
# Effective Unit Testing for Java EE

Version 2.1

# Contents

# Chapter 1

# Overview

Needle is a lightweight framework for testing Java EE components outside of the container in isolation. The main goals are the reduction of setup code and faster executions of tests, especially compared to running embedded or external server tests.

Needle will automatically analyse the dependencies of components and inject mock objects by default. The developer may freely provide default or custom objects instead.

Needle is an Open Source library, hosted at SourceForge.net https://sourceforge.net/projects/jbosscc-needle. It is licensed under GNU Lesser General Public License (LGPL) version 2.1 or later.

## 1.1 Features

- Instantiation of tested components
- Constructor, Method and Field based dependency injection
- Injection of Mock objects by default
- Extensible by providing custom injection providers
- Comfortable automatic wiring of dependency graphs
- Database testing via JPA Provider, e.g. EclipseLink or Hibernate
- EntityManager creation and injection
- Execute optional database operations during test setup and tear down
- Transaction Utilities
- Provide Utilities for Reflection, e.g. for private method invocation or field access
- Needle can be used with JUnit or TestNG.
- It supports EasyMock and Mockito out-of-the-box but could also be extended with other frameworks.

## 1.2 Links

- Needle Home Page: http://needle.spree.de/
- Downloads: https://sourceforge.net/projects/jbosscc-needle/
- Forums: https://sourceforge.net/projects/jbosscc-needle/forums
- Issue Tracking: https://sourceforge.net/tracker/?group_id=306915
- Source Code:

```
svn co https://jbosscc-needle.svn.sourceforge.net/svnroot/jbosscc-needle/
   trunk/jbosscc-needle/ jbosscc-needle
```

Listing 1.1: Source Code

# Chapter 2

# Getting Started

In this chapter, a very simple user management application is to be tested using Needle.

## 2.1 Sample Application

The example consists of two JPA entity classes User and Profile, with a `@OneToOne` relationship between them and two CDI components.

### 2.1.1 User

```
@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

   @Column(unique = true, nullable = false)
   private String username;

   @Column(nullable = false)
   private String password;

   @OneToOne(optional = false, cascade = CascadeType.ALL)
   private Profile profile;

   // Getters and setters
   ...
}
```

Listing 2.1: The user entity

### 2.1.2 Profile

```
@Entity
public class Profile {
   @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String language;

    // Getters and setters
    ...
}
```

Listing 2.2: The profile entity

### 2.1.3 Data Access Object

Now we add a simple DAO component to access the user data.

```
public class UserDao {
    @PersistenceContext
    private EntityManager entityManager;

    public User findBy(final String username, final String password) {
        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> query = builder.createQuery(User.class);

        Root<User> user = query.from(User.class);

        query.where(
                builder.and(builder.equal(user.get(User_.username), username))
                    ,
                builder.equal(user.get(User_.password), password));

        return entityManager.createQuery(query).getSingleResult();
    }

    public List<User> findAll() {
        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> query = builder.createQuery(User.class);

        return entityManager.createQuery(query).getResultList();
    }
}
```

Listing 2.3: The User DAO component

### 2.1.4 Authenticator

To authenticate a user, the application uses an authenticator component which itself depends on the User DAO.

```
public class Authenticator {
    @Inject
    private UserDao userDao;
```

```
    public boolean authenticate(final String username, final String
        password) {
        User user = userDao.findBy(username, password);

        return user != null ? true : false;

    }
}
```

Listing 2.4: The authenticator component

## 2.2 Using Needle with JUnit

Needle provides JUnit "Rules" to extend JUnit. Rules are basically wrappers around test methods. They may execute code before, after or instead of a test method.

The following example demonstrates hwo to write a simple JUnit Needle test with two rules. The database rule provides access to the database via JPA and may execute optional database operations, e.g. to setup the initial data. The Needle rule does the real magic: it scans the test for all fields annotated with `@ObjectUnderTest` and initializes these tested components by injection of their dependencies. I.e., the UserDao will get the `EntityManager` field injected automatically. Since we provided a database rule that entity manager will not be a mock, but a "real" entity manager.

