**Into the SPRING**

**09/09/2017**

**SPRING FRAMEWORK OVERVIEW**

The official website for spring: www.spring.io

Documentation, Tutorials

https://spring.io/docs/reference

Why spring?

Simplified Java Enterprise Development

* **Goal of Spring:**

\* Lighweight development with java POJOs

\* Dependency injection to promote loose coupling

\* Declarative programming with Aspect Oriented Programming (AOP)

\* Minimize boilerplate Java code

/Users/Prashanth/sagar/java/Spring/

* **Core Container**

Beans

Core

SpEL

Context

It has a bean factory for creating beans, reads Config files..And making those beans available

AOP

It is a kind of layer on the core.

\* How much time to find the response time

\* I’m going to inject that logging on the outer layer using the Config files

Without touching the code. Provide security on the outer layer....

* **Data Access Section**

JDBC Helper classes much easy to access databases (Reduce code by 50%)

ORM (Object to Relational Mapping)

Integration with hibernation and JPA

JMS

For sending a sync messages to a message broker

Add transaction support

Makes use of AOP behind the scenes

* **Web Layer**

Servlet

Web Socket

Web

Portlet

* **Infra**

AOP

Aspects

Instrumentation

Messaging

🡪 Java agent to remotely monitor our application using JMX

* **Test Layer**

Unit

Integration

Mock

Supports (TDD) -🡪 test driven deployment

**Spring Projects🡪 www.spring.io**

* **Additional Spring modules**

Spring cloud, Spring Data, Spring Batch, Spring Security, Spring Android, Spring Web Flow, Spring Web Services, Spring LDAP.

* **Set up Environment:**

Java App Server 🡪 Tomcat

/Users/Prashanth/downloads/apache-tomcat-8.0.46/bin

Java IDE 🡪 Eclipse EE

Mars 1

* **Download Spring Jar files**

🡪 Create Eclipse Project

🡪 Download Spring JAR files

http://repo.spring.io/release/org/springframework/spring/4.2.6.RELEASE/

🡪 Download Common Logging JAR files

https://commons.apache.org/proper/commons-logging/download\_logging.cgi

🡪 Add JAR files in a lib folder and then 🡪 Build Path 🡪 Add jars

The jar files in the reference libraries will not be there in the war files.

It is a good practice to save the jar files in the lib folder.

**Inversion of Control**

The approach of the outsourcing the construction and management of objects.

getDailyWorkout()

My App -----------------------------------------> Baseball coach

<---------------------------------------

App should be configurable

Easily change the coach for another sport

Hockey cricket tennis gymnastics

Creating the objects within the program by configuring the things in the Config file.

**Code Demo:**

MyApp.java 🡪 Main Method

BaseballCoach.java

Coach.java 🡪 Interface

TrackCoach.java

* We need a object factory
* Spring provides an object factory
* We give the configuration to the different classes

Spring Container:

* Primary function
  + Create and manage objects (Inversion of control)
  + Inject object’s Dependencies (Dependency Injection)

Configuring Spring Container:

* XML configuration file
* Java Annotations
* Java Source Code

Spring Development Process

1. Configure the Spring Beans
2. Create a Spring Container
3. Retrieve Beans from Spring Container

🡪 XML Config file

applicationContext.xml

🡪Creating the spring container:

Spring container is generally known as **Application Context**

* Specialized implementations
* ClassPathXmlApplcaitionContext
* AnnotationConfigApplicationContext
* GenericWebApplicationContext

And others

Application is going to ask the container hey give me a sports object

🡪 Retrieving Beans from Container:

In java there is concept called reflection (there is a way to create a object on the fly dynamically).

applicationContext.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"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.1.xsd"*>

<bean id= *"myCoach"*

class=*"com.codewithshan.springdemo.TrackCoach"*>

</bean>

</beans>

HelloSpringApp.java

Create a new class HelloSpringApp with main method and

// load the spring configuration file

ClassPathXmlApplicationContext context =

**new** ClassPathXmlApplicationContext("applicationContext.xml");

//retrieve bean from the spring container

Coach theCoach = context.getBean("myCoach",TrackCoach.**class**);

//call methods on the bean

System.***out***.println(theCoach.getDailyWorkout());

//close the context

context.close();

**Dependency Injection:**

The dependency inversion principle:

Dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

Give my car object from the car factory.

“Dependency is same as helper objects”

the baseball objects or the track coach objects can be injected automatically.

The coach will also provide daily fortunes (new objects)

Injection Types:

**Constructor Injection**

1. Define the dependency interface and object
2. Create a constructor in your class for injections
3. Configure the dependency injection in the Spring Config file

🡪Create a FortuneService interface and HelloFortuneClass

**public** **class** HappyFortuneService **implements** FortuneService {

@Override

**public** String getFortune() {

**return** "today is your lucky day";

}

}

--------------------------------------------------------------

**public** **interface** FortuneService {

**public** String getFortune();

}

🡪Create a constructor in the baseballCoach (fortuneService = theFortuneService)

//define a private

**private** FortuneService fortuneService;

//define a constructor for dependency injection

**public** BaseballCoach(FortuneService theFortuneService){

fortuneService = theFortuneService;

}

@Override

**public** String getDailyFortune() {

//use my fortuneService to get a fortune

**return** fortuneService.getFortune();

}

🡪Dependency injection in xml

<!-- Define beans -->

<!-- define dependency -->

<bean id= *"myFortune"*

class=*"com.codewithshan.springdemo.HappyFortuneService"*>

</bean>

<bean id= *"myCoach"*

class=*"com.codewithshan.springdemo.BaseballCoach"*>

<!-- Setup constructor injection -->

<constructor-arg ref=*"myFortune"*/>

</bean>

System.***out***.println(theCoach.getDailyFortune());

