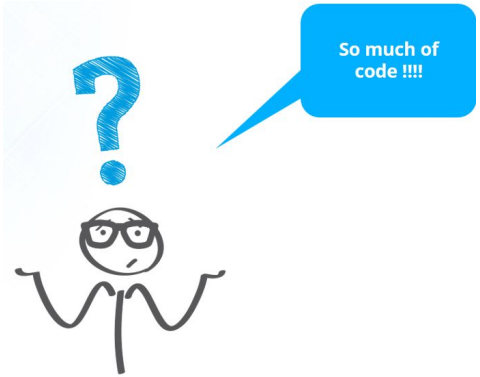
AN INTRODUCTION TO BACKEND FOR BEGINNERS

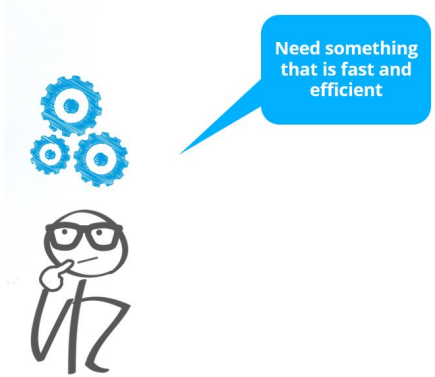
MODULO 4 – SPRING

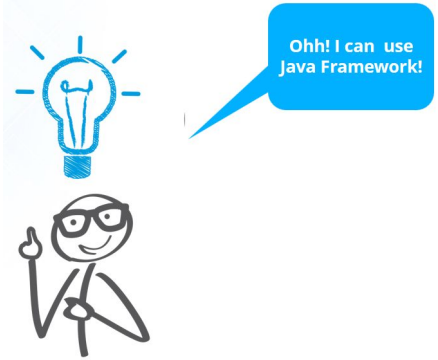
Objectives:

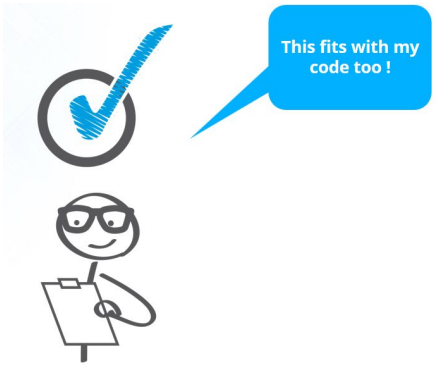
- Learn Spring basics
- Learn how Spring MVC works
- Modify a simple Spring MVC webapp

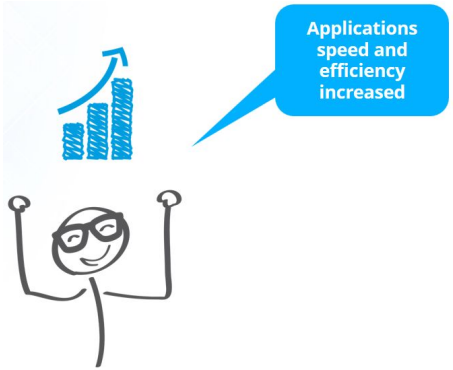
Why frameworks?





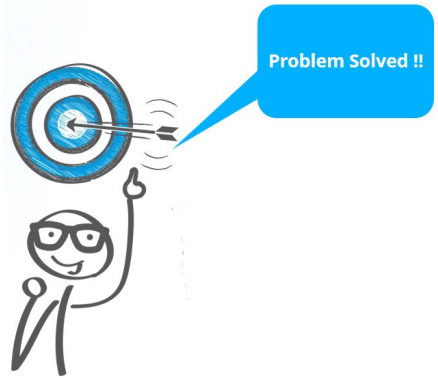






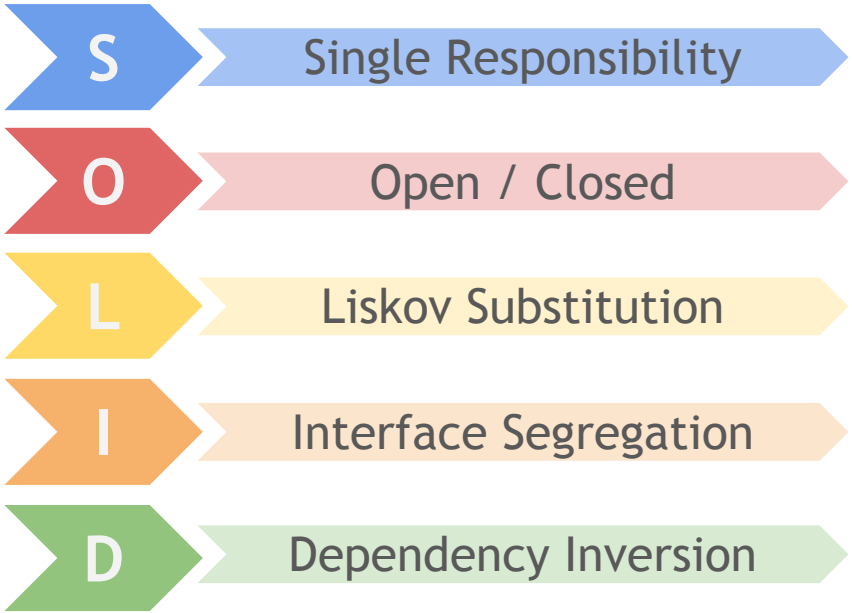
Frameworks are fast, efficient and light-weight.

They are large bodies of predefined codes which we can easily add to our own code to solve a specific problem.



SOLID principles

In computer programming, the term **SOLID** is a mnemonic acronym for five **design principles** intended to make **software designs** more **understandable**, **flexible** and **maintainable**. The principles are a subset of many principles promoted by **Robert C. Martin**



“a class should have only a single responsibility” (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).

“software entities … should be open for extension, but closed for modification.”

“objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”

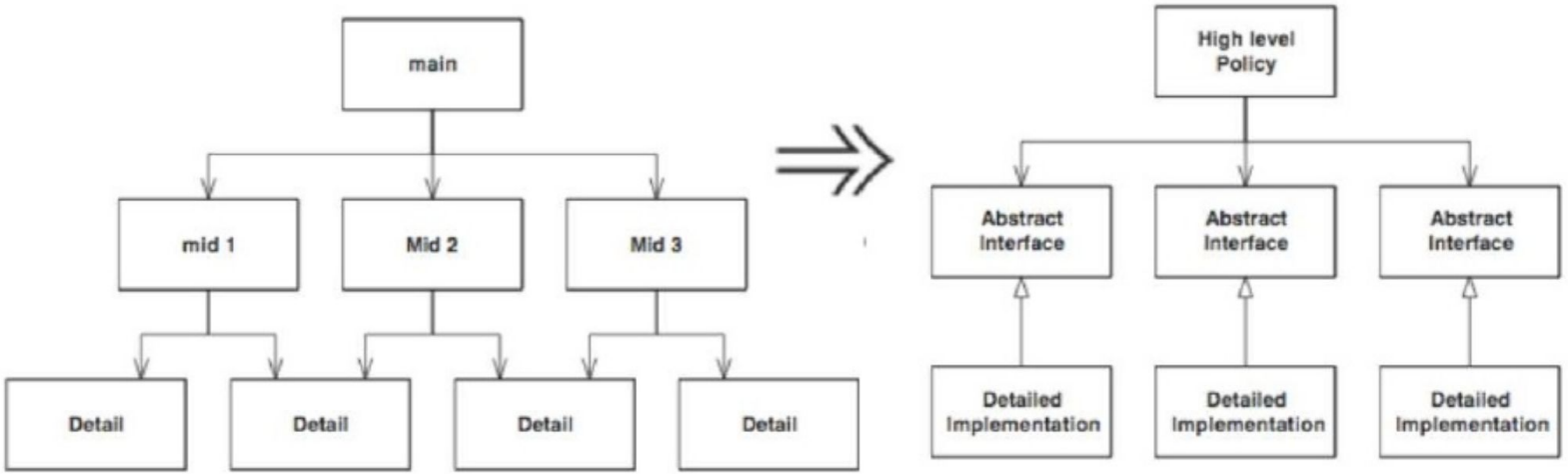
“many client-specific interfaces are better than one general-purpose interface.”