Supported injections are constructor injection, field injection and method injection.

```
public class UserDaoTest {
    @Rule
    public DatabaseRule databaseRule = new DatabaseRule();

    @Rule
    public NeedleRule needleRule = new NeedleRule(databaseRule);

    @ObjectUnderTest
    private UserDao userDao;

    @Test
    public void testFindByUsername() throws Exception {
        final User user = new UserTestdataBuilder(
                databaseRule.getEntityManager()).buildAndSave();

        User findBy = userDao.findBy(user.getUsername(), user.getPassword())
            ;

        Assert.assertEquals(user.getId(), findBy.getId());
    }

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

```
      new UserTestdataBuilder(databaseRule.getEntityManager()).
          buildAndSave();

      List<User> all = userDao.findAll();

      Assert.assertEquals(1, all.size());
    }
}
```

Listing 2.5: JUnit User DAO test

## 2.3 Using Needle with TestNG

Needle also supports TestNG. There are two abstract test cases that may be extended by concrete test classes.

The class `de.akquinet.jbosscc.needle.testng.AbstractNeedleTestcase` scans all fields annotated with `@ObjectUnderTest` and initializes the components.

The class `de.akquinet.jbosscc.needle.testng.DatabaseTestcase` can either be used as a special provider for `EntityManager` injection or as a base test case for JPA tests. In the first case, a new DatabaseTestcase instance is passed to the constructor of the AbstractNeedleTestcase:

```
public class UserDaoTest extends AbstractNeedleTestcase {
  public UserDaoTest() {
    super(new DatabaseTestcase());
  }

  @ObjectUnderTest
  private UserDao userDao;

  @Test
  public void testFindByUsername() throws Exception {
    final User user = new UserTestdataBuilder(getEntityManager())
        .buildAndSave();

    User findBy = userDao.findBy(user.getUsername(), user.getPassword());
    Assert.assertEquals(user.getId(), findBy.getId());
  }

  @Test
  public void testFindAll() throws Exception {
    new UserTestdataBuilder(getEntityManager()).buildAndSave();

    List<User> all = userDao.findAll();
    Assert.assertEquals(1, all.size());
   }
}
```

Listing 2.6: TestNG User DAO test

# Chapter 3

# Configuration

## 3.1 Requirements

Ensure that you have a JDK6+ installed.

## 3.2 Maven dependency configuration

If you are using Maven as your build tool add the following single dependency to your pom.xml file to get started with Needle:

```
<dependency>
    <groupId>de.akquinet.jbosscc</groupId>
    <artifactId>jbosscc-needle</artifactId>
    <scope>test</scope>
    <version>${needle.version}</version>
</dependency>
```

Listing 3.1: The Maven pom.xml

Where `needle.version` currently should have the value "2.0.1". Check for the most current version at our web site.

To reduce complexity Needle has no transitive dependencies, and does thus not restrict you to use specific versions of JUnit or TestNG. On the other hand, you will have to explicitely configure dependencies for the test scope, for example, to Hibernate as the JPA provider.

## 3.3 Needle configuration properties

The Needle default configuration may be modified in a `needle.properties` file in the classpath root. I.e., Needle will look for a file `/needle.properties` somewhere in the classpath.

Configuration of additional custom injection annotations and injection provider.

| Property name | Description |
|---|---|
| custom.injection.annotations | Comma separated list of the fully qualified name of the annotation classes. A standard mock provider will be created for each annotation. |
| custom.injection.provider.classes | Comma separated list of the fully qualified name of the injection provider implementations. |

Table 3.1: Custom Injection Provider

Configuration of mock provider.

| Property name | Description |
|---|---|
| mock.provider | The fully qualified name of an implementation of the MockProvider interface. There is an implementation of EasyMock de.akquinet.jbosscc.needle.mock.EasyMockProvider and Mockito de.akquinet.jbosscc.needle.mock.MockitoProvider. **EasyMock is the default configuration.** |

Table 3.2: Mock Provider

Configuration of JPA, Database operation and JDBC connection.

| Property name | Description |
|---|---|
| persistenceUnit.name | The persistence unit name. Default is TestDataModel |
| hibernate.cfg.filename | XML configuration file to configure Hibernate (eg. /hibernate.cfg.xml) |
| db.operation | Optional database operation on test setup and tear down. Value is the fully qualified name of an implementation of the AbstractDBOperation base class. There is an implementation for script execution de.akquinet.jbosscc.needle.db.operation.ExecuteScriptOperation and for the HSQL DB to delete all tables de.akquinet.jbosscc.needle.db.operation.hsql.HSQLDeleteOperation. |
| jdbc.url | The JDBC driver specific connection url. |
| jdbc.driver | The fully qualified class name of the driver class. |
| jdbc.user | The JDBC user name used for the database connection. |
| jdbc.password | The JDBC password used for the database connection. |

Table 3.3: JPA and JDBC Configuration

The JDBC configuration properties are only required if database operation and JPA 1.0 are used. Otherwise, the JDBC properties are related to the standard property names of JPA 2.0.

### 3.3.1 Example configuration

A typical `needle.properties` file might look like this:

```
db.operation=de.akquinet.jbosscc.needle.db.operation.hsql.
    HSQLDeleteOperation
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:mem:memoryDB
jdbc.user=sa
jdbc.password=
```

Listing 3.2: needle.properties

## 3.4 Logging

Needle uses the Simple Logging Facade for Java (SLF4J). SLF4J serves as a simple facade or abstraction for various logging frameworks. The SLF4J distribution ships with several JAR files referred to as "SLF4J bindings", with each binding corresponding to a supported framework.

For logging within the test, the following optional dependency may be added to the classpath:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <scope>test</scope>
</dependency>
```

Listing 3.3: SLF4J dependency

# Chapter 4

# Needle Testcase

## 4.1 ObjectUnderTest instantiation and initialization

Needle may automatically instantiate all objects under test for you. The Needle test case scans all fields of the test class for annotations and creates a completely initialized instance. Alternatively, the developer may of course instantiate the field herself, too.

Multiple fields can be annotated with the `@ObjectUnderTest` annotation. The annotation can optionally be configured with the implementation of the type and an id. The id may be used for additional injections (explained later). When an object under test is already instantiated, only the dependency injection will be done.

## 4.2 Injection

Needle supports field, constructor and method injection by evaluating `@EJB`, `@Resource`, `@PersistenceContext`, `@PersistenceUnit` and `@Inject` annotations. Note that the annotation class needs to be available one the classpath of the test execution. By default, Mock objects for the dependencies are created and injected.

The injected (mock or real) objects can be also injected into your testcase. Test case injection can be done in the same way as the injection of dependencies in the production code. I.e., you may write `@Inject` within your test case to access the generated components.

All standard Java EE annotations and any additionally configured injection annotation can be used to inject a reference into the testcase.

```
public class AuthenticatorTest {
   @Rule
   public NeedleRule needleRule = new NeedleRule();

   @ObjectUnderTest
```

```
    private Authenticator authenticator = new Authenticator();

    @Inject
    private EasyMockProvider mockProvider;

    @Inject
    private UserDao userDaoMock;

    @Test
    public void testAuthenticate() {
        ...
    }
}
```

Listing 4.1: Testcase injection

It is also possible to use the API to get a reference of an injected object.

```
UserDao injectedUserDao = needleRule.getInjectedObject(UserDao.class);
```

Listing 4.2: Injected Components

The key is generated from the respective injection provider. By default, the class object of the associated injection point is used as the key or – in the case of resource injection – the mapped name of the resource.

## 4.3 Custom injection provider

Needle is fully extensible, you may implement your own injection providers or register additional annotations.

