WFS Notes

1. Java 7 and Java 8
2. Spring
3. Spring Microservices
4. Angular Framework

Java has done major changes in Java 5, Java 7 and Java 8

Java 5 features

1. for each loop
2. Enums
3. Generics
4. Var-Args
5. Static Imports
6. Annotations

Enums: Fixed set of constants, so that at compile time itself you can verify the valid values, ex:

Gender must be male or female

Account must be Savings, Current, Recurring, Fixed Deposit

Departments must be Accounts, Sales, Finance,..

Gender.java



Employee.java



EmployeeDemo.java



Varying Arguments

It can accept 0 or more arguments

int sum(int... x)



Output:



Java 7 Features

1. Diamond Operator
2. Multi-Catch block
3. Resource Management with try-with resource syntax
4. Strings in Switch case
5. Underscores in numbers

Diamond Operator <>:

Before Java 7:

List<Integer> list = new ArrayList<Integer>();

From Java 7:

List<Integer> list = new ArrayList<>();

Multi-Catch Statements:

In one catch you can have one or more exceptions to handle

catch(ArithmeticException | NumberFormatException e) { }

Resource Management with try-with resource statement

This features automatically closes the resources like files, buffers, databases etc.

It automatically flushes the streams while performing write operations

Syntax:

try (resource instances; resource instances) {

}

Example:

try (fw = new FileWriter(“abc.txt”)) { }

This makes the Java to automatically perform close after performing the operations inside the try



Above code automatically closes the streams

Strings in Switch & Underscores in Numbers



Java 8 Features:

1. New Date & Time API’s
2. Static & Default methods inside interface
3. Functional Interface
4. Lambda Expressions
5. Stream API’s

New Date & Time API’s

Java introduced 3 main classes to work on Date, Time and DateTime

1. LocalDate
2. LocalTime
3. LocalDateTime

All these 3 classes have some static methods to create date & time instances

now(): to create current date or time instance

of(): to create specific date or time instance

parse(): to take a string and convert to date & time instance

LocalDate Demo



Output:



LocalTime & LocalDateTime also works the same way



Output:

Time: 11:45:17.188

Date & Time: 2020-10-05T11:45:17.189

Format Date & Time: 05/10/2020 T 11:45:17

Changes in the interface

You can have methods with body in the interface

1. default methods: it will have some default implementations but not necessary to override
2. static methods: you can call directly from the interface name

Lambda Expression:

It is a simplified form of implementing anonymous class, it has to be applied only on the interfaces having single abstract method (Functional Interface)



Lambda expressions are better when the statement is single line expression, but it also allows you to have multiline statements

() -> 10 // a method that returns an int

() -> “Hello” // a method that returns a String

(x, y) -> x + y // a method that returns addition of x & y

() -> { System.out.println(“...”); return 10; }

() -> System.out.println() // a method returns void

Note: return statement is required when a method has a return type if you write more than one line in lambda expression

Functional Interface:

It allows you to pass code directly to a method, it is an interface with only one abstract methods.



Exercise:



1st requirement: get all the employees added into the collection

2nd requirement: get only the top 3 employees sorted based on id, salary, dob

Java Streams:

Streams are collection of data that you want to operate while processing the collection, Java Streams processes the collection of data in a declarative way like SQL.

SQL statements are declarative statements they are easier to write, to select items, to filter items, to sum items

select \* from employee;

select \* from employee order by name;

select \* from employee order by name desc;

You can simplify sorting, filtering, iterating in the streams



Streams has many methods that can be chained that gives another stream and some methods also used at the end which is called as terminal operation.

There are two type of operations you can do on streams

1. Intermediate
   1. sorted
   2. filter
   3. distinct
   4. limit
   5. map
2. Terminal
   1. forEach
   2. count
   3. collect



Output:



filter: it is used to filter the data from a stream by applying some condition.

* get laptops of specific brand name
* get laptops of specific ram size



Spring Framework

Framework makes developers to develop complex applications in a simpler way, it will take care of lot repeating tasks like Exception Handling, Type Conversion, Design Patterns, Object creations, configurations and so on

Spring Framework is one of the java framework which is very popular because you develop many kinds of applications like standalone, web, mobile, microservices and so on.

Spring Framework or any other frameworks use lot of libraries to make the task simple hence you must download those libraries from the internet, which is why it’s recommended to use maven project which has a feature of downloading the libraries from the internet.

Spring provides many modules for different technologies

1. Spring Core : Fundamentals
2. Spring Web : Web applications
3. Spring Data JPA : Database purpose
4. Spring REST : Application Integration
5. Spring Boot : Simplifies spring application development

Spring Core: The basic unit of spring core is dependency injection.

