



# Testing Spring Applications

---

Testing in General, Spring and JUnit,  
Profile-based Testing, Database Testing

## Objectives

---

After completing this lesson, you should be able to

- Write tests using JUnit 5
- Write Integration Tests using Spring
- Configure Tests using Spring Profiles
- Extend Spring Tests to work with Databases

# Agenda

- **JUnit 5**
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- Appendices
  - JUnit 5, JUnit 4, Stubs & Mocks



# Testing with JUnit 5



- Labs use JUnit 5 for testing
  - JUnit 5 support is a major feature of Spring 5
  - Requires Java 8+ at runtime
    - Leverages Lambdas
- Components
  - **JUnit Platform**
    - A foundation for launching testing frameworks on the JVM
  - **JUnit Jupiter**
    - An extension model for writing tests and extensions in JUnit 5
  - **JUnit Vintage**
    - A *TestEngine* for running JUnit 3 & 4 tests on the platform

# JUnit 5: New Programming Models

- Replaces JUnit 4 annotations
  - `@Before` → `@BeforeEach`
  - `@BeforeClass` → `@BeforeAll`
  - `@After` → `@AfterEach`
  - `@AfterClass` → `@AfterAll`
  - `@Ignore` → `@Disabled`
- Introduces new annotations
  - `@DisplayName`
  - `@Nested`
  - `@ParameterizedTest`
  - ...

} JUnit 5

- *Use right annotations from correct package*
- *JUnit 5 ignores all JUnit 4 annotations*

# Writing Test – JUnit 5 Style

New package

```
import static org.junit.jupiter.api.Assertions.fail;
```

```
import org.junit.jupiter.api.AfterAll;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Disabled;  
import org.junit.jupiter.api.Test;
```

```
class StandardTests {
```

```
    @BeforeAll  
    static void initAll() {  
    }  
}
```

Replaces  
**@BeforeClass**

```
    @BeforeEach  
    void init() {  
    }  
    ...
```

Replaces  
**@Before**

```
@Test  
void succeedingTest() {  
}
```

```
@Test  
void failingTest() {  
    fail("a failing test");  
}
```

Replaces  
**@Ignore**

```
@Test  
@Disabled("for demo purposes")  
void skippedTest() {  
    // not executed  
}
```



# Agenda

- JUnit 5
- **Integration Testing with Spring**
- Testing with Profiles
- Testing with Databases
- Lab
- Appendices
  - JUnit 5, JUnit 4, Stubs & Mocks



# Unit Testing

## Unit Testing *Without Spring*

- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
  - Uses simplified alternatives for dependencies
    - Stubs and/or Mocks
    - *See Appendix for more details*



