Orchard| Mindtree

UNIT & INTEGRATION TESTING JAVA

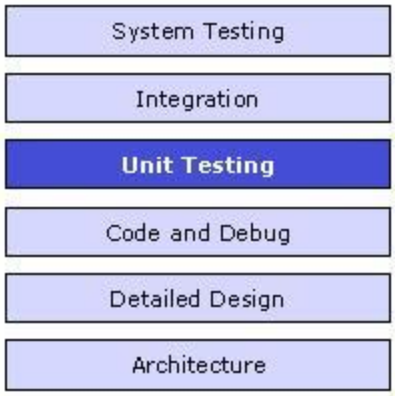
APRIL 5, 2018

by
Abhishek Pattanaik (M1027284)

Unit & Integration Testing Java

Unit Testing

- ➔ Unit tests are basically written and executed by software developers to make sure that code meets its design and requirements and behaves as expected.
- ➔ The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly.
- ➔ Unit testing is basically done before integration as shown in the image below.

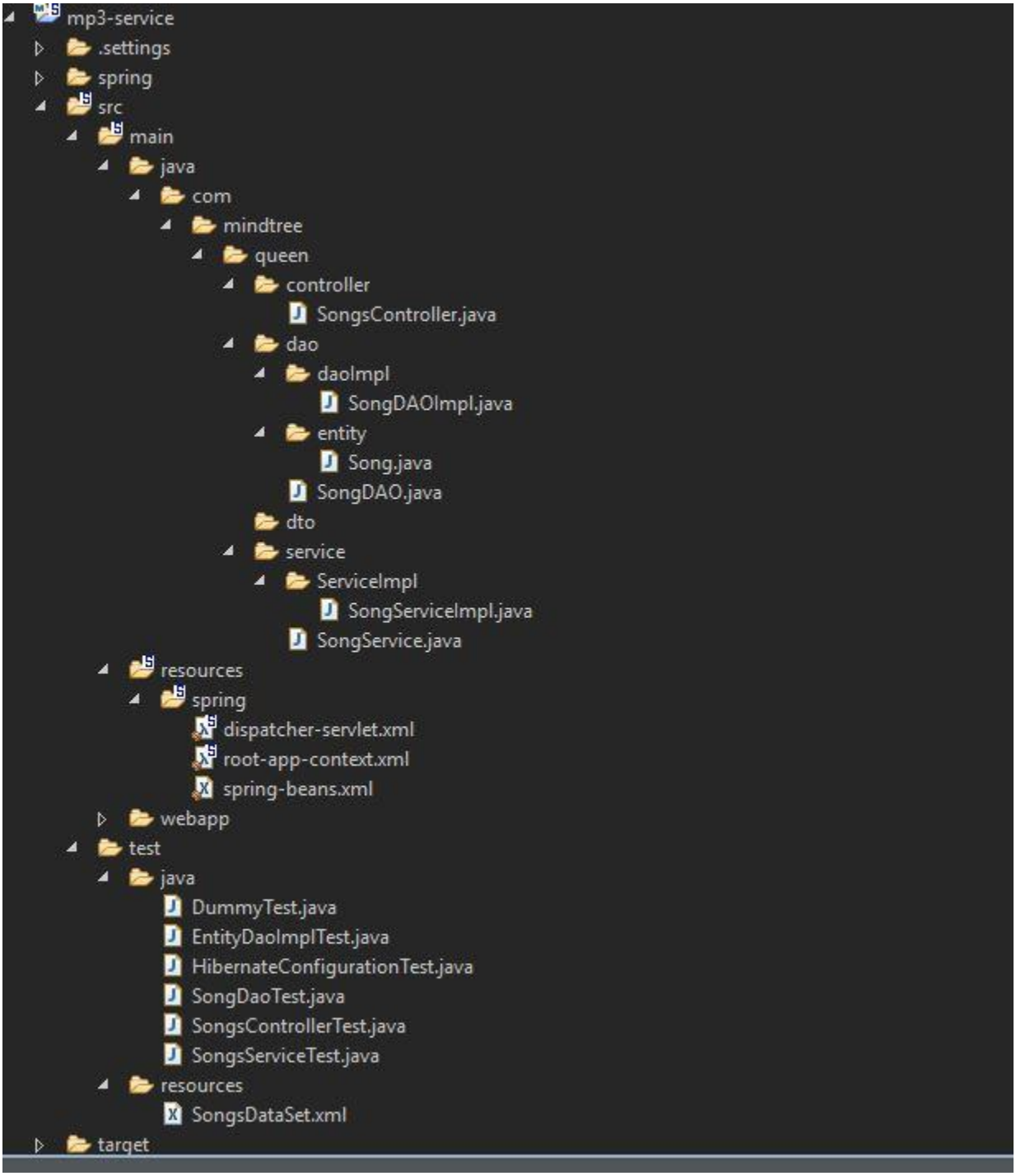


- ➔ Advantages of Unit Testing
  - The modular approach during Unit testing eliminates the dependency on other modules during testing.
  - We can test parts of a project without waiting for the other parts to be available.
  - Since the bugs are found early in unit testing hence it also helps in reducing the cost of bug fixes. Just imagine the cost of bug found during the later stages of development like during system testing or during acceptance testing.
  - Unit testing helps in simplifying the debugging process. If suppose a test fails then only latest changes made in code needs to be debugged.

❖ Code Examples

Problem Statement: - The application named as Queen MP3 aims to serve as repository of songs which can be accessed any time. Users would able to interact with database to *add, read, update and delete* the songs. Design a backend which will facilitate the user to perform the desired operations and should follow modular architecture such as service, data access object (DAO) and controller layers. The front end would be interacting with backend server through REST APIs.

