

JUnit Testing Using EasyMock

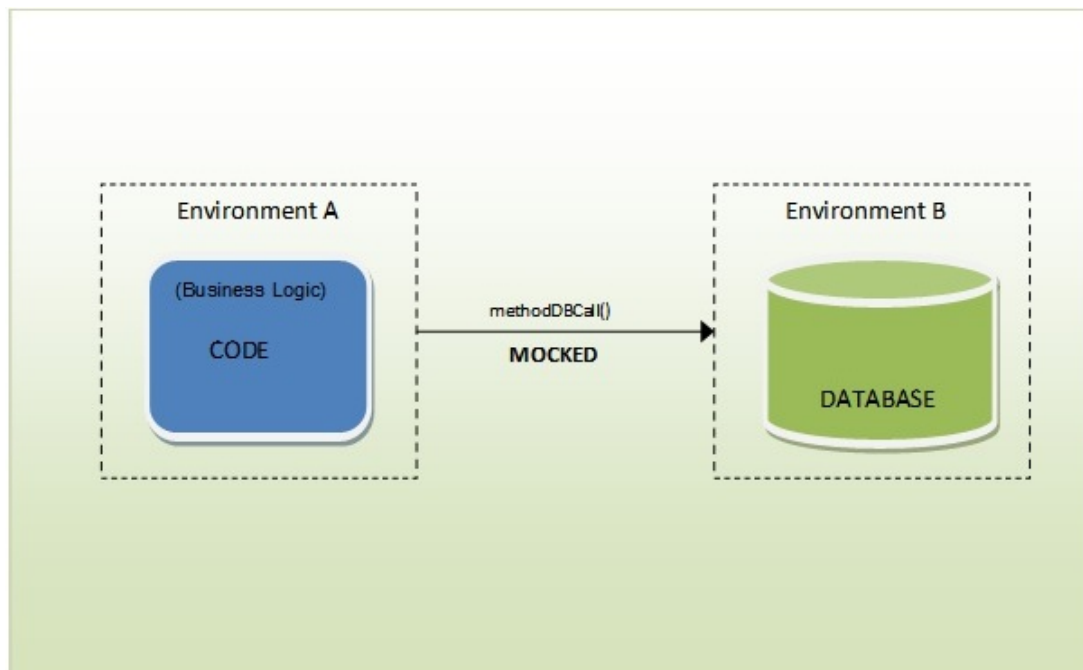
by Sourabh Bawage · Jan. 04, 17 · Java Zone · Tutorial

Start coding something amazing with our library of open source Cloud code patterns. Content provided by IBM.

JUnit is one of the most popular frameworks for performing Java UT. Mocking is also an aspect that goes hand in hand with JUnit. Unit testing is, of course, performed by developers to test the code they've written.

Let's consider:

- Developer 1 has developed business logic code.
- Developer 2 has developed persistence logic code (code that interacts with the database).

