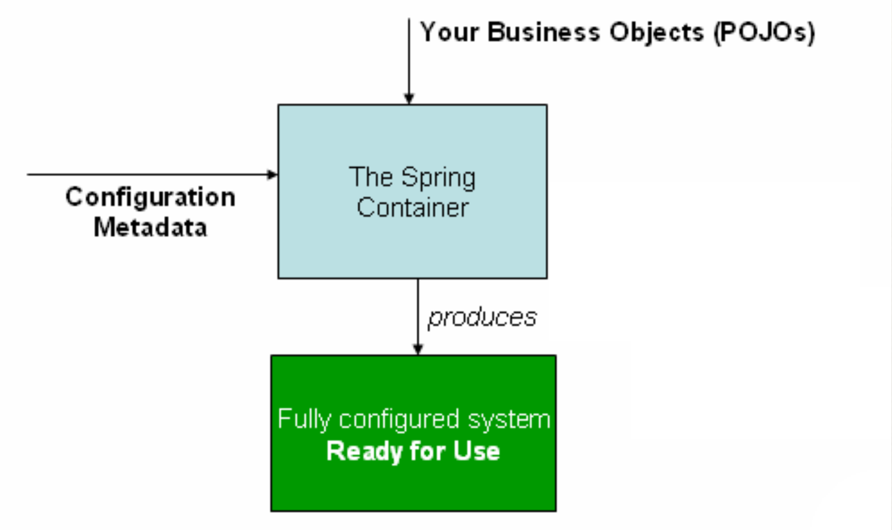
```
public Car getById(String id) {
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        String sql = "select * from CAR where ID = ?";
        con = DriverManager.getConnection("localhost:3306/cars");
        stmt = con.prepareStatement(sql);
        stmt.setString(1, id);
        rs = stmt.executeQuery();
        if(rs.next()) {
            Car car = new Car();
            car.setMake(rs.getString(1));
            return car;
}
else {
            return null;
        }
    } catch (SQLException e) { e.printStackTrace();}
    finally {
        try {
            if(rs != null && !rs.isClosed()) {
                rs.close();
            }
        } catch (Exception e) {}
    }
    return null;
}
```

Spring

```
public Car findCar(String id) {
    return getEntityManager().find(Car.class, id);
}
```

Spring Framework

What is IoC?

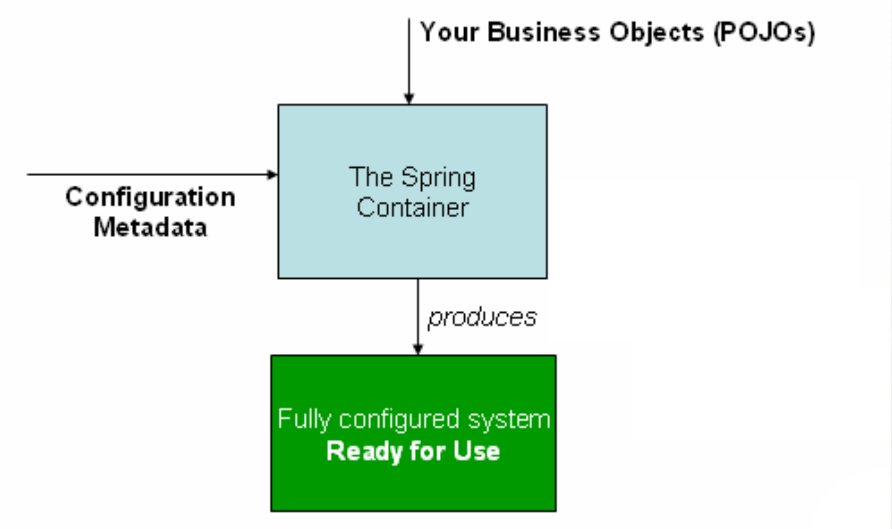


"IoC is a principle of a Software Engineering by which the control of objects or portion of a program is transferred to a container or framework".