Folder Structure – The folder structure image followed for this application



Service Layer – Below code snippets are present in Service layer package which interacts with DAO layer.

```java
@Service
public class SongServiceImpl implements SongService{

    @Autowired
    private SongDAO songDAO;

    public void setSongDAO(SongDAO songDAO) {
        this.songDAO = songDAO;
    }

    @Override
    @Transactional
    public String addSong(Song song) {
        return this.songDAO.addSong(song);

    }

    @Override
    @Transactional
    public String deleteSong(int songId) {
        return this.songDAO.deleteSong(songId);

    }
```

```java
    @Override
    @Transactional
    public List<Song> getALLSongs() {

        return songDAO.getALLSongs();
    }

    @Override
    @Transactional
    public Song getSong(int songId) {
        return songDAO.getSong(songId);
    }

    @Override
    @Transactional
    public Song updateSongDetail(Song song) {
        return this.songDAO.updateSongDetail(song);

    }
```

Let us try to achieve unit testing of *Service Layer* of an application with the help of **Mockito Framework, Spring-Test and TestNG framework** in the following code**.**

Imports

```
import static org.mockito.Mockito.when;
import static org.testng.AssertJUnit.assertEquals;

import java.util.ArrayList;
import java.util.List;

import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.Spy;
import org.mockito.runners.MockitoJUnitRunner;
import org.springframework.test.context.ContextConfiguration;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

import com.mindtree.queen.dao.SongDAO;
import com.mindtree.queen.dao.entity.Song;
import com.mindtree.queen.service.ServiceImpl.SongServiceImpl;
```

Note:-

1. Annotate with MockitoJUnitRunner if you want to run Mockito.
2. The XMLs are the spring framework config files.

Declarations

```
@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(locations = {
        "classpath:spring/dispatcher-servlet.xml",
        "classpath:spring/root-app-context.xml",
        "classpath:spring/spring-beans.xml" })
public class SongsServiceTest {
```

Global Declarations with Mockito and TestNG annotations

```
@Mock
private SongDAO songDao;

@InjectMocks
private SongServiceImpl songService;

@Spy
private List<Song> songs = new ArrayList<Song>();

@BeforeClass
public  void testSetUp() throws Exception {

    /*
     * MockitoAnnotations.initMocks(SongService.class);
     * MockitoAnnotations.initMocks(SongsController.class
     * MockitoAnnotations.initMocks(List.class);
     */
     MockitoAnnotations.initMocks(this);
     songs = getSongs();
     when(songDao.getALLSongs()).thenReturn(songs);

}
```

@BeforeClass – It belongs to TestNG annotations. It's get executed before any test cases method gets executed. So, it would be nice if we initialized the variables in this block.

Note: - We have to initialize some dummy data to our list of songs. Check ***getSongs***() at the end.

@Test -TestNG Annotations

Test cases for song update service

```
@Test
public void updateSongServiceTest() {
    Song s1 = new Song();
    s1.setId(3);
    s1.setName("Vande Mataram");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.google.com");
    s1.setArtists("Bankim Chandra Chatterjee");
    s1.setDuration("4:45");

    // updating songs list
    for(Song s : songs) {
        if(s.getId() == 3) {
            songs.set(songs.indexOf(s), s1);
        }
    }

    when(songDao.getSong(3)).thenReturn(songs.get(2));

    if(songDao.getSong(3).getId() == 3) {
        when(songDao.updateSongDetail(s1)).thenReturn(s1);
        assertEquals(songService.updateSongDetail(s1).getName(), songs.get(2).getName());
    }


}
```

when..then & verify are popular stubbing and verification techniques used in tests to define the behavior and then optionally verifying that behavior was indeed executed.

```
@Test
public void deleteSongServiceTest() {
    String deleteMsg = "Song 1 is deleted successfully";
    when(songDao.deleteSong(1)).thenReturn(deleteMsg);
    assertEquals(songService.deleteSong(1),deleteMsg);

}

@Test
public void addSongServiceTest() {
    Song s1 = new Song();
    s1.setId(4);
    s1.setName("Vande Mataram");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.google.com");
    s1.setArtists("Bankim Chandra Chatterjee");
    s1.setDuration("4:45");

    String addMsg = "Song "+s1.getId()+" is added successfully";

    when(songDao.addSong(s1)).thenReturn(addMsg);
    assertEquals(songService.addSong(s1),addMsg);

}
```

Test cases for deleting and adding a song service.

```java
@Test
public void getAllSongsServiceTest() {

    assertEquals(songService.getALLSongs().size(), 3);

}

@Test
public void getSongByIdServiceTest() {


    when(songDao.getSong(1)).thenReturn(songs.get(0));

    assertEquals(songService.getSong(1).getName(), "Bulleya");

}
```

Test cases for getting all songs and getting song by id.

```java
public static List<Song> getSongs() {
    List<Song> songs = new ArrayList<Song>();
    Song s1 = new Song();
    s1.setId(1);
    s1.setName("Bulleya");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.google.com");
    s1.setArtists("Amar, Vivek");
    s1.setDuration("3:45");
    songs.add(s1);
     s1 = new Song();
    s1.setId(2);
    s1.setName("Ae Zindagi");
    s1.setUploaderLocation("Bengaluru");
    s1.setUploaderName("Amar");
    s1.setUrl("www.youtube.com");
    s1.setArtists("Kavi");
    s1.setDuration("4:41");
    songs.add(s1);
     s1 = new Song();
    s1.setId(3);
    s1.setName("Dil Chori");
    s1.setUploaderLocation("Pune");
    s1.setUploaderName("Nitish");
    s1.setUrl("www.gaana.com");
    s1.setArtists("Aatish, Shruti");
    s1.setDuration("4:49");
    songs.add(s1);
    return songs;
}
```

Dummy data of songs

Integration Testing

Integration testing plays an important role in the application development cycle by verifying end-to-end behavior of the system.

1. **DAO Layer –** Below code snippets are present in DAO layer package which interacts with DAO layer.

```java
@Repository
public class SongDAOImpl implements SongDAO {

    private static final Logger logger = LoggerFactory.getLogger(SongDAOImpl.class);

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sf) {
        this.sessionFactory = sf;
    }

    public Session getSession() {
        return this.sessionFactory.getCurrentSession();
    }

    @Override
    public String addSong(Song song) {
        getSession().persist(song);
        return "Song " + song.getId() + " is added successfully";

    }

    @Override
    public String deleteSong(int songId) {
        Song song = getSession().load(Song.class, new Integer(songId));
        getSession().delete(song);
        return "Song " + songId + " is deleted successfully";

    }
```

```java
    @SuppressWarnings("unchecked")
    @Override
    public List<Song> getALLSongs() {
        List<Song> songs = getSession().createQuery("from Song").list();

        return songs;
    }

    @Override
    public Song getSong(int songId) {
        // return getSession().load(Song.class, new Integer(songId));
        return getSession().load(Song.class, new Integer(songId));
    }

    @Override
    public Song updateSongDetail(Song song) {

        Song song1 = null;

        getSession().update(song);
        song1 = getSong(song.getId());
        return song1;

    }
}
```

Let us try to achieve integration testing of *DAO Layer* of an application with the help of **Mockito Framework, Spring-Test and TestNG framework** in the following code**.**

If we think in terms of unit-test, then our goal becomes testing every line of DAO code while really mocking all the external systems/dependencies. Actually, we can't truly test a data-layer without really interacting with the database itself. And then it becomes an ***integration*** test.