The following example shows the registration of additional annotations in the `needle.properties`.

```
custom.injection.annotations=org.jboss.seam.annotations.In, org.jboss.seam
    .annotations.Logger
```

Listing 4.3: Additional Annotation

It is also possible to implement custom providers. A custom injection provider must implement the `de.akquinet.jbosscc.needle.injection.InjectionProvider` interface.

```
public class CurrentUserInjectionProvider implements InjectionProvider<
    User> {
    private final User currentUser = new User();

    @Override
    public User getInjectedObject(Class<?> injectionPointType) {
        return currentUser;
    }
```

```
    @Override
    public boolean verify(InjectionTargetInformation information) {
        return information.isAnnotationPresent(CurrentUser.class);
    }

    @Override
    public Object getKey(InjectionTargetInformation information) {
        return CurrentUser.class;
    }
}
```

Listing 4.4: javax.inject.Qualifier Injection Provider

A custom injection provider can be provided for a specific test or as a global provider (again in the `needle.properties` file).

```
@Rule
public NeedleRule needleRule = new NeedleRule(new
    CurrentUserInjectionProvider());
```

Listing 4.5: Custom injection provider for a specific test

```
custom.injection.provider.classes=de.akquinet.CurrentUserInjectionProvider
```

Listing 4.6: Global custom injection provider

## 4.4 Wiring of object dependencies

Sometimes you may want to provide your own objects as components or influence the wiring of complex object graphs. Typically, you may want to mix mock objects and "real" objects, depending on the focus of your test.

The object referenced by the field annotated with `@InjectIntoMany` is injected into all fields annotated with `@ObjectUnderTest`.

In the following example UserDao is not a mock, but a fully initialized component. It is injected into the Authenticator object (or any other component annotated with `@ObjectUnderTest`).

```
@ObjectUnderTest
private Authenticator authenticator;

@InjectIntoMany
@ObjectUnderTest
private UserDao userDao;
```

Listing 4.7: InjectIntoMany

If you need more fine-grained configuration you may also use the `@InjectInto` annotation to specify the component IDs where the object shall be injected. The ID corresponds to the field name of the target object by default.

```
@ObjectUnderTest
private Authenticator authenticator;

@InjectInto(targetComponentId="authenticator")
@ObjectUnderTest
private UserDao userDao;
```

Listing 4.8: InjectInto

# Chapter 5

# Database Testing

When unit-testing your application, it is usually recommended to mock calls to DAOs oder database dependent services in your test cases. However, you will also need to test against a real database, e.g. to make sure that your queries work as expected.

## 5.1 Database Testcase

In these cases, Needle can automatically create and inject the `EntityManager` instance into your objects under test. You simply need to provide a JPA persistence.xml and the JDBC driver to the classpath of the test execution. The `persistence.xml` file is the standard configuration file in JPA. The `persistence.xml` file must define a persistence-unit with a unique name and the transaction type RESOURCE_LOCAL for a Java SE environment.

The following listing below shows a complete example of a `persistence.xml` file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.
      sun.com/xml/ns/persistence/persistence_2_0.xsd"
   version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
   <persistence-unit name="TestDataModel"
      transaction-type="RESOURCE_LOCAL">

      <class>de.akquinet.jbosscc.needle.example.User</class>
      <class>de.akquinet.jbosscc.needle.example.Profile</class>

      <properties>
         <property name="javax.persistence.jdbc.driver" value="org.hsqldb.
            jdbcDriver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."
             />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <property name="javax.persistence.jdbc.password" value="" />
```

```
        <!-- EclipseLink should create the database schema automatically
            -->
        <property name="eclipselink.ddl-generation" value="create-tables"
            />
        <property name="eclipselink.ddl-generation.output-mode"
            value="database" />
      </properties>

   </persistence-unit>
</persistence>
```

Listing 5.1: test persistence.xml

The UserTest below checks the JPA mapping against a real database.

```
public class UserTest {
   @Rule
   public DatabaseRule databaseRule = new DatabaseRule();

   @Test
   public void testPersist() throws Exception {
      EntityManager entityManager = databaseRule.getEntityManager();

      User user = new UserTestdataBuilder(entityManager).buildAndSave();

      User userFromDb = entityManager.find(User.class, user.getId());
      Assert.assertEquals(user.getId(), userFromDb.getId());
   }
}
```

Listing 5.2: User persistence test

## 5.2 Transaction utilities

The EntityManager is the primary interface used by application developers to interact with the underlying database. Many operations must be executed in a transaction which is not started in the test component, because it is ususally maintained by the application server. To run your code using transactions the `TransactionHelper` Utility makes this more convenient.