- Mechanisms for IoC
  - Factory Pattern
  - Service Locator Pattern
  - Strategy Design Pattern
  - Dependency Injection
- Advantages
  - Easy context switch between implementations
  - Abstracting implementation from execution
  - Modular programming
  - Components communication via contracts

Spring Framework

What is IoC?

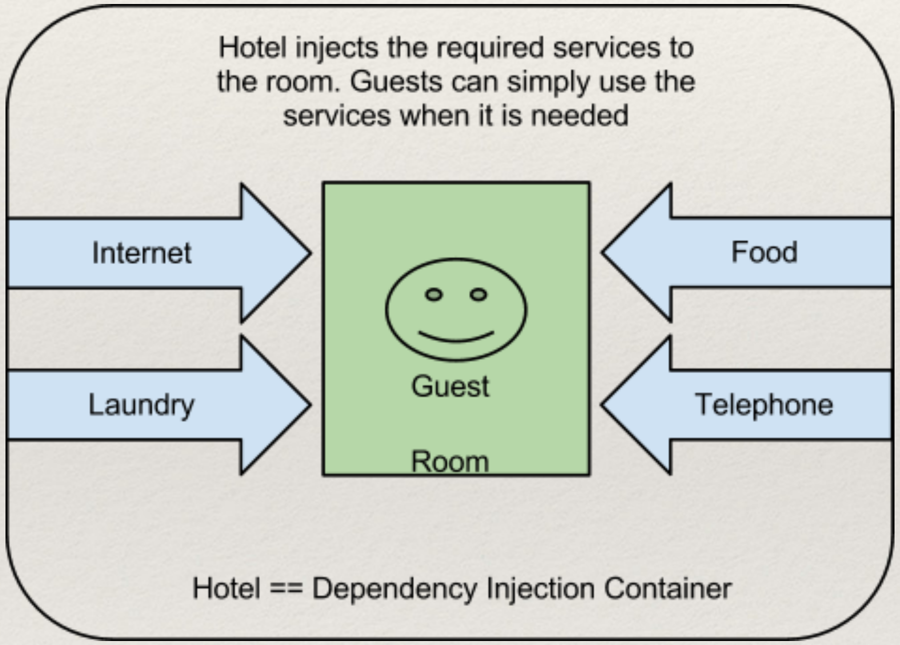


IoC Container

- Responsible for instantiating, configuring & assembling the beans
- The container is instructed through configuration metadata
- The configuration metadata is represented as XML or Java Annotations
- It allows to express objects that compose your application & rich interdependencies between those objects

Spring Framework

What is Dependency Injection (DI)?


Hotel injects the required services to the room. Guests can simply use the services when it is needed
Internet
Laundry
Food
Telephone
Guest
Room
Hotel == Dependency Injection Container

"DI is one of the implementation mechanism for IoC, where objects define their dependent objects with which they work. This is done by an assembler rather than by the objects themselves".

- Methods of DI
  - Setter methods
  - Constructor Arguments
- Advantages
  - The object does not lookup its dependencies
  - The object does not know the location of the dependencies

Spring Framework

Demo Setup

- spring_sample application

Spring Framework

Pain Points

- Service shouldn't be aware of some things
- Values not to be hard coded

Spring Framework

XML Configuration

- First Approach
- Simpler
- Separation of Concerns

Spring Framework

XML Configuration

- Add Spring dependency to spring_sample application
- Create a copy of spring_sample application

Spring Framework

applicationContext.xml

- Name doesn’t matter
- Spring context sort of a HashMap
- Can simply be a registry
- XML Configuration begins with this file
- Namespaces aid in configuration/validation

Spring Framework

Namespaces

```
<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">
```

Spring Framework

XML Declaration

```
<bean name="customerService"
   class="com.demo.service.CustomerServiceImpl"
   autowire="byName">
     <property name="0" ref="customerRepository">


   </bean>
```

Spring Framework

Beans

```
<bean name="customerRepository"
    class=“com.demo.repository.HibernateCustomerRepositoryImpl" />
```

Beans

Essentially Classes

Replaces keyword ‘new’

Define Class, use Interface

“We can now change our configurations without recompiling our code. We could switch ENV from DEV to TEST just by pointing to config files without recompiling sources. This technique is called as **Separation of Concerns**”