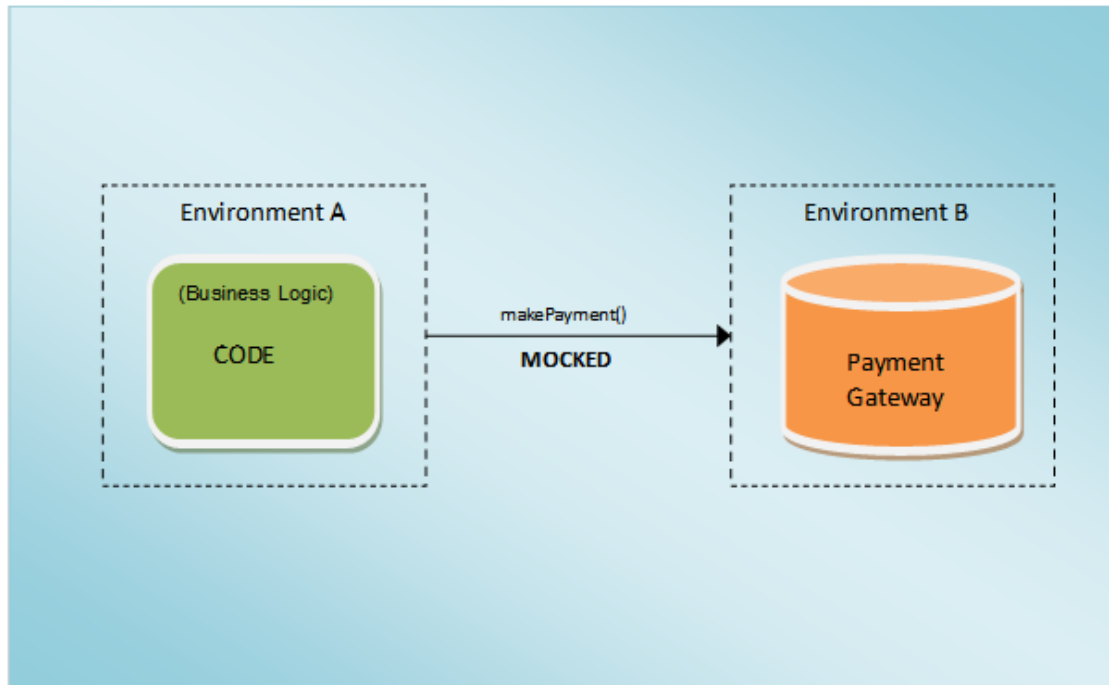


If Developer 1 has to use the persistence logic code (method call) written by Developer 2, then Dev 1 might mock that method call. Doing so will let the developer focus only on that code to test it. It's more like staying in an environment and just testing the code in that environment itself.

While unit testing, developers just test the code in the same environment and mocks the interaction with any

other environment.

A more real-time example would be where the code has to interact with the payment gateway. For UT, a developer might not want to send requests to the payment gateway.



Actual interactions with the payment gateway might be done during the SIT (System Integrated Testing) phase.

There are many frameworks for mocking; Mockito and EasyMock are two of the most popular frameworks. I will be using EasyMock in the example below. Mocking is the process of using a fake object during the UT phase. These frameworks do this by generating proxy objects (`java.lang.reflect.Proxy`). However, for these purposes, you don't need to know the details about proxy objects.

I will just be using a simple editor for the example, as it will allow for better understanding of the implementations that an IDE might not be able to show.

We will need following JARs:

- `mysql-connector-java-5.1.6-bin.jar` <Your JDBC driver, incase you don't mock>
- `junit-4.11-beta-1.jar`
- `hamcrest-core-1.3.RC2.jar`
- `easymock-3.2.jar`
- `cglib-nodep-2.2.2.jar`
- `obienesis-1.2.jar`

There are four java files for this example.

- Student.java (POJO Class)
- StudentDAO.java
- StudentCheck.java
- StudentTest.java <JUnit Test Class>

Here, I am using the data type as String for rollNo (though ideally, it should be int). I'm just using a String for the sake of this example.

```
1  public class Student
2  {
3      private String rollNo;
4      private String firstname;
5      private String lastname;
6      private String dept;
7
8      public String getRollNo(){
9          return rollNo;
10     }
11
12     public void setRollNo(String rollNo){
13         this.rollNo = rollNo;
14     }
15
16     public String getFirstname(){
17         return firstname;
18     }
19
20     public void setFirstname(String firstname){
21         this.firstname = firstname;
22     }
23
24     public String getLastname(){
25         return lastname;
26     }
27
28     public void setLastname(String lastname){
29         this.lastname = lastname;
30     }
31 }
```

```

29         this.lastname = lastname;
30     }
31
32     public String getDept(){
33         return dept;
34     }
35
36     public void setDept(String dept){
37         this.dept = dept;
38     }
39
40     public String toString(){
41         return "\n Roll No: "+rollNo+ "\n Firstname : " +firstname+ "\n Lastname : "+lastname-
42     }
43 }

```

This is the DAO class that actually hits the database.

```

1  import java.sql.*;
2  public class StudentDAO{
3
4      public Student getStudentDetails(String rollNo) throws Exception{
5
6          System.out.println("Actual DB call will occur.....");
7
8          Class.forName("com.mysql.jdbc.Driver");
9          Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/trial","root","root");
10         PreparedStatement ps = con.prepareStatement("select * from Student where rollno=?");
11         Student stud = new Student();
12         ps.setString(1,rollNo);
13         ResultSet rs= ps.executeQuery();
14         while(rs.next()){
15             stud.setRollNo(rs.getString("rollno"));
16             stud.setFirstname(rs.getString("firstname"));
17             stud.setLastname(rs.getString("lastname"));
18             stud.setDept(rs.getString("dept"));
19         }
20         return stud;
21     }
22 }