```
public class UserTest {
   @Rule
   public DatabaseRule databaseRule = new DatabaseRule();

   private TransactionHelper transactionHelper = databaseRule.
      getTransactionHelper();

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

```
        final User user = new User();

        ...

        transactionHelper.executeInTransaction(new VoidRunnable() {
            @Override
            public void doRun(EntityManager entityManager) throws Exception {
                entityManager.persist(user);
            }
        });

        ...

    }
}
```

Listing 5.3: Transaction utilities

The above example illustrates the use of the `TransactionHelper` class and the execution of a transaction. The implementation of the VoidRunnable is executed within a transaction. There is also `Runnable<T>` interface that contains a method run return a value to be used for further testing.

## 5.3 Database operation

A common issue in unit tests that access a real database is their effect on the state of the persistence store. They usually expect a certain state at the beginning and alter it at runtime which of course will affect other tests. To address that problem optional Database operations can be executed before and after test execution, in order to clean up or set up data.

There are two implementations:

1. `de.akquinet.jbosscc.needle.db.operation.ExecuteScriptOperation`
   Execute sql scripts during test setup and tear down. A before.sql and after.sql script must be provided on the classpath.

2. `de.akquinet.jbosscc.needle.db.operation.hsql.HSQLDeleteOperation`
   Deletes all rows of all tables of the hsql database.

To use own Database operation implementations, the abstract base class `de.akquinet.jbosscc.needl` must be implemented and configured in the `needle.properties` file.

## 5.4 Testdatabuilder

Using the Test Data Builder pattern, the builder class provides methods that can be used to configure the test objects. Properties that are not configured use default values. The builder methods can be chained together and provide transient or persistent testdata for the test case.

For this purpose Needle provides an abstract base class. The following code examples shows two implementations of Test Data Builder pattern. The Testdatabuilder inherit from de.akquinet.jbosscc.needle.
AbstractTestdataBuilder class.

```
public class UserTestdataBuilder extends AbstractTestdataBuilder<User> {

   private String withUsername;
   private String withPassword;
   private Profile withProfile;

   public UserTestdataBuilder() {
      super();
   }

   public UserTestdataBuilder(EntityManager entityManager) {
      super(entityManager);
   }

   public UserTestdataBuilder withUsername(final String username) {
      this.withUsername = username;
      return this;
   }

   private String getUsername() {
      return withUsername != null ? withUsername : "username";
   }

   public UserTestdataBuilder withPassword(final String password) {
      this.withPassword = password;
      return this;
   }

   private String getPassword() {
      return withPassword != null ? withPassword : "password";
   }

   public UserTestdataBuilder withProfile(final Profile profile) {
      this.withProfile = profile;
      return this;
   }

   private Profile getProfile() {
      if (withProfile != null) {
         return withProfile;
```

```
        }

        return hasEntityManager() ? new ProfileTestdataBuilder(
                getEntityManager()).buildAndSave()
                : new ProfileTestdataBuilder().build();

    }

    @Override
    public User build() {
        User user = new User();
        user.setUsername(getUsername());
        user.setPassword(getPassword());
        user.setProfile(getProfile());
        return user;
    }

}
```

Listing 5.4: User Testdatabuilder

```
public class ProfileTestdataBuilder extends AbstractTestdataBuilder<
    Profile> {

    private String withLanguage;

    public ProfileTestdataBuilder() {
        super();
    }

    public ProfileTestdataBuilder(EntityManager entityManager) {
        super(entityManager);
    }


    public ProfileTestdataBuilder withLanguage(final String language) {
        this.withLanguage = language;
        return this;
    }

    private String getLanguage() {
        return withLanguage != null ? withLanguage : "de";
    }

    @Override
    public Profile build() {
        Profile profile = new Profile();
        profile.setLanguage(getLanguage());
        return profile;
    }
```

```
}
```