Spring Framework

Injections

- Setter Injection
- Constructor Injection
- Used together

Spring Framework

Setter Injection

- Primitive & String based values
- Dependent Object (Contained Object)
- Supports Collection Values

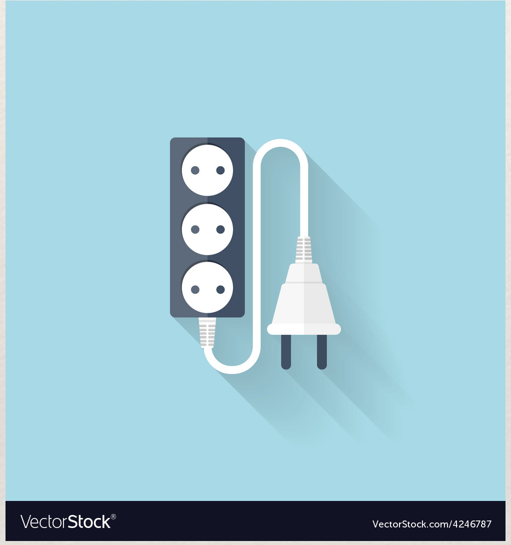
Spring Framework

Constructor Injection

- Guaranteed Contract
- Constructor Defined
- Used together
- Index based

Spring Framework

Autowire



- Spring automatically Wires Beans
- byType
- byName
- Constructor
- no

Spring Framework

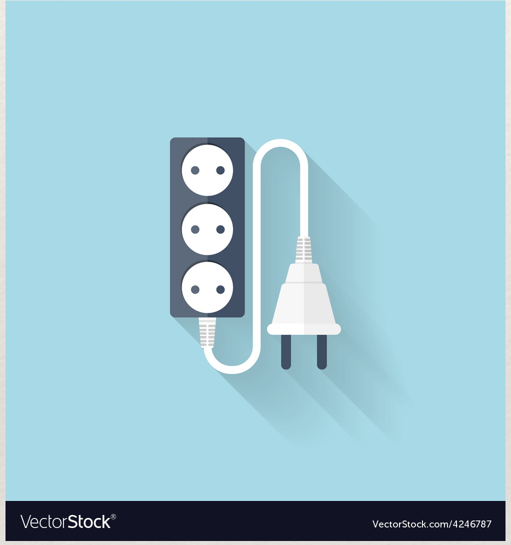
Spring Annotation Configuration Using XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
  ……
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-
context-4.3.xsd">

