31/01/18

Maven

- Many different application require the dependencies for their execution.

- To implement hibernate we need to download its own jar file or to implement h2 database we need to download it jar file etc.

- Similarly to implement spring based implementation we need to add the set of jar file.

- These jar files which are required for the application are called dependencies.

Disadvantages Manual Adding Dependencies

1. Version Conflict : When we do the manual addition of depedencies we will have the version conflict problem.

2. Error : Manual addition may arise error if any of the jar file will not be added.

- To avoid this manual adding of dependencies we are implementing built in tool.

- There are various built in tool for the building the application respective to the dependecies they are

1. Maven **2.Gradle**

- These two are the build tool by which we can make the automation of dependencies.

- Maven build tool runs using xml configuration and Gradle runs using json.

1. Maven

- All the configuration will be written using xml here the configuration is nothing but the dependencies which we need to add in our application.

- So when we create an application based on Maven then automatically there is xml file will be created which called pom.xml file.

- Here pom is Project Object Model.

Implementing Maven

- Maven based application uses a component called archetype.

- This is a templating toolkit for creating the project.

- An archetype is defined as an original pattern or model from which all other things of the same kind are made.

- These archetype are nothing but the project structure definition or project template.

- There are various archetypes which are supported by maven they are

1. Maven-archetype-webapp : An archetype to generate the web based project.

2. Maven-archetype-quickstart : An archetype to generate simple project

Group ID and Artifact ID

- When we define the maven based project we need to define the group id and artifact id.

Group ID : Here we can specify the package information.

ArtificatID : Here we can specify the project name which is unique in the workspace.

Note : When we are providing the package name we need to follow certain guide lines.

- A project is commercial domain then we can start with com.
- Then followed to that we can have the type company name com.bluechip
- Then followed by the module com.bluechip.controller, com.bluechip.model , com.bluechip.dao etc.
- Every package name should be small case.

Creating a Simple Maven Project

- Click on File
- Click on New
- Click Maven Project
- Select the Maven Archetype - Quickstart
- Click on Next
- Type the Group Id and Artifact Id
- Click Finish

Note : To create Maven project we need to be in java ee perspective.

pom.xml

- This file will contain the dependecies.
- Each dependency will have the following information.

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
</dependency>
```

It contains 1. Group ID 2. artifactId 3.version 4.scope

Basic pom.xml file without any Dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.niit</groupId>
 <artifactId>GadgetGuru</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>jar</packaging>

 <name>GadgetGuru</name>
 <url>http://maven.apache.org</url>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

 <dependencies>

 </dependencies>
</project>
```

- To implement the spring based implementation we need to add the few depdenecies

1. Spring Core

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.5.RELEASE</version>
</dependency>
```

2. Spring Context

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.5.RELEASE</version>
</dependency>
```

1/02/18

Creating a Simple Spring based Application

Step 1: Create a Maven Project of type Quickstart.

Step 2: Add the Spring dependency required for this project.

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.5.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
```

```
                <artifactId>spring-context</artifactId>
                <version>4.3.5.RELEASE</version>
        </dependency>
```

Step 3: Create a bean class which need to be stored in the Spring Container.

```
package com.niit;
public class MyBean
{
        public void display()
        {
                System.out.println("I am in Display Method-MyBean");
        }
}
```

Step 4: Create a Configuration Class which will create a Bean and put into Spring Container.

```
package com.niit;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class MyConfiguration
{
        @Bean("myBean")                 //<bean id="myBean" class="com.MyBean"/>
        public MyBean getMyBean()
        {
                System.out.println("==Get Bean Executed==");
                return new MyBean();
        }
}
```

i. @Bean : This particular annotation will enable us to store the instance in the spring container with the name provided. Here our bean name is "myBean" which can be refered and we can get this bean with this particular name.

ii. @Configuration : This annotation is used to load the class using AnnotationConfigApplicationContext.

Step 5: Create a simple Test Class and Test the Bean.

```
package com.niit;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MyBeanTest

{

	public static void main(String arg[])

	{

		AnnotationConfigApplicationContext context=new
AnnotationConfigApplicationContext();

		context.scan("com.niit");

		context.refresh();



		MyBean myBean=(MyBean)context.getBean("myBean");

		myBean.display();

	}

}
```

Creating Project Backend

Step 1: Create a Maven Quick start Project.

Step 2: Configure the java build path for jdk and compiler setting.

Step 3: In pom.xml file we need to add the various dependencies

1. Spring-Core 2.Spring-Context 3.Spring-ORM

4. Hibernate-Core 5.H2

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

	<modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.niit</groupId>
<artifactId>InteriorBackend</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>InteriorBackend</name>
<url>http://maven.apache.org</url>

<properties>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-core</artifactId>
		<version>4.3.5.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-orm</artifactId>
		<version>4.3.5.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-context</artifactId>
		<version>4.3.5.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.hibernate</groupId>
		<artifactId>hibernate-core</artifactId>
```

```
            <version>4.3.5.Final</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>1.4.193</version>
        </dependency>

    </dependencies>
</project>
```

