XSEMBLE 10Xofy

# Test Bundle User Guide

Corresponding to Xsemble 2.1

Commercial in Confidence

**10Xofy**,

Pune, India

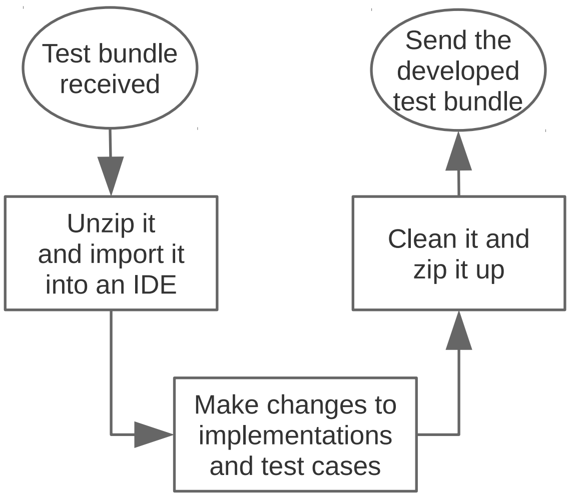
April 26, 2019

# Table of Contents

# 1 Introduction

This guide is for the programmers who work with Xsemble test bundles. A test bundle comes in as a zip file, containing a standalone Java J2EE web application. A test bundle corresponds to one or more modules (also called as software molecules); containing the module implementation code and unit test cases for each of them.

The following figure shows a flowchart of how a programmer works with a test bundle.


Test bundle received
Unzip it and import it into an IDE
Make changes to implementations and test cases
Clean it and zip it up
Send the developed test bundle

During the SDLC (Software Development Life Cycle), test bundles are used for the following purposes:

1. **First time development:** A test bundle is created with skeleton code for the module(s). The programmer is expected to complete the code by inserting the implementaiton code into the skeleton. The programmer needs to test the code by creating test cases. (Again, the skeleton of the test cases is included in the test bundle.) Once the development is complete, the programmer returns the test bundle along with the test cases for integrating with the application.
2. **Changing the Implementation:** The programmer receives the test bundle with the existing implementation. The programmer can make the changes to the module implementation as required and then run the test cases. If any of the old test cases break, then it needs to be re-checked whether the code changes have broken previously working functionality. The programmer edits existing test cases/ creates new ones based on the changed behaviour before returning the test bundle in such a way that all test cases should pass.
3. **Validation of Changes:** At any time during developemnt, when the code of a module is changed, the test cases can be run to ensure that old use cases are not broken.
4. **Validation of Codebase:** Running the test cases just to ensure that the codebase is in good condition.

Out of these 4, a programmer is needed for the first 2. The validations (last 2) are generally done on existing test cases, and that activity does not need any programming expertise. The persons doing so may jump from deploying the test bundle (section 3) to running test reports (section 4.3).

To create the modules and the test cases, a programmer needs to have a good understanding of Java JEE and unit testing frameworks, specifically JUnit, HttpUnit and Mockito; apart from any application specific frameworks/patterns used.

# 2 Concepts

There are three kinds of modules:

1. Method module
2. Page module
3. Entry Point module

They are explained below.

## 2.1 Method Module

A Method module is very similar to a method used in a program (some times called subroutines or functions). However, a Method module has zero or more input arguments, zero or more output arguments and one exit path. Use of spaces is allowed in these arguments and exit path. (On these points a Method module differs from a usual method).

The exit path is basically a string chosen from possible choices created at the time of definition of the Method (before the test bundle was created). The ability to define more than one exit paths makes a Method act like a decision box in a flowchart -- during Method execution, which path to take is decided.

## 2.2 Entry Point Module

An entry point, as its name suggests, is the point of entry to the server. Whenever a request hits the server, there is an entry point whose primary purpose is to translate the values received from the web request into a server side variable values. Typically, these requests are HTTP requests, but in some cases they are different, such as Websockets.

Like a Method module, an entry point also has input and output arguments, and exit paths. The output arguments are the server side variables, and exit paths are possibly differentiated based on the values received with the requests.

## 2.3 Page Module

A Page represents a user interface that is shown on the browser. The Page has input arguments, and exit paths. In case of a Page, the exit path typically corresponds to which link or button the user clicks. These exit paths would hit different urls which are not defined at the time of coding, and therefore the Pages make use of a utility that returns these urls. The Pages could use these URLs typically over HTTP but in some cases they could be something else, like Websockets.

The skeleton code generated for each of the modules makes the programmer's job easy. The code comes with helpful comments. Such skeleton code is generated for the test cases too.

**IMPORTANT: You should not change the definition of a module (the in/out arguments and exit path) within a test bundle, no matter how good a reason you have. Go to the test bundle creator and get her to recreate the test bundle with the changed definition.**

# 3 Test Bundle Setup

A Test Bundle is a runnable Java JEE application out of the box. It uses gradle as the build mechanism. We can use a Java JEE web container like Apache Tomcat or Jetty to deploy and run it. Below we shall use Apache Tomcat.

Assuming that you already have a test bundle, below we outline the steps to work with it.

## 3.1 Prerequisites – Java, Gradle, Tomcat and Eclipse

You need these on your system. While the detailed installation is out of scope of this document, here are some guidelines.

1. **Java**: Install JDK 1.8. We prefer that the Oracle JDK be downloaded and installed, but even OpenJDK could work. Set JAVA_HOME environmental variable and add its 'bin' to the system PATH. Starting link: http://www.oracle.com/technetwork/java/javase/downloads/
2. **Gradle**: It is a good idea to download gradle and install it on your machine. In our experience, gradle wrapper is slow, so do not use it. Put the bin folder in the system PATH. Starting link: https://gradle.org/install/
3. **Tomcat**: Download tomcat archive and extract it to a folder. It's easy to run the startup and shutdown scripts from its bin folder for starting and stopping tomcat. You may also use "catalina jpda start" (or "catalina.sh jpda start" on *nix systems) for starting tomcat in debugging mode. Starting link: http://tomcat.apache.org
4. **Eclipse**: Download Eclipse IDE for Java EE, so that most required plugins are already available. You will still need to add a couple of plugins. From the Eclipse marketplace (use menu Help -> Eclipse Marketplace after starting Eclipse), install "Buildship Gradle Integration" plugin. It's a good idea to add SonarLint plugin too, so that we can get immediate feedback on the code we write. JUnit is expected to be available with Eclipse EE, but you will need to install it if not. Starting link: http://eclipse.org

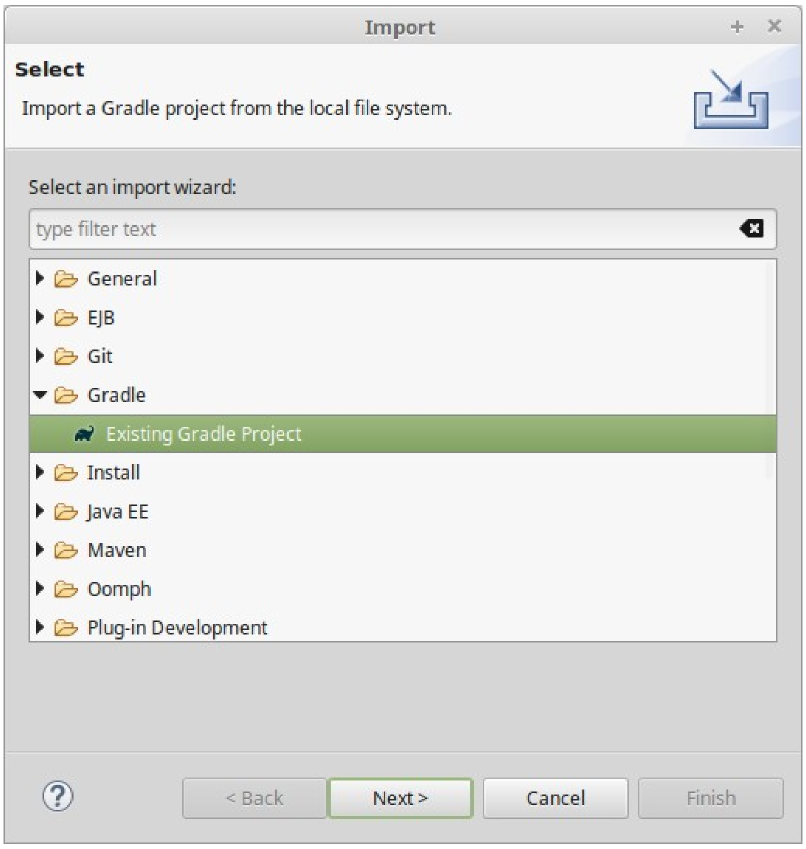
## 3.2 Unzip the Test Bundle

Unzipping the test bundle is trivial. On most systems, you should be able to browse to the folder and double-click on the archive and extract it. Extract it to some folder where you keep your source code.

## 3.3 Opening the Test Bundle in an IDE

As a gradle project, the test bundle may be opened in a Java IDE. Here we detail out the steps to open it in Eclipse, which is the most popular Java IDE.

In Eclipse, use File -> Import to open the Import dialog. In that dialog box, click on Gradle option which will open suboption named as "Existing Gradle Project". Click on that option.



Import

Select

Import a Gradle project from the local file system.

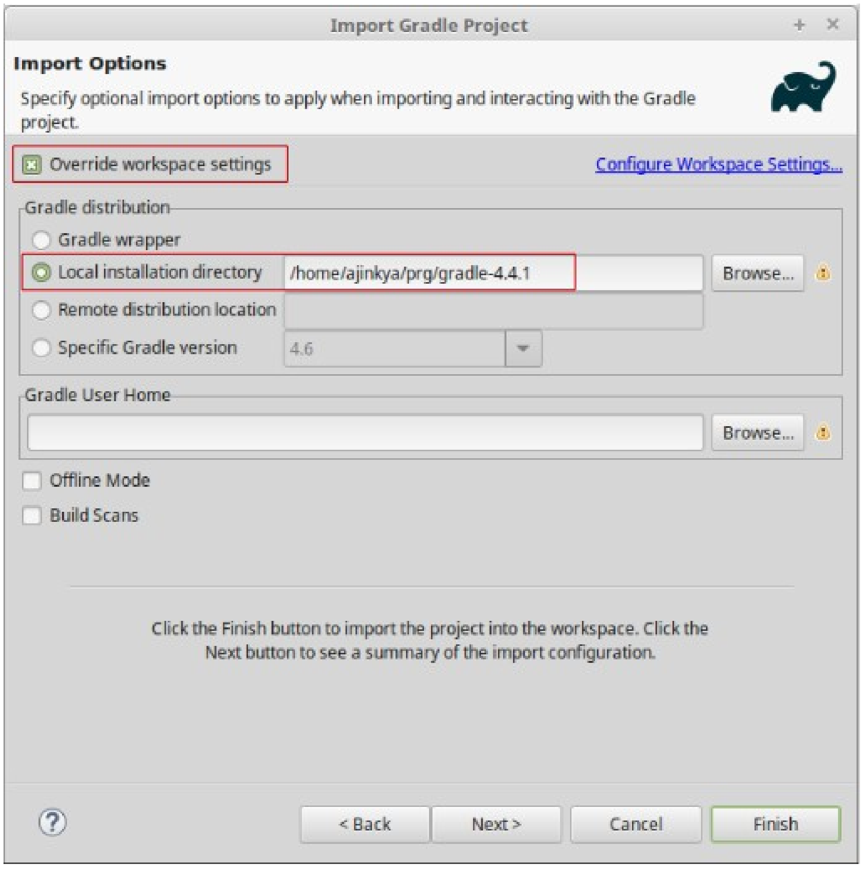
Select an import wizard:

type filter text

- General
- EJB
- Git
- Gradle
  - Existing Gradle Project
- Install
- Java EE
- Maven
- Oomph
- Plug-in Development

In the next screen, select root directory as the directory where you unzipped the test bundle and import the project. Eclipse should have no trouble in realizing this as a gradle project, as the gradle build file gradle.build would already be present in the folder. (Your test bundle is not proper if it is not present.)

Then you will need to choose gradle import options. If you have followed our gradle deployment instructions, then you would have gradle installed on your machine. Let’s use it as shown in the following dialog box.


Import Gradle Project

**Import Options**

Specify optional import options to apply when importing and interacting with the Gradle project.

[✓] Override workspace settings

Configure Workspace Settings...

Gradle distribution
- Gradle wrapper
- [✓ Local installation directory] /home/ajinkya/prg/gradle-4.4.1
- Remote distribution location
- Specific Gradle version 4.6

Gradle User Home

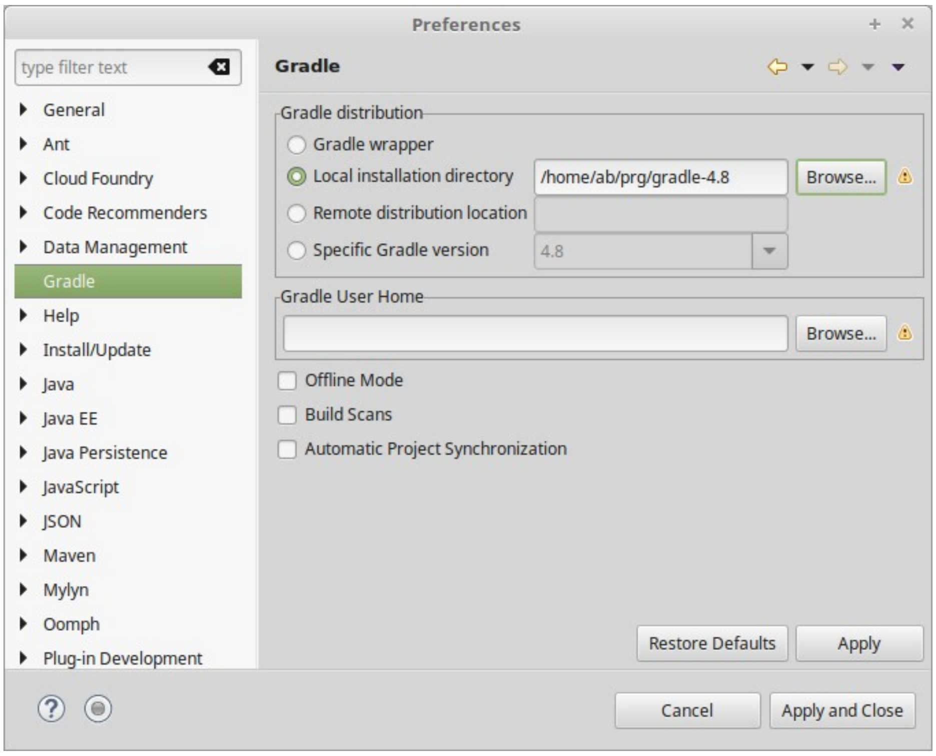
[ ] Offline Mode

[ ] Build Scans

Click the Finish button to import the project into the workspace. Click the Next button to see a summary of the import configuration.

Click Finish. Now the test bundle should be opened in Eclipse.

Note that overriding the default gradle settings every time is painful, so it is a good idea to have it configured as the default. You can do that through the preferences dialog box which can be opened up through Window -> Preferences and then selecting Gradle in the left pane and adjusting the settings, as shown below. Click “Apply” button so that the new settings are applied.


Preferences

**Gradle**

Gradle distribution
- Gradle wrapper
- [✓ Local installation directory] /home/ab/prg/gradle-4.8
- Remote distribution location
- Specific Gradle version 4.8

Gradle User Home

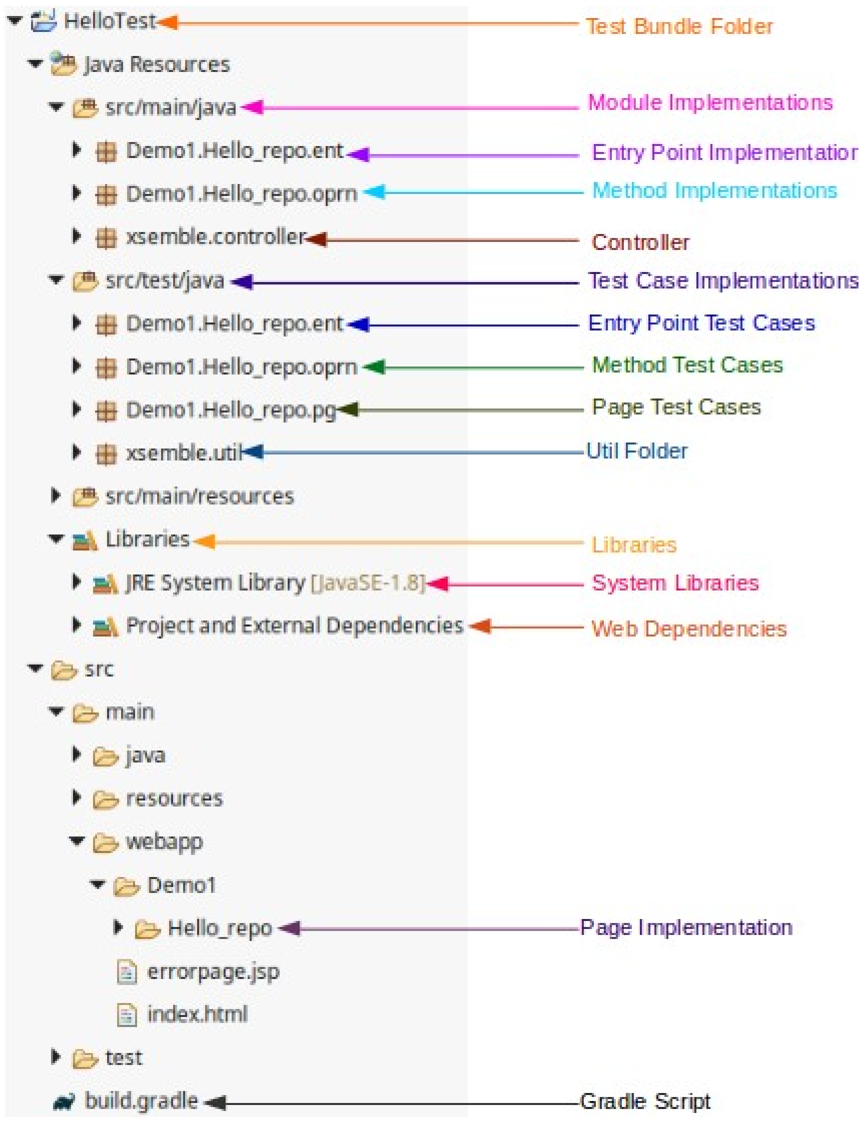
[ ] Offline Mode

[ ] Build Scans

[ ] Automatic Project Synchronization

## 3.4 Examining the Folder Structure

Since the test bundle is opened in Eclipse, it is easy to check out the folder structure of the test bundle. Those who are familar with the maven conventions will find themselves at ease, as those are the conventions followed. Please see the following figure.



### 3.4.1 Module Implementations

Here is where you find the module implementations.

1. **Method**: The implementation of a Method module is a Java class that goes inside a package ending with “oprn”. In the screenshot, under src/main/java, there is a package Demo1.Hello_repo.oprn. You will find the Java classes implementing the Methods modules in this package.
2. **Entry Point**: The Entry Point is implemented as Java class that goes inside a package ending with “ent”.
3. **Page**: Most page implementations are jsp files. In the screenshot, look at the folder “Demo1” under src/main/webapp folder. This folder contains all these JSP page implementations. In some other cases, pages are implemented as Java classes. In those cases, they are found under a package ending with “page”.

### 3.4.2 Test Cases

The test cases are found under Java packages under src/test/java as per the standard maven convention. A test case is typically under the same package as the module implementation, and it would be obvious to find test cases corresponding to a module from the names of the packages.

### 3.4.3 Module Dependencies

The module implementations may have dependencies on some other files.

1. **Types**: They are Java classes (actually Javabeans) that represent any custom types, found under the package ending with “type”.
2. **Jar libraries**: The jar files that are needed as dependencies are found under a folder “lib”, right under the project folder.
3. **Java classes**: The implementations may be calling some other Java classes. Like any standard Java project, these classes would be under some Java packages. By convention, packages starting with xsemble are reserved for supporting files and hence are not used as the locations of such files.
4. **Web dependencies**: The web pages may depend on some other files such as css, javascript, image files. They are typically under some subfolders of src/main/webapp.

### 3.4.4 Supporting Files

There are some supporting files that are present in the test bundle. In most cases, leave them as they are.

1. **Supporting JSP Files**: You will see index.html and errorpage.jsp under src/main/webapps folder. Leave them as they are. Index.html is loaded by default when the test bundle is deployed as the web application and it further redirects to the appropriate url.
2. **Supporting Utilities**: The packages xsemble.controller and xsemble.util contains the supporting controller code and the utility files respectively. You will largely leave these files as they are. They are needed for the execution of test bundle, but are not a part of any module or the test cases.

## *3.5 Deploying the Test Bundle in Web Container*

Deployment of the test bundle to a web container is needed only for testing Page modules. It is not needed for Method or Entry Point modules.

Below we shall see two ways in which it can be done.

### 3.5.1 Alternative 1: Direct Deployment in Eclipse

As long as you have Apache Tomcat (or Jetty or some other Java JEE web container) configured inside Eclipse, you may deploy this test bundle to that within Eclipse by dropping the Eclipse project onto the Tomcat server (under Servers) as shown in **Fig 3.5.1.**

However, having Tomcat configured under Eclipse slows down Eclipse, so many times it might be a good idea not to configure it there and keep it outside.

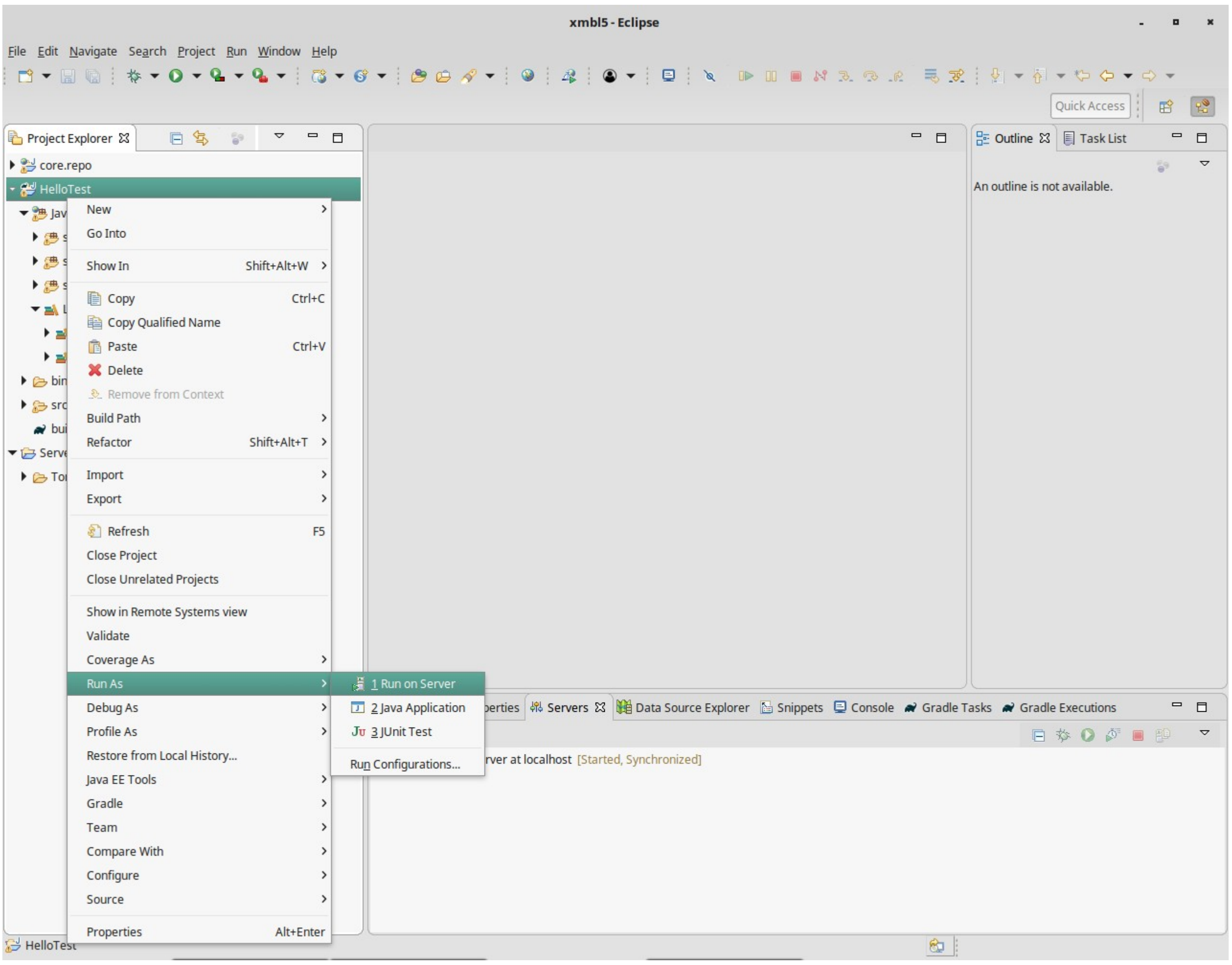
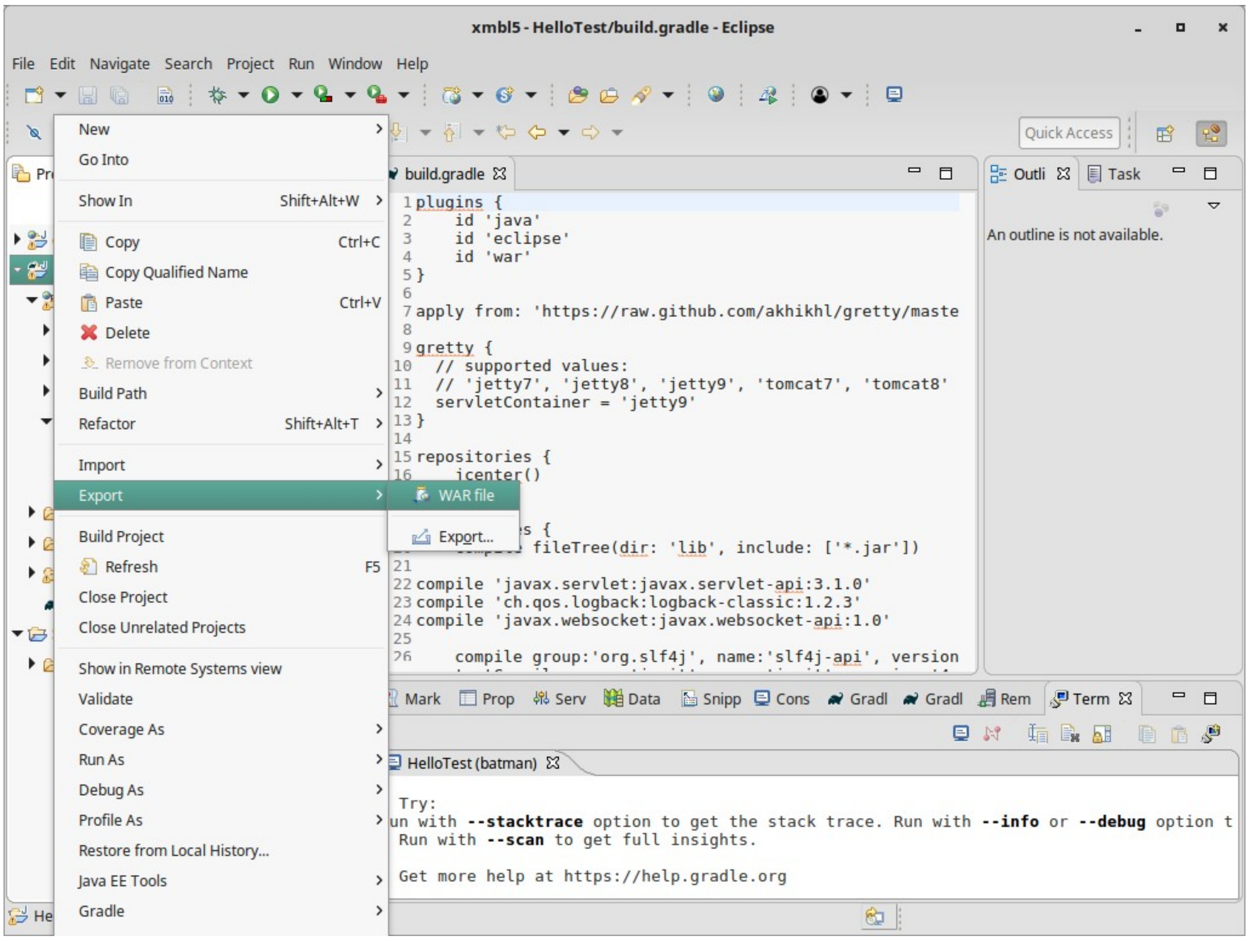


Fig 3.5.1

## 3.5.2 Alternative 2: Building a war File and Manually Deploying It

The alternative is to do gradle build to create the war (web archive) file, and then deploy it in Tomcat.Gradle build may be fired through Eclipse by creating war



```
plugins {
    id 'java'
    id 'eclipse'
    id 'war'
}

apply from: 'https://raw.github.com/akhikhl/gretty/maste

gretty {
  // supported values:
  // 'jetty7', 'jetty8', 'jetty9', 'tomcat7', 'tomcat8'
  servletContainer = 'jetty9'
}

repositories {
    icenter()

      s {
            fileTree(dir: 'lib', include: ['*.jar'])

compile 'javax.servlet:javax.servlet-api:3.1.0'
compile 'ch.qos.logback:logback-classic:1.2.3'
compile 'javax.websocket:javax.websocket-api:1.0'

    compile group:'org.slf4j', name:'slf4j-api', version
```

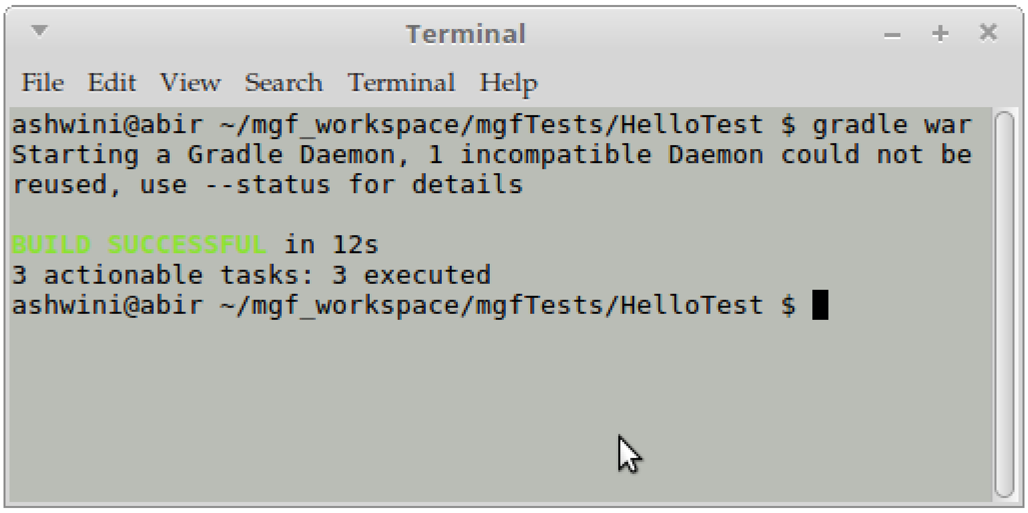
```
 Try:
un with --stacktrace option to get the stack trace. Run with --info or --debug option t
 Run with --scan to get full insights.

 Get more help at https://help.gradle.org
```

Gradle build may also be fired through command line. You need to navigate to the root of the test bundle, and fire “gradle war” command as shown in the following screenshot.

With any of these options, the war file gets built under build/libs folder under your test bundle folder.

The easiest way to deploy the war to Tomcat is to copy it to Tomcat’s webapps subfolder. It is Tomcat’s hot deploy folder. To elaborate, Tomcat will detect it as soon as it is copied (as long as it is already running) and try to deploy it. If Tomcat is not running, it will detect it next time it is started and automatically try to deploy it.

Note that while a newly copied war file gets deployed well on tomcat, replacing an existing war some times does not go through successfully. So it’s safe to reload tomcat when you replace the war file.

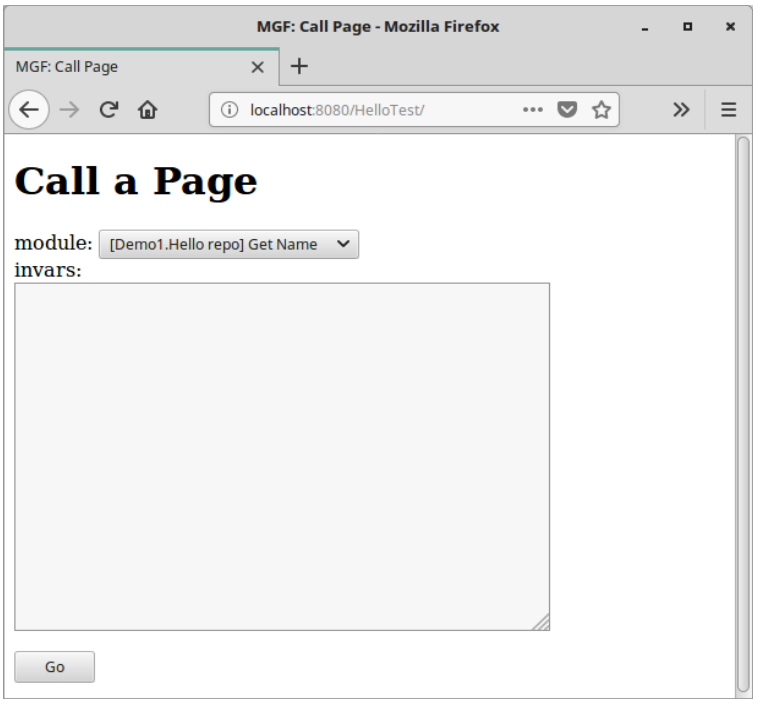


```
ashwini@abir ~/mgf_workspace/mgfTests/HelloTest $ gradle war
Starting a Gradle Daemon, 1 incompatible Daemon could not be
reused, use --status for details

BUILD SUCCESSFUL in 12s
3 actionable tasks: 3 executed
ashwini@abir ~/mgf_workspace/mgfTests/HelloTest $ 
```

### 3.6 Checking out the Web Application Post Deployment

Once deployed, you should be able to open your application in the web browser. if you have used the alternative of the war file deployment, then the deployment url is derived from the name of the war file by removing the trailing ".war". Thus, if your war file is named "HelloTest.war" then your application will be available at a url "http://localhost:8080/HelloTest".

When you point the browser to the correct url corresponding to your application, it will show the following screen.



Call a Page

module: [Demo1.Hello repo] Get Name

invars:

# 4 Writing Code

## 4.1 Writing / Editing Module Implementations

When a test bundle contains unfinished module implementations expected to be completed by the programmer, then you will need to make changes in the files that implement modules and the supporting ones.

A well-documented skeleton implementaiton is always in place, and a programmer is expected to fill in the business logic. Create/add module dependencies as needed (for example, a data access object used by a Method module that connects to the database, which in turn needs a jar file, or an image to be shown on the web page).

The process of making the code changes is iterative. You make the code changes and then execute the module by running test cases. If it does not run as expected, you make changes to your code and re-run the test cases. At the end of it, you will finally reach a stage where the module coding is complete and all test cases for the module run successfully.

## 4.2 Writing / Editing Test Cases

A test bundle always contains a JUnit test file containing test cases corresponding to every module. The file contains skeleton code to be used for creating the test cases. In the event there are pre-existing test cases then those are also found in this file.

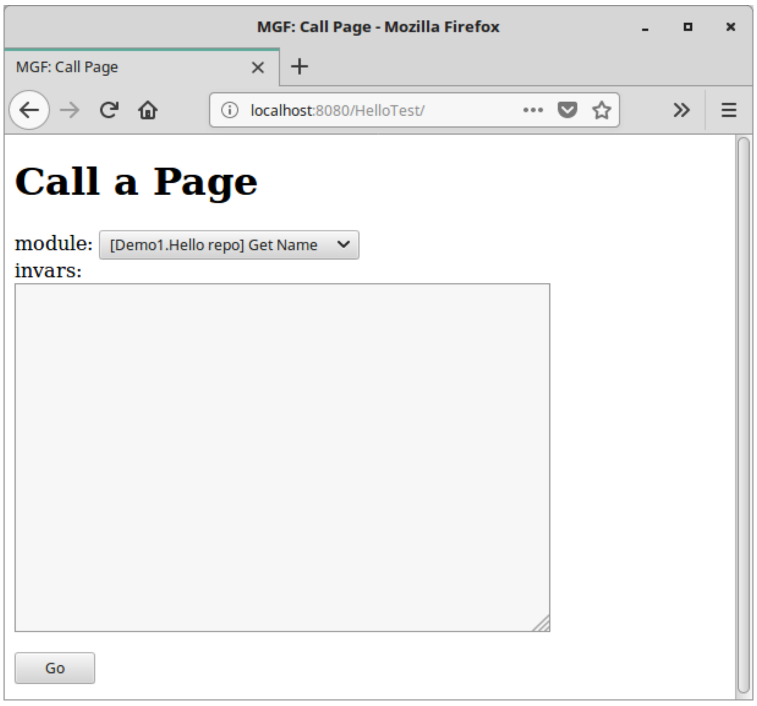
The skeleton code contains a commented test method that needs to be copy pasted outside the comment and changed. Note that the standard “@Test” annotation tells JUnit framework to treat the method as a test case. You should change the name of the method to something that’s descriptive to what the test case does.

Writing JUnit test cases is outside the scope of this guide. However, here are some useful hints.

1. **Method Module Test Cases**: One may optionally use the mockito framework to mock certain objects. This is useful in case the behaviour of external systems needs to be mocked. For instance, a mocked database connection can be made to return a pre-defined value against a call without actually connecting to a database.
2. **Entry Point Module Test Cases**: The request and response objects are mocked using Mockito, so that the test cases can be run without having a web request. For instance, the request object is mocked to return the desired values in the test case when its getParameter() method is called.
3. **Page Module Test Cases**: The Page module test cases need the test bundle deployed as a web application. The test cases connect to this deployed test bundle using HttpUnit framework. The base url of the this deployed web application is set in the function `getTestSystemBaseUrl()` under PageUtil.java, found under xsemble.util package. (It’s default setting should be good enough for most purposes, however in case your system has some other URL then that can be set over there.) The skeleton code contains appropriate comments as to how to pass parameters to the Page module and how to capture what is returned. Also see the section below.

### 4.2.1 Special Note on Testing a Page Module

When the browser is pointed to the deployed test bundle application, you will see the following user interface. Note that the page prompts the user to call a page (execute it) manually, which is an quick alternative to test the page.


MGF: Call Page - Mozilla Firefox
MGF: Call Page
localhost:8080/HelloTest/
Call a Page
module: [Demo1.Hello repo] Get Name
invars:
Go

The page contains a text area on which you would pass the values of the in arguments (short form invars) encoded as XML string. When you run the page test case, the same XML encoded invars need to be passed to the deployed test bundle. The key question is how to get these invars encoded in XML.

10Xofy provides a standalone Java file for this purpose. Click here to download the file XmlStr.java. Given a standalone file, it could be compiled even on command line. The purpose is to make changes to the invars as defined in the file as you want, and then compile it and execute it so that it shows the xml that may be fed to the test bundle.

Edit this file and set the appropriate invars that you intend to pass on to the Page module.

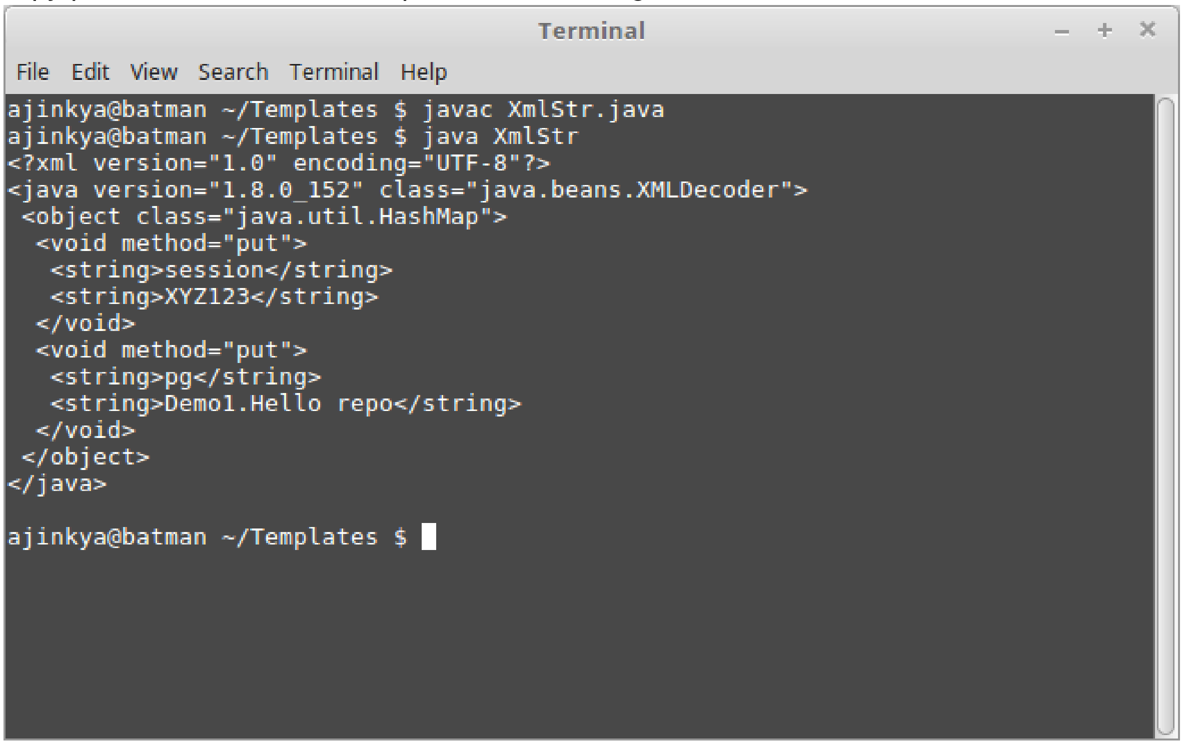
Now compile the file using command line as:

**javac XmlStr.java**

Then run it as:

**java XmlStr**

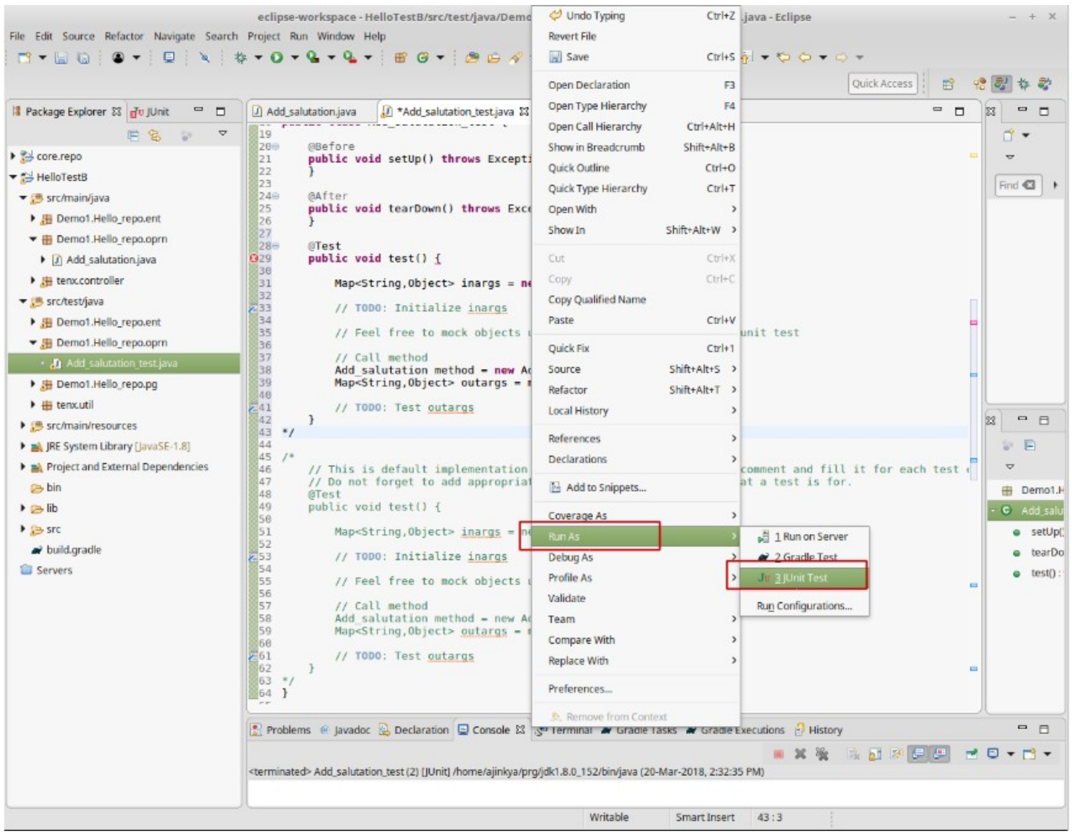
The below screenshot shows a successful execution on the command line. The output xml can be copy/pasted on the browser to pass on to the Page.



```
ajinkya@batman ~/Templates $ javac XmlStr.java
ajinkya@batman ~/Templates $ java XmlStr
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_152" class="java.beans.XMLDecoder">
 <object class="java.util.HashMap">
  <void method="put">
   <string>session</string>
   <string>XYZ123</string>
  </void>
  <void method="put">
   <string>pg</string>
   <string>Demo1.Hello repo</string>
  </void>
 </object>
</java>

ajinkya@batman ~/Templates $ 
```

## 4.3 Running the Test Cases

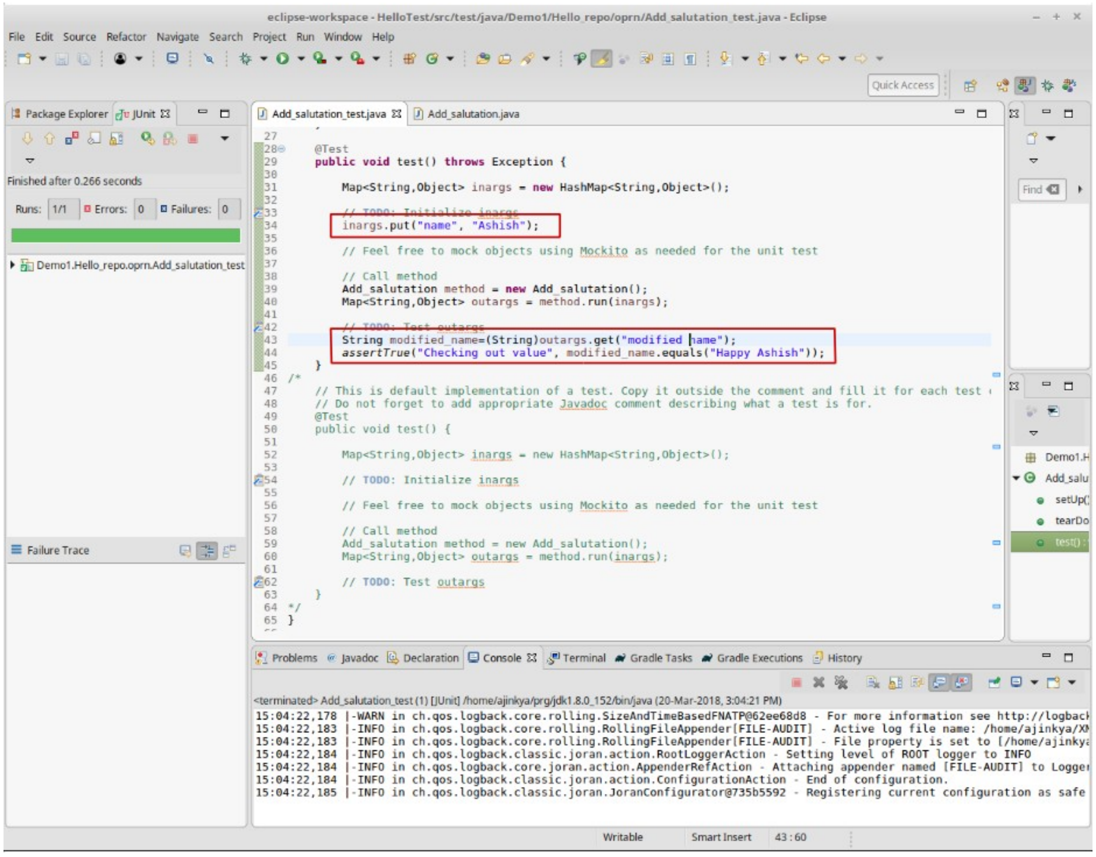
The easiest way to run the test cases is via the JUnit plugin in Eclipse. Right click on the test case class and choose “Run As...” and select JUnit test, as shown in the following figure.



```
    @Before
    public void setUp() throws Excepti
    }

    @After
    public void tearDown() throws Exce
    }

    @Test
    public void test() {

        Map<String,Object> inargs = ne

        // TODO: Initialize inargs

        // Feel free to mock objects 

        // Call method
        Add_salutation method = new Ad
        Map<String,Object> outargs = m

        // TODO: Test outargs
    }
*/

/*
    // This is default implementation                              comment and fill it for each test 
    // Do not forget to add appropriat                             at a test is for.
    @Test
    public void test() {

        Map<String,Object> inargs = ne

        // TODO: Initialize inargs

        // Feel free to mock objects                               unit test

        // Call method
        Add_salutation method = new Ad
        Map<String,Object> outargs = m

        // TODO: Test outargs
    }
*/
}
```

```
<terminated> Add_salutation_test (2) [JUnit] /home/ajinkya/prg/jdk1.8.0_152/bin/java (20-Mar-2018, 2:32:35 PM)
```

The following screenshot shows the successful execution of a test case. Notice the green bar that indicates success. It would be red if the test case is not successful.



```
    @Test
    public void test() throws Exception {

        Map<String,Object> inargs = new HashMap<String,Object>();

        ~~// TODO: Initialize inargs~~
        inargs.put("name", "Ashish");

        // Feel free to mock objects using Mockito as needed for the unit test

        // Call method
        Add_salutation method = new Add_salutation();
        Map<String,Object> outargs = method.run(inargs);

        ~~// TODO: Test outargs~~
        String modified_name=(String)outargs.get("modified name");
        assertTrue("Checking out value", modified_name.equals("Happy Ashish"));
    }
/*
    // This is default implementation of a test. Copy it outside the comment and fill it for each test 
    // Do not forget to add appropriate Javadoc comment describing what a test is for.
    @Test
    public void test() {

        Map<String,Object> inargs = new HashMap<String,Object>();

        // TODO: Initialize inargs

        // Feel free to mock objects using Mockito as needed for the unit test

        // Call method
        Add_salutation method = new Add_salutation();
        Map<String,Object> outargs = method.run(inargs);

        // TODO: Test outargs
    }
*/
}
```

```
<terminated> Add_salutation_test (1) [JUnit] /home/ajinkya/prg/jdk1.8.0_152/bin/java (20-Mar-2018, 3:04:21 PM)
15:04:22,178 |-WARN in ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP@62ee68d8 - For more information see http://logback
15:04:22,183 |-INFO in ch.qos.logback.core.rolling.RollingFileAppender[FILE-AUDIT] - Active log file name: /home/ajinkya/X
15:04:22,183 |-INFO in ch.qos.logback.core.rolling.RollingFileAppender[FILE-AUDIT] - File property is set to [/home/ajinky
15:04:22,184 |-INFO in ch.qos.logback.classic.joran.action.RootLoggerAction - Setting level of ROOT logger to INFO
15:04:22,184 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [FILE-AUDIT] to Logge
15:04:22,184 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - End of configuration.
15:04:22,185 |-INFO in ch.qos.logback.classic.joran.JoranConfigurator@735b5592 - Registering current configuration as safe
```

# 5 Best Practices for Writing Test Cases

We conclude this guide with a word on how the test cases should be designed. Good references on this topic can be found in the automated testing literature. For a good reference, click here.

A general principle is that the test cases should test how your system *should* always work. In a test bundle, you get to save a test case condition only when it works right. This ensures that the test case ran successfully at least at the time of creation of the test case. If a test case starts failing, then it typically means that some code change in the module has made it fail. Now you will either change the module code so that the test case does not fail, or edit / delete the test case if the test condition has no more remained current.

## *5.1 Have a Complete Coverage*

Try to cover all the functionality of your module in test cases. Typically, you will need separate positive and negative test cases. For example, the valid username and password submitted to a login method module is a positive test case, and invalid username and password submitted is a negative test case. For a negative test case, the module throwing an error is typically the condition of the test case passing.

## *5.2 Name Test Cases Appropriately*

The name of a test case is the natural place to document the purpose of the test case. You can have spaces in the test names but no slashes. (Spaces get converted to underscores when creating the name of the file).

## *5.3 Test Only One Thing in a Test Case*

Dedicate one test case for testing one thing. Thats way, when a test case fails, it's very easy to debug what could have made it fail.

## *5.4 Make FIRST class Test Cases (Fast, Independent, Repeatable, Small and Transparent)*

Out of these, Independence and Repeatability warrant a constant test bed approach. Choose a standard data set that your database has before running any test. Take a snapshot of it (by exporting it) so that it can be readily reinstated as required. Write your test cases against this standard data set, so your answers are always same and predictable. For example, if your standard data set contains 4 users, then a method module that counts the number of users can be tested on whether it returns 4. This approach works as long as the data set is always the same before running a test case. However, this is a challenge when some method module, in the course of its execution, modifies the data. For example, a “Create user” or a “Delete user” functionality would actually affect the user count in the data set.

The independence means that the test cases should be able to be run in any order, and the success of a test case should not depend on the success of another test case. This is possible only when the data set is brought to the standard data set before the beginning of every test case execution.