Dependency Injection: It supplies dependency of an object to another object, so that you don’t have to create object or initialize the object in another code.

Spring Framework maintains all the objects in its container called Spring Context, it initializes the objects by looking at the xml configuration.

pom.xml



Identifier.java



Pan.java



Aadhar.java



OldApproach.java



SpringApproach.java



Spring container can be accessed through

1. BeanFactory
2. ApplicationContext

BeanFactory is the super type for ApplicationContext.

You can also initialize the object variables in the xml file.

Spring Container maintains all the objects and it supplies the dependencies to other objects.

Spring Core or IoC: Inversion of Control, it is a mechanism where object creation is inverted, like instead of code creating the object, framework creates the object i.e, container will maintain the object.

There are three ways you can achieve dependency injection

1. setter injection
2. constructor injection
3. @Autowired

setter injection is a way where objects are supplied by calling setter methods, i.e., variables of an object is initialized via setter this is done using <property> tag in the xml

constructor injection is a way where objects are supplied by calling constructor, i.e., variables of an objects is initialized via constructor

@Autowired is a way where objects are directly supplied to the variable without using constructor or setter

Setter Injection Demo

DBUtility.java



beans.xml



SpringTest.java



Output:



Constructor Injection Demo

DBUtility.java



beans.xml



SpringTest.java



Output:



Note: getConnection() should be treated as a method that establishes the connection on a particular database & its called usually in DAO layer

The above examples supply values to the object, but you can also supply dependent object to the object.

<property name = “variable” ref = “idName”>

<constructor-arg index = “...” ref = “idName”>

Supplying an object to another object

AccountDao.java



JdbcBackedDao.java



beans.xml



Note: Client program doesn’t need DbUtility instead it can take DAO instance & call the methods

SpringTest.java



Output:



Assignment:

1. Pass DBUtility through constructor injection
2. Implement another DAO like HibernateBackedDAO for the AccountDAO perform the required dependency injection the same way and call the createAccount() & getAccounts() method
3. Perform a dependency injection by adding another layer i.e., Service Layer (AccountService) and make framework to supply DAO instance to the service layer and from client code(main method) you call the methods of service layer

Hint: AccountService interface with

* 1. createAccount
  2. getAccounts
  3. getAccountsSortByName

Spring Annotation based configuration:

It simplifies configuration of spring features so that xml configurations can be reduced

XML based way  
<bean id = “” class = “”>

Annotation based way:

@Component  
public class A { }

@Component specifies spring to instantiate the class

There are other annotations you can use

@Repository, @Service, @Controller, @RestController and so on

All these annotations are part of @Component, but you can use them based on the layer type

@Repository can be used in the DAO

@Service can be used in the Service

@Controller can be used in the Controller

Supplying dependencies can also be done through annotations using @Autowired on top the variable

XML based way:

<bean ....>   
 <property name = “..” ref = “..”>

Annotation based way:

@Autowired

private AccountDao dao;

To configure spring to support annotations you must use one tag in the xml called <context:component-scan>

beans.xml



CustomerDao.java



CustomerDaoImpl.java



CustomerService.java



CustomerServiceImpl.java



SpringTest.java



@Autowired: it automatically injects the object based on the type of the variable,

*CustomerDao customerDao* variable has *@Autowired*, spring framework checks for the object that implements CustomerDao, but the condition is there must be only one object of that particular type, if found more than one, spring raises error UnsatisfiedDependencyException, to resolve you can use *@Qualifier* or you can configure spring to have only one object of that type

