Spring / Hibernate

why Spring:

---

* Building JAVA enterprise applications

* simple and lightweight alternative to J2EE

what is J2EE?

---

client side --> server side --> service business logic --> database

(contains JSP, servlets, EJB, JMS, web services, JSF, JAXB, JAX-WS, sockets)

Release Timeline:

J2EE version 1.2 to 8

(1999)

Spring version 1.0 to 4.3

(2004-)

Goals of Spring core framework:

---

* Lightweight development with JAVA POJO (Plain old java objects)

* Dependency injection to promote loose coupling

* Declarative programming using AOP (Aspect oriented programming)

    (adding application wide services to your objects)

* Minimize java code.

Spring core container:

    * Beans

    * Core

-- factory for creating beans (BeanFactory) and manage bean dependencies

    * SpEL   -- Spring Expression Language (used in config files to refer other beans)

    * Context   -- holds the beans in the memory

Spring Infrastructure:

* AOP -- Logging, security, transactions etc. through annotations or configs

* Aspects

* Instrumentation -- java agents to remotely monitor your app with JMX (java mgmt extn) we use agents provided by the spring team and web server.

* Messaging

Spring Data Access Layer: (communicating with the database (RDBMS or NoSQL)

- JDBC -- contains JDBC helper class, reduces 50% of JDBC code
- ORM -- Object relational mapping (integration with Hibernate and JPA)
- Transactions -- add transaction support (make use of heavy AOP behind the scenes)
- OXM
- JMS -- Java messaging services (for sending async messages to a message broker (Queue)) Spring provides helper classes for JMS

Spring Web Layer: (spring MVC framework -- core / controller / view)

- Servlet
- WebSocket
- Web -- external client calls
- Portlet

Spring Test Layer: (supports TDD - test driven development)

- Unit
- Integration
- Mock -- MOCK objects for mocking out servlets and JNDI access

Spring Projects:

Spring modules built on top of the core module

- Spring Security
- Spring batch
- Spring boot
- Spring Cloud
- Spring webflow

First Spring Application:

- **Create a project**
- **Download Spring – common logging jars**

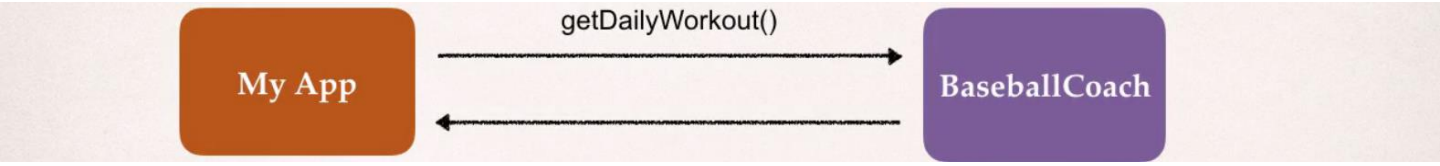
  http://repo.spring.io/release/org/springframework/spring/

  https://commons.apache.org/proper/commons-logging/download_logging.cgi

- **Build path → Add JARS → It gets downloaded into referenced libraries**

Spring Inversion of Control (IoC)

- **Construction and management of objects.**



- **App should be configurable, easily change the coach for another sport**
- **Keys things to build this myApp:**
  - **MyApp.java -- main method**
  - **BaseballCoach.java**
  - **Coach.java -- interface (best S/W engg practices)**
  - **TrackCoach.java**

MyApp.java

```
Coach theCoach = new BaseballCoach();
System.out.println(theCoach.getDailyWorkout());

Coach theCoach2 = new TrackCoach();
System.out.println(theCoach2.getDailyWorkout());
```

BaseballCoach.java:

```
public class BaseballCoach implements Coach{
	public String getDailyWorkout()
	{
		return "BaseballCoach: practice batting for 30 minutes....";
	}
}
```

Coach.java:

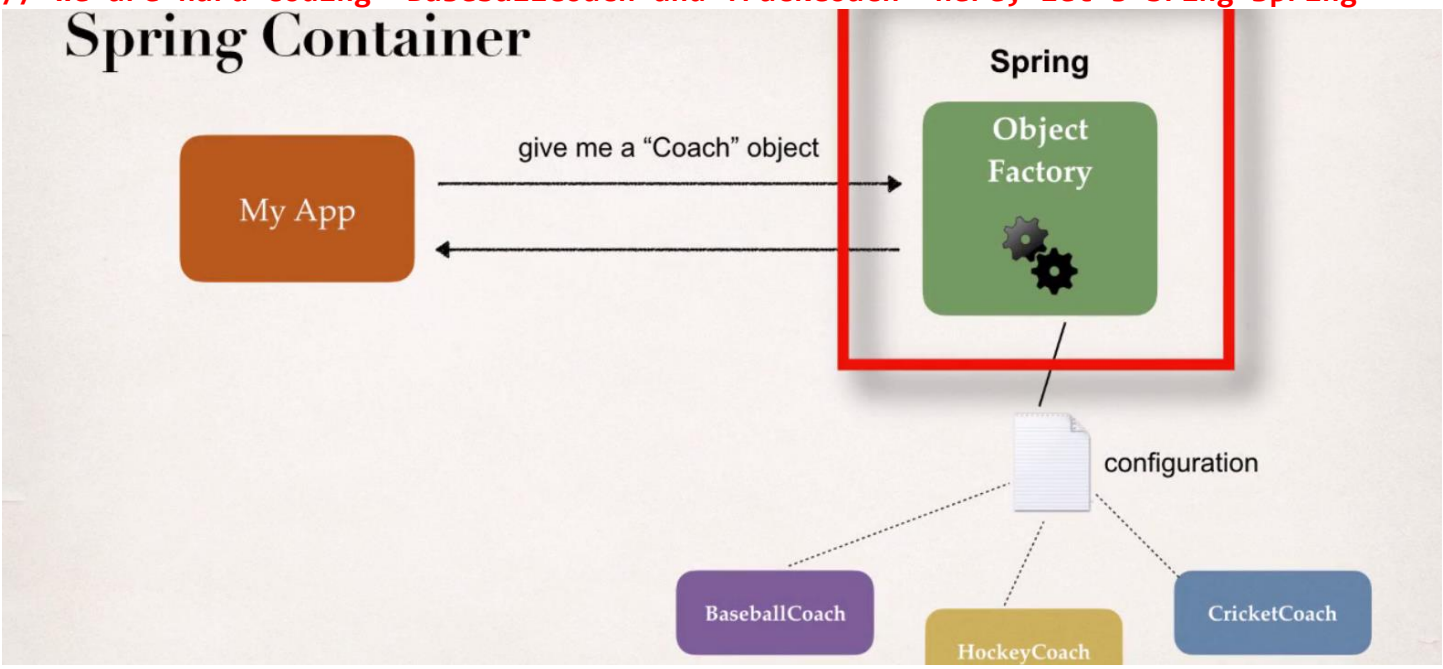
```
public interface Coach {
	public String getDailyWorkout();
}
```

TrackCoach.java:

```
public class TrackCoach implements Coach {

	@Override
	public String getDailyWorkout() {
		return "Track Coach: Run 5K miles...";
	}
}
```

```
<terminated> MyApp [Java Application] C:\Program Files\Java\jre\bin\javaw.exe (Feb 24, 2017, 9:43:53 PM)
BaseballCoach: practice batting for 30 minutes....
Track Coach: Run 5K miles...
```

// we are hard coding "BaseballCoach and TrackCoach" here, let's bring Spring



Spring Container:

- **IOC – creating and managing objects**
- **Dependency Injection – inject object dependencies**

Configuring Spring Container:

- **XML (legacy apps)**
- **JAVA annotations**
- **JAVA source code**

Spring Development process:

- **Configuring your spring beans**
- **Create a spring container**
- **Retrieve beans from spring container**

Configuring your spring bean:

applicationContext.xml (place in src folder)

```
<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">
       <bean id="baseballCoach" class="com.test.spring.BaseballCoach">
       </bean>
</beans>
```

Create a spring container:

- **Spring container = ApplicationContext**
  - **ClassPathXmlApplicationContext**
  - **AnnotationconfigApplicationContext**
  - **GenericWebApplicationContext….**

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

Retrieve beans from spring container:

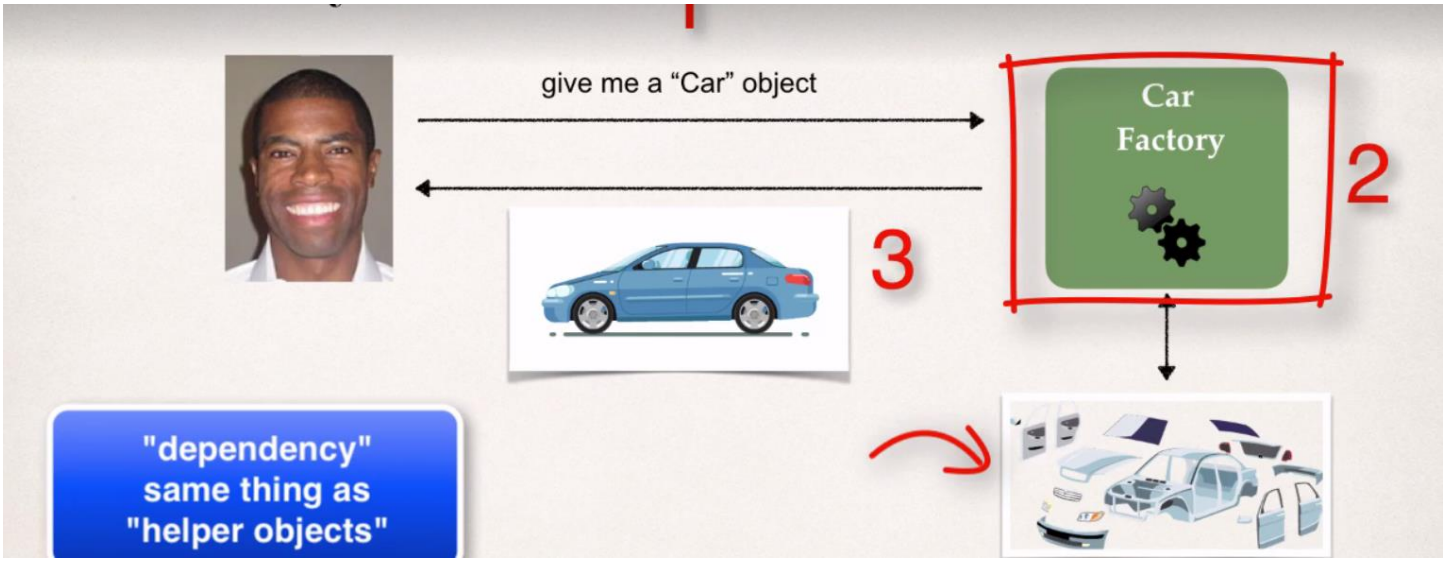
```
Scanner scan = new Scanner(System.in);

System.out.println("Enter game name (baseballCoach or trackCoach): ");
String name = scan.next();

Coach theCoach = context.getBean("baseballCoach", Coach.class);
// checks in configuration "baseballCoach" and interface of "baseball"
class. (When we pass the interface to the method, behind the scenes Spring will
cast the object for you.)

System.out.println(theCoach.getDailyWorkout());
context.close();
```

```
Enter game name (baseballCoach or trackCoach):
baseballCoach
Feb 24, 2017 11:45:32 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup date [Fr
BaseballCoach: practice batting for 30 minutes....
```

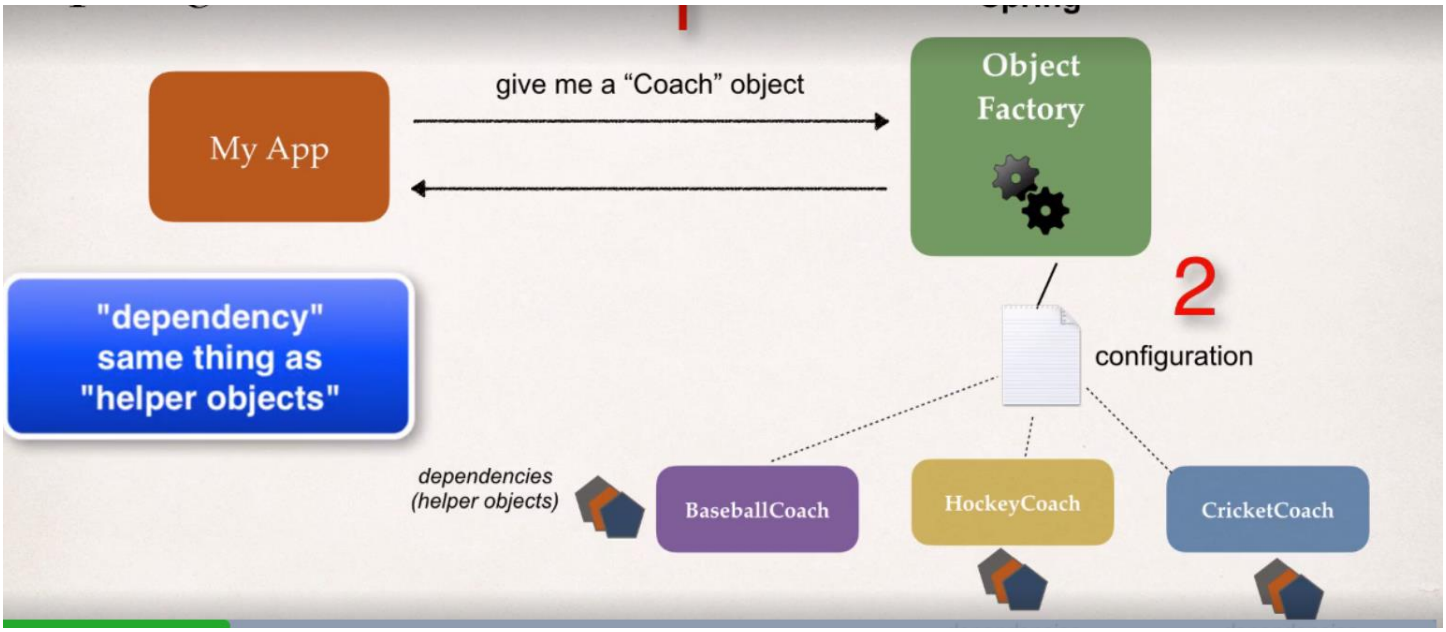
Spring Dependency injection:

- Car factory → build a car → tires, doors etc. are been injected

This is called dependency injection.



- These coach object might have some dependencies.



Coding: Extra feature added to coach → provide Fortune Service. (dependency now)

Injection Types:

- Constructor injection
- Setter injection

Constructor injection (Development process):

- Define the dependency interface and class
- Create a constructor in your class for injections
- Configure the dependency injection in spring config file

Define the dependency interface and class:

```
public interface Fortune {
	public String getFortune();
}


public class FortuneService implements Fortune{

	@Override
	public String getFortune() {
		return "Today you will enjoy in work....";	// read from
file system, DB, web service
	}
}
```

Create a constructor in your class for injections:

- **Injecting a dependency by calling in constructor**

```
private Fortune fortune;
	public BaseballCoach(Fortune fortune)
	{
		this.fortune=fortune;
	}
	public String getDailyFortune()			//dependency
	{
		return fortune.getFortune();
	}
```

Configure the dependency injection in spring config file:

```
public interface Coach {
	public String getDailyWorkout();
	public String getDailyFortune();
}


<bean id="baseballCoach" class="com.test.spring.constructorinjection.BaseballCoach">
		<constructor-arg ref="fortune" />		<!-- inject the dependency
"fortune" -->
</bean>
<bean id="fortune" class="com.test.spring.constructorinjection.FortuneService" />


	Coach theCoach = context.getBean("baseballCoach", Coach.class);
	System.out.println(theCoach.getDailyFortune());
	context.close();
```

```
<terminated> MyApp (2) [Java Application] C:\Program Files\Java\jre\bin\javaw.exe (Feb 25, 2017, 12:46:07 AM)
Feb 25, 2017 12:46:08 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefiniti
INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
BaseballCoach: practice batting for 30 minutes....
Today you will enjoy in work....
Feb 25, 2017 12:46:09 AM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup date
```

Setter injection:

- **Inject dependencies by calling setter method on your class**

Steps:

- **Creating a setter method in your class for injections**
- **Configure the dependency injection in spring config file**

Creating a setter method in your class for injections:

```
private Fortune fortune;

	public void setFortune(Fortune fortune) {
		this.fortune = fortune;
	}
	@Override
	public String getFortuneDetails() {
		return fortune.getFortune();
	}
```