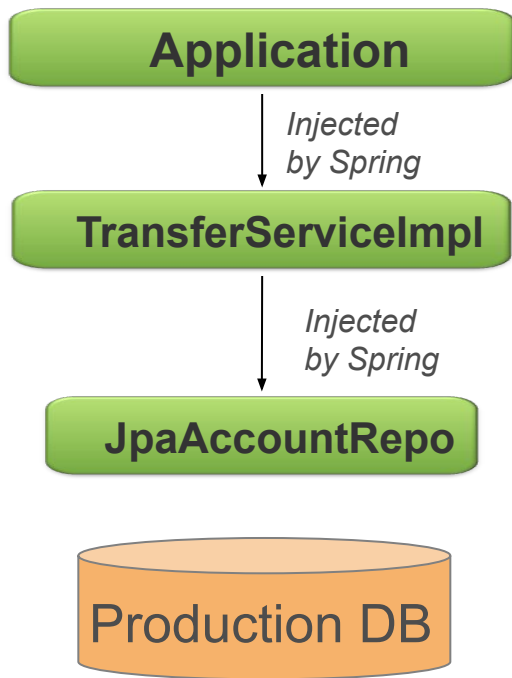
# Integration Testing

## Integration Testing *With Spring*

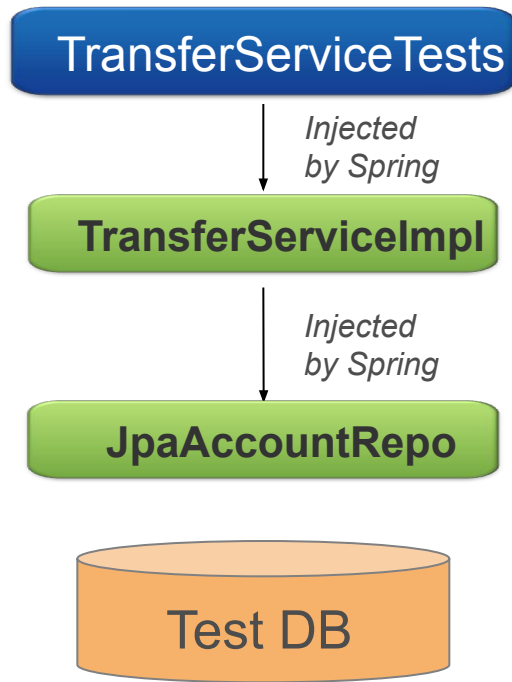
- Integration Testing
  - Tests the interaction of multiple units working together
    - All should work individually (unit tests showed this)
- Tests application classes in context of their surrounding infrastructure
  - Out-of-container testing, no need to run up full App. Server
  - Infrastructure may be “scaled down”
    - Use Apache DBCP connection pool instead of container-provider pool obtained through JNDI
    - Use ActiveMQ to save expensive commercial messaging server licenses

# Integration Test Example

- Production mode



- Integration test



# Spring Support for Testing

- Spring has rich testing support
  - Based on **TestContext** framework
    - Defines an **ApplicationContext** for your tests
  - **@ContextConfiguration**
    - Defines the Spring configuration to use
- Packaged as a separate module
  - **spring-test.jar**



<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#integration-testing>  
<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testcontext-framework>

## @ExtendWith in JUnit 5

- JUnit 5 has extensible architecture via @ExtendWith
  - Replaces JUnit 4's @RunWith
  - JUnit 5 supports *multiple* extensions - hence the name
- Spring's extension point is the **SpringExtension** class
  - A Spring aware test-runner



See *Appendix* (end of this section) for Spring's JUnit 4 support.

# Using Spring's Test Support

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes=SystemTestConfig.class)

public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void shouldTransferMoneySuccessfully() {
        TransferConfirmation conf = transferService.transfer(...);
        ...
    }
}
```

Run with Spring support

Point to system test configuration file(s)

Inject bean to test

Test the system as normal

No need for *@BeforeEach* method for creating *TransferService*

# @SpringJUnitConfig



- @SpringJUnitConfig is a “*composed*” annotation that combines
  - @ExtendWith(SpringExtension.class) from JUnit 5
  - @ContextConfiguration from Spring

**Recommended:** Use  
*composed* annotation

```
@SpringJUnitConfig(SystemTestConfig.class)
public class TransferServiceTests {
    ...
}
```

# Alternative Autowiring for Tests

```
@SpringJUnitConfig(SystemTestConfig.class)
```

```
public class TransferServiceTests {
```

```
    // @Autowired
```

```
    // private TransferService transferService;
```

```
    @Test
```

```
    public void shouldTransferMoneySuccessfully
```

```
        (@Autowired TransferService transferService) {
```

```
        TransferConfirmation conf = transferService.transfer(...);
```

```
        ...
```

```
    }
```

```
}
```

No longer required –  
dependency injected as  
test method *argument*

## NOTES:

- Test works either way – your choice
- This use of Autowired *unique to tests*



# Including Configuration as an Inner Class

```
@SpringJUnit4Config
public class JdbcAccountRepoTest {

    @Test
    public void shouldUpdateDatabaseSuccessfully() {...}

    @Configuration
    @Import(SystemTestConfig.class)
    static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

Don't specify any  
config classes

Looks for configuration  
*embedded* in test class

Override a bean with  
a test alternative

# Multiple Test Methods

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() { ... }

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

ApplicationContext  
instantiated only *once*

Both tests share *same*  
cached **Application-  
Context** & use *same*  
**TransferService** bean



Most Spring Beans are *stateless/immutable* singletons, never modified during any test. No need for a new context for each test.

# Dirty Context

- Forces context to be closed at end of test method
  - Allows testing of `@PreDestroy` behavior
- Next test gets a *new* Application Context
  - Cached context destroyed, new context cached instead

```
@Test
@DirtyContext
public void testTransferLimitExceeded() {
    transferService.setMaxTransfers(0);
    ... // Do a transfer, expect a failure
}
```

Context closed  
and destroyed at  
end of test

# Test Property Sources

- Custom properties *just* for testing
  - Specify one or more properties
    - Has higher precedence than sources
  - Specify location of one or more properties files to load
    - Defaults to looking for `[classname].properties`

```
@SpringJUnitConfig(SystemTestConfig.class)
@TestPropertySource(properties = { "username=foo", "password=bar" }
                    locations = "classpath:/transfer-test.properties")
public class TransferServiceTests {
    ...
}
```

# Benefits of Testing with Spring

- No need to deploy to an external container to test application functionality
  - Run everything quickly inside your IDE
  - Supports *Continuous Integration* testing
- Allows reuse of your configuration between test and production environments
  - Application configuration logic is typically reused
  - Infrastructure configuration is environment-specific
    - DataSources
    - JMS Queues

# Agenda

- JUnit 5
- Integration Testing with Spring
- **Testing with Profiles**
- Testing with Databases
- Lab
- Appendices
  - JUnit 5, JUnit 4, Stubs & Mocks



# Activating Profiles For a Test

- **@ActiveProfiles** inside the test class
  - Define one or more profiles
  - Beans associated with that profile are instantiated
  - Also beans not associated with *any* profile
- Example: Two profiles activated – *jdbc* **and** *dev*

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )

public class TransferServiceTests { ... }
```



# Profiles Activation with JavaConfig

- **@Profile** on *@Configuration* class or any of its *@Bean* methods

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests
{...}
```

```
@Configuration
@Profile("jdbc")
public class DevConfig {

    @Bean
    public ... {...}

}
```

```
@Configuration
public class DevConfig {

    @Profile("jdbc")
    @Bean
    public ... {...}

}
```



Only beans matching an active profile or with *no* profile are loaded

# Profiles Activation with Annotations

- **@Profile** on a *Component* class

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests {
    ...
}
```

```
@Repository
@Profile("jdbc")
public class JdbcAccountRepository {
    ...
}
```



Only beans with current profile / no profile are component-scanned

# Agenda

- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- **Testing with Databases**
- Lab
- Appendices
  - JUnit 5, JUnit 4, Stubs & Mock



# Testing with Databases

- Integration testing against SQL database is common
- In-memory databases useful for this kind of testing
  - No prior install needed
- Common requirement: populate DB before test runs
  - Use the `@Sql` annotation:

```
@Test
@Sql ( "/testfiles/test-data.sql" )
public void successfulTransfer() {
    ...
}
```

Run this SQL script *before* this test method executes.

# @Sql Examples

Run these scripts before *each* **@Test** method  
*unless* a method is annotated with its own **@Sql**

```
@SpringJUnitConfig(...)
@Sql( { "/testfiles/schema.sql", "/testfiles/load-data.sql" } )
public class MainTests {

    // schema.sql and load-data.sql only run before this test
    @Test
    public void success() { ... }

    @Test // Overrides to use own scripts
    @Sql ( scripts="/testfiles/setupBadTransfer.sql" )
    @Sql ( scripts="/testfiles/cleanup.sql",
            executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )
    public void transferError() { ... }
}
```

Run *before* **@Test** method

... run *after*  
**@Test** method

# @Sql Options

- When/how does the SQL run?
  - *executionPhase*: before (default) or after test method
  - *config*: Options to control SQL scripts
    - What to do if script fails? **FAIL\_ON\_ERROR**, **CONTINUE\_ON\_ERROR**, **IGNORE\_FAILED\_DROPS**, **DEFAULT\***
    - SQL syntax control: comments, statement separator

```
@Sql( scripts = "/test-user-data.sql",  
      config = @SqlConfig(errorMode = ErrorMode.FAIL_ON_ERROR,  
                          commentPrefix = "//", separator = "@@") )
```

**\*DEFAULT** = whatever @Sql defines at class level, otherwise **FAIL\_ON\_ERROR**

# Summary

- Testing is an *essential* part of any development
- Unit testing tests a class in isolation
  - External dependencies should be minimized
  - Consider creating stubs or mocks to unit test
  - *You don't need Spring to unit test*
- Integration testing tests the interaction of multiple units working together
  - Spring provides good integration testing support
  - Profiles for different test & deployment configurations
  - Built-in support for testing with Databases



A man with a beard and a woman are sitting at a desk, looking at a computer monitor. The man is pointing at the screen. The background is slightly blurred, showing other people in an office.

## ***Lab: Testing Spring Application***

**Lab project:  
24-test**

**Anticipated Lab time:  
30 Minutes**

**Optional Topics:** Appendices on JUnit 5, JUnit 4,  
Unit testing using Stubs and Mocks

# Agenda

- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- **Appendices**
  - **More on JUnit 5**
  - JUnit 4 Support
  - Unit Testing (Stubs & Mocks)



# JUnit 4 vs JUnit 5

- JUnit 5 enhances JUnit framework
  - New features are added over JUnit 4
- JUnit 5 supports multiple extensions
  - Solves “single Runner limitation” of JUnit 4



# JUnit 5: New Assertions – 1

- Introduces new assertions
  - `assertThrows`  
`(<exception.class>, <lambda-expression>)`
  - `assertTimeout`  
`(<duration>, <lambda-expression>)`
  - `assertAll`  
`(<description>, <multiple-assertions>)`
- Improves assumptions
  - Uses lambda expression
    - The code to be tested

## JUnit 5: New Assertions – 2

### *assertThrows(..) & assertTimeout(..)*

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
        () -> { /* Perform task that throws illegal argument exception */ }
    );
    assertEquals("some error message", exception.getMessage());
}
```

No need to specify  
expected attribute

```
@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}
```

No need to specify  
timeout attribute

## JUnit 5: New Assertions – 3

```
@Test
void standardAssertions() {
    assertEquals(2, 2);
    assertEquals(4, 4, "Optional assertion message is now last parameter.");
    assertTrue(2 == 2, () -> "Assertion messages can be lazily evaluated -- "
        + "to avoid constructing complex messages unnecessarily.");
}
```

```
@Test
void groupedAssertions() {
    // In a grouped assertion all assertions are executed, and any
    // failures will be reported together.
    assertAll("person",
        () -> assertEquals("John", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName())
    );
}
```

## JUnit 5: Assumptions – 1

- Run test only if a condition is true
  - Typically in a particular environment/platform/operating system

```
@Test void testOnlyOnCiServer() {  
    assertTrue("CI".equals(System.getenv("ENV")));  
    // Remainder of this test  
}
```

Only run test in  
Continuous  
Integration  
environment

```
@Test void testOnlyOnDeveloperWorkstation() {  
    assertTrue("DEV".equals(System.getenv("ENV")),  
        () -> "Aborting test: not on developer workstation");  
    // Remainder of test  
}
```

Only run test in  
Dev, return an  
error message  
otherwise



## JUnit 5: Assumptions – 2

- Only run test code if a condition is true

Run this part *only* in a CI environment

```
@Test void testInAllEnvironments() {  
    assumingThat("CI".equals(System.getenv("ENV")),  
        () -> {  
        // perform these assertions only on the CI server  
        assertEquals(2, 2);  
        });  
  
    // perform these assertions in all environments  
    assertEquals("a string", "a string");  
}
```

# Agenda

- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- **Appendices**
  - More on JUnit 5
  - **JUnit 4 Support**
  - Unit Testing (Stubs & Mocks)



# Spring's JUnit 4 Support

- Packaged as a separate module
  - `spring-test.jar`
- Consists of several JUnit test support classes
- Central support class is *SpringJUnit4ClassRunner*
  - Caches a *shared* ApplicationContext across test methods



See: [Spring Framework Reference – Integration Testing](https://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/htmlsingle/#integration-testing)

<https://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/htmlsingle/#integration-testing>

# Using Spring with JUnit 4

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void shouldTransferMoneySuccessfully() {
        TransferConfirmation conf = transferService.transfer(...);
        ...
    }
}
```

Run with Spring support

Point to system  
test configuration file

Inject bean to test

Test the system as normal

No need for @Before method

# Including Configuration as an Inner Class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class JdbcAccountRepoTest {

    private JdbcAccountRepo repo = ...;

    @Test
    public void shouldUpdateDatabaseSuccessfully() {...}

    @Configuration
    @Import(SystemTestConfig.class)
    static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

Don't specify any  
config classes

Looks for  
configuration  
*embedded* in test  
class

Override a bean  
with an alternative  
for testing

# Multiple Test Methods

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=SystemTestConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() { ... }

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

ApplicationContext  
instantiated only *once*

Both tests share *same* cached  
ApplicationContext - use *same*  
transferService bean

Add **@DirtyContext**  
to test method - forces *new*  
context for next test

# Alternative Runner Class

- Can use **SpringRunner** as an alternative to the **SpringJUnit4ClassRunner**
  - Simply a sub-class with a nicer name
  - Available from Spring 4.3

```
@RunWith(SpringRunner.class)
@ContextConfiguration(SystemTestConfig.class)
public class TransferServiceTests {
    ...
}
```



# Agenda

- JUnit 5
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases
- Lab
- **Appendices**
  - More on JUnit 5
  - JUnit 4 Support
  - **Unit Testing (Stubs & Mocks)**





# Unit Testing vs. Integration Testing

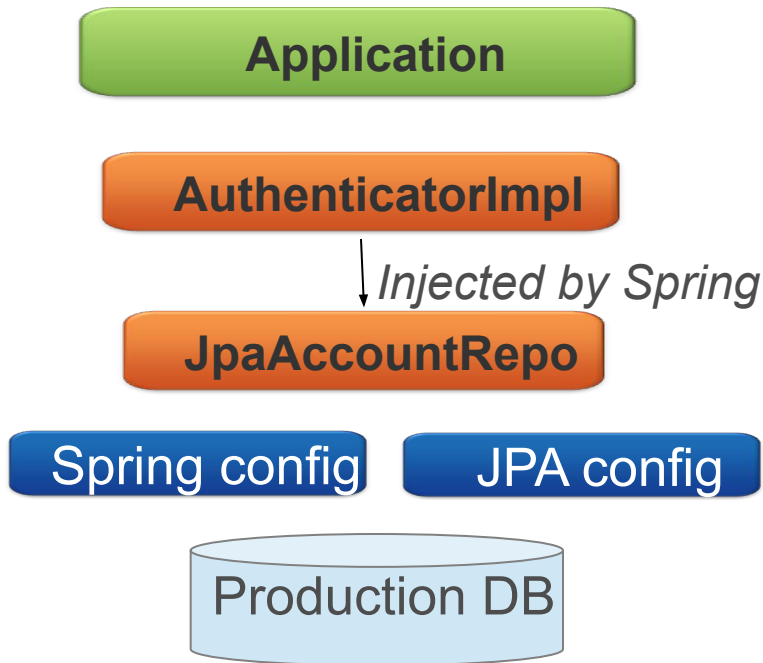
- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
- Integration Testing
  - Tests the interaction of multiple units working together
  - Integrates infrastructure
- Discussed Integration Testing earlier
  - Let's discuss Unit Testing here
  - Remember: *Unit Testing does not use Spring*

# Unit Testing

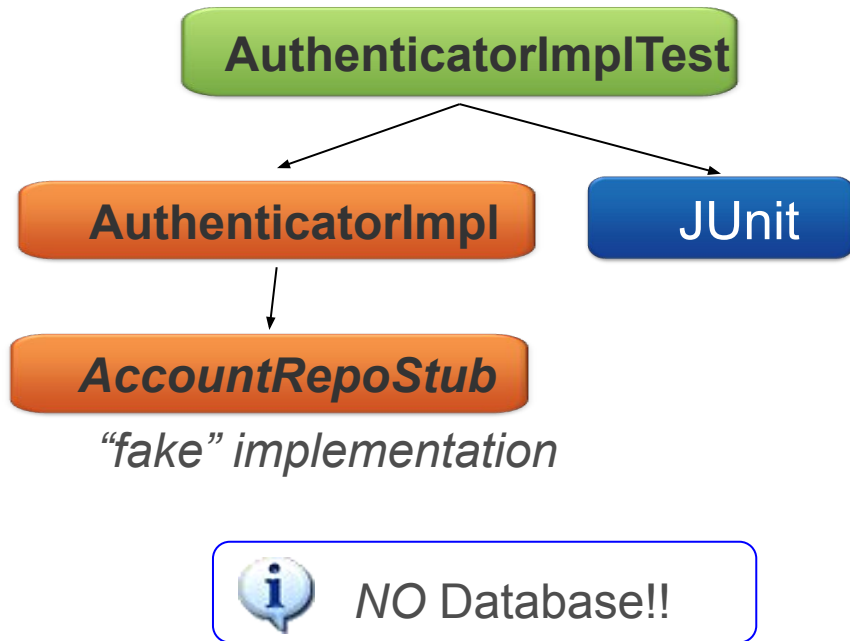
- Remove links with dependencies
  - The test shouldn't fail because of failure in external dependencies
  - Spring is also considered as a dependency
- 2 ways to create a “testing-purpose” implementation of your dependencies:
  - Stubs Create a simple test implementation
  - Mocks Dependency class generated at startup-time using a “Mocking framework”

# Unit Testing example

- Production mode



- Unit test with Stubs



# Example Unit to be Tested

**Note:** Validation failure paths ignored for simplicity

```
public class AuthenticatorImpl implements Authenticator {  
    private AccountRepository accountRepository;  
  
    public AuthenticatorImpl(AccountRepository accountRepository) {  
        this.accountRepository = accountRepository;  
    }  
  
    public boolean authenticate(String username, String password) {  
        Account account = accountRepository.getAccount(username);  
  
        return account.getPassword().equals(password);  
    }  
}
```

External dependency

Unit *business* logic  
– 2 paths: success or fail

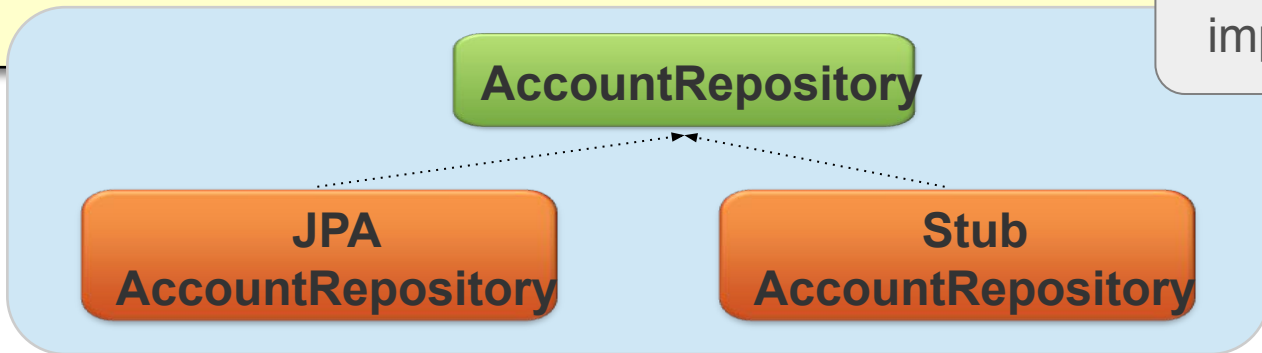
# Implementing a Stub

- Class created manually
  - Implements Business interface

- Only need *one* Account for testing
- Test *must only* use valid data (can only use “lisa”)

```
class StubAccountRepository implements AccountRepository {  
    public Account getAccount(String user) {  
        return “lisa”.equals(user) ? new Account(“lisa”, “secret”) : null;  
    }  
}
```

Simplest *minimal* implementation



# Unit Test using a Stub

Lisa is the *only* test user

```
public class AuthenticatorImplTests {  
    private AuthenticatorImpl authenticator;  
  
    @BeforeEach public void setUp() {  
        authenticator = new AuthenticatorImpl( new StubAccountRepository() );  
    }  
  
    @Test public void successfulAuthentication() {  
        assertTrue(authenticator.authenticate("lisa", "secret"));  
    }  
  
    @Test public void invalidPassword() {  
        assertFalse(authenticator.authenticate("lisa", "invalid"));  
    }  
}
```

Spring **not** in charge of injecting dependencies

OK scenario

KO scenario

# Unit Testing with Stubs

- Advantages
  - Easy to implement and understand
  - Reusable
- Disadvantages
  - Change to an interface requires change to stub
  - Your stub must implement all methods
    - even those not used by a specific scenario
  - If a stub is reused refactoring can break other tests

# Steps to Testing with a Mock

1. Use a mocking library to generate a mock object
  - Implements the dependent interface on-the-fly
2. Record the mock with expectations of how it will be used for a scenario
  - What methods will be called
  - What values to return
3. Exercise the scenario
4. Verify mock expectations were met



# Example: Using a Mock - I

EasyMock

- Setup
  - A Mock class is created at startup time

```
import static org.easymock.classextensions.EasyMock.*;  
  
public class AuthenticatorImplTests {  
    private AccountRepository accountRepository  
        = createMock(AccountRepository.class);  
  
    private AuthenticatorImpl authenticator  
        = new AuthenticatorImpl(accountRepository);  
  
    // continued on next slide ...
```

Static import

Creates an implementation of  
interface *AccountRepository*

## Example: Using a Mock - II

EasyMock

// ... continued from previous slide

```
@Test public void validUserWithCorrectPassword() {  
    expect(accountRepository.getAccount("lisa")).  
        andReturn(new Account("lisa", "secret"));
```

```
    replay(accountRepository);
```

```
    assertTrue( authenticator.  
        authenticate("lisa", "secret") );
```

```
    verify(accountRepository);
```

```
}
```

```
}
```

**Recording**

*What behavior to expect?*

*Recording → Playback*

**Playback  
mode - run test**

*Mock now fully available*

**Verification**

*No planned method call  
has been omitted*

# Same Example using Mockito

Mockito

No *replay* step  
with Mockito

```
import static org.mockito.Mockito.*;

public class AuthenticatorImplTests {
    private AccountRepository accountRepository
        = mock( AccountRepository.class );           // Create a mock object
    private AuthenticatorImpl authenticator
        = new AuthenticatorImpl(accountRepository);   // Inject the mock object

    @Test public void validUserWithCorrectPassword() {
        when(accountRepository.getAccount("lisa")).    // Train the mock
            thenReturn(new Account("lisa", "secret"));

        assertTrue( authenticator.authenticate("lisa", "secret") ); // Run test
        verify(accountRepository);                                     // Verify getAccount() was
                                                                    // invoked on the mock
    }
}
```

# Mock Considerations

- Several mocking libraries available
  - Mockito, JMock, EasyMock
- Advantages
  - No additional class to maintain
  - You only need to setup what is necessary for the scenario you are testing
  - Test behavior as well as state
    - Were all mocked methods used? If not, why not?
- Disadvantages
  - A little harder to understand at first

# Mocks or Stubs?

- You will probably use both
- General recommendations
  - Favor mocks for non-trivial interfaces
  - Use stubs when you have simple interfaces with repeated functionality
  - Always consider the specific situation
- Read “Mocks Aren’t Stubs” by Martin Fowler
  - <http://www.martinfowler.com/articles/mocksArentStubs.html>