```

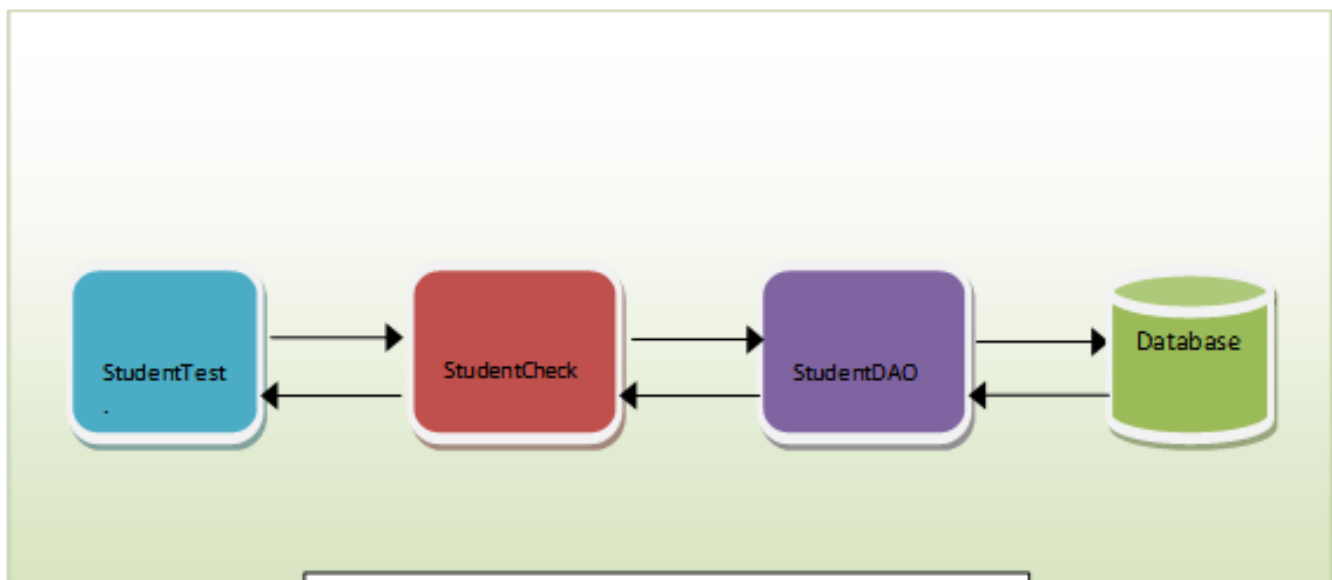
Please note that there is an SOP on line 6, which prints on the console if the code hits the database.

```
1 System.out.println("Actual DB call will occur.....");
```

StudentCheck.java is the class that calls the `getStudentDetails()` method of the `StudentDAO.java` class.