--🡪Print it out in the SpringHelloApp.java🡨--

**Using Setter Methods:**

The name is same as the myFortuneServices

Injecting

1. Create setter methods in the class for injections
2. Configure the injection in the applicationContext.xml

**private** String team;

**private** String email;

**public** **void** setTeam(String team) {

**this**.team = team;

System.***out***.println("CricketCoach:Set Team Name");

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

System.***out***.println("CricketCoach:Set email address");

}

applicationContext.xml

<bean id=*"myCricketCoach"*

class=*"com.codewithshan.springdemo.CricketCoach"*>

<!-- Setup setter injection -->

<property name=*"fortuneService"* ref=*"myFortuneServices"*/>

<!-- inject literal values -->

<property name =*"email"* value=*"shanboi99@gmail.com"*/>

<property name =*"team"* value=*"Beachizag"*/></bean>

🡪Create a properties File:

Sports.properties

Give the name value pairs.

foo.email=rcb@gmail.com

foo.team=Royal Challengers Bangalore

🡪Load Properties file in the Spring Config File

<context:property-placeholder location=*"classpath:sport.properties"*/>

🡪 Reference Values from properties file

<property name =*"email"* value=*"${foo.email}"*/>

<property name =*"team"* value=*"${foo.team}"*/>

**Bean Scopes**

By def. scope is singleton scope

A single bean is taken as ref when we create a ref

**Scope**

Singleton Scope: (Stateless)

Spring creates only one instance of the bean, by default

It is cached in the memory.

Prototype Scope: (State full)

New object for each request!!

Request Session global

//Load the spring config file

ClassPathXmlApplicationContext context =

**new** ClassPathXmlApplicationContext("beanScope-applicationContext.xml");

//retrieve the spring container

Coach theCoach = context.getBean("myCoach",Coach.**class**);

Coach alphaCoach = context.getBean("myCoach",Coach.**class**);

//check to see if they are the same

**boolean** result = (theCoach == alphaCoach);

System.***out***.println("Poiting to the same object:" + result);

System.***out***.println("\nMemory location for theCoach:"+theCoach);

System.***out***.println("\nMemory location for theCoach:"+alphaCoach+"\n");

//close the context

context.close();

When the class is run:

Pointing to the same object: false

Memory location for theCoach:com.codewithshan.springdemo.TrackCoach@76707e36

Memory location for theCoach:com.codewithshan.springdemo.TrackCoach@614ddd49-> Prototype

**Bean Lifecycle:**

**Container Started**

**🡪 Bean Instantiated**

**🡪 Dependencies Injected**

**🡪 Internal Spring Processing**

**🡪 Your Custom init Method**

**🡪 Bean is ready for use**

**-------------------------------------------**

**🡪 Container Is Shutdown**

**🡪 Your Custom Destroy Method**

**🡪 Stop**

Development process:

Init and destroy must be | public void

|| No args

1. Define methods for init and destroy
2. Configure the method names in Spring Config file

Init and destroy methods in TrackCoach.java

//Add an init method

**public** **void** doMyStartupStuff(){

System.***out***.println("trackcoach: inside method doMyStartupStuff");

}

//Add a destroy method

**public** **void** doMyCleanupStuff(){

System.***out***.println("trackcoach: inside method doMyCleanupStuff");

}

BeanLifeCycle-applicationContext.xml

<!-- Define beans -->

<!-- define dependency -->

<bean id= *"myFortuneServices"*

class=*"com.codewithshan.springdemo.HappyFortuneService"*>

</bean>

<bean id= *"myCoach"*

class=*"com.codewithshan.springdemo.TrackCoach"*

init-method=*"doMyStartupStuff"*

destroy-method=*"doMyCleanupStuff"*>

<!-- Setup constructor injection -->

<constructor-arg ref=*"myFortuneServices"*/>

</bean>

</beans>

BeanLifeCycleDemo.java

//Load the spring config file

ClassPathXmlApplicationContext context =

**new** ClassPathXmlApplicationContext("beanLifeCycle-applicationContext.xml");

//retrieve the spring container

Coach theCoach = context.getBean("myCoach",Coach.**class**);

System.***out***.println(theCoach.getDailyWorkout());

//close the context

context.close();

}

}

When the class is run:

trackcoach: inside method doMyStartupStuff

Run a hard 5k

Sep 09, 2017 5:29:34 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose

INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@497470ed: startup date [Sat Sep 09 17:29:33 EDT 2017]; root of context hierarchy

trackcoach: inside method doMyCleanupStuff

Note: If you have a bean with singleton and another with non singleton

In the singleton bean it will always create object.

For other scopes only, when the context is called then only the object is created.

For prototype scoped beans, Spring does not call the destroy method.

**Spring Java Annotations**

🡪Provide Meta data about the class

🡪Special Labels/markers added to a class

**@Override**

🡺Tells the compiler, we are overriding a method.

🡺At compile time, Complier will check/verify the override.

Spring will scan your java classes for the annotations.

Automatically registers the beans in the spring container.

Development Process:

1. Enable component scanning in the Spring Config file.
2. Add the @Component annotation to your java classes.
3. Retrieve the bean from the Spring Container.