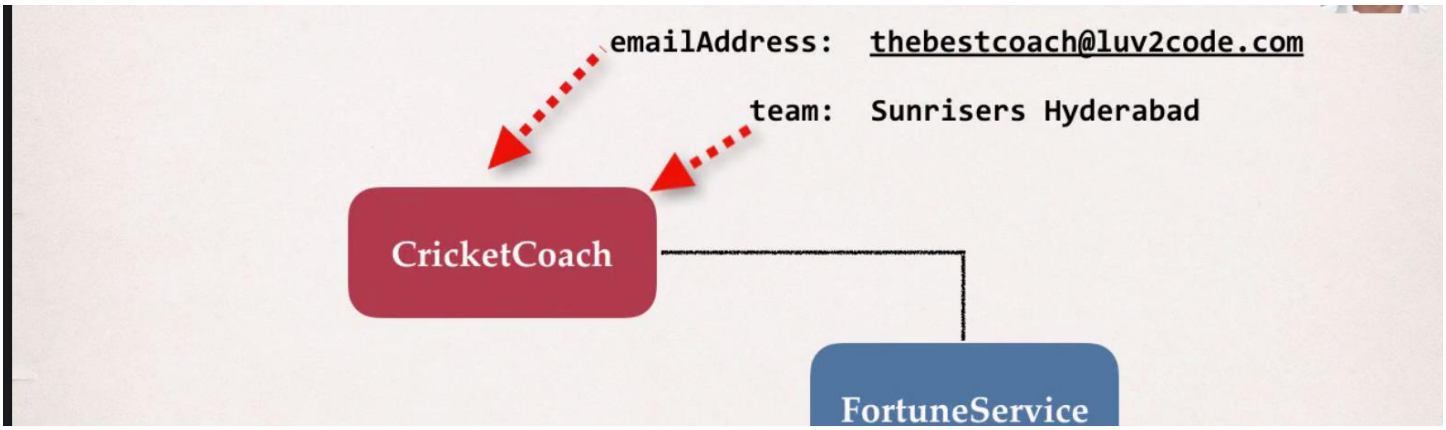
Configure the dependency injection in spring config file:

```
<bean id="trackCoach" class="com.test.spring.TrackCoach">
	<property name="fortune" ref="fortune"></property>  <!-- setter injection -->
</bean>
<bean id="fortune" class="com.test.spring.FortuneService" />
```

```
<terminated> myApp [Java Application] C:\Program Files\Java\jre\bin\javaw.exe (Feb 25, 2017, 12:54:57 PM)
Feb 25, 2017 12:54:57 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
TrackCoach: Run 5K miles today....
FortuneService: Today you will the winner....
```

Injecting Literal Values:

- **Adding some values into our spring objects.**



Create a setter method for those private fields:

```
public class BaseballCoach implements Coach {

	private String email;
	private String team;
```

```java
public void setEmail(String email) {
	this.email = email;
}

public void setTeam(String team) {
	this.team = team;
}

public String getEmail() {
	return email;
}

public String getTeam() {
	return team;
}
```

Configure those fields in the spring config file:

```xml
<bean id="baseballCoach" class="com.test.spring.BaseballCoach">
	<property name="email" value="yunus@gmail.com" />    <!-- literal value injection -->
	<property name="team" value="Chennai super kings" />
</bean>
```

Injecting values from properties file: (XML)

- **This can help in avoiding hardcoded values in config file**
- **Read values from properties file**
- **Steps:**
  - **Create a properties file**
  - **Load properties file in spring config file**
  - **Reference values from properties file**

Create a properties file: (src folder)

```
coach.email=yunusirshad@yahoo.com
coach.team=Chennai Super kings
```

Load properties file in spring config file:

```xml
<context:property-placeholder location="classpath:coach.properties"/>
```

Reference values from properties file:

```xml
<bean id="baseballCoach" class="com.test.spring.BaseballCoach">
	<property name="email" value="${coach.email}" />          <!-- ${prop name} -->
	<property name="team" value="${coach.team}" />
</bean>
```

Bean Scopes:

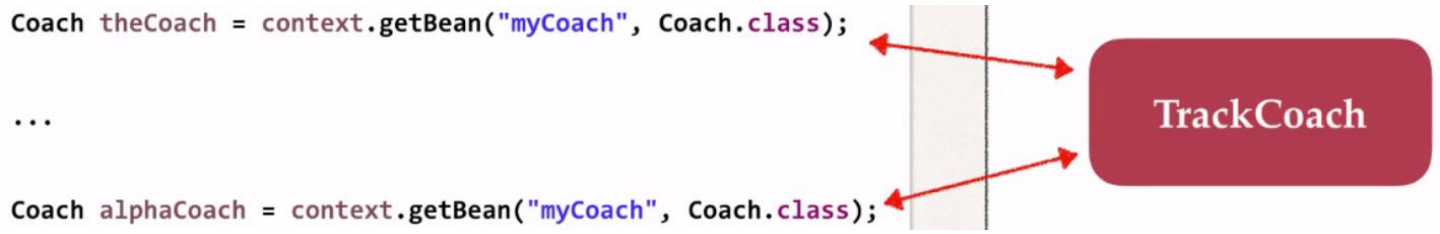
- **Scopes = lifecycle of a bean**
  - **How long it will live?**
  - **How many instances are created?**
  - **How is the bean shared?**

Types of Bean Scopes:

- **Singleton (default)**
- **Prototype -- creates a new bean instance for every container request.**
- **Request -- scoped to a HTTP request. (web apps)**
- **Session -- scoped to a HTTP session. (web apps) shopping cart.**
- **global-session -- scoped to a global HTTP web session. (web apps)**

Singleton Bean Scope: (stateless bean)

- **Spring container creates only one instance of the bean**
- **It is cached in main memory**
- **All requests of the bean will return shared reference to same bean. (like alphaCoach)**




```
<bean id="coachBean" class="com.test.spring.BaseballCoach" scope="singleton"/>
```

```
Coach theCoach = context.getBean("coachBean", BaseballCoach.class);
Coach theCoach2 = context.getBean("coachBean", BaseballCoach.class);

        System.out.println(theCoach.hashCode());          //1620303253
        System.out.println(theCoach2.hashCode());         //1620303253
```

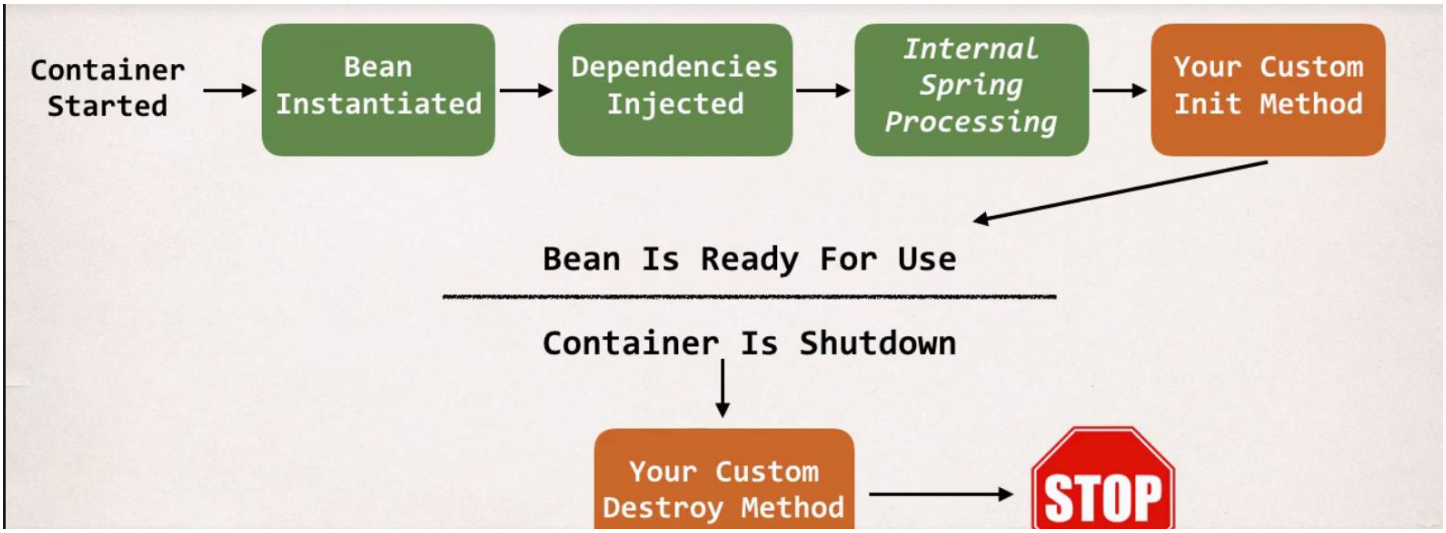
Prototype Bean Scope:

- **New object for each request.**


```
<bean id="coachBean" class="com.test.spring.BaseballCoach" scope="prototype"/>
```

```
        System.out.println(theCoach.hashCode());          //1547425104
        System.out.println(theCoach2.hashCode());         //152134087


        boolean result = (theCoach == theCoach2);
        System.out.println(result);                  // false memory location
```

Bean Lifecycle methods:



Container is Shutdown = context.close();

Bean initialization:

- **You can add custom code during bean initialization.**
- **Setting up handles to resources (db, sockets, file etc)**
- **This method name must be "public void" method and contains NO argument**

Bean destruction:

- **You can add custom code during bean destruction**
- **Cleaning up handles to resources (db, sockets, file etc)**
- **This method name must be "public void" method and contains NO argument.**
- **In scope "prototype", destruction are not called.**

Steps:

- **Define your methods for init and destroy**

```
public void insertCustomLogic()
{
	System.out.println("BaseballCoach: Insert some custom logic through init()method....");
}

public void destructCustomLogic()
{
	System.out.println("BaseballCoach: Destruct through destroy() method...");
}
```

- **Configure the methods into spring config file.**

```
<bean id="coachBean" class="com.test.spring.BaseballCoach" init-method="insertCustomLogic" destroy-method="destructCustomLogic" scope="singleton"/>
```

```
<terminated> MyApp (4) [Java Application] C:\Program Files\Java\jre\bin\javaw.exe (Feb 25, 2017, 5:30:16 PM)
Feb 25, 2017 5:30:17 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefr
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup
Feb 25, 2017 5:30:17 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefiniti
INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
BaseballCoach: Insert some custom logic through init()method....
90320863
90320863
true
Feb 25, 2017 5:30:17 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup dat
BaseballCoach: Destruct through destroy() method...
```

Spring JAVA Annotations:

- **Special labels added to JAVA classes**
- **Provide meta-data about the class (eg. Shoe box contains meta-data "size… color… model.")**
- **Processed at runtime or compile time for special processing.**

@Override = telling compiler that we are overriding a method from parent class or interface.

Why Spring needs annotations?

- **XML configuration is verbose (if we have 100 beans like that…)**
- **Configure your spring beans with annotations**
- **Annotations minimizes XML configuration**

Scanning of Component classes:

- **Spring will scan all JAVA classes in search of annotations.**
- **If find, then automatically register into spring container.**

Development process:

- **Enable component scanning in config file. (Spring scan recursively)**
- **Add @Component annotation in JAVA class (declare spring bean)**
- **Retrieve your bean from spring container.**

```
<context:component-scan base-package="com.test.spring" />
```

```
@Component("coachBean")
public class BaseballCoach implements Coach {
```

```
Coach theCoach = context.getBean("coachBean",BaseballCoach.class);
```

```
INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
BaseballCoach: practice bowling today....
Feb 25, 2017 6:48:13 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup
```

Default component names:

- **It is good practice to use default component names**

```
@Component            // it will take default bean id ("baseballCoach")
public class BaseballCoach implements Coach {
```

- **special case of when BOTH the first and second characters of the class name are upper case, then the name is NOT converted.**

RESTFortuneService --> RESTFortuneService

- **No conversion since the first two characters are upper case.**
- **Behind the scenes, Spring uses the Java Beans Introspector to generate the default bean name.**
- **In this case, we can give explicit name ("restFortuneService")**

Spring Dependency injection (Autowiring using Annotations)

Auto Wiring:

- **Spring will match up the objects together**
- **Spring will look for a class that matches the property**
  - **Matches by type: class or interface**
- **Spring will inject it automatically…. Hence it is autowired.**
- **Spring supports multiple @autowire methods**

Steps:

  - **Create a dependency interface and class.**
  - **Inject the dependency into a class**
  - **Spring will scan components**
  - **Any one implements?**
  - **Once Spring finds it, then automatically inject into our class.**

Auto Wiring Implementation Types:

- **Constructor**
- **Setter**
- **Field**

Auto Wiring (Constructor Dependency Injection):

Steps:

- **Define dependency interface and class**
- **Create a constructor in your class for injections**
- **Configure dependency injection with @Autowired annotation.**

```
@Component
public class FortuneService implements Fortune {
	public String getFortune() {
		return "FortuneService: Today you will the winner....";
	}
```

```java
@Component
public class TrackCoach implements Coach {

	private FortuneService FS;

	@Autowired
	public TrackCoach(FortuneService FS) {
		this.FS=FS;
	}
```

```
TrackCoach: Run 5K miles today....
FortuneService: Today you will the winner....
Feb 25, 2017 7:56:17 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup
```

Auto Wiring: (Setter Injection)

Steps:

- **Create setter method in your class for injections.**
- **Configure the dependency injection with @Autowired annotation.**

```java
@Component
public class TrackCoach implements Coach {

	private FortuneService FS;

	@Autowired
	public TrackCoach(FortuneService FS) {
		this.FS=FS;
	}
```

```
BaseballCoach: practice bowling today....
FortuneService: Today you will the winner....
Feb 25, 2017 8:23:39 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup
```

Auto Wiring Method Injection:

- **You can inject any method.**
- **Some sort of input we need to get, so connection is required.**

```java
@Component
public class BaseballCoach implements Coach {

	private FortuneService FS;

/*	@Autowired
	public void setFS(FortuneService fS) {
		FS = fS;
	}*/

	@Autowired
	public void methodInjectionFS(FortuneService FS) {
```

```
        this.FS=FS;
    }
```

Auto Wiring (Field Injection):

- **Using this you can directly inject dependencies on the fields even though it is private.**
- **It is accomplished by using JAVA reflection**
- **No need of constructor or setter injection.**

```
@Autowired
private FortuneService FS;
```

Annotation Auto Wiring and Qualifiers:

- **Anyone implements FortuneService interface?**
  - **Multiple implementations from one interface**
- **Which one to pick?**
  - **We will get NoUniqueBeanDefinitionException**

Logs → we expected single match but we found 4.

```
@Component
public class BaseballCoach implements Coach {

    @Autowired
    @Qualifier("fortuneService2")          // declare bean id
    private Fortune FS;                    // use interface to declare a field
```

```
BaseballCoach: practice bowling today....
FortuneService2: Today you will the winner....
Feb 26, 2017 12:29:53 AM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup d
```

Constructor injection:

```
    @Autowired
    public TrackCoach(@Qualifier("fortuneService") Fortune FS) {
    //constructor injection
        this.FS=FS;
    }
```

Bean Lifecycle methods Annotations for bean initialization and destruction:

```
// define my init method
    @PostConstruct
    public void doMyStartupStuff() {
        System.out.println(">> TrackCoach: inside of doMyStartupStuff()");
    }

// define my destroy method
    @PreDestroy
    public void doMyCleanupStuff() {
```

```
        System.out.println(">> TrackCoach: inside of doMyCleanupStuff()");
    }
```

Injecting values from properties file: (Annotation)

- **Configure spring config file for loading properties file**
- **Declare @Value on top of the field.**

```
    <context:property-placeholder location="classpath:coach.properties"/>
</beans>
```

```
    @Value("${coach.team}")
    private String team;

    public String getTeam() {
        return team;
    }

    public void setTeam(String team) {
        this.team = team;
    }
```

```
Chennai Super kings
Feb 26, 2017 12:53:55 AM org.springframework.context.support.ClassPathXmlApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup d
>> TrackCoach: inside of doMyCleanupStuff()
```