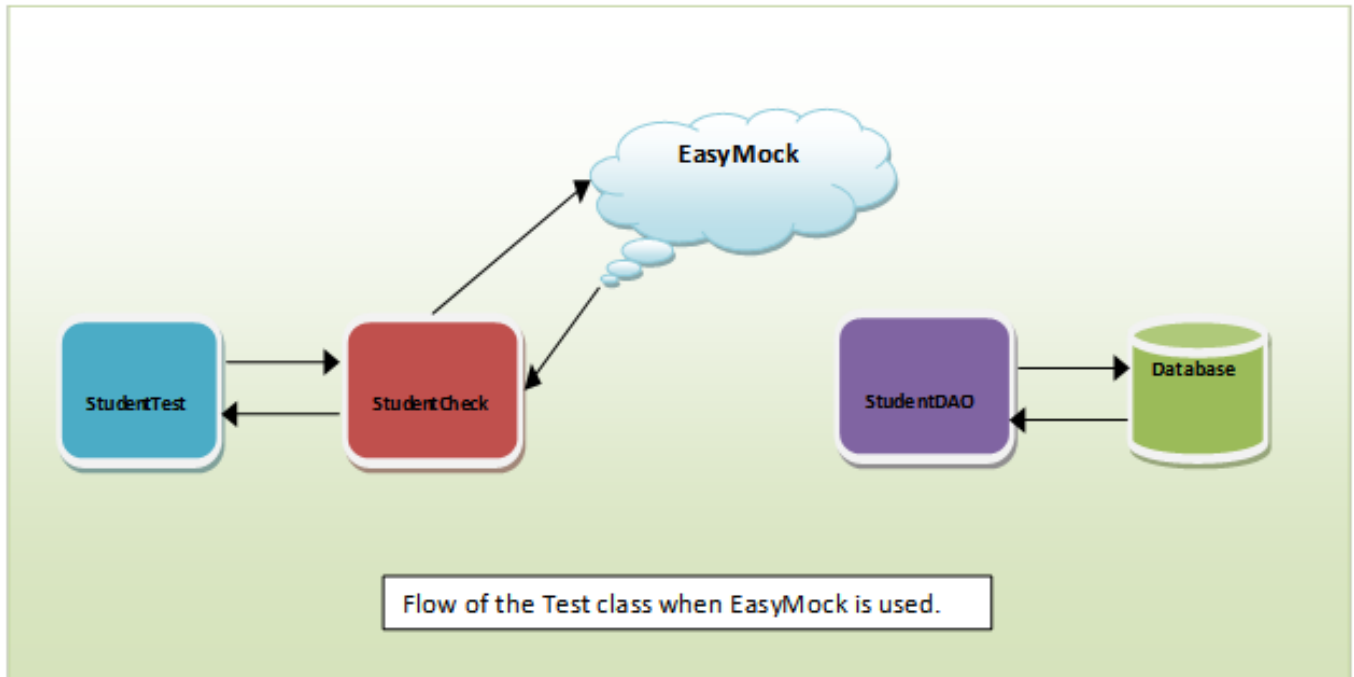
```
1 public class StudentCheck{
2
3     private StudentDAO studDAO = null;
4
5     public Student getDetails(String rollNo) throws Exception{
6
7         System.out.println("Before DB Call");
8
9         Student stud = studDAO.getStudentDetails(rollNo);
10
11        System.out.println("After DB Call");
12        return stud;
13    }
14
15    public void setStudDAO(StudentDAO studDAO){
16        this.studDAO = studDAO;
17    }
18 }
```

Now, `StudentTest.java` is the Test class where the actual action happens. This class has JUnit and the EasyMock API.



Normal Flow of the Test class without Mocking.

The EasyMock framework creates a proxy object/fake object of the class that we want to mock, and this proxy object can be used to return whatever you want as a return value of any of the methods of this class.

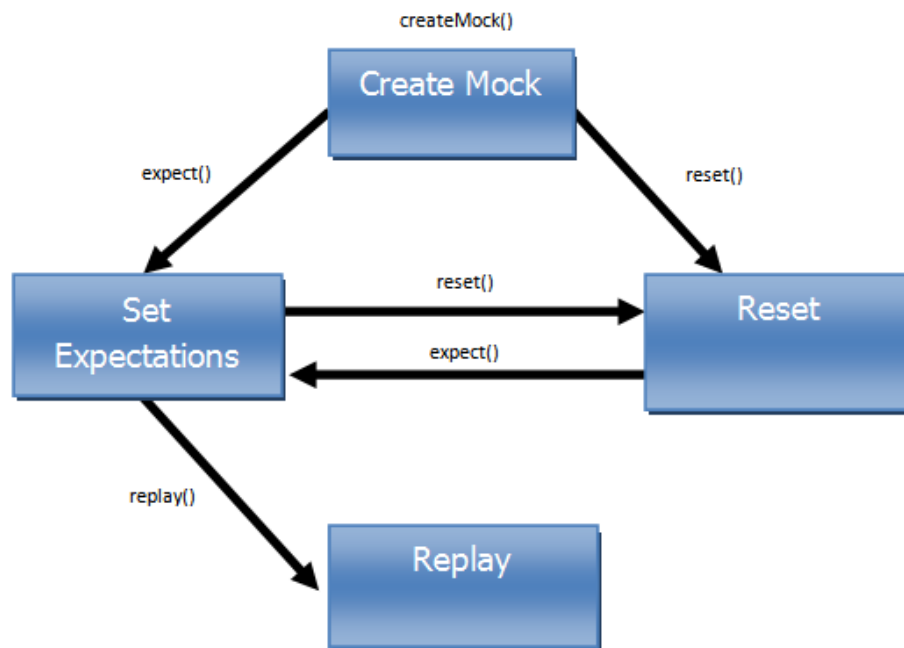


```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3  import org.easymock.*;
4
5  public class StudentTest {
6      @Test
7      public void testGetDetails001() throws Exception{
8          String rollNo= "111";
9          StudentDAO studentDAO = EasyMock.createMock(StudentDAO.class);
10         Student student = new Student();
11         student.setRollNo("111");
12         EasyMock.expect(studentDAO.getStudentDetails(EasyMock.isA(String.class))).andReturn(
13             EasyMock.replay(studentDAO);
14             StudentCheck studentCheck = new StudentCheck();
15             studentCheck.setStudDAO(studentDAO);
16             Student stud = studentCheck.getDetails(rollNo);
17             assertEquals("111",stud.getRollNo());
18     }
```

```
18  
19 }
```