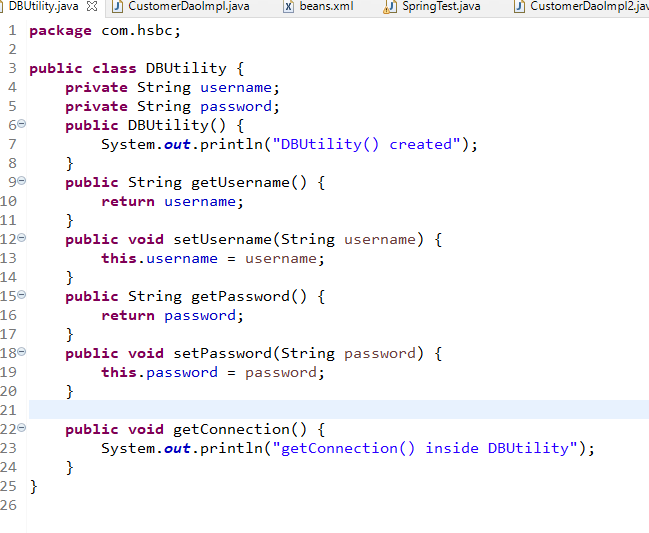
CustomerServiceImpl.java



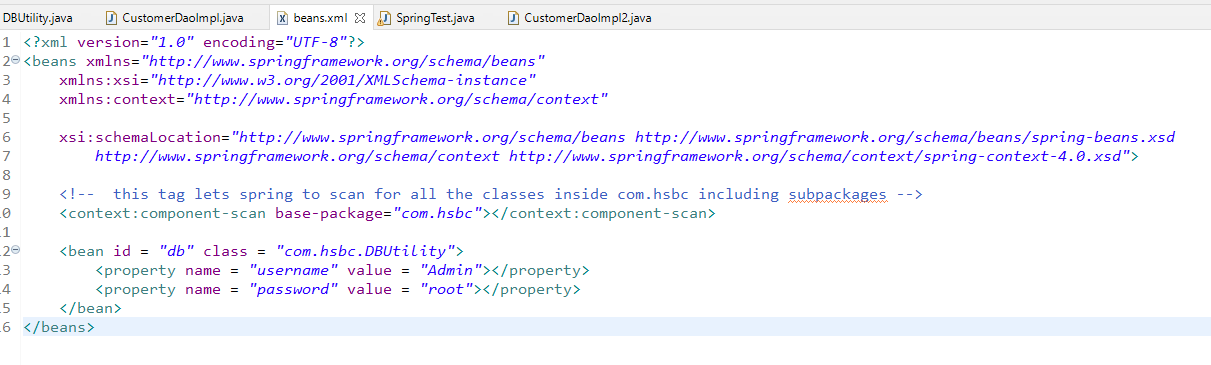
Since there are 2 Dao implementations we must use @Qualifier

Can we configure the class in the xml and inject the object using annotation

DBUtility.java

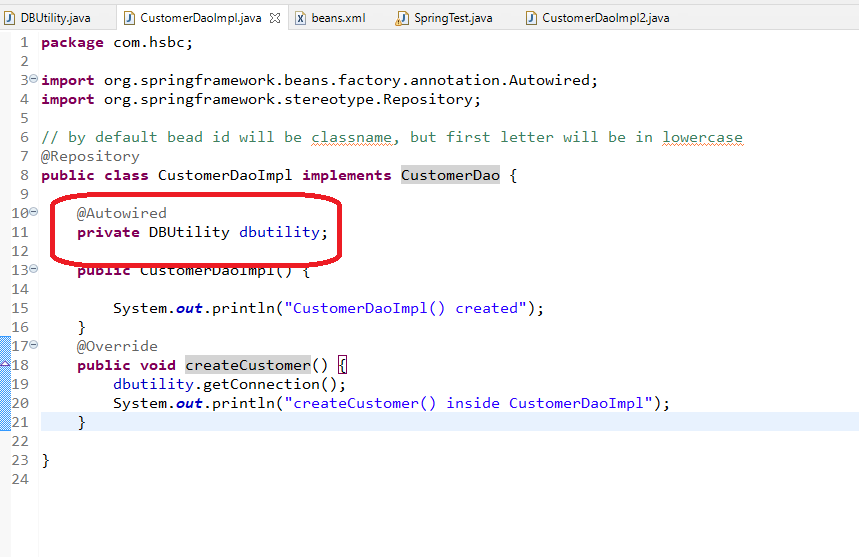


beans.xml



Note: When DBUtility has some properties you can initialize those properties in XML or suppose you don’t have DBUtility class with any stereo type annotations and it is provided by some third party vendors, then you can configure in the xml only

CustomerDaoImpl



Spring MVC Module

It allows you to develop web applications in a simplified way using spring features.

Spring MVC uses FrontController it can take all the requests coming to the web application & route to the appropriate controller, the front controller name is DispatcherServlet.

DispatcherServlet task:

1. it maps the request to the appropriate controller
2. it takes care of creating ApplicationContext
3. it takes care of mapping the response to appropriate view (JSP)

You must add spring mvc library to your project, it automatically adds dependent library i.e., spring-context

web.xml



DispatcherServlet is a front controller, which creates ApplicationContext object and specifies the xml file name should be *<servlet-name>-servlet.xml*