Package: Com.codewithshan.annotations

Project: spring-demo-annotations

applicationContext.xml

<!-- add entry to enable component scanning -->

<context:component-scan base-package=*"com.codewithshan.annotations"*/>

AnnotationDemoApp.java

//read spring config file

ClassPathXmlApplicationContext context =

**new** ClassPathXmlApplicationContext("applicationContext.xml");

//get the bean from spring container

Coach theCoach = context.getBean("thatSillyCoach",Coach.**class**);

//call a method on the bean

System.***out***.println(theCoach.getDailyWorkout());

//close the context

context.close();

TennisCoach.java

**package** com.codewithshan.annotations;

**import** org.springframework.stereotype.Component;

//spring will automatically register this bean

@Component("thatSillyCoach") //this is the bean id

**public** **class** TennisCoach **implements** Coach {

@Override

**public** String getDailyWorkout() {

**return** "hello tennis coach: Practice your backhand";

}

}

Coach.java

**package** com.codewithshan.annotations;

**public** **interface** Coach {

**public** String getDailyWorkout();

}

Output:

INFO: Loading XML bean definitions from class path resource [applicationContext.xml]

hello tennis coach: Practice your backhand

Sep 09, 2017 6:33:25 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose

Tennis Coach 🡺 remove the bean id then you can call using the class name. and when you invoke it use the lower case in this case tennisCoach.

**Spring AutoWiring**

For dependency injection spring will automatically look for a class and matches the property

Spring will scan the @components

Any one implements the FortuneService interface???

If so, lets inject them.

* Constructor injection
* Setter injection
* Field injection

Development Process: **Constructor injection**

1. Define the dependency interface and class.
2. Create constructor in your class for injections
3. Configure the dependency injection using Autowired annotation.

FortuneService.java

**package** com.codewithshan.annotations;

**public** **interface** FortuneService {

**public** String getFortune();

}

**HappyFortuneService.java**

**package** com.codewithshan.annotations;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** HappyFortuneService **implements** FortuneService {

@Override

**public** String getFortune(){

**return** "today is your lucky day!!!";

}

}

Component //this is the bean id

**public** **class** TennisCoach **implements** Coach {

**private** FortuneService fortuneService;

@Autowired

**public** TennisCoach(FortuneService fortuneService) {

**this**.fortuneService = fortuneService;

}

@Override

**public** String getDailyWorkout() {

**return** "hello tennis coach: Practice your backhand";

}

@Override

**public** String getDailyFortune() {

**return** (fortuneService).getFortune();

}

}

TennisCoach.java

applicationContext.xml

<!-- add entry to enable component scanning -->

<context:component-scan base-package=*"com.codewithshan.annotations"*/>

</beans>

Run:

hello tennis coach: Practice your backhand

today is your lucky day!!!

Development process: **Setter injection**

1.Creating a setter methods in the class

2. Configure the dependency injection using the autowired annotation

//define a default constructor

**public** TennisCoach(){

System.***out***.println(">>Tennis coach: inside the default constructor");

}

//Creating a setter method

@Autowired

**public** **void** setFortuneService(FortuneService theFortuneService){

fortuneService =theFortuneService;

}

Output:

>>Tennis coach: inside the default constructor

>>Tennis coach: inside the setter method

hello tennis coach: Practice your backhand

today is your lucky day!!!

**Field Injection**

Configure the dependency injection with Autowired

Unlike the dependency injection or setter injection, field injection can be done.

@Autowired

**private** FortuneService fortuneService;

Note:

Choose a injection type and stick to it.

@Component

**import** org.springframework.stereotype.Component;

@Autowired

**import** org.springframework.beans.factory.annotation.Autowired;

@Override

What happens if there are multiple interfaces,

FortuneService

1. HappyFortuneService
2. RandomFortuneService
3. DatabaseFortuneService
4. RestfullFortuneService

Spring will give the following error:

NouniqueBeanDefinition

Inorder to resolve this we need to give spring a unique bean

So

@Autowired

**@Qualified**(“happyFortuneService”)

**import** org.springframework.beans.factory.annotation.Qualifier;

🡺 Can apply @Qualifier injection to any of the injections

@Autowired

@Qualifier("happyFortuneService")

**private** FortuneService fortuneService;

By keeping the qualifier we can avoid this error

Exception in thread "main" org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'tennisCoach': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire field: private com.codewithshan.annotations.FortuneService com.codewithshan.annotations.TennisCoach.fortuneService;

databaseService,happyFortuneService,randomService,restFortuneService

@Qualifier("randomFunctionService")

The first letter should be small alphabet

@Value is the new annotation for when using the properties file(data in key value pairs).

@Value(“${foo.email}”)

@Value(“${foo.team}”)

BEAN ANNOTATION METHOD WITH ANNOTATIONS

1. Define methods for init and destroy
2. Add annotations @PostConstruct AND @PreDestroy

Write the init and destroy methods.

**Spring Configuration with Java Code**

3 ways of configuring Spring Container

**Full XML Config**

**XML Component Scan**

**Java Configuration Class (No XML)**

Development Process:

1. Create a Java class and annotate **@Configuration**
2. Add component scanning support **@ComponentScan** (optional)
3. Read Spring Java Configuration Class
4. Retrieve bean from Spring Container

SportConfig.java

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.codewithshan.annotation")

**public** **class** SportConfig {

}

In the main method

Step 3: //read spring config file

AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext("applicationContext.xml");

Step 4:

//get the bean from spring container

Coach theCoach = context.getBean("tennisCoach",Coach.**class**);

Development Process

1. Define method to expose bean

2. Inject bean Decencies

3. Read Spring Java Configuration class

4. Retrieve the bean from the spring container

SwimjavaConfigDemoApp.java 🡺 Main method

SwimCoach.java 🡺

SportsConfig.java 🡺 Beans

@PropertySource

@Value(“${foo.email}”)

@Value(“${foo.team}”)

**SPRING MVC**

* Model View Controller
* Framework for building web applications in java

https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

Web Browser ======🡺 Front Controller

**|| model**

**||**

**||**

**Controller**

**|| model**

**||**

**||**

**Web Browser<======= View Template**

**Spring MVC benefits:**

* The Spring way of **building web app UIs** using Java
* Leverage a **set of reusable UI Components**
* **Help manage application state** for web requests
* Process form data: **validation, conversion** etc.,
* **Flexible configuration** for the **view layer**.