2/02/18

Creating the SessionFactory Bean.

```
package com.niit.config;
import java.util.Properties;
import javax.sql.DataSource;

import org.hibernate.SessionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate4.HibernateTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.orm.hibernate4.LocalSessionFactoryBuilder;

import com.niit.model.*;
import com.niit.dao.*;

@Configuration
@ComponentScan("com.niit")
```

```
@EnableTransactionManagement
public class DBConfig
{
	public DataSource getH2DataSource()
	{
		DriverManagerDataSource dataSource=new DriverManagerDataSource();

		dataSource.setDriverClassName("org.h2.Driver");
		dataSource.setUrl("jdbc:h2:tcp://localhost/~/DT255");
		dataSource.setUsername("dteja");
		dataSource.setPassword("dteja");

		System.out.println("---Data Source Created---");
		return dataSource;
	}

	@Bean(name="sessionFactory")
	public SessionFactory getSessionFactory()
	{

		Properties hibernateProp=new Properties();

		hibernateProp.setProperty("hibernate.hbm2ddl.auto", "update");
		hibernateProp.put("hibernate.dialect","org.hibernate.dialect.H2Dialect");

		LocalSessionFactoryBuilder factoryBuilder=new
LocalSessionFactoryBuilder(getH2DataSource());
		factoryBuilder.addAnnotatedClass(Category.class);
		factoryBuilder.addProperties(hibernateProp);

		System.out.println("Creating SessionFactory Bean");
```

```
		return factoryBuilder.buildSessionFactory();
	}



	@Bean(name="categoryDAO")
	public CategoryDAO getCategoryDAO()
	{
		System.out.println("----DAO Implementation---");
		return new CategoryDAOImpl();
	}


	@Bean(name="txManager")
	public HibernateTransactionManager getTransactionManager(SessionFactory
sessionFactory)
	{
		System.out.println("---Transaction Manager----");
		return new HibernateTransactionManager(sessionFactory);
	}



}
```

Creating a Model Component : Category

- The model component are simple POJO (Plain Old Java Object).
- This model class can be created using simple java class.

Category.java

```
package com.niit.model;


import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table
public class Category
{
	@Id
	@GeneratedValue
	private int categoryId;

	private String categoryName;
	private String cateogryDesc;

	public int getCategoryId()
	{
		return categoryId;
	}
	public void setCategoryId(int categoryId)
	{
		this.categoryId = categoryId;
	}
	public String getCategoryName()
	{
		return categoryName;
	}
	public void setCategoryName(String categoryName)
	{
		this.categoryName = categoryName;
	}
	public String getCateogryDesc()
```

```
	{
		return cateogryDesc;
	}
	public void setCateogryDesc(String cateogryDesc)
	{
		this.cateogryDesc = cateogryDesc;
	}
}
```

@Entity : This is an annotation which specifies the class instance will be stored in an persistence location i.e in database.

@Table : This is an annotation which specifies that the table need to be created for that particular entity class.

@Id : This is the annotation which specifies that there will be a primary key need to be specified for the property.

@GenerateValue : This annotation is used to generate the values.

Create a DAO Component

- DAO - Data Access Object is the full form.

- This particular component will have all the functionality code to do the CRUD operations.

- CRUD - Create , Retrieve , Update , Delete

- This is also a simple java class where we write the method to do all these operations.

CategoryDAO.java

```
package com.niit.dao;
import com.niit.model.Category;
public interface CategoryDAO
{
	public boolean addCategory(Category category);
}
```

CategoryDAOImpl.java

```
package com.niit.dao;
import javax.transaction.Transactional;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.niit.model.Category;

@Repository("categoryDAO")
public class CategoryDAOImpl implements CategoryDAO
{

	@Autowired
	SessionFactory sessionFactory;



	@Transactional
	@Override
	public boolean addCategory(Category category)
	{
		sessionFactory.getCurrentSession().save(category);
		return false;
	}

}
```

Test Implementation

```
package com.niit.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.niit.dao.CategoryDAO;
import com.niit.model.Category;

public class DemoTest
{
	public static void main(String arg[])
	{
		AnnotationConfigApplicationContext context=new
AnnotationConfigApplicationContext();
		context.scan("com.niit");
		context.refresh();

		CategoryDAO categoryDAO=(CategoryDAO)context.getBean("categoryDAO");

		Category category=new Category();
		category.setCategoryName("Chimney");
		category.setCateogryDesc("Hindware Chimney-Kitchen Decor");

		categoryDAO.addCategory(category);

	}
}
```

5/02/18

Junit Testing

- Testing is the process of checking the functionality of an application to ensure it runs as per requirement.

- Unit testing is nothing but testing of a single entity at a given point of time the entity can be a class or a method.

- Unit testing plays a critical role in helping software company to deliver the quality product.