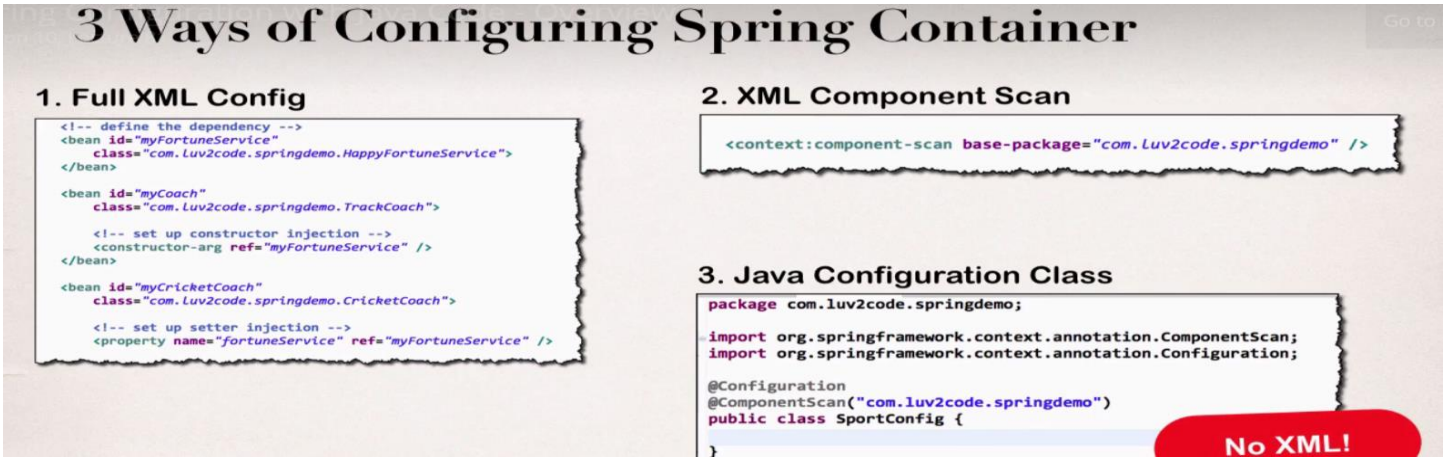
Spring bean scopes: (Annotations)

- **Singleton and Prototype can used through annotations**

```
@Component
@Scope("prototype")
public class BaseballCoach implements Coach {
```

Spring Configuration: (JAVA code)

- **No need of XML file**
- **Configure spring container using JAVA code**



Steps:

- **Create a JAVA class and annotate with @Configuration**
- **Add component scanning support (OPTIONAL)**
- **Read spring configuration class**
- **Retrieve bean from spring container.**

```
@Configuration
@ComponentScan("com.test.spring")
public class BaseballCoach implements Coach {
```

```
		AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(BaseballCoach.class);

		Coach theCoach = context.getBean("baseballCoach",BaseballCoach.class);
		System.out.println(theCoach.getDailyDetails());
		context.close();
```

```
BaseballCoach: practice bowling today....
>> TrackCoach: inside of doMyCleanupStuff()
```

Defining Spring beans with JAVA code:

- **Define method to expose bean**
- **Inject the method as bean dependencies**
- **Read spring configuration class**
- **Retrieve bean from spring container**

```
@Component
@Configuration
public class SwimCoach implements Coach {

	private Fortune fortune;

	public SwimCoach(Fortune fortune) {
		this.fortune=fortune;
	}

	@Override
	public String getDailyDetails() {
		return "Swim 1 miles...";
	}

	@Bean
	public Fortune fortuneService()			//any method name...in which
bean id is registered
	{
		return new FortuneService();
	}
```

```java
public Coach swimCoach(@Qualifier("fortuneService") Fortune fortune)
// define bean for our swimcoach and inject dependency
{
      return new SwimCoach(fortuneService());          // inject the method
}
```

```java
Coach theCoach2 = context.getBean("swimCoach",Coach.class);
      System.out.println(theCoach2.getDailyDetails());
      context.close();
```

```
Swim 1 miles...
Feb 26, 2017 10:55:15 AM org.springframework.context.annotation.AnnotationCor
```

Inject Values from properties file (JAVA Code):

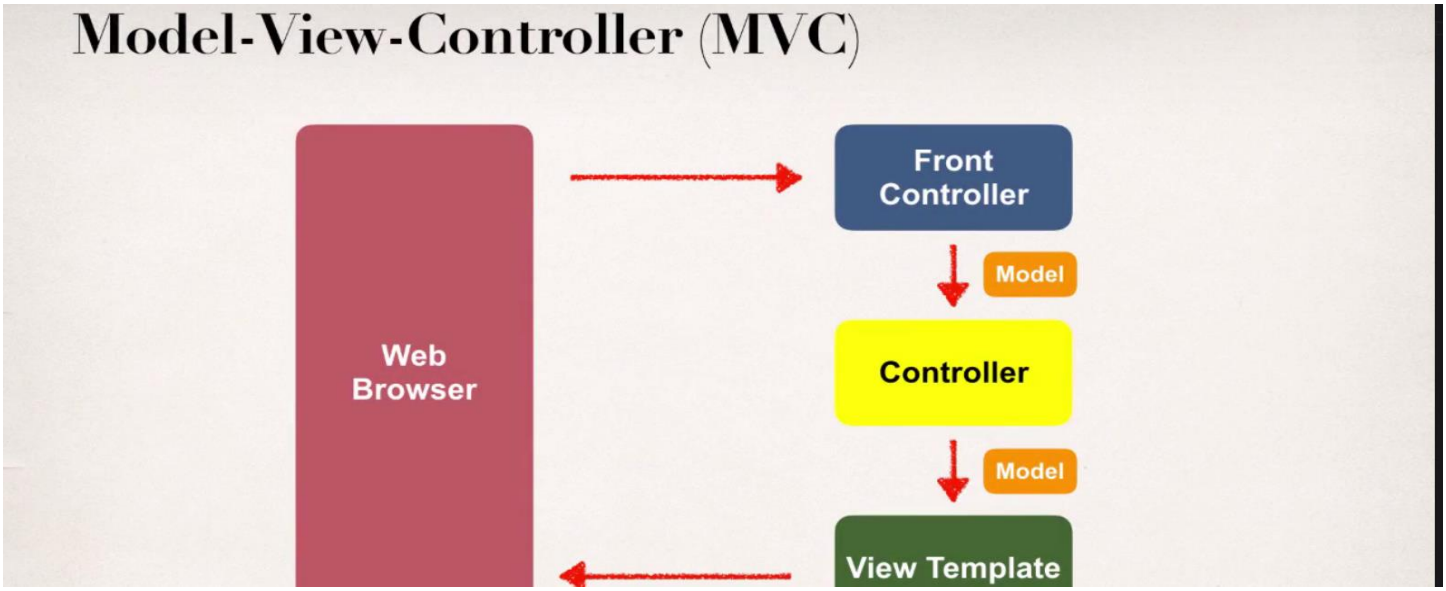
- **Create properties file**
- **Load properties file**
- **Reference the values in JAVA class**

```
coach.team=Chennai Super kings
```

```java
@PropertySource("classpath:coach.properties")
public class BaseballCoach implements Coach {

@Value("${coach.team}")
private String team;
```