one should **“depend upon abstractions, [not] concretions.”**

Dependency Inversion Principle

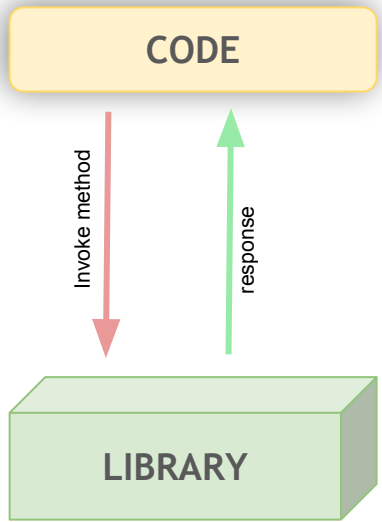
*Formulated by **Robert C. Martin** consists in reversing the conventional dependency relationships established from high-level modules to low-level one:*

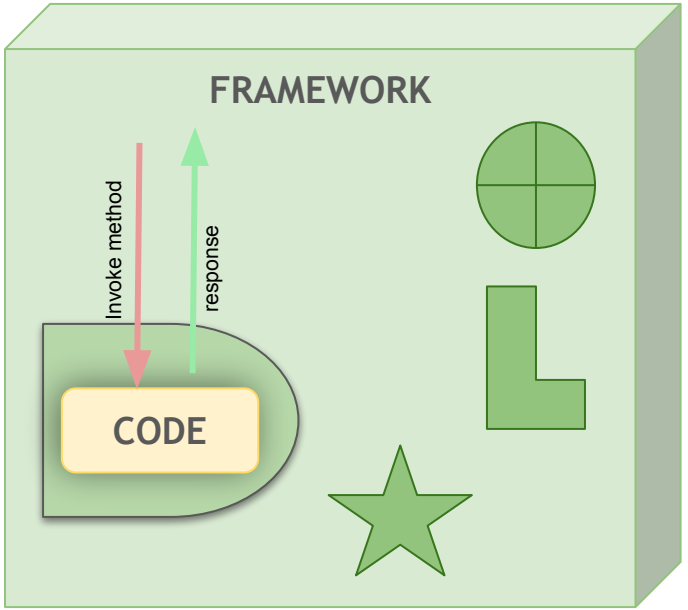
- ***High-level modules** should not depend on low-level modules. Both **should depend on abstractions**.*
- *Abstractions should not depend on details. **Details should depend on abstractions**.*



Inversion Of Control

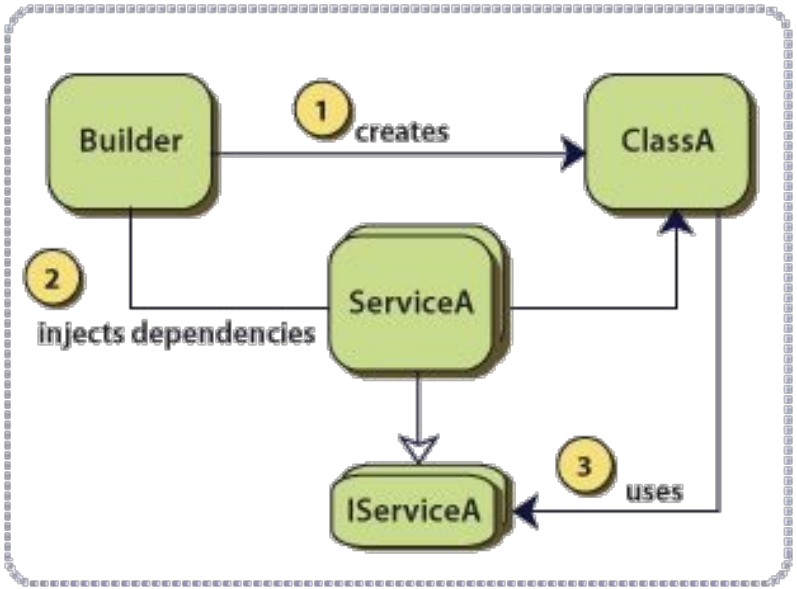
Objects rely on their environment to provide dependencies rather than actively obtaining them. Inversion of Control can make the difference between a library and a framework.





Dependency Injection

Dependency Injection is a design pattern which purpose is to have a separate object, an assembler, that populates a field defined by a contract in the class with an appropriate implementation for that contract.

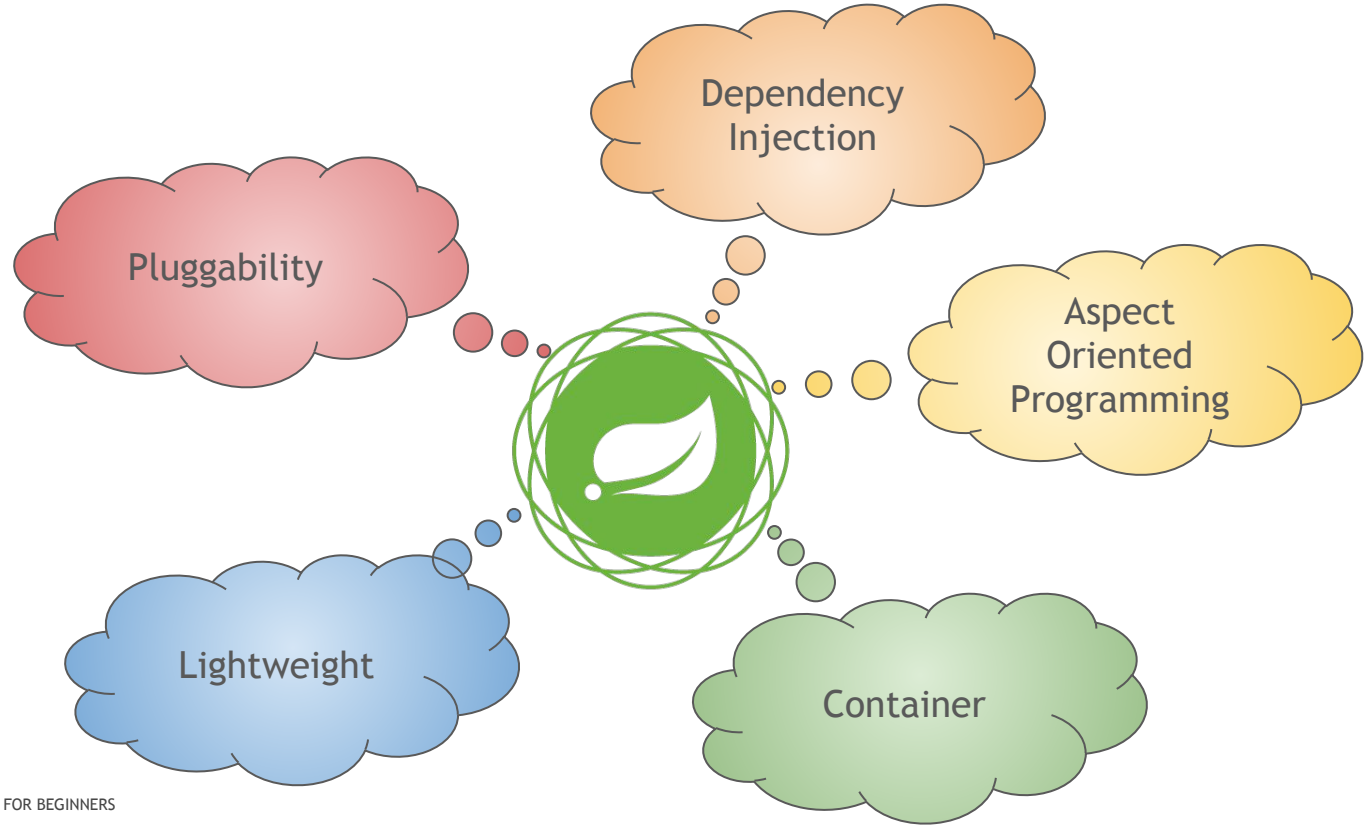


There are three main styles of dependency injection:

- By Constructor
- By Setter
- By Interface

What is Spring Framework

Spring Framework is a powerful lightweight application development framework used for Enterprise Java (JEE).

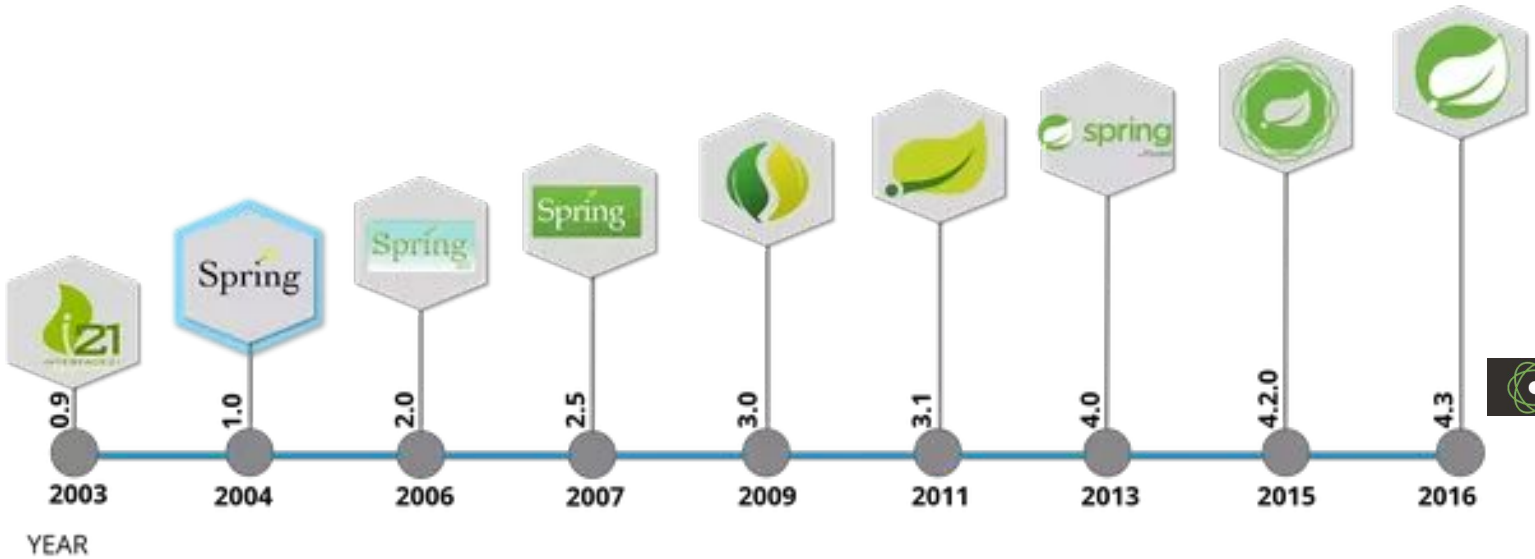


Story?

Created by **Rod Johnson in 2003**

*Thinked as a **simpler solution than J2EE** design for applications based on ordinary Java classes (POJO) and dependency injection*

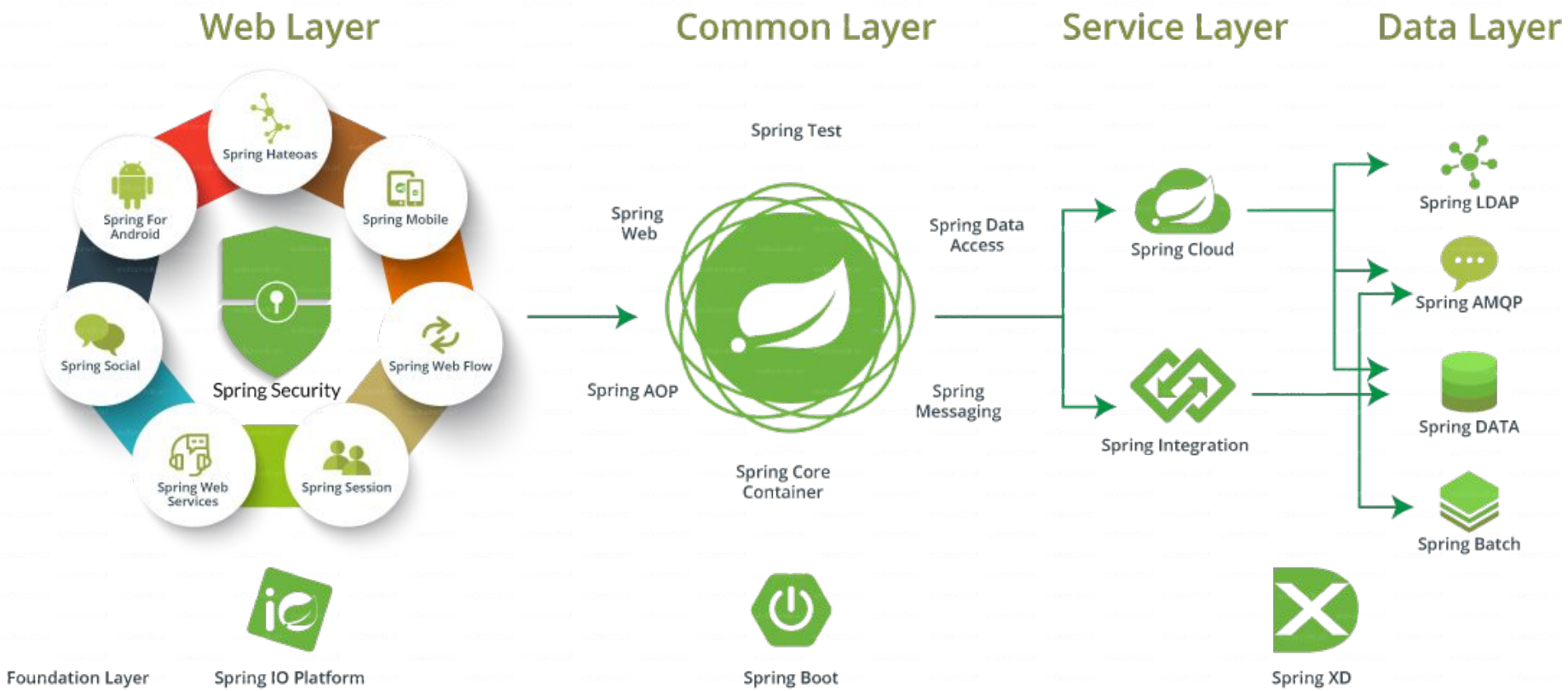
The name “**Spring**” was given as it meant a fresh start after “**Winter**” of **traditional J2EE**



PROJECT **REACTOR**

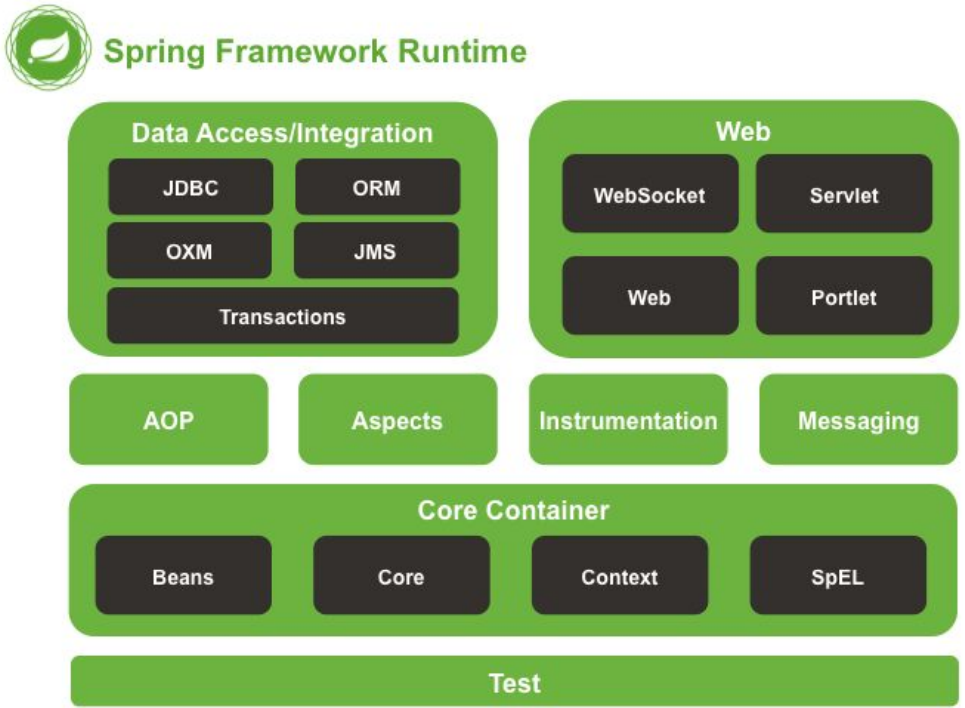
2018

Spring ecosystem



Spring architecture

The Spring Framework provides about 20 modules which can be used based on an application requirement.



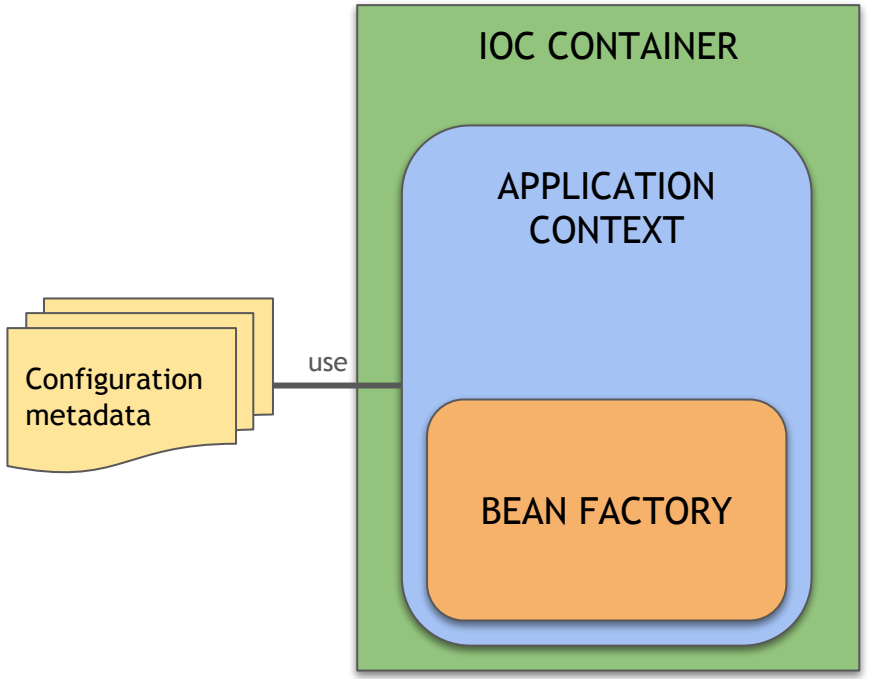
IOC Container

In the Spring framework, the IoC container is represented by the interface ApplicationContext. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle

BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.

ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory:

- Internationalization
- Annotation based dependency
- Lazy and Eager initialization



What is a Bean?

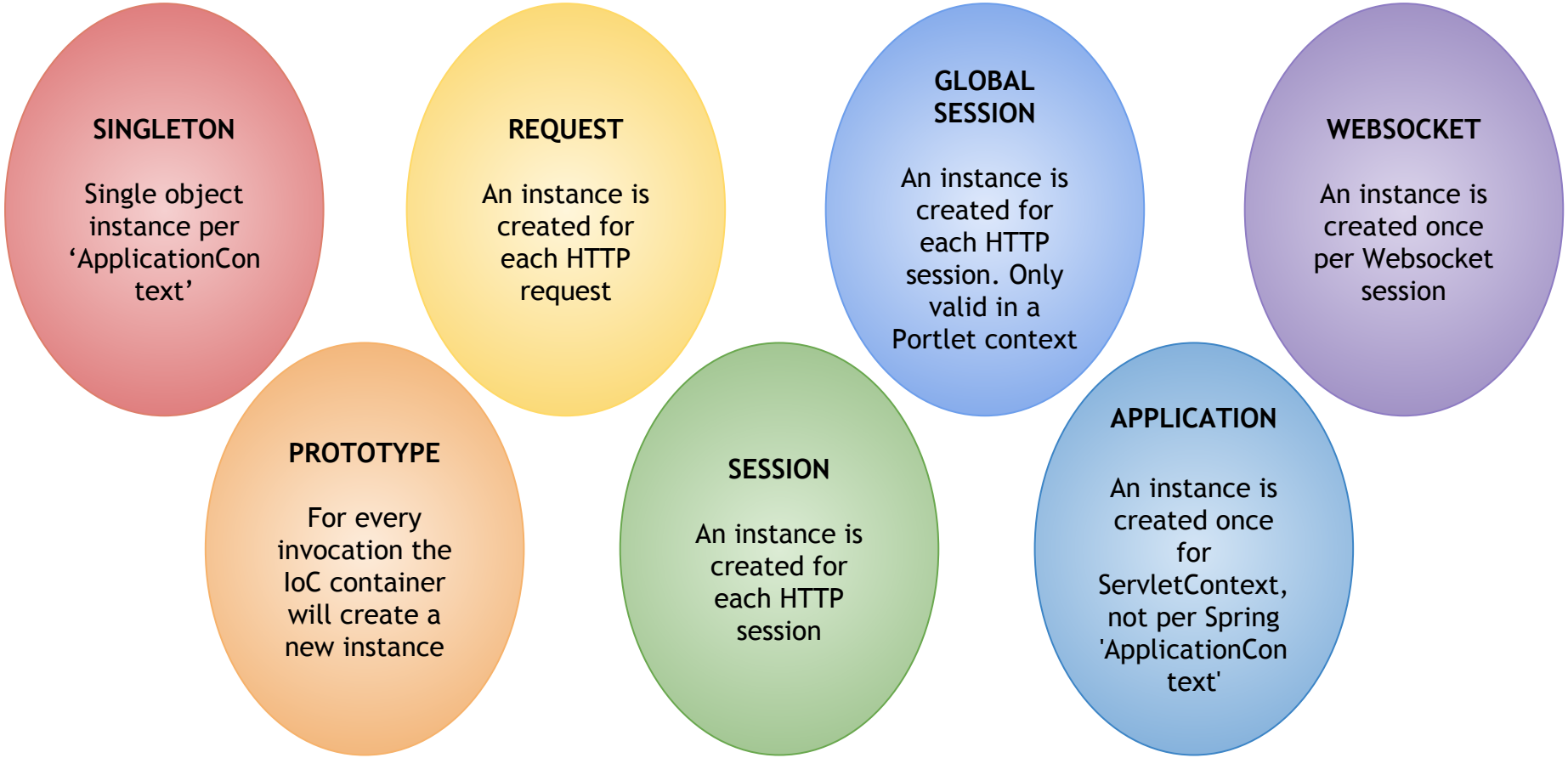
A bean is an POJO (Plain Old Java Object) that is instantiated, assembled, and otherwise managed by a Spring DI Framework.

Key attributes:

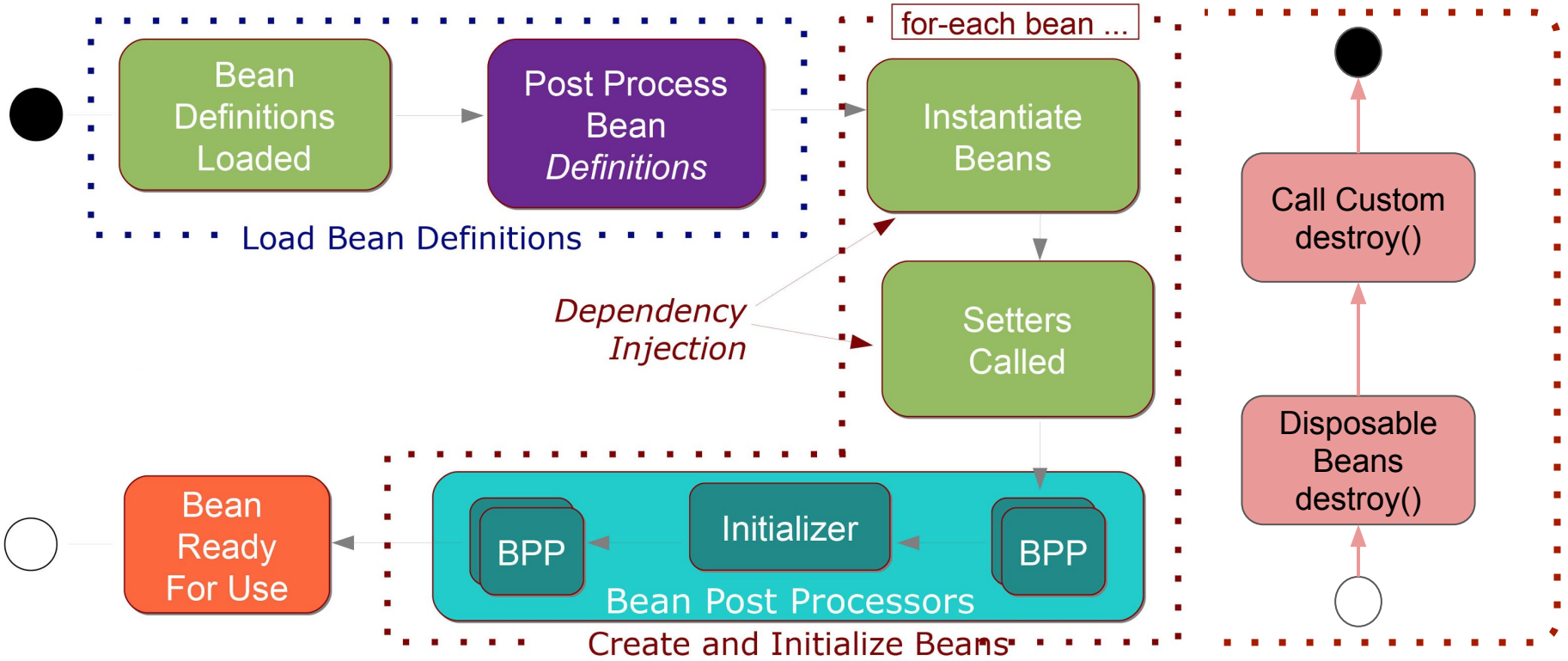
- **class** (required): fully qualified java class name
- **id**: the unique identifier for this bean
- **config-attributes**: (scope, init-method, etc.)
- **constructor-arg**: arguments to pass to the constructor at creation time
- **property**: arguments to pass to the bean setters at creation time, can be values or dependencies, specified in property or constructor-arg

```
<bean id="myBeanId" class="example.BeanClass"scope="singleton"
init-method="eager" abstract="false" parent="parentBean">
    <constructor-arg index="0" value="${property.val}" />
    <constructor-arg index="1" ref="otherBean" />
    <property name="dependency2" ref="myOtherBeanId" />
    <property name="prop2" value="value" />
</bean>

public class BeanClass {
   private OtherBean dependency1;
   private OtherBean dependency2;
   private String prop1;
   private String prop2;

   public BeanClass(OtherBean bean, String val) {
       this.dependency1 = bean;
       this.prop1=val;
   }

   public void setProp2(String val) {
       this.prop2=val;
   }

   public void setDependency2(OtherBean bean) {
       this.dependency2=bean;
   }
 }
```

Bean Scopes


SINGLETON
Single object instance per 'ApplicationContext'
REQUEST
An instance is created for each HTTP request
GLOBAL SESSION
An instance is created for each HTTP session. Only valid in a Portlet context
WEBSOCKET
An instance is created once per Websocket session
PROTOTYPE
For every invocation the IoC container will create a new instance
SESSION
An instance is created for each HTTP session
APPLICATION
An instance is created once for ServletContext, not per Spring 'ApplicationContext'

Spring Bean Lifecycle



Lifecycle examples

```xml
<bean id="exampleInitBean" class="examples.ExampleBean"
init-method="init"/>
```

```
public class ExampleBean {

   public void init() {
       // do some initialization work
   }
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" />
public class ExampleBean implements InitializingBean {

   @Override
   public void afterPropertiesSet() {
       // do some initialization work
   }
}
```

```
public class ExampleBean implements InitializingBean {

   @PostConstruct
   public void postConstruct() {
       // do some initialization work
   }
}
```

```
@Bean(initMethod = "init")
public ExampleBean exampleBean() {
   return new ExampleBean();
}
```

INIT

```xml
<bean id="exampleInitBean" class="examples.ExampleBean"
destroy-method="cleanup"/>
```

```
public class ExampleBean {

   public void cleanup() {
       // do some destruction work
   }
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" />
public class ExampleBean implements DisposableBean {

   public void destroy() {
       // do some destruction work
   }
}
```

```
public class ExampleBean implements DisposableBean {

   @PreDestroy
   public void preDestroy() {
       // do some destruction work
   }
}
```

```
@Bean(destroyMethod = "cleanup")
public ExampleBean exampleBean() {
   return new ExampleBean();
}
```

DESTROY

Spring configuration

XML-Based

Dependencies and beans are specified in XML configuration files.

```
<beans>
<bean id="myBeanId"
      class="example.BeanClass"
      scope="singleton">
   <property name="dependency"
             ref="myOtherBeanId" />
</bean>

<bean id="myOtherBeanId"
      class="example.OtherBeanClass"
      scope="singleton">
   <property name="prop"
   value="myValue" />
</bean>
</beans>
```

Annotation-Based

Dependencies and beans are configured into the component class itself by using stereotype annotations on the relevant class, method, or field declaration. Must be enabled.

```
<beans>
<context:annotation-config/>
<!-- bean definitions go here -->
</beans>

@Controller
public class MyBean {
   @Autowired
   private MyOtherBean dependency;
}

@Service
public class MyOtherBean {
   private String prop = "myvalue";
}
```

Java-based configuration

Java-configuration support are **@Configuration** annotated classes and **@Bean** annotated methods.

```
@Configuration
public class MyConfig {
 @Bean
 @Scope(BeanDefinition.SCOPE_SINGLETON)
 public MyBean myBean() {
   //constructor injection
   return new MyBean(myOtherBean());
 }

 @Bean
 @Scope(BeanDefinition.SCOPE_SINGLETON)
 public MyOtherBean myOtherBean() {
   MyOtherBean bean = new MyOtherBean();
   bean.setProp("myValue");
   return bean;
 }
}
```

Dependency Injection in Spring

Dependency Injection in Spring can be done **manually** through **constructors** or **setters**, it is also possible to leave framework do it **automatically** by **autowiring**.

Constructor-based: should be used for mandatory dependencies. In constructor, we should assign constructor args to final member fields.

Setter-based: should be used for optional dependencies.

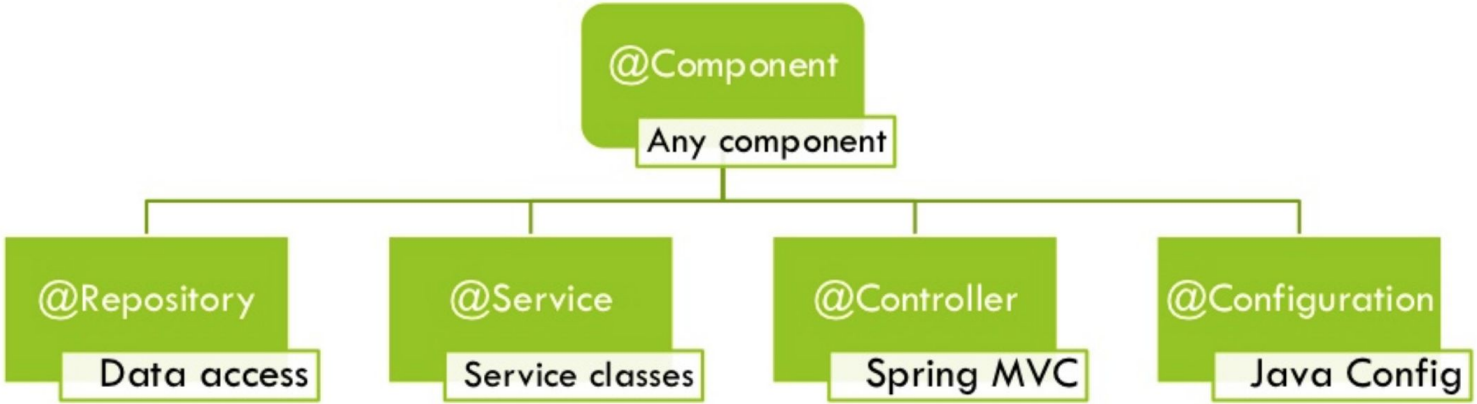
Autowire-based: allows the Spring container to automatically resolve dependencies:

- By **Type**
- By **Name**

```
@Component("fooFormatter")
public class FooFormatter implements Formatter {}

@Component
public class FooService {

    @Autowired
    private Formatter formatter; //by type

    @Autowired
    private Formatter fooFormatter; //by name

    @Autowired
    @Qualifier("fooFormatter")  //by name
    private Formatter formatter;
}

public class FooService {
    private Formatter formatter;
    @Autowired  //setter
    public void setFormatter(Formatter formatter) {
            this.formatter = formatter;
    }
}

public class FooService {
    private Formatter formatter;
    @Autowired  //constructor
    public FooService(Formatter formatter) {
        this.formatter = formatter;
    }
```

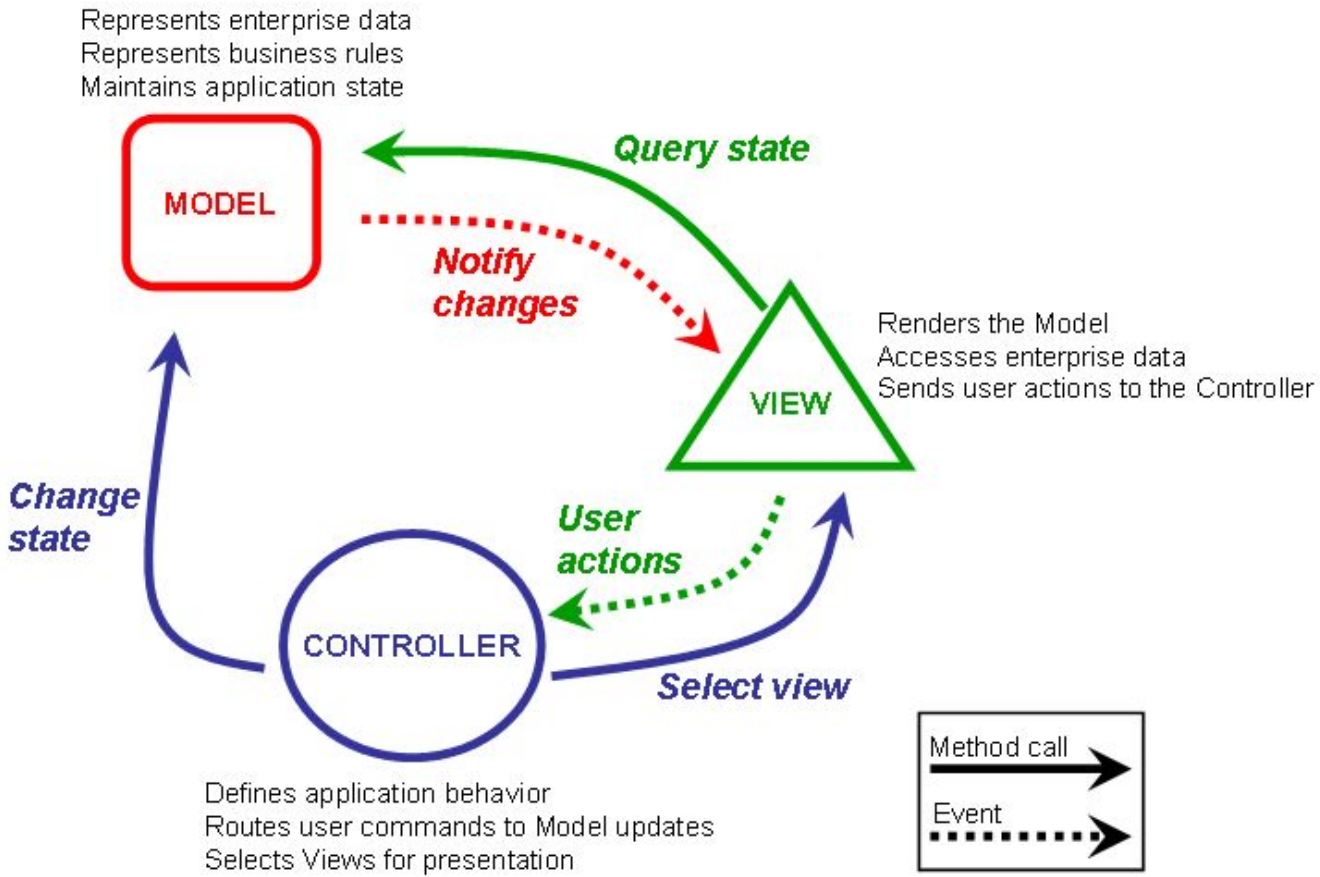
Spring Stereotypes

*Spring framework provides following **stereotypes** that **frees** you **from defining beans explicitly with XML** configuration with support of **autowiring**.*




```
<!-- looks for annotations on beans -->
<context:annotation-config />

<!-- define component scan for stereotype annotations -->
<context:component-scan base-package="org.example.package,com.example.api" />
```

