Testing Compendium

Why?

- You as a developer is **expected to test your code**. It is very important to identify problems and bugs as early as possible in development.

- Well tested code builds confidence in your application.

Our tools:

- In this course we are going to focus on using Junit as our testing framework.
- Later in the course we need to cover some more frameworks like Mockito in order to test our Spring applications.

What is a Unit Test?

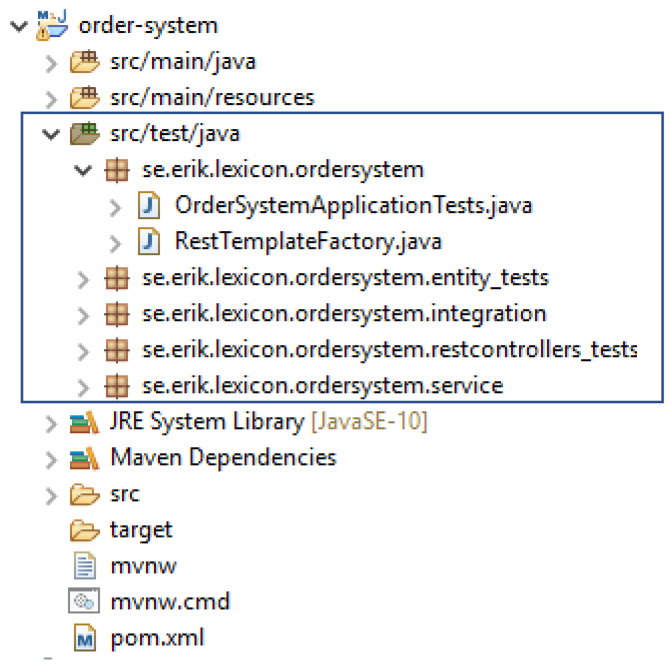
- A unit test is a test that tests a unit. A unit is the smallest testable part of the application.

A unit basically means a method!

Where do I write my tests?

Write your tests separate from your application code.

Maven by default create a special location where you should write your tests.



```
order-system
  src/main/java
  src/main/resources
  src/test/java
    se.erik.lexicon.ordersystem
      OrderSystemApplicationTests.java
      RestTemplateFactory.java
    se.erik.lexicon.ordersystem.entity_tests
    se.erik.lexicon.ordersystem.integration
    se.erik.lexicon.ordersystem.restcontrollers_tests
    se.erik.lexicon.ordersystem.service
  JRE System Library [JavaSE-10]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Basic unit test

```
public class App
{

    public static boolean isAdult(int age) {
            return age >= 18;
    }

}
public class BasicTest {

    @Test
    public void test_IsAdult_true() {
        int age = 18;
        Assert.assertTrue(App.isAdult(age));
    }

    @Test
    public void test_IsAdult_false() {
        int age = 12;
        Assert.assertFalse(App.isAdult(age));
    }

}
```

Another basic unit test example

```
public class App
{

    public static double getProfit(double costPerUnit, double pricePerUnit, int unitAmount) {
        double profit = (pricePerUnit * unitAmount) - (costPerUnit * unitAmount);
        return profit;
    }

}

public class CalculateProfitTest {
    @Test
    public void test_CalculateProfit_returns_expected_profit() {
        double costPerItem = 10, pricePerItem = 25;
        int givenAmount = 10;

        double expectedProfit = 150;

        double actualProfit = App.getProfit(costPerItem, pricePerItem, givenAmount);

        Assert.assertEquals(expectedProfit, actualProfit, 0);
    }
}
```

Testing instance methods

```java
public class Account {

    private double balance;

    public Account(double balance) {this.balance = balance;}

    public Account() {this(0);}

    public double withdraw(double amount) {
        if(validWithdraw(amount)) {
            this.balance -= amount;
            return amount;
        }else {
            return -1;
        }
    }

    public void deposit(double amount) {
        if(amount > 0) {
            this.balance += amount;
        }
    }

    public double getBalance() {return this.balance;}

    private boolean validWithdraw(double amount) {
        if((this.balance - amount) < 0) return false;
        if(amount < 0) return false;
        return true;
    }
}
```

Should test that return value and balance are correct

Should test that passing a value that is greater than the balance does not affect the balance and returns -1

Should test that passing in a negative value does not affect the balance and returns -1

Should test that passing in a positive amount adds the amount to the balance.

Should test that passing in a negative amount don't affect the balance.

Setting up the test with @Before and making a static import on all Assert methods

```
import static org.junit.Assert.*;

public class AccountTest {

    private Account testAccount;

    @Before
    public void setup() {
        testAccount = new Account(1000);
    }

   //Write your tests here

}
```

In order to make the code cleaner i make a static import on the Assert class. This saves us from having to use Assert class name Before the static method we want to call.

The @Before annotation makes the method run before each test in the class.

This saves you from manually making a test object inside each test method.

Tests for withdraw(double amount)