Listing 5.5: Profile Testdatabuilder

In the example the Testdatabuilder is using defaults for everything except the username.

```
final User user = new UserTestdataBuilder(databaseRule.getEntityManager())
    .withUsername("user").buildAndSave();
```

Listing 5.6: Build an persistent User object

```
final User user = new UserTestdataBuilder().withUsername("user").build()
```

Listing 5.7: Build an transient User object

# Chapter 6

# Testing with Mock objects

Mock objects are a useful way to write unit tests for objects that have collaborators. Needle generates Mock objects dynamically for dependencies of the components under test by default. Out-of-the-box Needle has implementations for EasyMock and Mockito. To use other mock frameworks, the interface `de.akquinet.jbosscc.needle.mock.MockProvider` must be implemented and configured in the `needle.properties` file.

## 6.1 Create a Mock Object

To create a Mock object, you can annotate a field with the annotation `@Mock`.

```
public class Test {
   @Rule
   public NeedleRule needleRule = new NeedleRule();

   @Mock
   private EntityManager entityManagerMock;

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

Listing 6.1: Mock annotation

The dependencies of an object under test are automatically initialized by the corresponding InjectionProvider. These dependencies can also injected into the testcase by using the corresponding injection annotation.

## 6.2 EasyMock

The EasyMockProvider creates "Nice" Mock objects by default. Such nice mocks allow all method calls and returns appropriate empty values e.g. 0, null or false. If needed, all mocks can also be converted to use another policy by calling resetAllToNice(), resetAllToDefault() or resetAllToStrict().

The EasyMockProvider implementation is a subclass of EasyMockSupport. EasyMockSupport is a class that meant to be used as a helper or base class to your test cases. It will automatically register all created mocks and to replay, reset or verify them in batch instead of explicitly.

The following test illustrates the usage of EasyMock with Needle and the injection of generated mock objects.

```
public class AuthenticatorTest {

   @Rule
   public NeedleRule needleRule = new NeedleRule();

   @ObjectUnderTest
   private Authenticator authenticator;

   @Inject
   private EasyMockProvider mockProvider;

   @Inject
   private UserDao userDaoMock;

   @Test
   public void testAuthenticate() throws Exception {
      final User user = new UserTestdataBuilder().build();
      final String username = "username";
      final String password = "password";

      EasyMock.expect(userDaoMock.findBy(username, password)).andReturn(
         user);

      mockProvider.replayAll();
      boolean authenticated = authenticator.authenticate(username,
         password);
      Assert.assertTrue(authenticated);
      mockProvider.verifyAll();
   }
}
```

Listing 6.2: Testing with EasyMock

EasyMock is the default mock provider. Only the EasyMock library must be added to the test classpath.

For more details about EasyMock, please refer to the EasyMock [http://easymock.org] documentation.

## 6.3 Mockito

Needle has also an mock provider implementation for Mockito. Mockito generates Mock objects, where by default the return value of a method is null, an empty collection or the appropriate primitive value.

The following test illustrates the usage of Mockito with Needle.

```
public class AuthenticatorTest {
   @Rule
   public NeedleRule needleRule = new NeedleRule();

   @ObjectUnderTest
   private Authenticator authenticator;

   @Inject
   private UserDao userDaoMock;

   @Test
   public void testAuthenticate() throws Exception {

      final User user = new UserTestdataBuilder().build();
      final String username = "username";
      final String password = "password";

      Mockito.when(userDaoMock.findBy(username, password)).thenReturn(user
         );

      boolean authenticated = authenticator.authenticate(username,
         password);
      Assert.assertTrue(authenticated);

   }
}
```

Listing 6.3: Testing with Mockito

To use Mockito, the mockito provider must be configured in the `needle.properties` file and the mockito library must be present on test classpath.

```
mock.provider=de.akquinet.jbosscc.needle.mock.MockitoProvider
```

Listing 6.4: Mockito configuration

For more details about Mockito, please refer to the Mockito [http://mockito.org] documentation.