- Unit testing can be done in two different ways they are

    1. Manual Testing

    2. Automated Testing

1. Manual Vs Automated Testing

- Manual -

- The test cases done manually without any tool or framework support.

- Time consuming and tedius process.

- Huge investment of human resources.

- Less reliable , No customization of the testing process.

-Automated

- It will use the tool support and executing the test cases.

- It is fast compare to the manual testing

- Less human resources are used.

- More reliable , Tests can be customizable using the programs.

Junit

- Junit is unit testing framework for java programming langauge.

- Junit promotes the idea of First testing and then coding.

- Junit approaches for incremental coding that is we write little code then test it and we can increment the code.

- This process will reduce the debugging the huge code as we already know by which code the error has came.

Features of Junit

- It is a open source framework and used for writing and running tests.

- It provides variety of annotations to identify the test methods.

- It provides the test runners for running the tests.

Unit Case

- Unit test case is the part of the code which ensures that another part of the code works as expected.

- Any unit test case is characterized by a known input and expected result.

- JUnit testing framework can easily be integrated with the different IDE they are

1. Eclipse 2.NetBeans

- It can be integrated with the build tool like Maven and Ant.

Junit Maven Dependency

```
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
   <scope>test</scope>
</dependency>
```

Implemening Test Case

- To implement the test case we need to add a Junit test case class.

- There are various annotations which we are going to implement they are

@BeforeClass : This annotation we will implement to the method which should be executed before any other unit test cases will run and before the class.

- This method should be static and it has to execute before the class.

```
package com.niit.test;


import static org.junit.Assert.*;
```

```
import org.junit.BeforeClass;
import org.junit.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.niit.dao.CategoryDAO;

public class CategoryUnitTest
{
	static CategoryDAO categoryDAO;

	@BeforeClass
	public static void executeFirst()
	{
		AnnotationConfigApplicationContext context=new
AnnotationConfigApplicationContext();
		context.scan("com.niit");
		context.refresh();

		categoryDAO=(CategoryDAO)context.getBean("categoryDAO");
	}

	@Test
	public void addCategoryTest()
	{
		assertTrue("Problem",true);
	}

}
```

- Here in the method executeFirst() we need to create the CategoryDAO bean as we are going to use the bean in all the unit test case method.

- Here @Test is the annotation which is used for specify each test. So here we will be creating the number of test methods depending on the number of DAO methods.

@Test

```
public void addCategoryTest()
{
        Category category=new Category();
        category.setCategoryName("Sofa");
        category.setCateogryDesc("All the Sofa Types");
        assertTrue("Problem in Category Insertion",categoryDAO.addCategory(category));
}
```

- Here the categoryTest() method has assertTrue as it's assert method.

- assertTrue will return true if the expected output returns true and will return false and if the output returns false.

Syntax

```
assertTrue("Text if any Error or Unexpected Output", booleanValue);
```

CateogoryUnitTest

```
package com.niit.test;

import static org.junit.Assert.*;

import org.junit.BeforeClass;

import org.junit.Ignore;

import org.junit.Test;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;


import com.niit.dao.CategoryDAO;

import com.niit.model.Category;
```

```
public class CategoryUnitTest
{
	static CategoryDAO categoryDAO;

	@BeforeClass
	public static void executeFirst()
	{
		AnnotationConfigApplicationContext context=new
AnnotationConfigApplicationContext();
		context.scan("com.niit");
		context.refresh();

		categoryDAO=(CategoryDAO)context.getBean("categoryDAO");
	}

	@Ignore
	@Test
	public void addCategoryTest()
	{
		Category category=new Category();
		category.setCategoryName("Sofa");
		category.setCateogryDesc("All the Sofa Types");
		assertTrue("Problem in Category Insertion",categoryDAO.addCategory(category));
	}

	@Test
	public void getCategoryTest()
	{
		assertNotNull("Problem in get Category",categoryDAO.getCategory(2));
	}
```

```
}
```

CategoryDAO.java

```
package com.niit.dao;
import com.niit.model.Category;
public interface CategoryDAO
{
	public boolean addCategory(Category category);
	public Category getCategory(int categoryId);
}
```

CategoryDAOImp.java

```
package com.niit.dao;

import javax.transaction.Transactional;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;



import com.niit.model.Category;

@Repository("categoryDAO")
public class CategoryDAOImpl implements CategoryDAO
{

	@Autowired
	SessionFactory sessionFactory;
```

```
	@Transactional
	@Override
	public boolean addCategory(Category category)
	{
		try
		{
		sessionFactory.getCurrentSession().save(category);
		return true;
		}
		catch(Exception e)
		{
		System.out.println("Exception Arised:"+e);
		return false;
		}
	}

	//getCategory()
	@Override
	public Category getCategory(int categoryId)
	{
		Session session=sessionFactory.openSession();
		Category category=(Category)session.get(Category.class,categoryId);
		return category;
	}
		//deleteCategory()
		//updateCategory()
		//listCategories()

}
```