We can compare an EasyMock object to a tape.

- `createMock()`: Blank Tape.
- `expect()`: Recording your favorite song.
- `replay()`: Playing your favorite song.
- `reset()`: Erasing the tape, making it blank to record a new song.



Here, we are using `EasyMock.isA()`, but if we expect that a specific input is to be received by the method that mock, then we can set such expectations

```
1 EasyMock.expect(studentDAO.getStudentDetails("111")).andReturn(student);
```

This ensures that `getStudentDetails()` is expected ONLY when the input to the method is "111." For anything else, we get an exception.

```
1 import static org.junit.Assert.*;  
2 import org.junit.Test;  
3 import org.easymock.*;  
4
```

```

5 public class StudentTest {
6     @Test
7     public void testGetDetails001() throws Exception{
8
9
10        String rollNo= "111";
11
12        StudentDAO studentDAO = EasyMock.createMock(StudentDAO.class);
13
14        Student student = new Student();
15        student.setRollNo("111");
16
17        EasyMock.expect(studentDAO.getStudentDetails(EasyMock.isA(String.class))).andReturn(student);
18
19        EasyMock.replay(studentDAO);
20
21        StudentCheck studentCheck = new StudentCheck();
22        studentCheck.setStudDAO(studentDAO);
23
24        Student stud = studentCheck.getDetails(rollNo);
25        assertEquals("111", stud.getRollNo());
26    }
27 }
28
29 }

```

JUnit Test Annotation.

This method creates a Mock object for StudentDAO class.

We are setting the expectations of getStudentDetails(), we can control what is to be returned when getStudentDetails() is called. Here, Student obj with rollNo 111 is returned. You can set any property that you want.

Lets EasyMock know that all expectations are set and the mock object is ready to be used. }

With isA(), we are telling EasyMock that when getStudentDetails() will be called, the parameter will be of String type. isA() accepts the class type as its parameter, it can be XXX.class.

Its like telling EasyMock "I don't care what values it has as long as it is of XXX.class type."

Finally, we run our test class with the following command.

```

C:\> Select Administrator: C:\Windows\system32\cmd.exe

E:\> java org.junit.runner.JUnitCore StudentTest
JUnit version 4.11-beta-1
.Before DB Call
After DB Call

Time: 0.176

OK (1 test)

E:\>

```

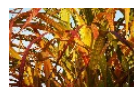
Please note "Actual DB call will occur....." is not printed on the console. This shows the DB was not hit.

Use this tool to look at the contents of GitHub and classify code based on the programming language used. Content provided by IBM Developer.

Like This Article? Read More From DZone



A Guide to Mocking With Mockito



The Concept of Mocking




JUnit4, JUnit5, and Spock: A Comparison



Free DZone Refcard Java Containerization

Topics: JUNIT , EASYMOCK , JAVA , UNIT TESTING , TUTORIAL

Published at DZone with permission of Sourabh Bawage . [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Java Partner Resources

Developing Reactive Microservices: Enterprise Implementation in Java

ghtbend

|

ilding Reactive Microservices in Java: Asynchronous and Event-Based Application Design

ed Hat Developer Program

|

va Ecosystem 2018 Infographic by JetBrains

tBrains

|

crosservices for Java Developers: A Hands-On Introduction to Frameworks & Containers

ed Hat Developer Program

|