```
@Test
public void testWithdraw() {
    double expectedReturn = 500;
    double expectedNewBalance = 500;
    double actualReturn = testAccount.withdraw(500);
    double actualNewBalance = testAccount.getBalance();
    assertTrue(expectedReturn == actualReturn && expectedNewBalance == actualNewBalance);
}
@Test
public void testWithdraw_negativeAmount() {
    double expectedReturn = -1;
    double expectedBalance = 1000;
    assertTrue(expectedReturn == testAccount.withdraw(-500) && expectedBalance == testAccount.getBalance());
}

@Test
public void testWithdraw_value_greater_than_balance() {
    double expectedReturn = -1;
    double expectedBalance = 1000;
    assertTrue(expectedReturn == testAccount.withdraw(1500) && expectedBalance == testAccount.getBalance());
}
```

Tests for deposit(double amount)

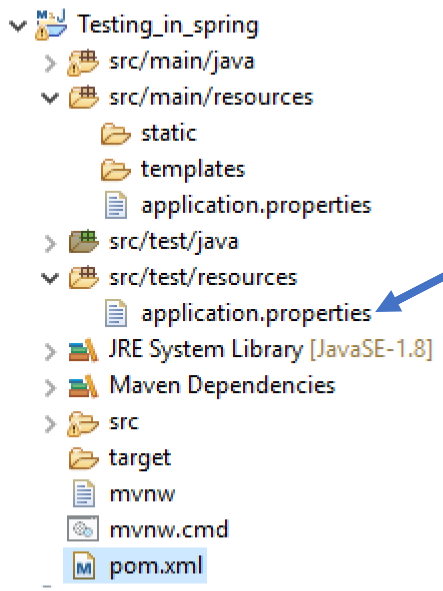
```java
@Test
public void testDeposit() {
    double depositAmount = 500;
    double expectedNewBalance = 1500;
    testAccount.deposit(depositAmount);
    assertEquals(expectedNewBalance, testAccount.getBalance(), 0);
}

@Test
public void testDeposit_negative_amount_will_not_change_balance() {
    double depositAmount = -500;
    double expectedBalance = 1000;
    testAccount.deposit(depositAmount);
    assertEquals(expectedBalance, testAccount.getBalance(), 0);
}
```

Integration testing DAO’s (Data Access Object)

- Integration tests often involves integrating a database.
- Ideally you want a separate in memory database like H2.
- You need to add a dependency for H2 in your maven .pom file.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

Configuring our H2 test database


```
Testing_in_spring
  src/main/java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  src/test/resources
    application.properties
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Need to supply an application.properties in src/test/resources (source folder need to be created). It will override your ordinary application.properties.

application.properties

```
#Example configuration of H2 in a test context
spring.datasource.url=jdbc:h2:mem:test
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver
```

Method we want to test:

```
@Repository
public class CustomerDaoJPAImpl implements CustomerDao {

    @PersistenceContext
    private EntityManager em;

    //more methods..

    //Our example method to test
    @Override
    @Transactional
    public Customer findById(int id){
        Customer customer = em.find(Customer.class, id);
        return customer;
    }

    //more methods

}
```

Creating our test class

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = TestingInSpringApplication.class)
@Transactional
public class CustomerDaoTest {

    @Autowired
    private CustomerDao customerDao;
    private Customer testCustomer;
    private int customerId;

    @Before
    public void setup() {
        Customer customer = new Customer("Test", "Testsson");
        testCustomer = customerDao.persist(customer);
        customerId = testCustomer.getId();
    }

    //write your test methods here

}
```

We need to point to our class that activates Component Scanning. Our main class does that.

All methods need to be transactional. It will roll back transaction after each test.

Our Unit test:

```java
@RunWith(SpringRunner.class)
@SpringBootTest(classes = TestingInSpringApplication.class)
@Transactional
public class CustomerDaoTest {

    //setup
    //more tests

    @Test
    public void testFindById() {
        Customer expected = testCustomer;
        assertEquals(expected, customerDao.findById(customerId));
    }

    //more tests

}
```

Integration testing Spring Data repositories

- Integration testing repositories are quite simple
- You want to setup a test H2 database as previously shown.

Repository we want to test:

```
public interface BookRepository extends CrudRepository<Book, Integer> {

    List<Book> findByTitleLike(String title);
    List<Book> findByDescriptionLike(String description);
    List<Book> findByOnLoanTrue();
    List<Book> findByOnLoanFalse();
    @Query("SELECT book FROM Book book WHERE book.customer.id = :customerId")
    List<Book> findBooksOnLoanToCustomer(@Param("customerId")int customerId);

}
```

We will focus on this method.

Creating our test class

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class BookRepositoryTest {
    @Autowired
    private TestEntityManager testEm;
    @Autowired
    private BookRepository bookRepo;

    private Book testBook;
    private int customerId;

    @Before
    public void setup() {
        testBook = new Book("Test title1", "Test description1");
        Customer customer = new Customer("Test", "Testsson");
        testBook.setCustomer(customer);
        bookRepo.save(testBook));
        this.customerId = customer.getId();
    }

    @After
    public void tearDown() {
        bookRepo.deleteAll();
        testEm.flush();
        testEm.remove(testEm.find(Customer.class, customerId));
        testEm.flush();
    }
    //Write your tests here
}
```

@DataJpaTest provides autoconfiguration when using Spring Data JPA.