**Components of Spring MVC Architecture:**

* A set of web pages to layout UI components
* A collection of Spring Beans (controllers, Services etc.,)
* Spring Configuration (XML, Annotations or Java)

Front Controller known, as Dispatcher Service is a part of spring framework.

Model Controller and view Templates 🡺 we need to design

**Controller:**

Business logic!

* Handles request.
* Store/ retrieve data (Db, web service)
* Place data in model
* Send to appropriate view Templates

**Model:**

* Contains your data
* Store/retrieve data via backend systems
* Database, web services etc.,
* Use a spring bean if you like
* Place your data in the model
* Data can be any java object/collection

**View Template:**

* Spring MVC is flexible
  + Supports many view templates
* Most common is JSP + JSTL
* Developer creates a page
  + Displays data

Thymeleaf, Groovy, Velocity, Freemarker etc.,

**SPRING MVC CONFIGURATION PROCESS**

Add configurations to WEB-INF/web.xml

1. Configure Spring MVC dispatcher servlet
2. Set up URL mapping to Spring MVC Dispatcher Servlet

Add configurations to file WEB-INF/spring-mvc-demo-servlet.xml

1. Add support for Spring component scanning
2. Add support for conversion, formatting and validation
3. Configure Spring MVC View Resolve

Development steps:

1. Create Controller class

2. Define Controller method

3. Add Request Mapping to controller method

4. Return View Name

5. Develop View Page

HomeController.java

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

@Controller //Controller class Step 1

**public** **class** HomeController {

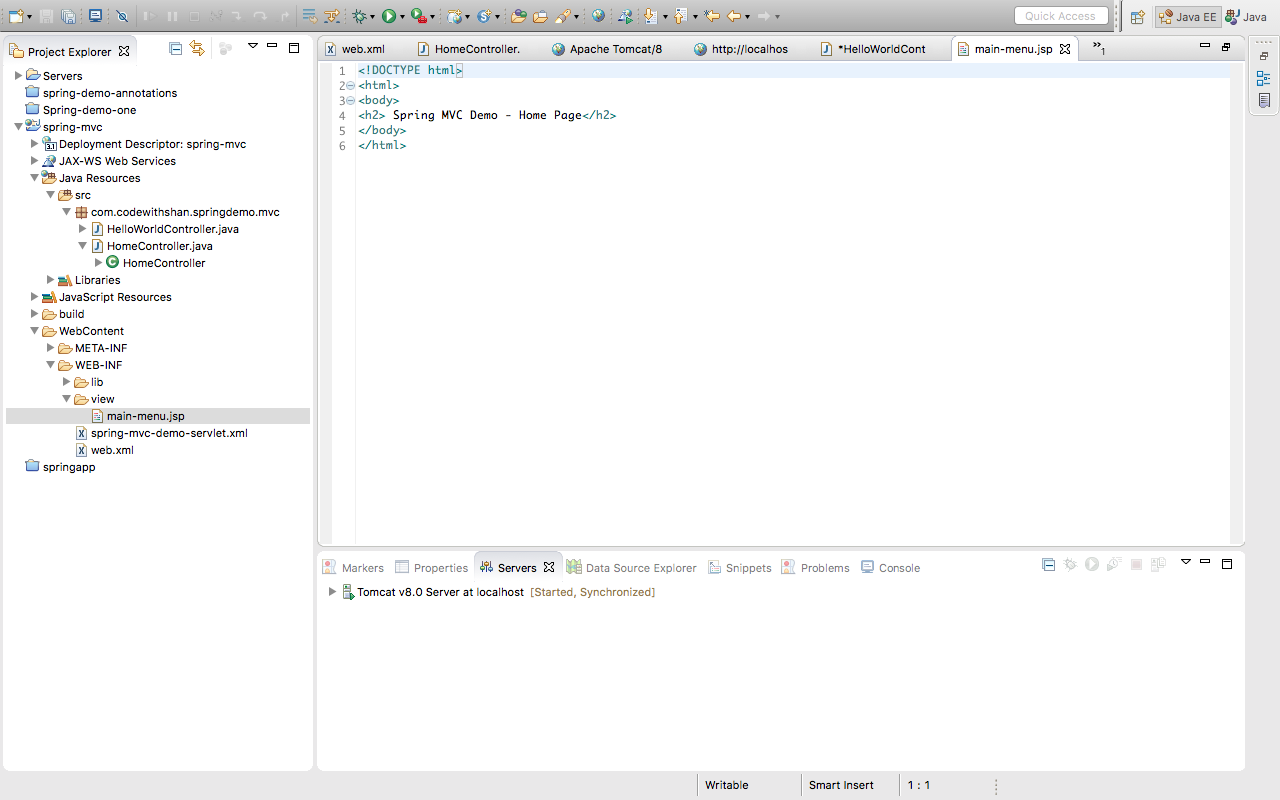
@RequestMapping("/")//Controller method Step 2

**public** String showPage(){

**return** "main-menu";

}

}



WEB-INF/view/Helloworld-form.jsp

<!DOCTYPE html>

<html>

<head>

<title>Hello world - input form</title>

</head>

<body>

<form action=*"processForm"* method=*"get"*>

<input type =*"text"* name=*"studentName"*

placeholder=*"Whats your name?"*/>

<input type=*"submit"*/>

</form>

</body>

</html>

WEB-INF/view/Helloworld.jsp

<!DOCTYPE html>

<html>

<body>

Code easy with Shan!