  <context:annotation-config />

  <context:component-scan base-package="com.demo" />

</beans>
```

Stereotype Annotations

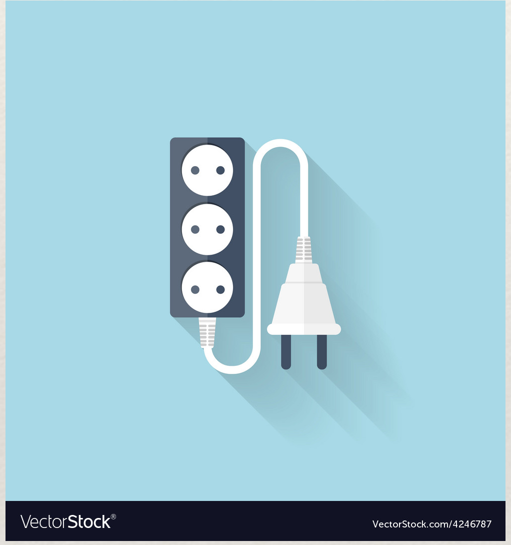
- @Component, @Service, @Repository
- Semantically the same
- @Component - any POJO
- @Service - business logic layer
- @Repository - data layer

Autowire Annotation



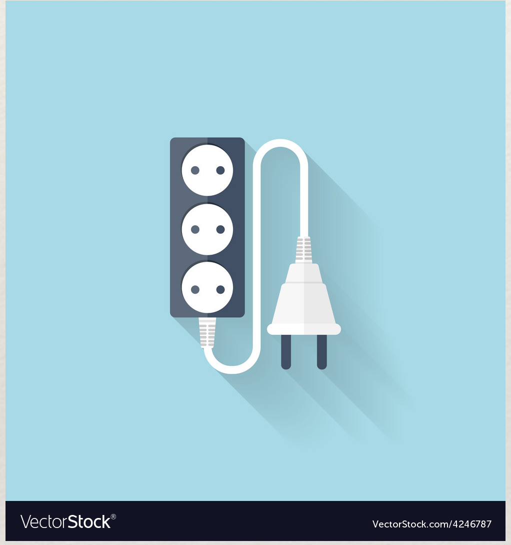
- Better with Annotations
- Tied to location
- Member Variables
- Constructor
- Setter

Autowired Member Variable



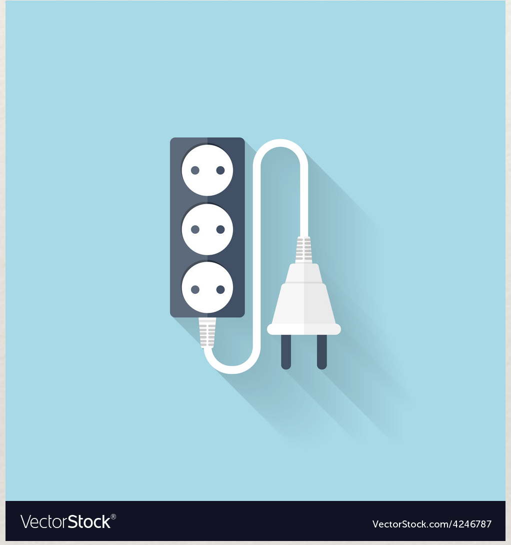
```
@Autowired
private CustomerRepository customerRepository;
```

Autowired Setter Injection



```
@Autowired
    public void setCustomerRepository(CustomerRepository
customerRepository) {
        System.out.println("We are using Setter Injection
@Autowired...");
        this.customerRepository = customerRepository;
    }
```

Autowired Constructor Injection



```
@Autowired
    public CustomerServiceImpl(CustomerRepository
customerRepository) {
        System.out.println("We are using Constructor
@Autowired");
        this.customerRepository = customerRepository;
    }
```

Spring Configuration Using JAVA

Completely Annotation based Spring Application

Spring Configuration Using JAVA

```
@Configuration
public class AppConfig {

    @Bean(name = "customerService")
    public CustomerService getCustomerService() {
        return new CustomerServiceImpl();
    }

}
```

@Configuration

applicationContext.xml replaced by @Configuration

@Configuration at Class level

Spring Beans defined by @Bean

@Bean at method level

Spring Configuration Using JAVA

Setter Injection

```java
@Bean(name = "customerService")
    public CustomerService getCustomerService() {
        CustomerServiceImpl service = new CustomerServiceImpl();
        service.setCustomerRepository(getCustomerRepository());

        return service;
    }

@Bean(name  = "customerRepository")
    public CustomerRepository getCustomerRepository() {
        return new HibernateCustomerRepositoryImpl();
    }
```

Setter Injection

Simple as a Method Call

“Mystery” of injection goes away

Setter Injection simply calling a setter

Spring Configuration Using JAVA

Constructor Injection