SPRING MVC:

- **Framework for building web apps in JAVA**
- **Based on Model View Controller design pattern.**
- **Adds features of Spring Core (IoC and DI).**



Controller: contains the business logic

View Template: JSP or HTML page render the data on page.

Benefits of Spring MVC:

- Leverage use of reusable UI components.
- Help manage application state of web requests.
- Process form data: validation, conversion etc..
- Flexible configuration for the view layer. (Velocity or Free marker)

Components of Spring MVC:

- A set of web pages to layout UI components.
- A collection of spring beans (controller, services etc…)
- Spring configuration file (XML, Annotation or JAVA)

Spring MVC Front Controller:

- Front Controller = DispatcherServlet
  - Part of spring framework
  - Already developed by spring team
- We need to create
  - Model objects
  - Controller classes
  - View templates

Controller:

- Code created by developer handles the web request.
- Contains business logic
  - Handle request
  - Store/retrieve data from DB, web service
  - Place that date into model. (container for our data
- Send to appropriate view template

Model:

- Contains data
- Store and retrieve data from DB, web service
  - Using a spring bean
- Data can be any object or collection.
- Model data is passed onto the JSP view template to display data.

View:

- Spring MVC is flexible supports many view templates (freemarker, JSP)
- Most common is JSP/JSTL (JSP standard Tag libraries)
- Developer creates a page and display data.
- For eg: airport website → confirmation page.

SPRING MVC Configuration:

- Part 1
  - Add configurations to file = WEB-INF/web.xml

- Configure Spring MVC DispatcherServlet
- Setup URL mappings to Spring MVC DispatcherServlet

- **Part 2**
  - Add configurations to file = WEB-INF/spring-mvc-demo-servlet.xml
  - Add support for Spring component scanning
  - Add support for conversion, formatting and validation
  - Configure Spring MVC View Resolver.

Configure DispatcherServlet:

Web.xml:

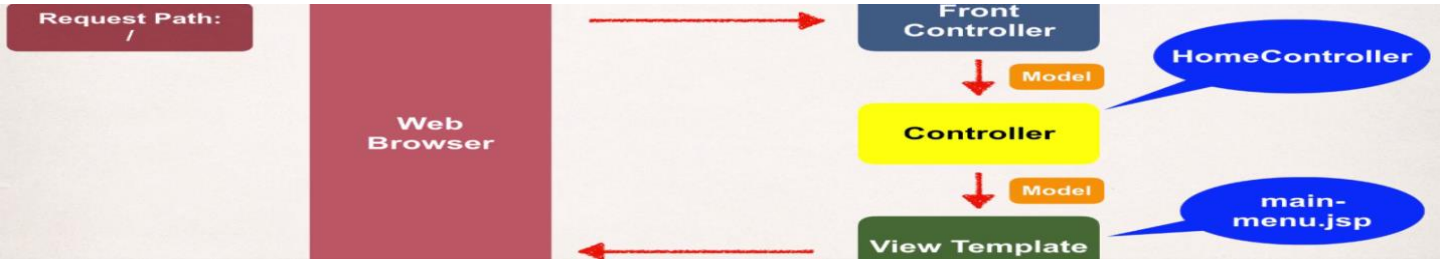
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
	<servlet-name>dispatcher</servlet-name>
	<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

	<!-- initialize the spring configuration file -->
	<init-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
	</init-param>

	<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
	<servlet-name>dispatcher</servlet-name>
	<url-pattern>/</url-pattern>
			<!-- all requests coming in must be handled and display the page -->
</servlet-mapping>
</web-app>
```

Spring Configuration File 2:

spring-mvc-demo-servlet.xml:

```
   <context:component-scan base-package="com.test.spring" />

   <!-- Add support for conversion, formatting and validation -->
   <mvc:annotation-driven/>

   <!-- configuring view resolver -->
   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="prefix" value="/WEB-INF/view/"/>
		<property name="suffix" value=".jsp"></property>
   </bean>
</beans>
```

Creating Spring Controller and View:



Move all jars to WEB-INF/lib folder

Steps:

- **Create a controller class**
- **Define controller method**
- **Add request mapping to controller method**
- **Return view name**
- **Develop view page**

Create a controller class:

- **Annotate with @Controller**
- **@Controller inherits from @Component… supports scanning**

Define controller method:

- **Controller class contains a method, which returns view name**

Add request mapping to controller method:

- **Add some type of web request used by annotating @RequestMapping("/");**
- **Acts as a path from request to controller method.**
- **Hit URL, → request mapping → controller method → showmypage view name**

Return View name:

- **Controller method returns the view name**

Develop View page:

- **JSP page with HTML tags should be created.**