<br><br>

Student name: ${param.studentName}

</body>

</html>

HelloWorldController.java

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

@Controller

**public** **class** HelloWorldController {

//need a controller method to show the intital jsp file

@RequestMapping("/showForm")

**public** String showForm(){

**return** "helloworld-form";

}

//need a controller method to process the HTML form

@RequestMapping("/processForm")

**public** String processForm(){

**return** "helloworld";

}

}

Adding data to the Spring Model

* The model is a container for your application data.
* In your container
  + You can put anything in your model

Your View Page (JSP) can access data from the model.

View Template –JSP

HelloWorldController.java

//need a controller method to read form data and

//add data to the model

@RequestMapping("/processFormVersionTwo")

**public** String letsShoutDude(HttpServletRequest request,Model model){

//read the request parameter from the HTML form

String theName = request.getParameter("studentName");

//Convert the data to all caps

theName = theName.toUpperCase();

//create the message

String result = "Yo!" + theName;

//add message to the model

model.addAttribute("message",result);

**return** "helloworld";

Note:

A "parameter" is a name/value pair sent from the client to the server - typically, from an HTML form. Parameters can only have [String](http://www.coderanch.com/t/410859/java/java/string-stringbuffer-stringbuilder-performance" \o "String vs. StringBuilder/StringBuffer" \t "_new) values. Sometimes (e.g. using a GET request) you will see these encoded directly into the URL (after the ?, each in the form name=value, and each pair separated by an &). Other times, they are included in the body of the request, when using methods such as POST.  
  
An "attribute" is a server-local storage mechanism - nothing stored in scoped attribues is ever transmitted outside the server unless you explicitly make that happen. Attributes have String names, but store Object values. Note that attributes are specific to [Java](http://www.javaranch.com/" \o "JavaRanch" \t "_new) (they store Java Objects), while parameters are platform-independent (they are only formatted strings composed of generic bytes).  
  
There are four scopes of attributes in total: "page" (for JSPs and tag files only), "request" (limited to the current client's request, destroyed after request is completed), "session" (stored in the client's session, invalidated after the session is terminated), "application" (exist for all components to access during the entire deployed lifetime of your application).  
  
The bottom line is: use parameters when obtaining data from the client, use scoped attributes when storing objects on the server for use internally by your application only.

**Data Binding**

* Spring MVC from tags can make use of data binding
* Automatically setting/retrieving data from a Java object/bean

Spring MVC Form Tags

Form:form

Form:input

Form:textarea

Form:Checkbox

Form:radiobutton

Form:select

…

<html>

regular html

Spring mvc tags

regular html

</html>

**In your Spring Controller**

**\*** Before you show the form, you must add a model attribute.

\* This is a bean that will hold form data for the data binding.

When form is loaded, fields are populated.

When the form is loaded spring MVC will call the student.getFirstName()

When the form is submitted spring mvc will call the student.setFirstName()

This is what we are adding in the studentcontroller.java

@RequestMapping(“/processForm”)

public String processForm(@ModelAttribute(“student”) Student theStudent) {

//log the input data

System.out.println(“theStudent:” + theStudent.getLastName());

Return “student-configuration”;

}

StudentController.java

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

@RequestMapping("/student")

public class StudentController {

@RequestMapping("/showForm")

public String showForm(Model theModel){

//create a new student object

Student theStudent = new Student();

//add student object to the model

theModel.addAttribute("student", theStudent);

return "student-form";

}

@RequestMapping("/processForm")

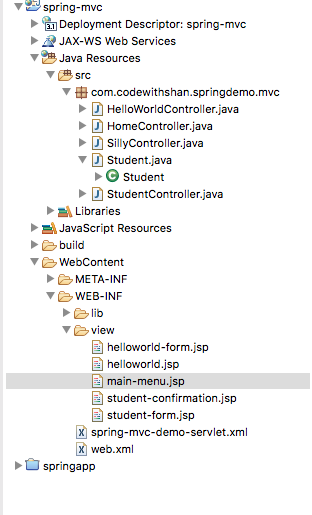
public String processForm(@ModelAttribute("student") Student theStudent){

//log the input data

System.out.println("theStudent:" +theStudent.getFirstName() + "" + theStudent.getLastName());

return "student-confirmation";