We will perform integration-test on our DAO layer to make sure that it works as expected. We will be using ***in-memory H2 database*** to do our integration-tests. We are not directly interacting with our real time database because we do not want to insert data or perform any kind of manipulation into our actual database while doing unit testing. So, we will leverage the in-memory H2 database technology to mitigate the real time database overheads.

❖ **<u>Set up H2 Database configuration file in java or xml –</u>**

- It creates a *SessionFactory* using a *dataSource* which is configured to work with in-memory database H2. In order to make hibernate work with H2, we also need to specify the dialect being used [H2 Dialect].
- This *SessionFactory* will be injected in our *SongDao* interface class defined in the above code snippet. And from then on, the actual DAO implementation classes [*SongDaoImpl*] will use this *sessionFactory* when running tests against them.

```
import java.util.Properties;

import javax.sql.DataSource;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
```

```
@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.mindtree.queen.dao" })
public class HibernateConfigurationTest {
```

Mention your DAO and entity packages to scan the components.

```
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] { "com.mindtree.queen.dao.entity" });
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }

    @Bean(name = "dataSource")
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        properties.put("hibernate.hbm2ddl.auto", "create-drop");
        return properties;
    }
```

```java
    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory s) {
        HibernateTransactionManager txManager = new HibernateTransactionManager();
        txManager.setSessionFactory(s);
        return txManager;
    }
}
```

❖ **<u>Create *Base class* for DAO layer testing</u>**

- We will be using ***DBUnit*** to clean-insert sample data in test database[H2] before each test case execution, in order to prepare database before each Dao method execution. This way we make sure that the tests method do not interfere with each other.
- The ***HibernateConfigurationTest*** class which we create in the previous step will be used here. It is used in the ***@ContextConfiguration***.
- ***AbstractTransactionalTestNGSpringContextTests*** can (at some extent) be considered as *JUnit* equivalent of *RunWith*. This abstract class integrates Spring TestContext support in TestNG environment. It requires a class-level *@ContextConfiguration* in order to load ApplicationContext using XML configuration files or annotated *@Configuration* classes.

```java
import javax.sql.DataSource;

import org.dbunit.database.DatabaseDataSourceConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.operation.DatabaseOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests;
import org.testng.annotations.BeforeMethod;
```

```java
@ContextConfiguration(classes = { HibernateConfigurationTest.class })
public abstract class EntityDaoImplTest extends AbstractTransactionalTestNGSpringContextTests {

    @Autowired
    DataSource dataSource;

    @BeforeMethod
    public void setUp() throws Exception {
        IDatabaseConnection dbConn = new DatabaseDataSourceConnection(
                dataSource);
        DatabaseOperation.CLEAN_INSERT.execute(dbConn, getDataSet());
    }

    protected abstract IDataSet getDataSet() throws Exception;

}
```

- It also requires a datasource and a transactionManager to be defined in ApplicationContext in order to provide data-access support during testing. We have already defined both datasource & transactionManager in our @Configuration class.

Thanks to transaction support, by default a transaction will be started before each test, and then this transaction will be rolled back at the end of test. You may override the rollback behavior.

- Look at setup method annotated with *@BeforeMethod*. Method annotated with *@BeforeMethod* is called before each test, so it is an ideal place to do something which is required before each test. In our case, we want the in-memory database to be clean and predefined sample data to be inserted before each test. We will do it right here.

- Additionally, for ***DBUnit*** to connect to database in order to perform clean-insert, we have to provide a dataSource for it. That's why we declared a dataSource here, which will be autowired with dataSource defined in ***HibernateTestConfiguration*** class.

- As shown in above setUp method, firstly we create a connection to database using dataSource available (which will be test dataSource), and execute clean-insert on DB via DBUnit.

- Notice the abstract method ***getDataSet*** above. This method will be implemented in our tests classes in order to provide the actual test data to be inserted before each test.

Below is the content of input file used by DBUnit:
src/test/resources/SongsDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <song id="1" name="Meri Zindagi" artists="Amar, Vivek"  url="www.google.com"  album="" duration="3:45"  uploaderName="Abhishek" uploaderLocation="Kalinga"/>
  <song id="2" name="Ae Zindagii" artists="Kavi"  url="www.youtube.com"  album="" duration="4:41"  uploaderName="Amar" uploaderLocation="Bengaluru"/>
  <song id="3" name="Dil Chorii" artists="Aatish, Shruti"  url="www.gaana.com"  album="" duration="4:49"  uploaderName="" uploaderLocation="Pune"/>
</dataset>
```

❖ **<u>DAO layer testing utilizing H2 Database configuration file and extending base class created in the above step.</u>**

```
import static org.testng.AssertJUnit.assertEquals;

import java.util.ArrayList;
import java.util.List;

import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.hibernate.SessionFactory;
import org.mockito.Spy;
import org.springframework.beans.factory.annotation.Autowired;
import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

import com.mindtree.queen.dao.daoImpl.SongDAOImpl;
import com.mindtree.queen.dao.entity.Song;
```

Load the dataset as mentioned in the previous step.

```java
@Override
protected IDataSet getDataSet() throws Exception{
    IDataSet dataSet = new FlatXmlDataSet(this.getClass().getClassLoader().getResourceAsStream("SongsDataSet.xml"));
    return dataSet;
}
```

```java
public class SongDaoTest extends EntityDaoImplTest{

    @Autowired
    private SessionFactory sessionFactory;

    @Autowired
    private SongDAOImpl songDao;

    @Spy
    private static List<Song> songs = new ArrayList<Song>();

    @BeforeClass
    public  void testSetUp() throws Exception {

        songs = getSongs();
        songDao.setSessionFactory(sessionFactory);

    }
```

```java
    @Test
    public void getSongByIdDaoTest() {

        Assert.assertEquals(songDao.getSong(1).getName(),"Meri Zindagi");
    }

    @Test
    public void getAllSongsDaoTest() {


        assertEquals(songDao.getALLSongs().size(), 3);

    }

    @Test
    public void deleteSongDaoTest() {
        String deleteMsg = "Song 3 is deleted successfully";
        assertEquals(songDao.deleteSong(3),deleteMsg);
        assertEquals(songDao.getALLSongs().size(), 2);
    }
```

```java
@Test
public void updateSongDaoTest() {
    Song s1 = new Song();
    s1.setId(3);
    s1.setName("Vande Mataram");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.google.com");
    s1.setArtists("Bankim Chandra Chatterjee");
    s1.setDuration("4:45");


    assertEquals(songDao.updateSongDetail(s1).getName(), "Vande Mataram");


}
```

```java
@Test
public void addSongDaoTest() {
    Song s1 = new Song();
    //s1.setId(4);
    s1.setName("Ban Ja Tu Meri Rani");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.gaana.com");
    s1.setArtists("Rockstar");
    s1.setDuration("4:49");
    System.out.println("name ="+s1.getName());

    String addMsg = "Song "+4+" is added successfully";

    assertEquals(songDao.addSong(s1),addMsg);
    assertEquals(songDao.getALLSongs().size(),4);

}
```

Let's take ***addSongDaoTest*** test case and understand how things happening here.

1) Before any of the test from the classes (which are extending *EntityDaoImplTest*) starts executing, Spring will load the text context from the configuration classes associated with *@ContextConfiguration* annotation & create the beans instances defined in those classes, thanks to *AbstractTransactionalTestNGSpringContextTests*. This will happen only once.

2) During *@Bean* instance creation, Spring will create the *SessionFactory* Bean which will be injected with dataSource bean (as defined in *HibernateTestConfiguration* class) based on database & hibernate properties. Look at the following property

properties.put("hibernate.hbm2ddl.auto", "create-drop");

Thanks to this hbm2ddl property, when the *SessionFactory* is created, the schema related to our Model classes will be validated and exported to database. That means Song table will be created in H2 database.

3) Before the test start, *@BeforeMethod* will be called, which will instruct DBUnit to connect to database and perform clean-insert. It will insert 2 rows in Song table (look at SongsDataSet.xml content)

4) Now the actual test case addSongDaoTest is about to start. Just before execution start, a transaction will be started. Method itself will run within this transaction. Once method finished it's execution, transaction will be rolled back which is default setup. You can override this behavior by annotating the test method with *@Rollback*(true) annotation. It is defined in [org.springframework.test.annotation.Rollback]

4) Now the actual test case addSongDaoTest finally starts it's execution. It will call songDao.addSong(Song song); which in-turn will insert the one pre-defined Song object into H2 database using hibernate. This is the core logic of ***addSong*** method anyway. After this operation there will be total 3 rows in Song table in H2 database. We will assert it for success/failure. Test completes.

5) For next test case, again *@BeforeMethod* will be called which will delete everything from table and re-insert two rows as defined in SongsDataSet.xml. Story continues…

6) When all our tests are done, session will be closed and schema will be dropped.

2. **Controller Layer –** Below code snippets are present in Controller layer package which interacts with Controller layer.

```
@RestController
@RequestMapping(value="/song")
public class SongsController {
    @Autowired
    private SongService songService;



    @Qualifier(value="songService")
    public void setSongService(SongService songService) {
        this.songService = songService;
    }


    @RequestMapping(value="/", method = RequestMethod.POST)
    public String addSong(@RequestBody Song song) {

        return this.songService.addSong(song);

    }

    @RequestMapping(value="/all", method = RequestMethod.GET, produces={ "application/json"})
    public List<Song>  getAllSongs() {
        return this.songService.getALLSongs();
    }
    @RequestMapping(value="/{songId}", method = RequestMethod.GET,produces={ "application/json"})
    public Song getSong(@PathVariable("songId") int songId ) {
        return this.songService.getSong(songId);
    }
```

```java
    @RequestMapping(value="/{songId}", method = RequestMethod.DELETE)
    public String deleteSong(@PathVariable("songId") int songId) {
        return this.songService.deleteSong(songId);


    }

    @RequestMapping(value="/", method = RequestMethod.PUT)
    public Song updateSongDetail(@RequestBody Song song) {
        return this.songService.updateSongDetail(song);
    }

}
```

Let us try to achieve integration testing of *Controller Layer* of an application with the help of **Mockito Framework, Spring-Test and TestNG framework** in the following code**.**

❖ **<u>Configuring Mockito and MockMvc</u>**

The *MockMvc* is initialized using the *MockMvcBuilders*#standaloneSetup(...).build() method. Optionally we can add filters, interceptors or etc. using the .addFilter() or .addInterceptor() methods.

```java
import static org.hamcrest.Matchers.is;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.delete;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.testng.AssertJUnit.assertEquals;
import java.util.ArrayList;
import java.util.List;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.Spy;
import org.mockito.runners.MockitoJUnitRunner;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
import com.fasterxml.jackson.databind.ObjectMapper;
```

```java
@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(locations = {
        "classpath:spring/dispatcher-servlet.xml",
        "classpath:spring/root-app-context.xml",
        "classpath:spring/spring-beans.xml" })
public class SongsControllerTest {

    private  MockMvc mockMvc;

    @Mock
    @Autowired
    private SongService songService;

    @InjectMocks
    private SongsController songController;


    @Spy
    private static List<Song> songs = new ArrayList<Song>();

    @BeforeClass
    public void testSetUp() throws Exception {
         MockitoAnnotations.initMocks(this);

         mockMvc = MockMvcBuilders
                 .standaloneSetup(songController)
                 .build();
         songs = getSongs();
         when(songService.getALLSongs()).thenReturn(songs);


    }
```

- **<u>REST APIs unit tests</u>**

  Now that we have configured Mockito with Spring Test Framework, we can start writing our unit tests for our spring mvc rest service. The endpoints in the rest service represent common CRUD operations like GET, POST, PUT and DELETE, as such we are going to unit test each operation for successes and failures. Here is an overview of each HTTP method:

  - Unit Test HTTP GET getting all users.
  - Unit Test HTTP GET/PathVariable get a user by id.
  - Unit Test HTTP POST create a new user.
  - Unit Test HTTP PUT update an existing user.
  - Unit Test HTTP DELETE delete a user by id.
  - Unit Test HTTP HEADERS verify if headers are correctly set.

- **Code Snippets**

```java
/*
 * Convert Java object into JSON
 * This piece of code is used to write an object into JSON representation.
 */
public static String asJsonString(final Object obj) {
    try {
        return new ObjectMapper().writeValueAsString(obj);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Code to Convert Java Objects into Json. It is required for REST APIs request and response.

```java
    public static List<Song> getSongs() {
        List<Song> songs = new ArrayList<Song>();
        Song s1 = new Song();
        s1.setId(1);
        s1.setName("Bulleya");
        s1.setUploaderLocation("Kalinga");
        s1.setUploaderName("Abhishek");
        s1.setUrl("www.google.com");
        s1.setArtists("Amar, Vivek");
        s1.setDuration("3:45");
        songs.add(s1);
         s1 = new Song();
        s1.setId(2);
        s1.setName("Ae Zindagi");
        s1.setUploaderLocation("Bengaluru");
        s1.setUploaderName("Amar");
        s1.setUrl("www.youtube.com");
        s1.setArtists("Kavi");
        s1.setDuration("4:41");
        songs.add(s1);
         s1 = new Song();
        s1.setId(3);
        s1.setName("Dil Chori");
        s1.setUploaderLocation("Pune");
        s1.setUploaderName("Nitish");
        s1.setUrl("www.gaana.com");
        s1.setArtists("Aatish, Shruti");
        s1.setDuration("4:49");
        songs.add(s1);
        return songs;

    }

}
```

- **Create test data which'll be returned as a response in the rest service**

  HTTP GET Unit Test

  Configure mock object to return the test data when the getAllSongs() method of the SongService is invoked.

  Invoke an HTTP GET request to the ***/all*** URI.

  - Validate if the response is correct.
  - Verify that the HTTP status code is 200 (OK).
  - Verify that the content-type of the response is application/json.
  - Verify that the collection contains 2 items.
  - Verify that the id attribute of the first element equals to 1.
  - Verify that the name attribute of the first element equals to Bulleya.
  - Verify that the getAllSongs method of the SongService is invoked exactly once.
  - Verify that after the response, no more interactions are made to the SongService

```java
@Test
public void getAllSongsRestApiTesting() throws Exception {


    mockMvc.perform(get("/song/all")).andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    /* .andExpect(jsonPath("$", hasSize(2))); */

    // hasSize is from Hamcrest library

    //verify(songService, times(1)).getALLSongs();
    //verifyNoMoreInteractions(songService);

}

@Test
public void getSongRestApiTesting() throws Exception {
    Song song = songs.get(0);
    when(songService.getSong(1)).thenReturn(song);

    mockMvc.perform(get("/song/{songId}", 1))
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.id", is(1)))
            .andExpect(jsonPath("$.name", is("Bulleya")));


    //verify(songService, times(1)).getSong(1);
    //verifyNoMoreInteractions(songService);

}
```

- **<u>HTTP POST/DELETE/PUT Unit Test: add and delete Song</u>**

  <u>POST /song/</u>

  <u>DELETE/song/{songID}</u>

  <u>PUT/song/</u>

  Configure mocked responses for the SongService and create methods.

  - Invoke an HTTP POST request to the ***/song*** URI. Make sure the content-type is set to application/json. Convert the Song object to JSON and add it to the request.
  - Validate if the response is correct.
  - Verify that the HTTP status code is 200.
  - Verify that the location header is set with the path to the created resource.
  - Verify that the addSong() methods of the SongService are invoked exactly once.
  - Verify that after the response, no more interactions are made to the SongService

```java
@Test
public void deleteSongRestApiTesting() throws Exception {
    String deleteMsg = "Song 1 is deleted successfully";
    Song s1 = new Song();
    s1.setId(1);

    when(songService.getSong(1)).thenReturn(s1);

    when(songService.deleteSong(1)).thenReturn(deleteMsg);
    mockMvc = MockMvcBuilders.standaloneSetup(songController).build();
    mockMvc.perform(delete("/song/{songId}", 1))
    .andExpect(status().isOk())
    .andExpect(content().string(deleteMsg));
    verify(songService, times(1)).deleteSong(s1.getId());
//  .andExpect(jsonPath("{}", is("Song Vande Mataram added successfully")));



}
```

```java
@Test
public void addSongRestApiTesting() throws Exception {

    Song s1 = new Song();
    s1.setId(4);
    s1.setName("Vande Mataram");
    s1.setUploaderLocation("Kalinga");
    s1.setUploaderName("Abhishek");
    s1.setUrl("www.google.com");
    s1.setArtists("Bankim Chandra Chatterjee");
    s1.setDuration("4:45");

    when(songService.addSong(s1)).thenReturn("Song "+s1.getId()+" is added successfully");

    mockMvc.perform(post("/song/")
    .contentType(MediaType.APPLICATION_JSON)
    .content(asJsonString(s1)))
    .andExpect(status().isOk())
    .andExpect(content().string("Song "+s1.getId()+" is added successfully"));





}
```

```java
@Test
public void updateSongRestApiTesting() throws Exception {

    Song s1 = new Song();
    s1.setId(1);
    s1.setName("Ae Dil Hai Mushkil");

    when(songService.getSong(1)).thenReturn(s1);
    Song updatedSong = s1;

    System.out.println(this.songService.getSong(1).getName());
    when(songService.updateSongDetail(updatedSong)).thenReturn(updatedSong);
    mockMvc = MockMvcBuilders.standaloneSetup(songController).build();
    mockMvc.perform(put("/song/")
            .contentType(MediaType.APPLICATION_JSON)
            .content(asJsonString(s1)))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name", is("Ae Dil Hai Mushkil")));

    verify(songService, times(1)).updateSongDetail(s1) ;
//  .andExpect(jsonPath("{}", is("Song Vande Mataram added successfully")));



}
```

Outputs

- **Maven Test Ouptut**
  - Run Maven test on the project, you will see below output on the console

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running TestSuite
Configuring TestNG with: org.apache.maven.surefire.testng.conf.TestNG652Configurator@221a3fa4
log4j:WARN No appenders could be found for logger (org.springframework.test.context.BootstrapUtils).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Tests run: 16, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 15.732 sec

Results :

Tests run: 16, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- jacoco-maven-plugin:0.7.5.201505241946:report (post-unit-test) @ mp3-service ---
[INFO] Analyzed bundle 'Queen MP3' with 4 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 40.464 s
[INFO] Finished at: 2018-04-05T02:33:35+05:30
[INFO] Final Memory: 14M/90M
[INFO] ------------------------------------------------------------------------
```

- **Jacoco Test Report**
  - After Maven run test, go to this path /target/jacoco-ut/

.resources	4/4/2018 9:48 PM	File folder	
com.mindtree.queen.controller	4/4/2018 9:48 PM	File folder	
com.mindtree.queen.dao.daoImpl	4/4/2018 9:48 PM	File folder	
com.mindtree.queen.dao.entity	4/4/2018 9:48 PM	File folder	
com.mindtree.queen.service.ServiceImpl	4/4/2018 9:48 PM	File folder	
.sessions.html	4/5/2018 2:33 AM	Chrome HTML Do...	448 KB
index.html	4/5/2018 2:33 AM	Chrome HTML Do...	5 KB
jacoco.csv	4/5/2018 2:33 AM	Microsoft Excel C...	1 KB
jacoco.xml	4/5/2018 2:33 AM	XML File	21 KB

- Open index.html, you will see nice report analysis of your code as shown below:

Queen MP3 Sessions

Queen MP3

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mindtree.queen.dao.entity		90%		56%	6	27	5	46	0	19	0	1
com.mindtree.queen.dao.daoImpl		100%		n/a	0	9	0	17	0	9	0	1
com.mindtree.queen.controller		100%		n/a	0	7	0	8	0	7	0	1
com.mindtree.queen.service.ServiceImpl		100%		n/a	0	7	0	8	0	7	0	1
Total	13 of 276	95%	7 of 16	56%	6	50	5	79	0	42	0	4

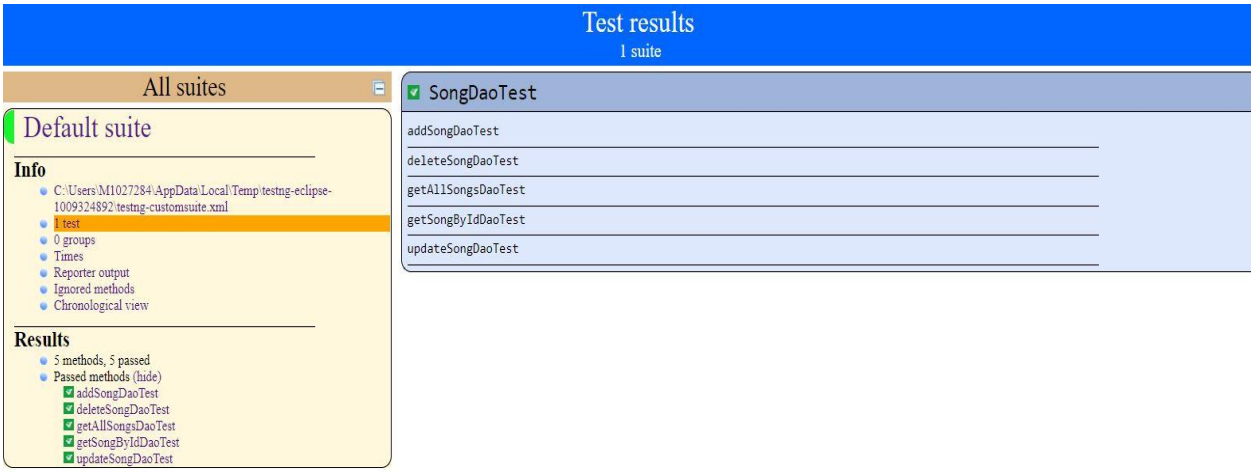
Created with JaCoCo 0.7.5.201505241946

❖ **TestNG Report**

- You can also run via TestNG on each test classes on right click, and you can check your console as shown below:-

```
log4j:WARN No appenders could be found for logger (org.springframework.test.context.BootstrapUtils).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
[TestNG] Running:
  C:\Users\M1027284\AppData\Local\Temp\testng-eclipse-1009324892\testng-customsuite.xml

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
PASSED: addSongDaoTest
PASSED: deleteSongDaoTest
PASSED: getAllSongsDaoTest
PASSED: getSongByIdDaoTest
PASSED: updateSongDaoTest

===============================================
    Default test
    Tests run: 5, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 5, Failures: 0, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@3a03464: 55 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@246ae04d: 116 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 0 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@735f7ae5: 178 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@6be46e8f: 15 ms
[TestNG] Time taken by org.testng.reporters.EmailableReporter2@6b143ee9: 7 ms
```

- Go to your test-output folder in the project directory and click the index.html



Miscellaneous

- To run the test cases – Run "**maven test**" or "**TestNG test**"

- **org.hamcrest:hamcrest** We use hamcrest for writing assertions on the response. We can use a variety of Matchers to validate if the response is what we expect.

- **org.springframework:spring-test** contains MockMvc and other test classes which we can use to perform and validate requests on a specific endpoint.

- **org.mockito:mockito-core** mocking framework for mocking data.

- **com.jayway.jsonpath:json-path-assert** Using jsonPath() we can access the response body assertions, to inspect a specific subset of the body. We can use hamcrest Matchers for asserting the value found at the JSON path.

- By annotating the **SongService** with the **@Mock** annotation, we can return mocked data when we call a method from this service. Using the @InjectMocks annotation, we can inject the mocked service inside our SongController. Before each test, we must initialize these mocks using the **MockitoAnnotations#initMocks(this)**.

- The **MockMvc** is initialized using the **MockMvcBuilders#standaloneSetup(...).build()** method. Optionally we can add filters, interceptors or etc. using the .addFilter() or .addInterceptor() methods.

- **Spring-test** : We will be using spring-test annotations in our test classes.
- **TestNG** : We will be using TestNG as our testing framework
- **Mockito** : We would be time to time doing some mocking, like mocking dao when testing service.
- **DBUnit** : We will use DBUnit to mange our data during data/dao layer testing
- **H2 Database** : For database layer, it's more of integration-test than unit-test. Unit tests does not bring real value while testing data layer. We will be using in-memory H2 database to do our integration-tests.

Maven Dependencies

Dependency	Detail
1. Spring-Test	<groupId>org.springframework</groupId> <artifactId>spring-test</artifactId> <version>4.3.0.RELEASE</version> <scope>test</scope>
2. TestNG	<groupId>org.testng</groupId> <artifactId>testng</artifactId> <version>6.9.4</version> <scope>test</scope>
3. Mockito	<groupId>org.mockito</groupId> <artifactId>mockito-all</artifactId> <version> 1.10.19</version> <scope>test</scope>
4. DBUnit	<groupId>dbunit</groupId> <artifactId>dbunit</artifactId> <version> 2.2</version> <scope>test</scope>
5. H2 Database	<groupId>com.h2database</groupId> <artifactId>h2</artifactId> <version>1.4.187</version> <scope>test</scope>
6. Hamcrest	<groupId>org.hamcrest</groupId>

	<artifactId>hamcrest-library</artifactId> <version>1.3</version> <scope>test</scope>
7. Json-Path-Assert	<groupId>com.jayway.jsonpath</groupId> <artifactId>json-path-assert</artifactId> <version>2.2.0</version> <scope>test</scope> <exclusions> <exclusion> <groupId>org.hamcrest</groupId> <artifactId>hamcrest-core</artifactId> </exclusion> <exclusion> <groupId>org.slf4j</groupId> <artifactId>slf4j-api</artifactId> </exclusion> </exclusions>
8. Jacoco	<plugin> <groupId>org.jacoco</groupId> <artifactId>jacoco-maven-plugin</artifactId> <version>0.7.5.201505241946</version> <executions> <execution> <id>prepare-agent</id> <goals> <goal>prepare-agent</goal> </goals> </execution> <execution> <id>report</id> <phase>prepare-package</phase> <goals> <goal>report</goal> </goals> </execution> <execution> <id>post-unit-test</id>

```
        <phase>test</phase>


       <goals>
        <goal>report</goal>
        </goals>
        <configuration>
<!-- Sets the path to the file which contains the
execution data. -->

       <dataFile>target/jacoco.exec</dataFile>

<!-- Sets the output directory for the code coverage
report. -->
        <outputDirectory>
         target/jacoco-ut
       </outputDirectory>
        </configuration>
     </execution>
     </executions>
<configuration>
    <systemPropertyVariables>
<jacoco-agent.destfile>target/jacoco.exec</jacoco-
agent.destfile>
    </systemPropertyVariables>
</configuration>
</plugin>
```

Explanation

In our example, we do it using **Mockito** framework. We provide mock of **SongService** by applying **@Mock** annotation on them.

- ➔ It's important to understand that **Mockito's** **@Mock** objects are not real instances, they are just bare-bones of instance created using Class of type. But their main capability is that they can remember all the interactions [operations performed] on them.

- **@Spy** objects are on the other hand real instances, but with additional capabilities of remembering all the interactions [operations performed] on them.

- **@InjectMocks** creates an instance of the class and injects the mocks that are created with the **@Mock/@Spy** objects in it.

- **MockitoAnnotations.initMocks(this);** initializes objects annotated with Mockito annotations **[@Mock, @Spy, @Captor, @InjectMocks]**

- Make sure to call **MockitoAnnotations.initMocks** when using Mockito annotations, else those mocks will be useless for your tests.

- Annotations **@Test & @BeforeClass are TestNG** specific annotations.

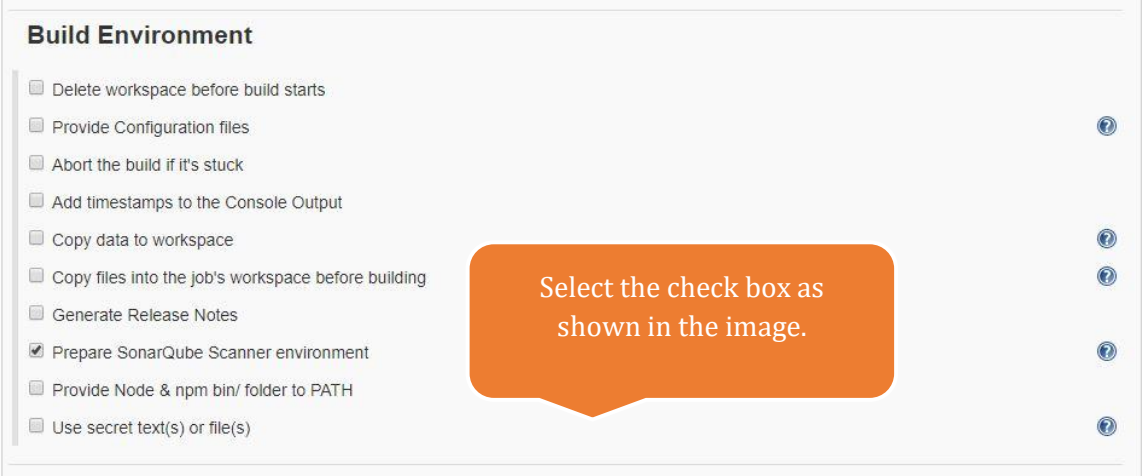
- Assert is the **TestNG** api for doing assertions on expected result and actual result.

- **when..then** & verify are popular stubbing and verification techniques used in tests to define the behavior and then optionally verifying that behavior was indeed executed.

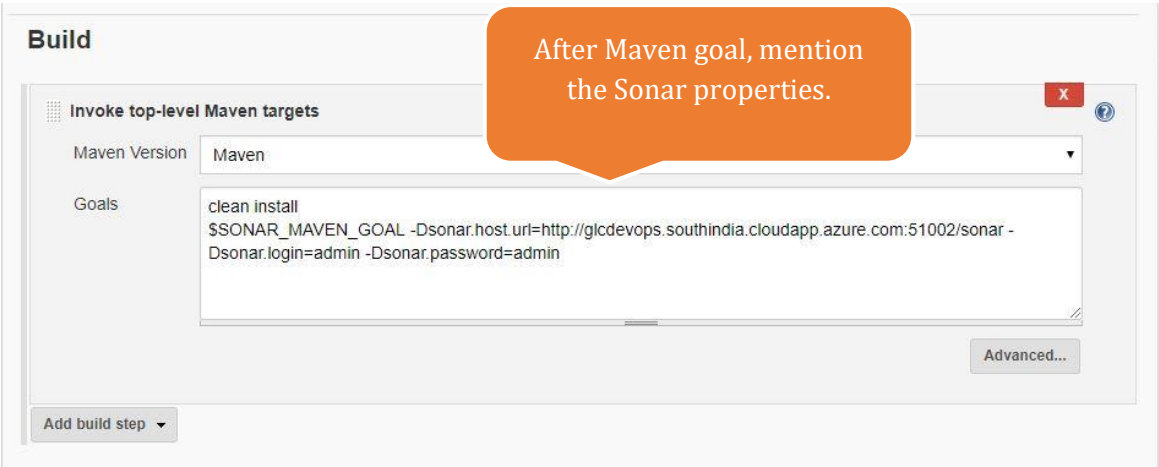
Jenkins

Let us try to configure Sonar with Jenkins to generate Java code coverage of the above application.

- Step 1



- Step 2



- Step 3

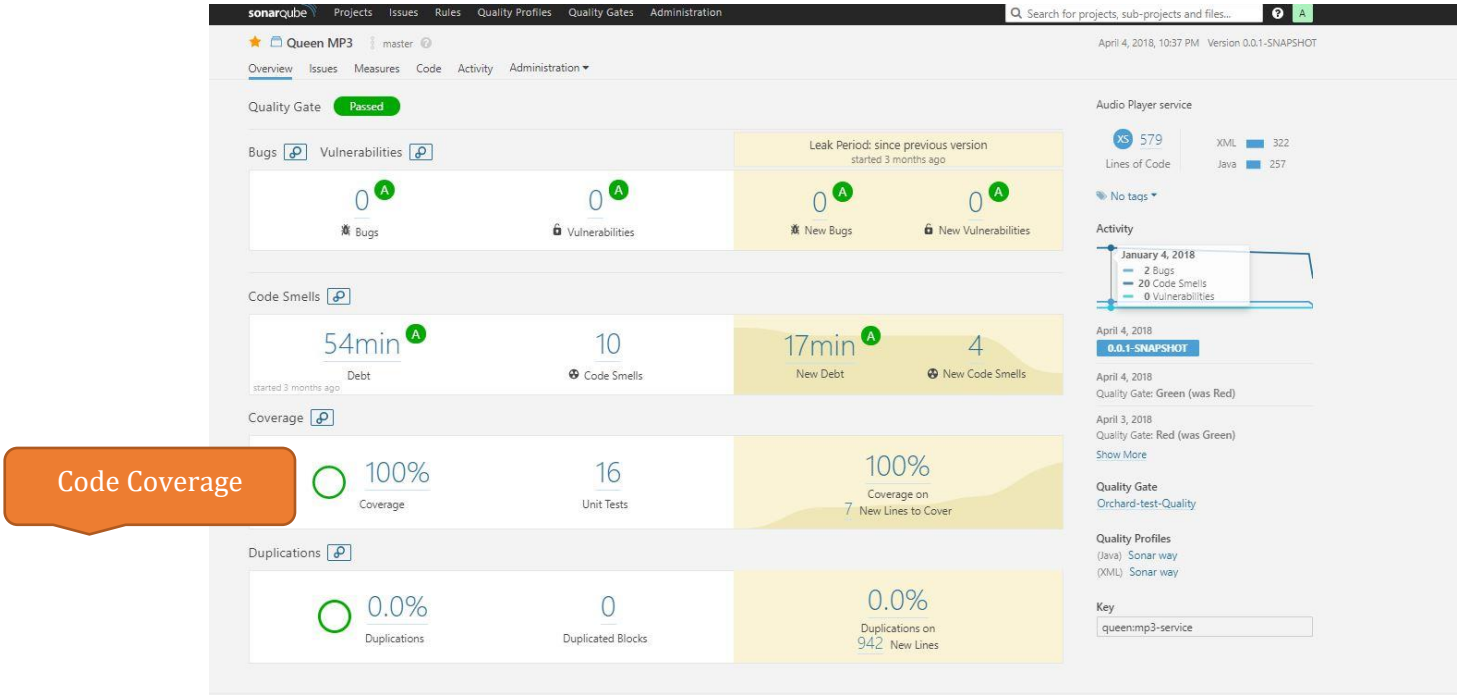
```
[INFO] Sensor CPD Block Indexer (done) | time=39ms
[INFO] SCM provider for this project is: git
[INFO] 2 files to be analyzed
[INFO] 2/2 files analyzed
[INFO] 2 files had no CPD blocks
[INFO] Calculating CPD for 4 files
[INFO] CPD calculation finished
[INFO] Analysis report generated in 124ms, dir size=122 KB
[INFO] Analysis reports compressed in 45ms, zip size=52 KB
[INFO] Analysis report uploaded in 63ms
[INFO] ANALYSIS SUCCESSFUL, you can browse http://glcdevops.southindia.cloudapp.azure.com:51002/sonar/dashboard/index/queen:mp3-service
[INFO] Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
[INFO] More about the report processing at http://glcdevops.southindia.cloudapp.azure.com:51002/sonar/api/ce/task?id=AWKRoG3_ha6pYpe6e0wD
[INFO] Task total time: 7.846 s
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Click Build Now→ Go to console output of the project and if the test cases are passed, you will get a link to check report on Sonar.

Sonar

Finally, we would be able to see the Sonar report of our application. Note we are able to get code coverage of 100%.

- **Sonar Report**



- **Exclusions of files**

If you want to exclude the files which does not need code coverage, you can achieve by adding below lines in the pom.xml

GitLab Link

git@glcgitlab.southindia.cloudapp.azure.com:playground/queen-player-service-codebase.git