TestEntityManager is an EntityManager that provides methods commonly used in tests. You typically use it to handle setup of data needed in the tests.

The @After annotation provided by Junit is used to clean up after each test.

Our Unit test:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class BookRepositoryTest {

    //setup
    //more tests

    @Test
    public void testFindBooksOnLoanToCustomer() {
        List<Book> expected = Arrays.asList(testBook);
        assertEquals(expected, bookRepo.findBooksOnLoanToCustomer(customerId));
    }

    //more tests

}
```

Testing Service Classes

- Testing a Service class can be done with Mockito.
- Mockito comes bundled with the sprint-boot-test-starter.
- Testing a Service we sometimes don’t need or want to test its dependencies (injected beans).
- Ideally we only want to test the code of our Service class.
- Mockito’s role is to mock the behaviour of the dependencies.

Our Service Class: BookServiceImpl

```java
@Service
@Transactional
public class BookServiceImpl implements BookService {

    //Dependencies we need to mock
    private BookRepository bookRepository;
    private StringUtil stringUtil;

    @Autowired
    public BookServiceImpl(BookRepository bookRepository, StringUtil stringUtil) {
        this.bookRepository = bookRepository;
        this.stringUtil = stringUtil;
    }

    //More service methods...

    @Override
    public List<Book> findBooksOnLoanToCustomer(int customerId){
        return bookRepository.findBooksOnLoanToCustomer(customerId);
    }

    @Override
    public List<Book> findBooksByTitle(String searchWord) throws IllegalArgumentException{
        if(searchWord == null) {
            throw new IllegalArgumentException("String searchWord was " + searchWord);
        }
        return bookRepository.findByTitleLike(stringUtil.addWildCardsToString(searchWord));
    }

    //More service methods...

}
```

These two dependencies we need to mock.

The methods we will test

Setting up our Test Class

```java
@RunWith(SpringRunner.class)
public class BookServiceTest {

    @TestConfiguration
    public static class BookServiceTestConfig{
        @Bean
        public BookService bookService(BookRepository bookRepository, StringUtil stringUtil) {
            return new BookServiceImpl(bookRepository, stringUtil);
        }
    }

    @Autowired
    private BookService bookService;
    @MockBean
    private BookRepository bookRepositoryMock;
    @MockBean
    private StringUtil stringUtilMock;

    private Book testBook1;
    private Book testBook2;

    @Before
    public void setup() {
        testBook1 = new Book("Test Title1", "Test Description1");
        testBook2 = new Book("Test Title2", "Test Description2");
    }
    //test methods

}
```

Defined a static config class that create our BookService bean

We use @MockBean to add our mock objects to the application context

Writing the first test:

```
@RunWith(SpringRunner.class)
public class BookServiceTest {

     //setup
     //more tests

    @Test
    public void testFindBooksOnLoanToCustomer_returns_List_with_testBook1() {
        testBook1.setCustomer(new Customer("Test", "Testsson"));

        when(bookRepositoryMock.findBooksOnLoanToCustomer(anyInt()))
            .thenReturn(Arrays.asList(testBook1));

        List<Book> result = bookService.findBooksOnLoanToCustomer(anyInt());

        assertEquals(Arrays.asList(testBook1), result);

    }

    //more tests
}
```

We define the behaviour of our mock repository by telling it how to behave when called.

You need a static import for Mockito to access when() and anyInt() without the classname.

Writing the second test

```
@RunWith(SpringRunner.class)
public class BookServiceTest {

    //setup
    //more tests

    @Test
    public void testFindBooksByTitle_return_list_of_two_books() {
        String given = "Test";
        String parsedString = "%Test%";
        List<Book> expected = Arrays.asList(testBook1, testBook2);
        when(stringUtilMock.addWildCardsToString(given)).thenReturn(parsedString);
        when(bookRepositoryMock.findByTitleLike(parsedString)).thenReturn(Arrays.asList(testBook1, testBook2));

        assertEquals(expected, bookService.findBooksByTitle(given));
    }

    @Test(expected = IllegalArgumentException.class)
    public void testFindBooksByTitle_with_null_throws_IllegalArgumentException() {
        bookService.findBooksByTitle(null);
    }
    //more tests
}
```

1. We are mocking the StringUtil depencency.
2. We are mocking the BookRepository depencency.
3. We call the real method with the given parameter.