MVC


Represents enterprise data
Represents business rules
Maintains application state
MODEL
Query state
Notify changes
VIEW
Renders the Model
Accesses enterprise data
Sends user actions to the Controller
Change state
User actions
CONTROLLER
Select view
Defines application behavior
Routes user commands to Model updates
Selects Views for presentation
Method call
Event

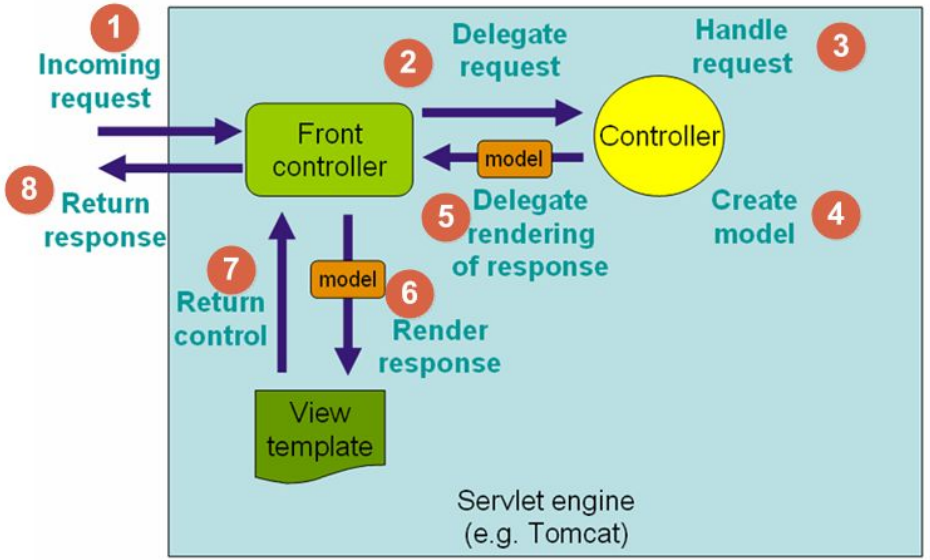
Spring MVC

Spring Web MVC is the **original web framework built on the Servlet API** and **included in the Spring Framework** from the very beginning.

The formal name "**Spring Web MVC**" comes from the name of its source **module spring-webmvc** but it is more commonly known as "Spring MVC".

Spring MVC provide support for:

- Servlets
- Portlet
- Websocket



MVC Request lifecycle

The client requests for a resource from the server and the request is intercepted by the **DispatcherServlet**.

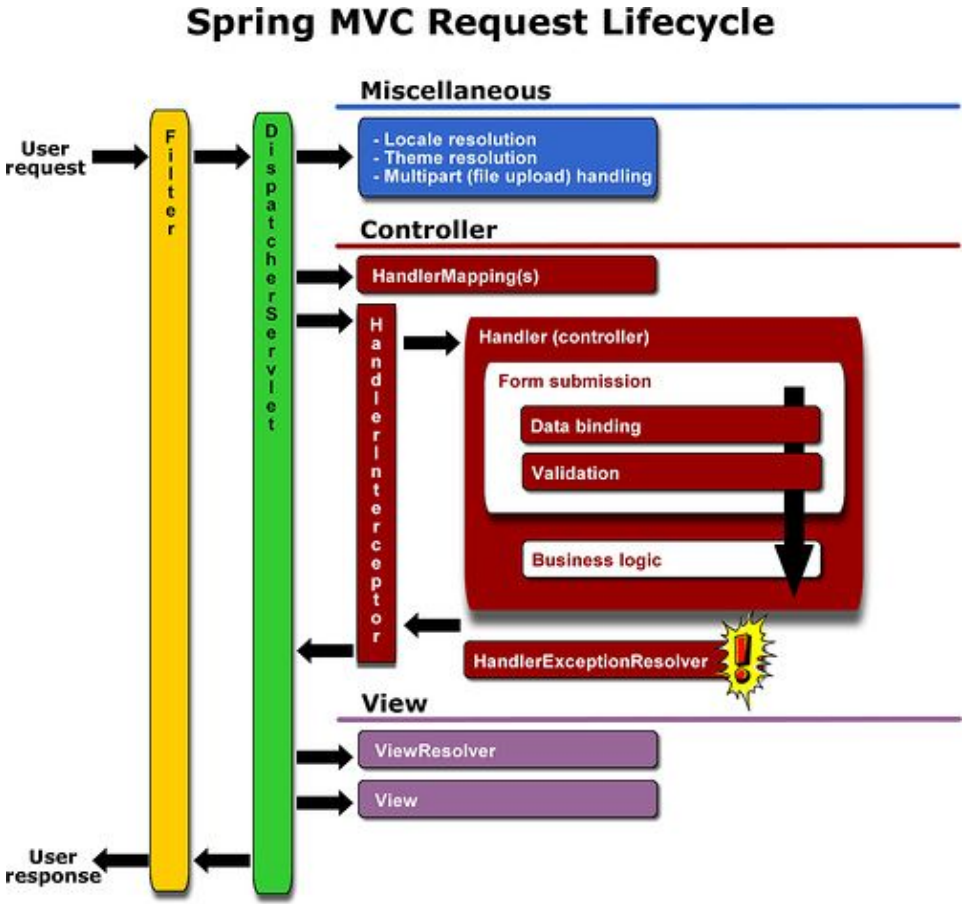
The DispatcherServlet finds the appropriate **HandlerMapping**.

The HandlerMapping is responsible for mapping the client request and the appropriate Controller dispatching the request to the **Controller**.

The Controller executes the necessary business logic as necessary and then returns **ModelAndView** object back to the DispatcherServlet.

Looking at the values in the ModelAndView object and with the help from **ViewResolver** object, Spring MVC derives the actual View to be rendered.

The **View** object is generated and then sent to the DispatcherServlet, which will send it to the **Servlet Container** to generate the final out which will be sent back to the client.



Dispatcher Servlet and WebContext

The **DispatcherServlet** is **provided** with the **Spring MVC** framework and accomplish the role of **FrontController**:

- You **must configure** the Web application to direct all requests into this servlet.
- As is the typical case in Java Web development, this is accomplished via the servlet/servlet mapping elements in the standard **web.xml** file.

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>WEB-INF/dispatcher-servlet.xml</param-value>
</context-param>

<servlet>
   <servlet-name>DispatcherServlet</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <init-param>
       <param-name>contextConfigLocation</param-name>
       <param-value></param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
```

web.xml

```
<context:annotation-config />
<mvc:annotation-driven />
<context:component-scan base-package="example" />
```

dispatcher-servlet.xml

Controller

Controllers provide access to the application business logic, take care about user inputs transforming them into model that are represented by the view and are responsible of flow navigation. Spring implements a Controller in a very abstract way, which enable developer to create a wide variety of controllers.

```java
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloWorldController {

    @RequestMapping("/")
    public String hello() {
        return "hello";
    }

    @RequestMapping(value = "/hi", method = RequestMethod.GET)
    public String hi(@RequestParam("name") String name, Model model) {
        String message = "Hi " + name + "!";
        model.addAttribute("message", message);
        return "hi";
    }

}
```

Model and View

ModelAndView is returned by the Controller object back to the Dispatcher Servlet.
This class is just a Container class for holding the Model and the View information.
This way of specifying a View is called a Logical View.

```
@GetMapping("/test")
public ModelAndView passParametersWithModelAndView() {
    ModelAndView modelAndView = new ModelAndView("viewPage");
    modelAndView.addObject("spring", "mvc");
    modelAndView.addObject("message", "Test");
    return modelAndView;
}
```

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags" %>
<!DOCTYPE HTML>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <div>Web Application. Passed parameter : <s:message code="subscription.email" /></div>
</body>
</html>
```

```
@GetMapping("/test")
public String passParametersWithModelMap(ModelMap map) {
    map.addAttribute("spring", "mvc");
    map.addAttribute("message", "Test");
    return "viewPage";
}
```

```
@GetMapping("/test")
public String passParametersWithModel(Model model) {
    Map<String, String> map = new HashMap<>();
    map.put("spring", "mvc");
    model.addAttribute("message", "Test");
    model.mergeAttributes(map);
    return "viewPage";
}
```

View Resolver

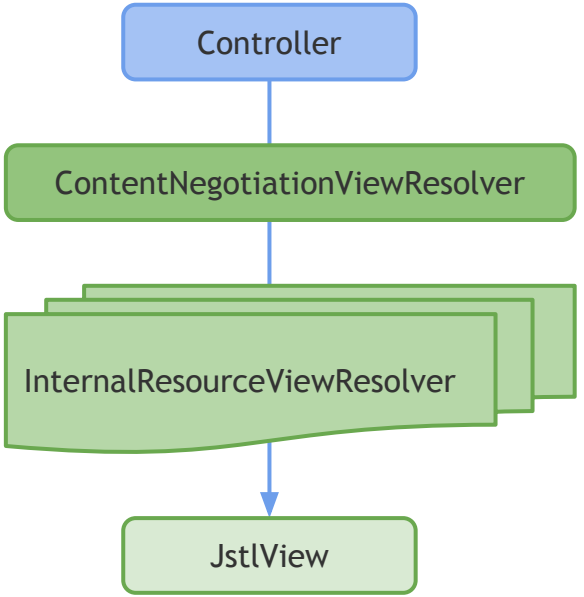
*All MVC frameworks provide a way of working with views. Spring does that via the **view resolvers**, which enable you to **render models** in the browser without tying the implementation to a specific view technology.*

*The ViewResolver **maps view names to actual views**.*

Spring framework comes with quite a few view resolvers like ***InternalResourceViewResolver***, *XmlViewResolver*, *ResourceBundleViewResolver*

```java
@Bean
public ViewResolver internalResourceViewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setViewClass(JstlView.class);
    bean.setPrefix("/WEB-INF/view/");
    bean.setSuffix(".jsp");
    return bean;
}
```

```xml
<bean id="jstlViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
</bean>
```

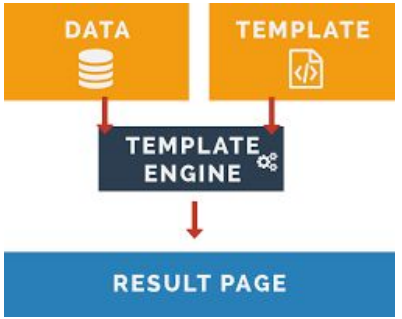


View - Templating

Spring MVC support a variety of templating technologies to develop views.

To do that in Spring MVC there is an implementation of TemplateEngine that take care about the translation.

A single project could use more than one template engine.



FreeMarker
Velocity
Groovy

```
<html>
  <head>
    <meta charset="UTF-8" />
      <title>User Registration</title>
  </head>
  <body>
    <form action="#" th:action="@{/register}"
          th:object="${user}" method="post">
    Email:<input type="text" th:field="*{email}" />
    Password:<input type="password" th:field="*{password}" />
    <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

Thymeleaf

```
<%@ taglib prefix="form"
  uri="http://www.springframework.org/tags/form"%>
<html>
   <head>
       <meta http-equiv="Content-Type"
         content="text/html; charset=UTF-8">
       <title>User Registration</title>
   </head>
   <body>
       <form:form method="POST" modelAttribute="user">
           <form:label path="email">Email: </form:label>
           <form:input path="email" type="text"/>
           <form:label path="password">Password: </form:label>
           <form:input path="password" type="password" />
           <input type="submit" value="Submit" />
       </form:form>
   </body>
</html>
```

JSP
Java Server Pages

Validation

Annotation	Type	Description
`@Min(10)`	Number	must be higher or equal
`@Max(10)`	Number	must be lower or equal
`@AssertTrue`	Boolean	must be true, null is valid
`@AssertFalse`	Boolean	must be false, null is valid
`@NotNull`	any	must not be null
`@NotEmpty`	String / Collection's	must be not null or empty
`@NotBlank`	String	`@NotEmpty` and whitespaces ignored
`@Size(min,max)`	String / Collection's	must be between boundaries
`@Past`	Date / Calendar	must be in the past
`@Future`	Date / Calendar	must be in the future
`@Pattern`	String	must math the regular expression

Spring have simple ways of triggering the validation process by just using annotations. Or it is possible to do it programmatically.

SIMPLE VALIDATION

```
public class UserDetails {
   @NotEmpty
   private String user;
   @NotEmpty
   private String email;
}

@RequestMapping("/register")
public String registerCheck(@Valid UserData userDetails,
                            BindingResult result,
                            ModelMap model) {
   if (result.hasErrors()) {
       return "registerPage";
   } else {
       model.addAttribute("registered", true);
       return "success";
   }
}
```

CUSTOM VALIDATOR

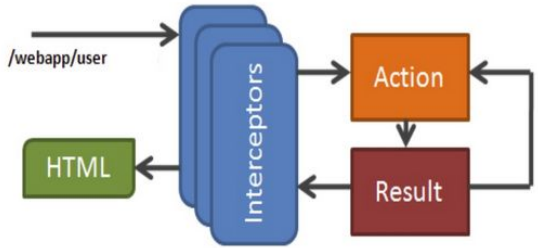
```
//create custom annotation
@Constraint(validatedBy=CapitalLetterValidator.class))
public @interface CapitalLetter {
 String message() default "{carbase.error.capital}";
}

//implement constraint validator
public class CapitalLetterValidator
   implements ConstraintValidator<CapitalLetter, String> {
}

//define a default error message inside properties
carbase.error.capital=The name must begin with a capital
letter
```

Interceptors

Spring provide a mechanism to intercept all HTTP requests. **HandlerInterceptor** perform actions **before handling**, **after handling** or **after completion** (when the view is rendered) **of a request**. The interceptors can be used for **cross-cutting concerns** and to avoid repetitive handler code.

Interceptors working with the ***HandlerMapping*** on the framework must implement:

- Implement ***HandlerInterceptor*** interface
- extends ***HandlerInterceptorAdapter***.



This interface contains three main methods:

- ***prehandle()*** - called before the actual handler is executed, but the view is not ge
- ***postHandle()*** - called after the handler is executed
- ***afterCompletion()*** - called after the complete request has finished and view was generated

```
public class CustomInterceptor
  extends HandlerInterceptorAdapter
  @Override
  public boolean preHandle(HttpServletRequest request,
       HttpServletResponse response, Object handler)
       throws Exception {
   //code here
   return true;
   }
}
```

```
// inside config that extends WebMvcConfigurerAdapter
@Override
public void addInterceptors(InterceptorRegistry registry) {
   registry.addInterceptor(new CustomInterceptor());
}
// OR
<mvc:interceptors>
   <bean id="loggerInterceptor" class="example.CustomInterceptor"/>
</mvc:interceptors>
```

Useful links

Inversion of Control Solid Principle

Martin Fowler - IOC

Spring Controllers

Spring Validation Basics Spring Validation Custom

Spring Interceptors

Spring ViewResolvers

Spring Framework Reference Guide

Github project course repository and Lessons documentation:
https://github.com/mcolombosperoni/an-introduction-to-backend-for-beginners