```
public CustomerServiceImpl(CustomerRepository
customerRepository) {
     System.out.println("We are using
constructor injection");
     this.customerRepository =
customerRepository;
   }
```

Spring Configuration Using JAVA

Autowired Annotation for JAVA Configuration

@ComponentScan({“com.demo”})

@Bean

Instance Type

Spring Configuration Using JAVA

Bean Scopes

5 Scopes

Valid in any configuration

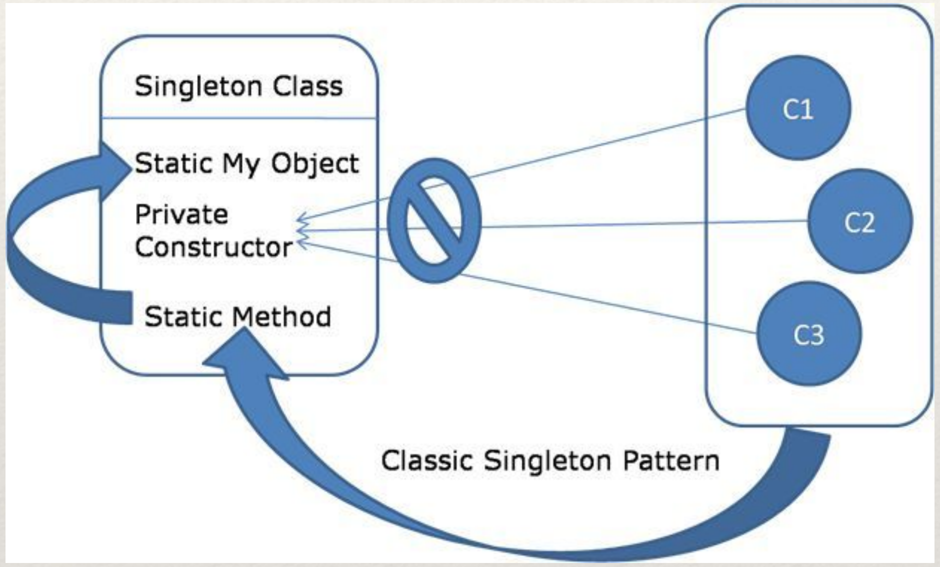
- Singleton
- Prototype

Valid only in Web-aware Spring Applications

- Request
- Session
- Global

Spring Configuration Using JAVA

Singleton



Singleton in Java

- Single Instance per JVM Instance
- Lazy / Eager types

Singleton Bean in Spring

- One Instantiation
- Default Bean Scope
- Single Instance per Spring Container

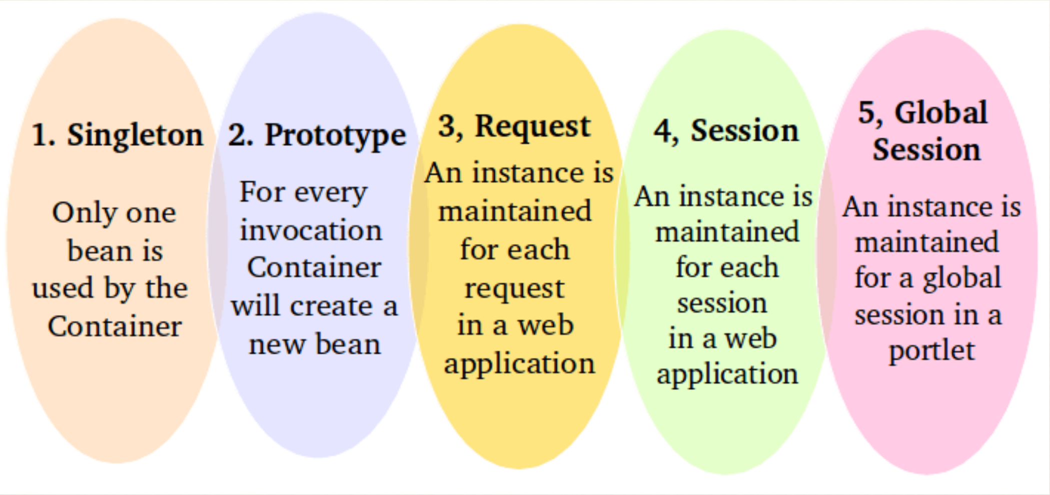
Spring Configuration Using JAVA

Singleton

```
For JAVA Configuration
@Service("customerService")
@Scope("singleton")
public class CustomerServiceImpl implements CustomerService {


For XML Configuration
<bean name="customerService"
    class="com.demo.service.CustomerServiceImpl"
    autowire="byType"
    scope=“singleton”>

@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
```

Spring Configuration Using JAVA

Prototype



1. Singleton

Only one bean is used by the Container

2. Prototype

For every invocation Container will create a new bean

3, Request

An instance is maintained for each request in a web application

4, Session

An instance is maintained for each session in a web application

5, Global Session

An instance is maintained for a global session in a portlet

Per request

Guaranteed Unique

Opposite of Singleton

Spring Configuration Using JAVA

@Component

- @Component is one Stereotype annotation used to mark a Class as a Component
- @Service & @Repository annotations inherit @Component
- All these 3 annotations are complimentary to each other

@Component vs @Service vs @Repository

@Component
Is the most generic stereotype and marks a bean as a Spring-managed component
Both @Service and @Repository annotations are the specializations over the @Component annotation

@Repository
Is a stereotype used for persistence layer. It translates any persistence related exceptions into a Spring’s DataAccessException

@Service
Is used for the beans at the service layer. Currently, it doesn’t offer any additional functionality over @Component

It’s always preferable to use @Repository and @Service annotations over @Component, wherever applicable. It communicates the bean’s intent more clearly

Spring Configuration Using JAVA

Properties

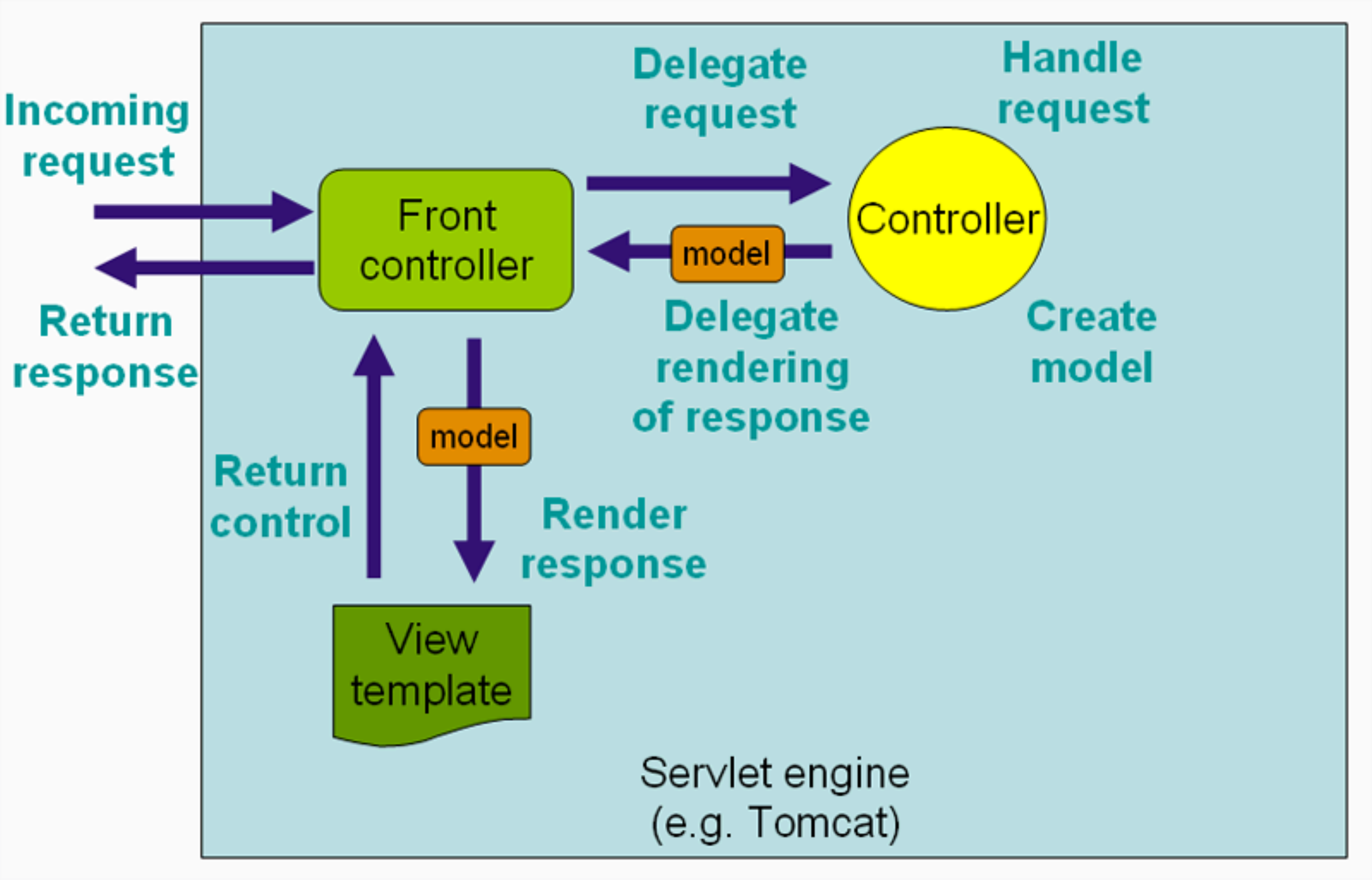
```
For JAVA Configuration (No setter required)
@Value("${dbUsername}")
private String dbUsername;