}



Student.java

**package** com.codewithshan.springdemo.mvc;

**public** **class** Student {

**private** String firstName;

**private** String lastName;

**public** Student(){

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

}

StudentConfirmation.jsp

<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

<!DOCTYPE html>

<html>

<head>

<title> Student Confirmation</title>

</head>

<body>

The student is confirmed : ${student.firstName} ${student.lastName}

</body>

</html>

StudentForm.jsp

<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

<!DOCTYPE html>

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<h1>Student</h1>

<form:form action=*"processForm"* modelAttribute= *"student"*>

First Name : <form:input path=*"firstName"*/>

<br></br>

Last Name : <form:input path=*"lastName"*/>

<br></br>

<input type = *"submit"* value=*"Submit"* />

</form:form>

</body>

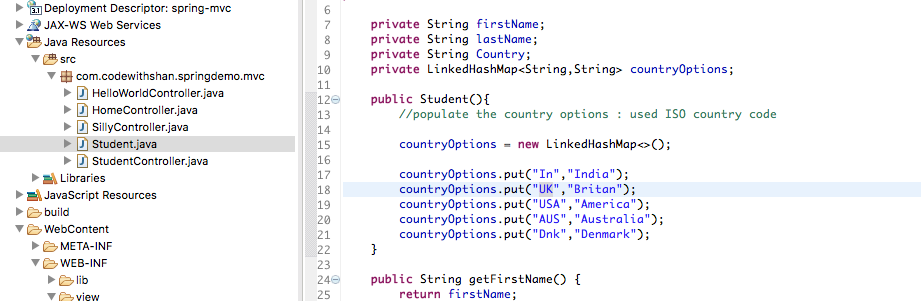
</html>

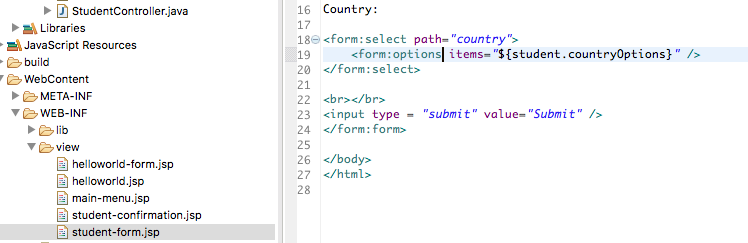
Drop Down List

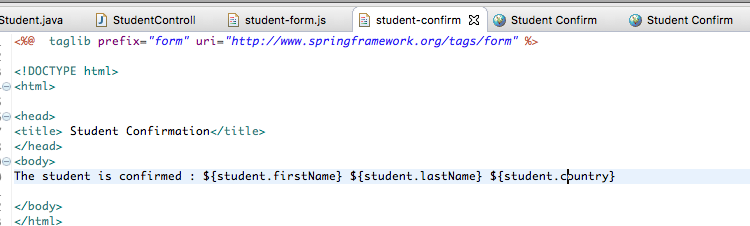
Spring MVC Tag!!!

<form:select path=”xxxxxx”>

<form:option value= “” option=””>







**Spring MVC Form Validation**

* Java has a standard Bean Validation API (JSR-303/309)
* Defines a metadata model and API for entity validation
* Not tied to either the web tier or persistence tier
* Available for server-side apps and also client side JavaFx/Swing Apps

<http://www.beanvalidation.org>

Bean Validation 🡪 Constrain once, Validate Everywhere.

* Spring Version 4 and higher supports Bean Validation API
* Preferred method for Validation when building Spring apps

Validation Annotations

@NotNull

@Min

@Max

@Size

@Pattern

@Future/@Past

Other…

@InitBinder

<http://www.hibernate.org/validator>

* Download the validator jar files from the hibernate website.
* Add jar files to the project.

1. Add validation rule to Customer class.
2. Display error message on HTML form.
3. Perform validation in the Controller class.
4. Update confirmation page.

@InitBinder

* We will use it to trim strings
* Removes whitespaces!!

147 video Customer last name validation missed!

Custom Validation Annotation!!!

1. Create custom validation rule…
2. Create the @CourseCode annotation
3. Create CourseCodeConstraintValidation

Step 1a: Create @CourseCode annotation

Usage Example

@Constraint(validatedBy = CourseCodeConstrauntValidator.class)

@Target( { ElementType.METHOD, ElementType.FIELD })

@Retention (RetentionPolicy.RUNTIME)

public @interface CourseCode {

}

**Hibernate:**

**https://www.slideshare.net/brmeyer/hibernate-orm-performance-31550150**

What is Hibernate?

A framework for persisting/saving Java Objects in a database.

* Hibernate handles all the low-level SQL
* Minimize the amount of JDBC code you have to develop
* Hibernate provides the Object-to-Relational Mapping.

OBJECT TO RELATIONAL MAPPING (ORM)

* The developer defines mapping between Java class and database table

🡺 Saving a Java Object with Hibernate

**//Create a Java Object**

**// Save it to the database**

**Hiberanate uses JDBC in the Background to communicate to the JDBC.**

**To Build Hibernate Applications:**

1. **IDE**
2. **Database Server**
3. **JAR File and JDBC driver**
4. 2017-09-11T21:05:44.830499Z 1 [Note] A temporary password is generated for root@localhost: t/X8s\*3v!zO+