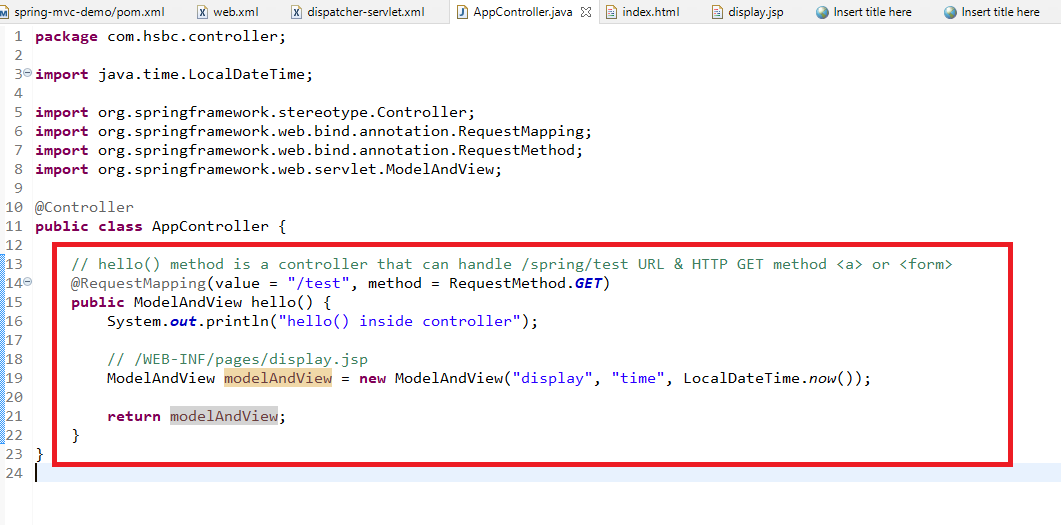
i.e., dispatcher-servlet.xml

dispatcher-servlet.xml



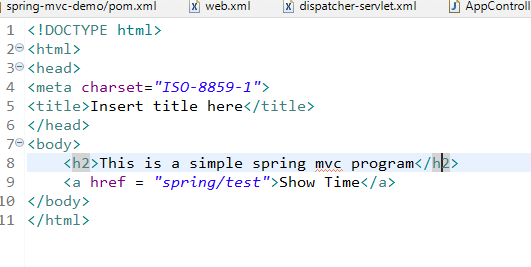
InternalResrouceViewResolver locates the view by adding prefix & suffix to the view name coming from the ModelAndView instance.

AppController.java



*@Controller* configures the class to be controller & its instantiated at the time component scan mentioned in the dispatcher-servlet.xml

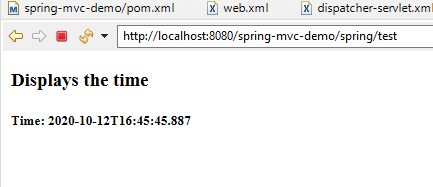
WebContent/index.html



WebContent/WEB-INF/pages/display.jsp



Output:



Implementing simple MVC program with Spring MVC

UserDao.java

**package** com.hsbc.model.dao;

**public** **interface** UserDao {

**public** String fetchUserById(**int** id);

}

UserDaoImpl.java

**package** com.hsbc.model.dao;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** String fetchUserById(**int** id) {

**if**(id == 100)

**return** "MS Dhoni";

**return** "Virat Kholi";

}

}

UserService.java

**package** com.hsbc.model.service;

**public** **interface** UserService {

**public** String fetchUser(**int** id);

}

UserServiceImpl.java

**package** com.hsbc.model.service;

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

**import** org.springframework.stereotype.Service;

**import** com.hsbc.model.dao.UserDao;

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** UserDao userDao;

@Override

**public** String fetchUser(**int** id) {

**return** userDao.fetchUserById(id);

}

}

AppController.java

**package** com.hsbc.controller;

**import** java.time.LocalDateTime;

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

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

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

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

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

**import** org.springframework.web.servlet.ModelAndView;

**import** com.hsbc.model.service.UserService;

@Controller

**public** **class** AppController {

@Autowired

**private** UserService service;

// hello() method is a controller that can handle /spring/test URL & HTTP GET method <a> or <form>

@RequestMapping(value = "/test", method = RequestMethod.***GET***)

**public** ModelAndView hello() {

System.***out***.println("hello() inside controller");

// /WEB-INF/pages/display.jsp

ModelAndView modelAndView = **new** ModelAndView("display", "time", LocalDateTime.*now*());

**return** modelAndView;

}

@RequestMapping(value = "/user", method = RequestMethod.***POST***)

**public** ModelAndView getUser(@RequestParam("userid") **int** id) {

String username = service.fetchUser(id);

ModelAndView modelAndView = **new** ModelAndView("demo", "name", username);

**return** modelAndView;

}

}

demo.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<h2>Hello ${name}</h2>

</body>

</html>

Output:

