Spring Start Here

Chapter-15:
Testing your Spring app

Upcode Software
Engineer Team

-2024-

CONTENT

1. Writing correctly implemented tests
2. Implementing tests in Spring apps
   2.1 Implementing unit tests
   2.2 Implementing integration tests
3. Conclusion
4. Reference

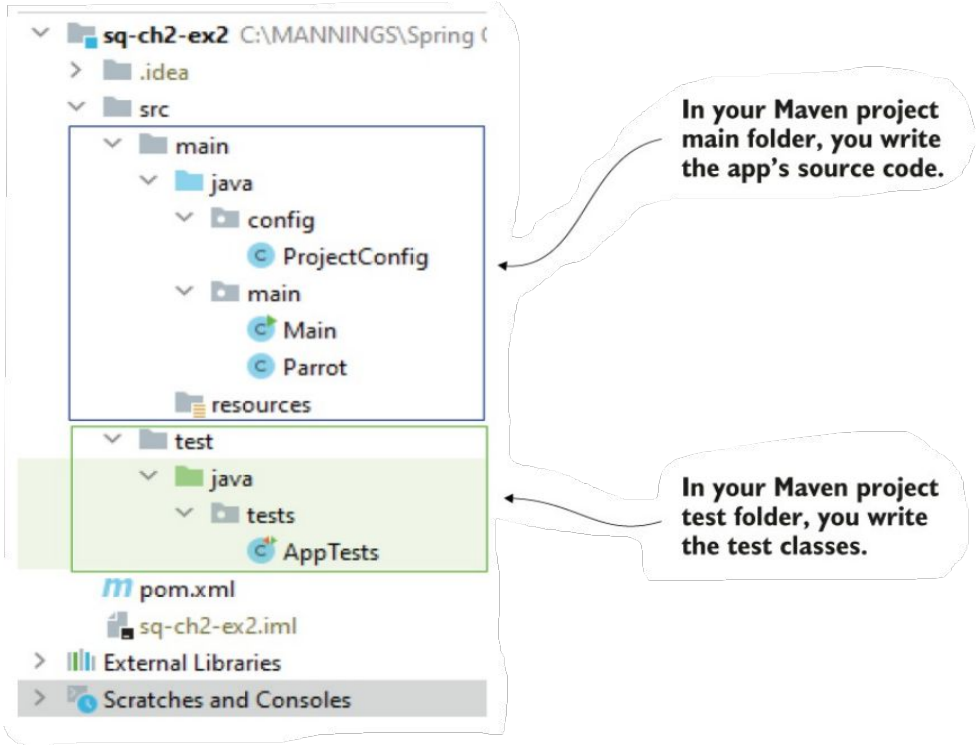
Introduction

- **What is test**

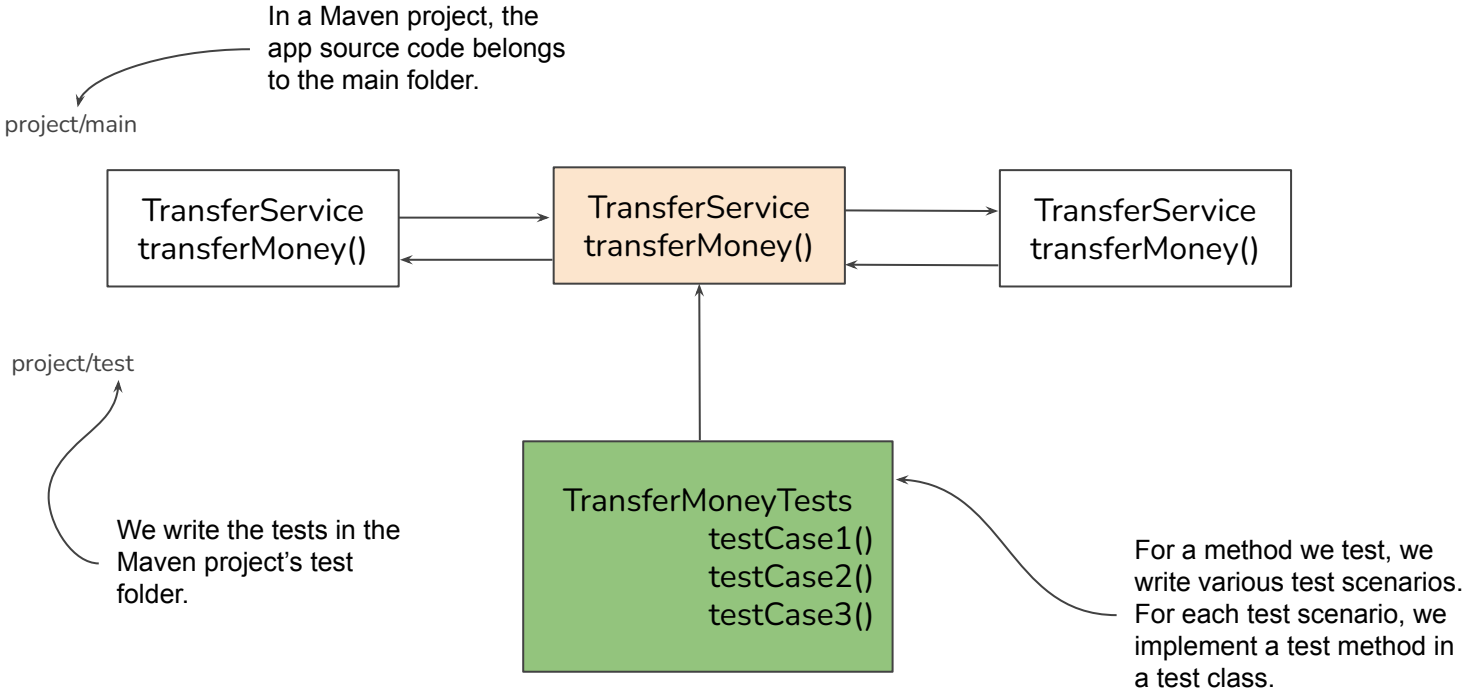
  A test is a small piece of logic whose purpose is to validate that a specific capability our app implements works as expected. We'll classify the tests into two categories:

  - Unit tests—Focus only on an isolated piece of logic
  - Integration tests—Focus on validating that multiple components correctly interact with each other

Writing correctly implemented tests(1/2)



Writing correctly implemented tests(2/2)



Implementing tests in Spring apps

- Writing unit tests to validate a method's logic. The unit tests are short, fast to execute, and focus on only one flow. These tests are a way to focus on validating a small piece of logic by eliminating all the dependencies.

- Writing Spring integration tests to validate a method's logic and its integration with specific capabilities the framework provides. These tests help you make sure your app's capabilities still work when you upgrade dependencies.

Implementing unit tests(1/23)

- For the first unit test, we'll build a wire transfer project

TransferRequest.java

```
4   import lombok.Setter;
5
6   import java.math.BigDecimal;
7
8   @Getter  6 usages
9   @Setter
10
11
12  public class TransferRequest {
13      private long senderAccountId;
14      private long receiverAccountId;
15      private BigDecimal amount;
16  }
```

request class for receiving transfer details

Implementing unit tests(2/23)

Account class for Account data

```
@Getter  9 usages
@Setter
@Entity
public class Account {
    @Id
    private long id;
    private String name;
    private BigDecimal amount;
}
```

AccountRepository class for communication with DataBase

```
@Repository  3 usages
public interface AccountRepository
        extends CrudRepository<Account , Long> {
    List<Account> findAllByName(String name);  no usages
    @Query("SELECT * FROM account " +  1 usage
            "WHERE name = :name")
    List<Account> findAccountsByName(String name);
    @Modifying  2 usages
    @Query("UPDATE account " +
            "SET amount = :amount " +
            "WHERE id = :id")
    void changeAmount(long id, BigDecimal amount);
}
```

Implementing unit tests(3/23)

- AccountController class for transfer details

```java
@RestController
public class AccountController {
    private final TransferServiceInterface transferService;  2 usages

    public AccountController(TransferService transferService) {
        this.transferService = transferService;
    }

    @PostMapping("/transfer")
    public ResponseEntity<?> transferMoney(@RequestBody TransferRequest request) {
        try {
            transferService.transferMoney(request);
        } catch (AccountNotFoundException e) {
            throw new RuntimeException(e);
        }
        return ResponseEntity.ok().build();
    }
}
```

Implementing unit tests(4/23)

- Service class to write the logic of the money transfer

```java
@Service
public class TransferService implements TransferServiceInterface {
    private final AccountRepository accountRepository;  5 usages

    public TransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }
    @Override  2 usages
    @Transactional
    public void transferMoney(long senderAccountId, long receiverAccountId, BigDecimal amount)
            throws AccountNotFoundException {

        Account sender = accountRepository.findById(senderAccountId)
                .orElseThrow(() -> new AccountNotFoundException("Account sen not found"));
        Account receiver = accountRepository.findById(receiverAccountId)
                .orElseThrow(() -> new AccountNotFoundException("Account des not found"));

        BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
        BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

        accountRepository.changeAmount(senderAccountId, senderNewAmount);
        accountRepository.changeAmount(receiverAccountId, receiverNewAmount);
    }
}
```

We find the details of the sender and receiver account.

We calculate the accounts' amounts.

We update the new amount in the sender and receiver account.

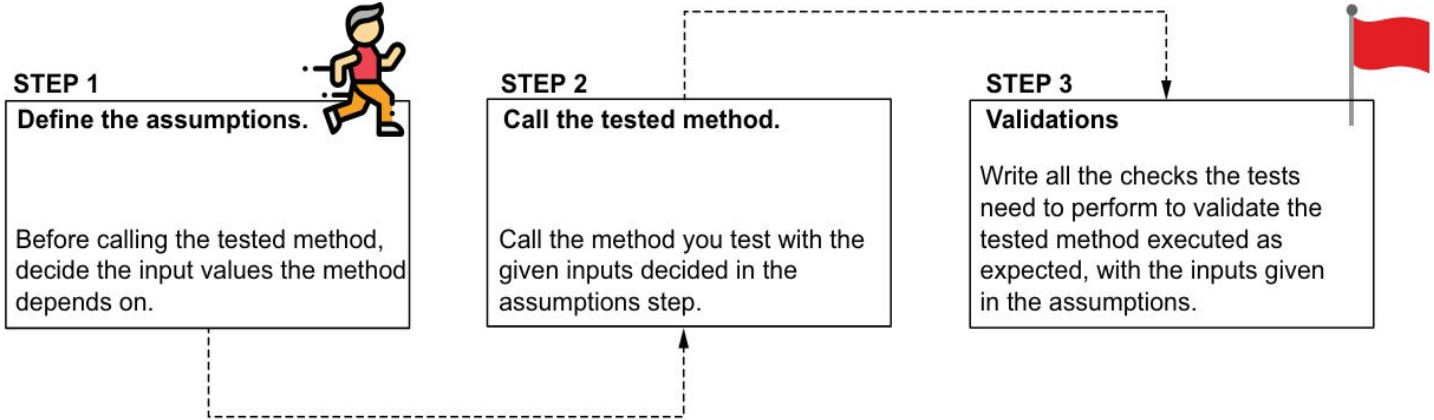
Implementing unit tests(5/23)

One of the scenarios (usually the first one for which we write tests), is the “happy flow”: the scenario in which no exception or error occurs.

```
    @Transactional
   public void transferMoney(long senderAccountId, long receiverAccountId, BigDecimal amount)
 throws AccountNotFoundException {
      BigDecimal amount = request.getAmount();
      Account sender = accountRepository.findById(request.getSenderAccountId())
        .orElseThrow(() -> new AccountNotFoundException("Account sen not found"));

      Account receiver = accountRepository.findById(request.getReceiverAccountId())
        .orElseThrow(() -> new AccountNotFoundException("Account des not found"));

      BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
      BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

      accountRepository.changeAmount(request.getSenderAccountId(), senderNewAmount);
      accountRepository.changeAmount(request.getReceiverAccountId(), receiverNewAmount);
    }
}
```

Implementing unit tests(6/23)

- Any test has three main parts



Implementing unit tests(7/23)

1. Assumptions(Given)—We need to define any input and find any dependency of the logic we need to control to achieve the desired flow scenario. For this point, we'll answer the following questions: what inputs should we provide, and how should dependencies behave for the tested logic to act in the specific way we want?

2. Call/Execution(When)—We need to call the logic we test to validate its behavior.

3. Validations(Then)—We need to define all the validations that need to be done for the given piece of logic. We'll answer this question: what should happen when this piece of logic is called in the given conditions?

Implementing unit tests(8/23)

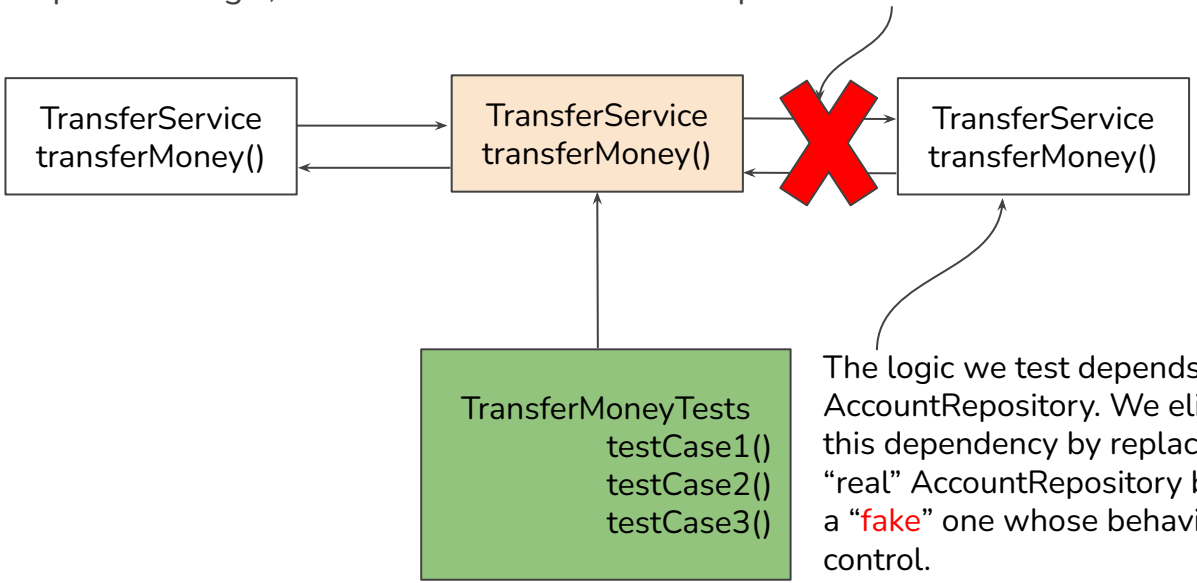
Parameters are execution dependencies. Based on their value, the method might behave one way or another

```
    @Transactional
    public void transferMoney(long senderAccountId, long receiverAccountId, BigDecimal amount)
            throws AccountNotFoundException {

        Account sender = accountRepository.findById(senderAccountId)
                .orElseThrow(() -> new AccountNotFoundException("Account sen not found"));
        Account receiver = accountRepository.findById(receiverAccountId)
                .orElseThrow(() -> new AccountNotFoundException("Account des not found"));

        BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
        BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

        accountRepository.changeAmount(senderAccountId, senderNewAmount);
        accountRepository.changeAmount(receiverAccountId, receiverNewAmount);
    }
}
```

Other objects external to the method, but that the method uses to implement its logic are also execution dependencies. Based on their behavior, the method might behave one way or another

Implementing unit tests(9/23)

A unit test only focuses on testing a certain piece of logic. To have the test validate only one component's logic, we need to remove all the dependencies.



The logic we test depends on the AccountRepository. We eliminate this dependency by replacing the "real" AccountRepository bean with a "fake" one whose behavior we control.

Implementing unit tests(10/23)

We use the Mockito mock() method to create a mock instance for the AccountRepository object.

```
public class TransferServiceUnitTest {
    @Test
    @DisplayName(" Test to transferMoney  ")
    public void transferServiceTest() {
        AccountRepository accountRepository =
                mock(AccountRepository.class);
        TransferService transferService =
                new TransferService(accountRepository);

    }
}
```

We create an instance of the TransferService object whose method we want to test. Instead of a real AccountRepository instance, we create the object using a mock AccountRepository. This way, we replace the dependency with something we can control.

Implementing unit tests(11/23)

We control the mock's findById() method to return the sender account instance when it gets the sender account ID.

```
public class TransferServiceUnitTest {
    @Test
    @DisplayName(" Test to transferMoney  ")
    public void transferServiceTest() {
        AccountRepository accountRepository =
                mock(AccountRepository.class);
        TransferService transferService =
                new TransferService(accountRepository);
        Account sender = new Account();
        sender.setId(1L);
        sender.setAmount(new BigDecimal( val: 1000));
        Account destination = new Account();
        destination.setId(2L);
        destination.setAmount(new BigDecimal( val: 1000));
        given(accountRepository.findById(sender.getId()))
                .willReturn(Optional.of(sender));
        given(accountRepository.findById(destination.getId()))
                .willReturn(Optional.of(destination));
        try {
            transferService.transferMoney(
                    sender.getId(),
                    destination.getId(),
                    BigDecimal.valueOf(300)
            );
```

We create the sender and the destination Account instances, which hold the Account details, which we assume the app would find in the database.

We control the mock's findById() method to return the destination account instance when it gets the destination account ID. You can read this line as "If one calls the findById() method with the destination ID parameter, then return the destination account instance.

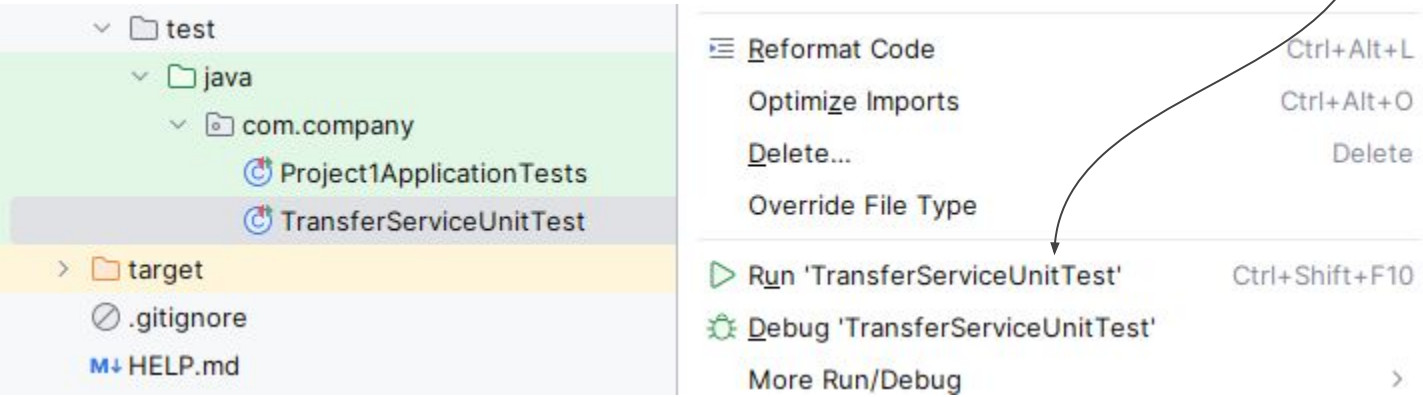
Implementing unit tests(12/23)

```java
public void transferServiceTest() {
    AccountRepository accountRepository =
            mock(AccountRepository.class);
    TransferService transferService =
            new TransferService(accountRepository);
    Account sender = new Account();
    sender.setId(1L);
    sender.setAmount(new BigDecimal( val: 1000));
    Account destination = new Account();
    destination.setId(2L);
    destination.setAmount(new BigDecimal( val: 1000));
    given(accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));
    given(accountRepository.findById(destination.getId()))
            .willReturn(Optional.of(destination));
    try {
        transferService.transferMoney(
                sender.getId(),
                destination.getId(),
                BigDecimal.valueOf(300)
        );
        verify(accountRepository)
                .changeAmount( id: 1L, new BigDecimal( val: 700));
        verify(accountRepository)
                .changeAmount( id: 2L, new BigDecimal( val: 1300));
    } catch (AccountNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Verify that the changeAmount() method in the AccountRepository was called with the expected parameters.

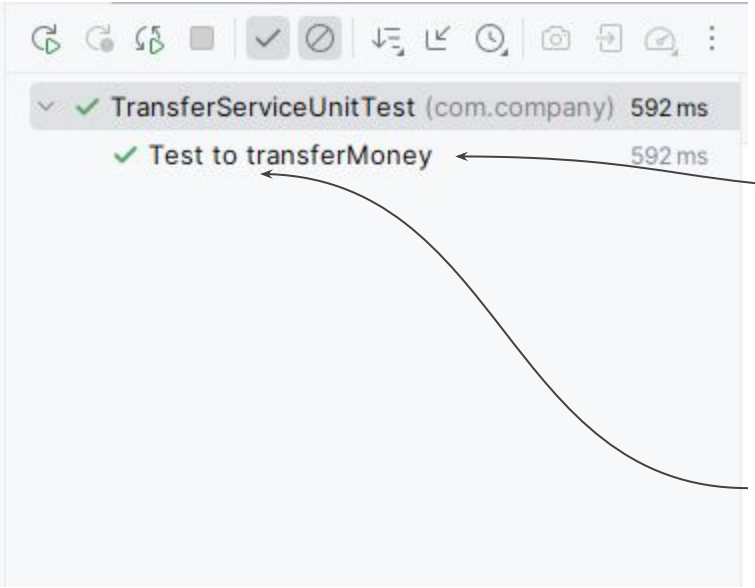
Implementing unit tests(13/23)

In any IDE, a simple way to run a test suite is to right-click on the test class and select “Run.”



Implementing unit tests(14/23)

- Running a test. An IDE usually offers several ways you can run a test



The IDE displays the results as shown in this image. A green check means the test passed. If the IDE displayed a red or yellow X, it means the tests failed. When a test fails, you can check the app's console to find out more about the reason it failed.

This is a @DisplayName

Implementing unit tests(15/23)

- **Using annotations for mock dependencies(1/2)**

```java
@ExtendWith(MockitoExtension.class)
public class TransferServiceUnitTest {
    @Mock  4 usages
    private  AccountRepository accountRepository;

    @InjectMocks  1 usage
    TransferService transferService ;

    @Test
    @DisplayName(" Test to transferMoney  ")
    public void transferServiceTest() {

        Account sender = new Account();
        sender.setId(1L);
        sender.setAmount(new BigDecimal( val: 1000));
        Account destination = new Account();
        destination.setId(2L);
        destination.setAmount(new BigDecimal( val: 1000));
        given(accountRepository.findById(sender.getId()))
                .willReturn(Optional.of(sender));
        given(accountRepository.findById(destination.getId()))
                .willReturn(Optional.of(destination));
        try {
            transferService.transferMoney(
                    sender.getId(),
                    destination.getId(),
                    BigDecimal.valueOf(300)
            );
```

Enable the use of @Mock and @InjectMocks annotations.

Use the @Mock annotation to create a mock object and inject it into the test class's annotated field.

Use the @InjectMocks to create the tested object and inject it into the class's annotated field.

Implementing unit tests(16/23)

- ## Using annotations for mock dependencies(2/2)

- @Mock: This annotation creates a mock object for the field it annotates. A mock object is a simulated object that mimics the behavior of real objects in controlled ways, allowing you to specify how it should behave during tests.
- @InjectMocks: This annotation creates an instance of the class you are testing and injects the mock objects (created with @Mock) into it, either through the constructor or setters. This allows you to test the class with its dependencies mocked.
- @ExtendWith(MockitoExtension.class): This annotation enables the Mockito extension which allows the framework to read the @Mock and @InjectMocks annotations and manage the annotated fields. This extension initializes the mocks and injects them into the fields as needed.

Implementing unit tests(17/23)

```
@ExtendWith(MockitoExtension.class)
public class TransferServiceUnitTest {
  @Mock
  private  AccountRepository accountRepository;
  @InjectMocks
  TransferService transferService ;

  @Test
  @DisplayName(" Test to transferMoney  ")
  public void transferServiceTest() {

  Account sender = new Account();
      sender.setId(1L);
      sender.setAmount(new BigDecimal(1000));
      Account destination = new Account();
      destination.setId(2L);
      destination.setAmount(new BigDecimal(1000));
      given(accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));
      given(accountRepository.findById(destination.getId()))
            .willReturn(Optional.of(destination));
     try {
        transferService.transferMoney(sender.getId(),destination.getId(), BigDecimal.valueOf(300));

        verify(accountRepository).changeAmount(1L, new BigDecimal(700));
        verify(accountRepository).changeAmount(2L, new BigDecimal(1300));
     } catch (AccountNotFoundException e) {
        throw new RuntimeException(e);
     }
  }
}
```

1 Assumptions—Enumerate and control the dependencies.

2 Call—Execute the tested method.

3 Validations—Verify the executed method had the expected behavior.

Implementing unit tests(18/23)

The happy flow is not the only one our tests should cover. We need to identify all the executuion scenarios that are relevant to our use case and implement tests to validate the app's behavior. For example, what can we expect if the receiver account details cannot be fetched? In this case, we expect the app to throw a specific exception.

```
@Transactional
public void transferMoney(
  long idSender,
  long idReceiver,
  BigDecimal amount) {

  Account sender = accountRepository.findById(idSender)
    .orElseThrow(() -> new AccountNotFoundException());

  Account receiver = accountRepository.findById(idReceiver)
    .orElseThrow(() -> new AccountNotFoundException());

  BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
  BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

  accountRepository.changeAmount(idSender, senderNewAmount);
  accountRepository.changeAmount(idReceiver, receiverNewAmount);
}
```

In such a scenario where fetching the receiver account details fails, the calculations of the new amounts and the accounts' update should no longer happen.

Implementing unit tests(19/23)

- assertThrows() function

1. The assertThrows() function is an important part of the Spring testing framework. It is used to verify that an expected exception is thrown and that the exception is of the expected type.
2. The main parameters of the assertThrows() function are:
   Class<? extends Throwable> - the expected type of the exception.
   Executable - the code block where the exception is expected to be thrown.

Implementing unit tests(20/23)

```java
@ExtendWith(MockitoExtension.class)
public class TransferServiceUnitTest {
    @Mock  3 usages
    private  AccountRepository accountRepository;
    @InjectMocks  1 usage
    TransferService transferService ;
    @Test
    @DisplayName(" Test to transferMoney  ")
    public void transferServiceTest() {
        Account sender = new Account();
        sender.setId(1L);
        sender.setAmount(new BigDecimal( val: 1000));
        given(accountRepository.findById(sender.getId()))
                .willReturn(Optional.of(sender));
        given(accountRepository.findById(2L))
                .willReturn( t: Optional.empty());
        assertThrows(
                AccountNotFoundException.class,
                () -> transferService.transferMoney( senderAccountId: 1L,  receiverAccountId: 2L, new BigDecimal( val: 300))
        );
        verify(accountRepository, never())
                .changeAmount(anyLong(), any());

    }
}
```

We control the mock AccountRepository to return an empty Optional when the findById() method is called for the destination account.

We assert that the method throws an AccountNotFoundException in the given scenario.

We use the verify() method with the never() conditional to assert that the changeAmount() method hasn't been called.

Implementing unit tests(21/23)

- The implementation of the login controller action we want to unit test

```java
@Controller
public class LoginController {
    private final LoginProcessor loginProcessor;  4 usages

    public LoginController(LoginProcessor loginProcessor) {
        this.loginProcessor = loginProcessor;
    }
    @GetMapping("/")
    public String loginGet() {
        return "login.html";
    }
    @PostMapping ("/")
    public String loginPost(
            @RequestParam String username,
            @RequestParam String password,
            Model model
    ) {
        loginProcessor.setUsername(username);
        loginProcessor.setPassword(password);
        boolean loggedIn = loginProcessor.login();
        if (loggedIn) {
            model.addAttribute( attributeName: "message",  attributeValue: "You are now logged in.");
        } else {
            model.addAttribute( attributeName: "message",  attributeValue: "Login failed!");
        }
        return "login.html";
    }
}
```

Implementing unit tests(22/23)

- Testing the returned value in a unit test

```
@ExtendWith(MockitoExtension.class)
class LoginControllerUnitTests {
    @Mock  2 usages
    private Model model;
    @Mock  1 usage
    private LoginProcessor loginProcessor;
    @InjectMocks  1 usage
    private LoginController loginController;
    @Test
    public void loginPostLoginSucceedsTest() {
        given(loginProcessor.login())
                .willReturn( t: true);
        String result =
                loginController.loginPost( username: "username",  password: "password", model);
        assertEquals( expected: "login.html", result);
        verify(model)
                .addAttribute( attributeName: "message",  attributeValue: "You are now logged in.");
    }
}
```

We define the mock objects and inject them into the instance whose behavior we test.

We control the LoginProcessor mock instance, telling it to return true when its method login() is called.

We verify the tested method returned value

Implementing unit tests(23/23)

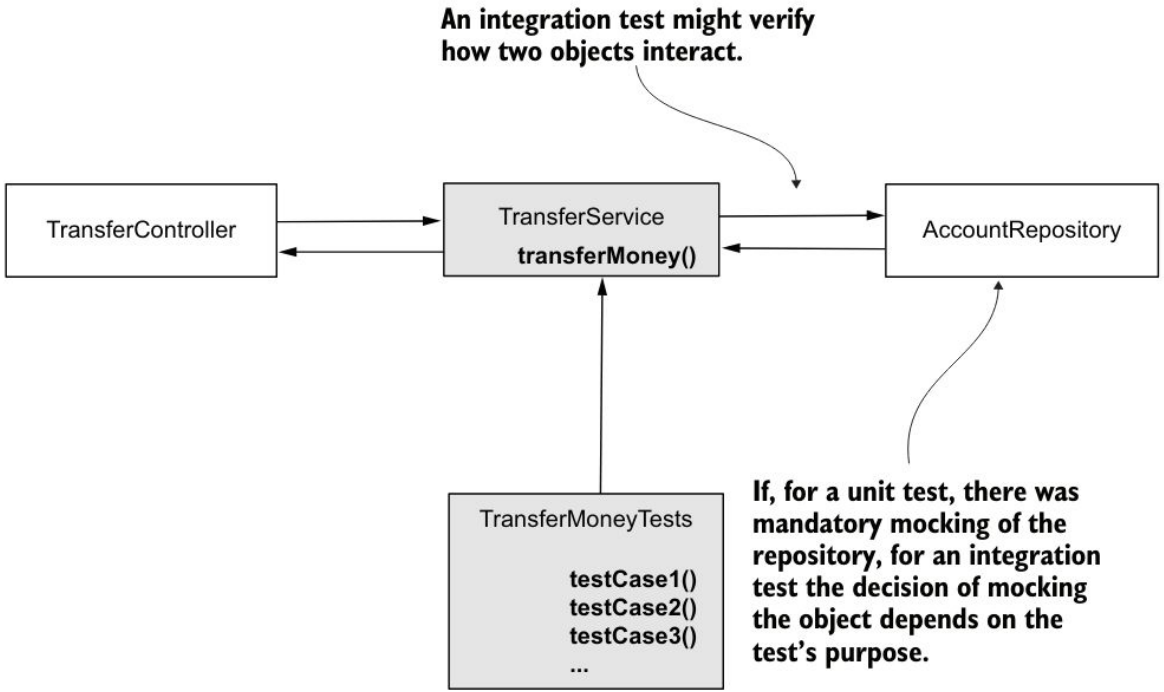
- Adding the test to validate the failed login scenario

```
@Test
public void loginPostLoginFailsTest() {
    given(loginProcessor.login())
            .willReturn( t: false);
    String result =
            loginController.loginPost( username: "username",  password: "password", model);
    assertEquals( expected: "login.html", result);
    verify(model)
            .addAttribute( attributeName: "message",  attributeValue: "Login failed!");
}
```

Implementing integration tests(1/3)

1. *Integration between two (or more) objects of your app.* Testing that the objects interact correctly helps you identify problems in how they collaborate if you change one of them.
2. *Integration of an object of your app with some capability the framework enhances it with.* Testing how an object interacts with some capability the framework provides helps you identify issues that can occur when you upgrade the framework to a new version. The integration test helps you immediately identify if something changed in the framework and the capability the object relies on doesn't work the same way.
3. *Integration of the app with its persistence layer (the database).* Testing how the repository works with the database ensures you quickly identify problems that might occur when upgrading or changing a dependency that helps your app work with persisted data.

Implementing integration tests(2/3)



Implementing integration tests(3/3)

```java
@SpringBootTest
class TransferServiceSpringIntegrationTests {
    @MockBean
    private AccountRepository accountRepository;
    @Autowired
    private TransferService transferService;

    @Test
    void transferServiceTransferAmountTest() {
        Account sender = new Account();
        sender.setId(1);
        sender.setAmount(new BigDecimal( val: 1000));
        Account receiver = new Account();
        receiver.setId(2);
        receiver.setAmount(new BigDecimal( val: 1000));
        when(accountRepository.findById(1L))
                .thenReturn(Optional.of(sender));
        when(accountRepository.findById(2L))
                .thenReturn(Optional.of(receiver));
        try {
            transferService
                    .transferMoney( senderAccountId: 1,  receiverAccountId: 2, new BigDecimal( val: 300));

            verify(accountRepository)
                    .changeAmount( id: 1L, new BigDecimal( val: 700));
            verify(accountRepository)
                    .changeAmount( id: 2L, new BigDecimal( val: 1300));

        } catch (AccountNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Create a mock object that is also part of the Spring context.

Inject the real object from the Spring context whose behavior you'll test.

Define all the assumptions for the test

Call the tested method

Validate the tested method call has the expected behavior.

Conclusion(1/2)

- A test is a small piece of code you write to validate the behavior of certain logic implemented in your app. Tests are necessary because they help you ensure that future app developments don't break existing capabilities. Tests also help as documentation.

- Any test has three main parts:
  – Assumptions(Given)—Define the input values and the way the mock objects behave.
  – Call/Execution(When)—Call the method you want to test.
  – Validations(Then)—Verify the way the tested method behaved.

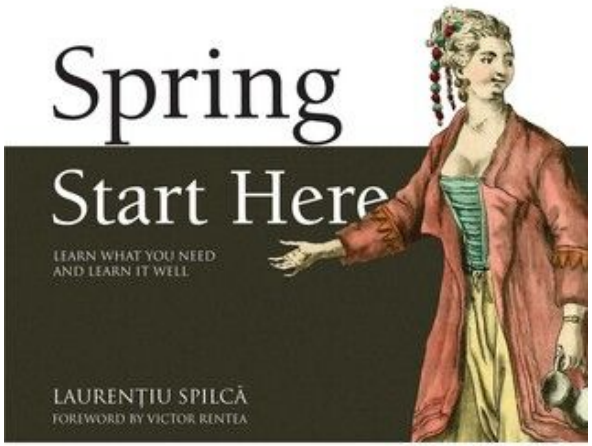
Conclusion(2/2)

- Tests fall into two categories: unit tests and integration tests. Each has its purposes.

  – A unit test only focuses on an isolated piece of logic and validates how one simple component works without checking how it integrates with other features. Unit tests are helpful because they execute fast and point us directly to the problem a specific component might face.

  – An integration test focuses on validating the interaction between two or more components. They are essential because sometimes two components might work correctly in isolation but don’t communicate well. Integration tests help us mitigate problems generated by such cases.

Resources



MANNING

Reference

1. Spring Start Here

Thank you!

Presented by

Qodirov Hudoberdi & Temurmalik Nomozov

(qodirovhudoberdi4@gmail.com)

(temirmaliknomozov@gmail.com)