**Setup Database Table:**

mysql -h **prashanth.cij7b4gpzzdo.us-east-1.rds.amazonaws.com:3306**

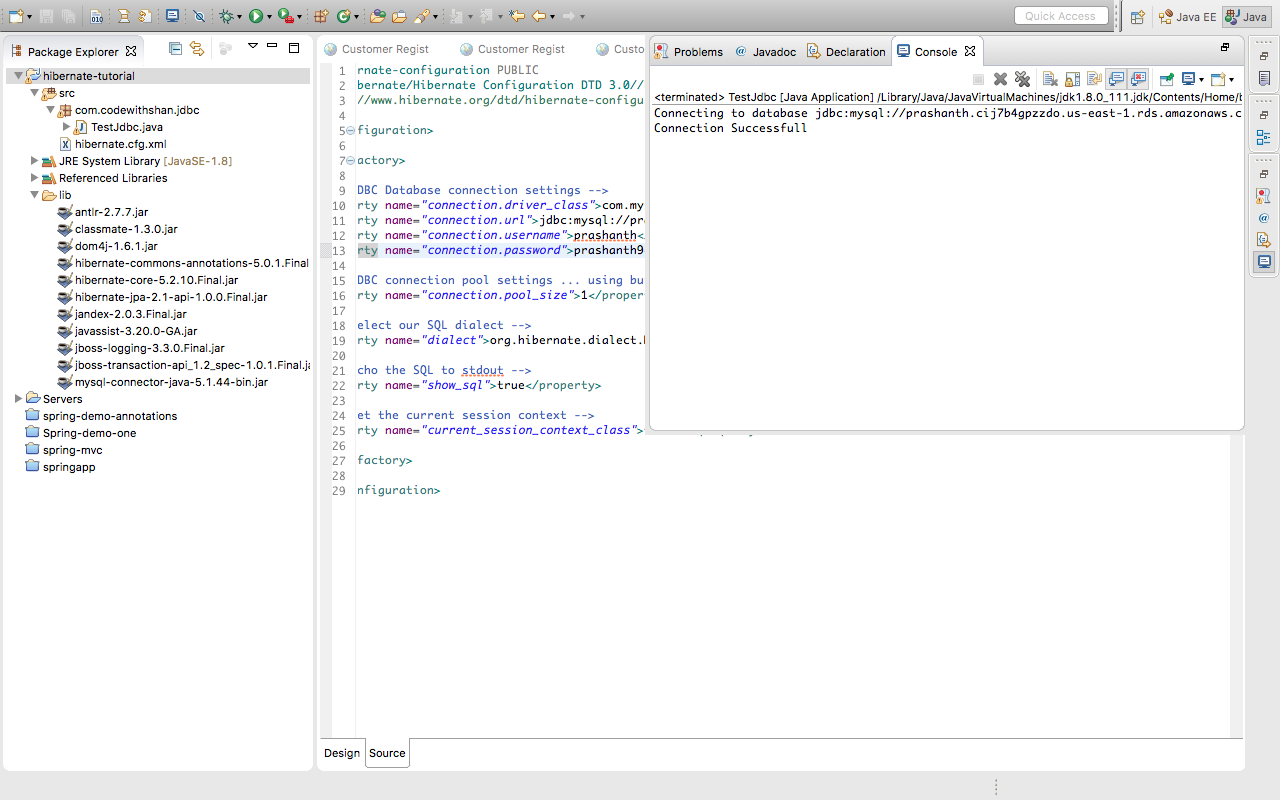
-P 3306 -u prashanth -p

GRANT ALL ON ` **prashanth.cij7b4gpzzdo.us-east-1.rds.amazonaws.com:3306**`.\* TO hbstudent@` **prashanth.cij7b4gpzzdo.us-east-1.rds.amazonaws.com:3306**`;

\* Add the hibernate configuration file

\* Annotate a Java Class:

Entity Class: Plain old java class that is mapped to the database table. With special Annotations on it



Two options for mapping a class:

1. XML Config file -🡪 More of legacy
2. Java Annotations 🡪 More Modern approach

Step1: Map the class to the database table

Step2: Map the fields to the Column in table

Save Java Object

SessionFactory Reads the hibernate Config file

Creates Session objects

Heavy Weighted object

Only create once in your app

Session Wraps a JDBC Connection

Main Object used to save/retrieve objects

Short-lived object

Retrieved from SessionFactory

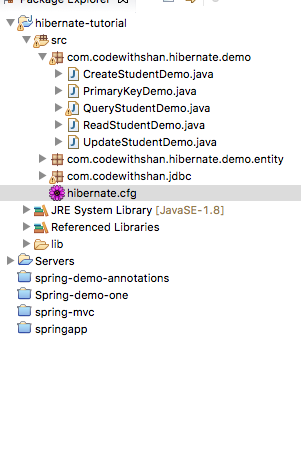
You can write your own generator strategy

Create a subclass of org.hibernate.id.SequenceGenerator

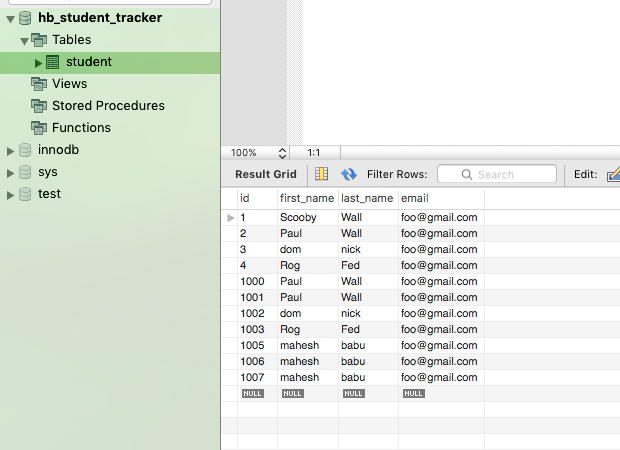
Override the method public Serializable generate(…)

Hibernate Query Language

* List<Student> theStudents = session.createQuery(“from Student”).getResultList();
* List<Student> theStudents = session.createQuery(“from Student s where s.lastName=’Doe’”).getResultList();



/users/Prashanth/documents/spring/workspace/hibernate-tutorial



Deleting the values in the database..

// delete the student

System.out.println("Deleting student: " + myStudent);

session.delete(myStudent);

// delete student id=2

System.***out***.println("Deleting student id=2");

session.createQuery("delete from Student where id=1001").executeUpdate();

Advanced Mappings:

* One-to-One
  + An instructor can have an instructor detail entity
  + Similar to an “instructor profile”
* One-to-Many
  + An instructor can have many courses
* Many-to-Many
  + A Course can have many students
  + A student can have many courses

Primary Key

* Identify a unique row in a table

Foreign Key

* Link tables together
* A field in one table links to another table field

Cascade Key

* You can cascade operations
* Apply the same operation to related entities

If we delete an instructor we should also delete the instructor\_detail

Cascade delete normally depends on the use case.