For XML Configuration
<property name="dbUsername" value=“${dbUsername}"></property>

<context:annotation-config />
<context:property-placeholder location="app.properties" />
```

Spring MVC

MVC Frameworks

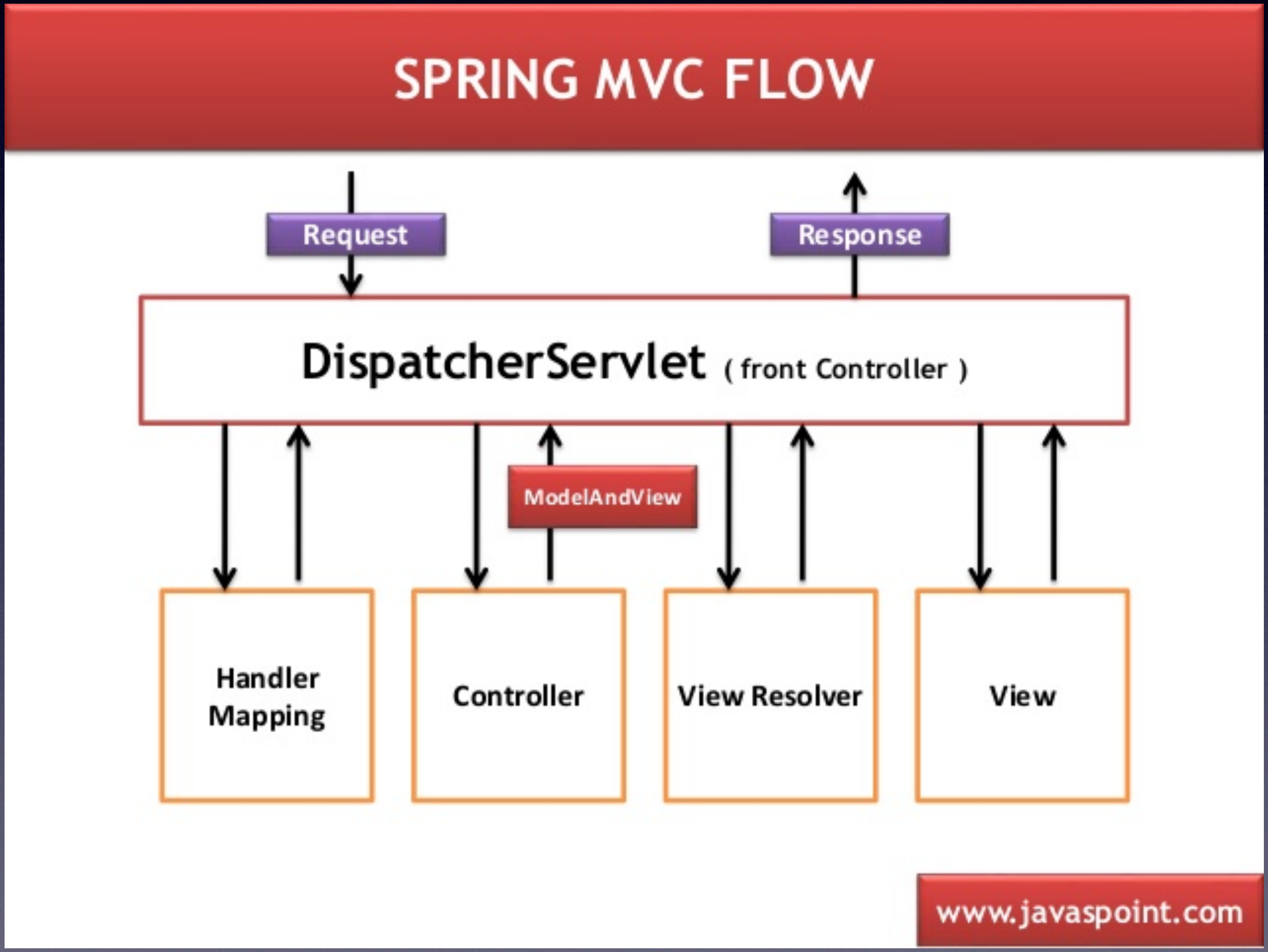
- Struts 2
- Tapestry
- OFBiz- eCommerce
- JSF
- Oracle ADF
- Spring MVC
- Spring Boot
- GWT
- Groovy & Grails
- Ruby on Rails

Spring Framework
Spring MVC

- *Front controller / Dispatcher Servlet*
- *Form Bean*
- *Controller*
- *Action method*
- *View Resolver*
- 

Spring Framework
Spring MVC

•



Spring Framework
Spring MVC

- ***http://localhost:8080/hello-springmvc/mymvc/hello***

- ***mymvc - Spring MVC filter URL pattern***
- ***web.xml***

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/mymvc/*</url-pattern>
    </servlet-mapping>
```

- ***hello - Action mapping / Request mapping which points to sayHello() action in a Controller***

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView sayHello() {
```

Spring Framework
Auto Component Scanning

- *stereotype annotations for spring-managed components:*

- *@Component*
- *The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context. To use this annotation, apply it over class*

- *@Controller*
- *@Controller annotation marks a class as a Spring Web MVC controller. It too is a @Component specialization, so beans marked with it are automatically imported into the DI container. When you add the @Controller annotation to a class, you can use another annotation i.e. @RequestMapping; to map URLs to instance methods of a class.*

- *@Service*
- *The @Service annotation is also a specialization of the component annotation. It doesn't currently provide any additional behavior over the @Component annotation, but it's a good idea to use @Service over @Component in service-layer classes because it specifies intent better.*

- *@Repository*
- *Although above use of @Component is good enough but you can use more suitable annotation that provides additional benefits specifically for DAOs i.e. @Repository annotation. The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.*
-

Spring Configuration Using JAVA

Q&A Links

1. How do you refer to .properties file which is outside the Spring project?
This is what I tried in our example and it worked! Its my MacOS and hence one extra '/' for /Users

```
@PropertySource("file:///Users/nanda/Documents/app.properties")
public class AppConfig {
…
}
```

2. The order of execution for the pointcuts, can we change?
https://stackoverflow.com/questions/15109101/how-to-control-the-execution-order-of-two-pointcuts-on-the-same-package

3. The Location of default validation messages in Hibernate-Validator
https://github.com/hibernate/hibernate-validator/blob/master/engine/src/main/resources/org/hibernate/validator/ValidationMessages.properties

Spring Expression Language (SpEL)

SpEL is a powerful library that supports querying and manipulating an object graph @ runtime

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

Spring Expression Language (SpEL)

```
@Value("#{19 + 1}")
private double add;

@Value("#{'Some string ' + 'plus other string'}")
private String addString;

@Value("#{20 - 1}")
private double subtract;

@Value("#{10 * 2}")
private double multiply;

@Value("#{36 / 2}")
private double divide;
@Value("#{36 div 2}")
private double divideAlphabetic;

@Value("#{37 % 10}")
private double modulo;
@Value("#{37 mod 10}")
private double moduloAlphabetic;

@Value("#{2 ^ 9}")
private double powerOf;

@Value("#{(2 + 2) * 2 + 9}")
private double brackets;
```