Should we do cascade delete here??

Developer can configure cascading

Fetch Types: Eager vs Lazy Loading

When we fetch/retrieve data, should we retrieve Everything?

* Eager will retrieve everything
* Lazy will retrieve data on request

Uni-Directional

Instructor 🡪 Instructor Detail

Bi-Directional

Instructor 🡨--🡪Instructor Details

**HIBERNATE ONE-ONE MAPPING:**

An instructor can have an instructor detail entity

Similar to an “instructor profile”

Uni-Directional for now.

Development process:

1. Define database tables
2. Create instructor detail class
3. Create Instructor class
4. Create Main App

Table: instructor\_detail

Table: instructor

Foreign Key

* Link Tables together
* A field in one table that refers to primary key in one table
* Main purpose is to maintain relationships between tables
* Referential Integrity
* Prevents operations that would destroy relationship
* Ensures only valid data is present in the table

Entity LifeCycle

Detach

Merge

Persist

Remove

Refresh

One-one 🡪 Cascade Type

* Persist
* Remove
* Refresh
* Detach
* Merge
* All

Detach will remove the relationship between the object and the entity

@OnetoOne(Cascade=CascadeType.ALL)

Configure Multiple Cascade Types:

@OnetoOne(Cascade=CascadeType.PERSIST)

Cascade Type is from the org.hibernate class

//Load on startup is kept 1

//The init method is read in servlet

JTA 🡪 java transaction Api

The transaction manager is done automatically inside the container

Even though it is a get operation the transaction manager takes care of it

J2ee has two types of transaction managers

1. User maintained transactions
2. Container maintained transaction

EIS 🡪 Enterprise Information System.

**BUILDING A DATABASE WEBAPP!**

Setup the Environment!

🡪 Testjdbc connection

🡪 Add hibernate and other jars in the lib folder

🡪 add spring-mvc-crud-demo-servlet.xml

🡪 add web.xml

spring-mvc-crud-demo-servlet.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"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<!-- Add support for component scanning -->

<context:component-scan base-package=*"com.codewithshan.springdemo"* />

<!-- Add support for conversion, formatting and validation support -->

<mvc:annotation-driven/>

<!-- Define Spring MVC view resolver -->

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/view/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

<!-- Step 1: Define Database DataSource / connection pool -->

<bean id=*"myDataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*

destroy-method=*"close"*>

<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"jdbcUrl"* value=*"jdbc:mysql://prashanth.cij7b4gpzzdo.us-east-1.rds.amazonaws.com:3306/web\_customer\_tracker?useSSL=false"* />

<property name=*"user"* value=*"prashanth"* />

<property name=*"password"* value=*"prashanth9"* />

<!-- these are connection pool properties for C3P0 -->

<property name=*"minPoolSize"* value=*"5"* />

<property name=*"maxPoolSize"* value=*"20"* />

<property name=*"maxIdleTime"* value=*"30000"* />

</bean>

<!-- Step 2: Setup Hibernate session factory -->

<bean id=*"sessionFactory"*

class=*"org.springframework.orm.hibernate5.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"myDataSource"* />

<property name=*"packagesToScan"* value=*"com.codewithshan.springdemo.entity"* />

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</prop>

<prop key=*"hibernate.show\_sql"*>true</prop>

</props>

</property>

</bean>

<!-- Step 3: Setup Hibernate transaction manager -->

<bean id=*"myTransactionManager"*

class=*"org.springframework.orm.hibernate5.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"*/>

</bean>

<!-- Step 4: Enable configuration of transactional behavior based on annotations -->

<tx:annotation-driven transaction-manager=*"myTransactionManager"* />

</beans>

list-customers.jsp

Web Browser -------------------customers/list------------------🡪Customer controller

|

|

|

| |

|

WebBrowse<----------------------------------------------------------- list-customers.jsp

1. Create Customer.java 🡪 Entity
2. Create CustomerDAO.java 🡪 DAO
3. CustomerDAOImpl.java 🡪 DAO
4. CustomerController.java 🡪 Controller
5. List-customers.jsp

Step1:

Define DAO interface

Step2:

Define DAO implementation

Step 3:

@Transactional magic is done by spring in the background

Spring handles transactions in the back

@Component

@Controller @Repository

Updates for the DAO implementation:

@Repository

@Autowired

@Transactional

Its okay if you fail. It is not okay if you ain’t trying.

To redirect:

<%response.sendRedirect("customer/list"); %>

New Annotations:

--------------------------------------------------------------------------------------------------------

@GetMapping

@PostMapping

Refactoring Service Layer

Customer Customer Service Customer DAO DB

Controller

* Service Façade design pattern
* Intermediate layer for custom business logic
* Integrate data from multiple sources (DAO/repositories)

Integrate Multiple Data Sources

Customer Controller Customer Service Customer DAO/SALES DAO DB

Specialized annotation for services

Component

Repository Service Controller

Spring provides the @Service annotation

* Added the customer button
* Updated the form

Updated the action in the header!

So now if we save it

!-- Construct a update link with customer id -->

<c:url var=*"updateLink"* value=*"/customer/showFormForUpdate"*>

<c:param name=*"customerId"* value=*"*${tempCustomer.id}*"*></c:param>

</c:url>

<td> ${tempCustomer.firstName} </td>

<td> ${tempCustomer.lastName} </td>

<td> ${tempCustomer.email} </td>

<td> <!-- update link to the action head -->

<a href= *"*${updateLink}*"*>Update</a>

</td>

</tr>

<http://localhost:8080/web-customer-tracker/customer/showFormForUpdate?customerId=9>