BANKING SYSTEM PROJECT - DAY 8: BDD TESTING IMPLEMENTATION

===========================================================

OVERVIEW

--------

Day 8 focuses on implementing Behavior Driven Development (BDD) testing using Cucumber framework with JUnit 5. This implementation provides comprehensive test coverage for transfer operations and other banking system functionalities with clear, readable test scenarios.

BDD TESTING FRAMEWORK SETUP

---------------------------

1. DEPENDENCIES CONFIGURATION

- io.cucumber:cucumber-java:7.14.0

- io.cucumber:cucumber-spring:7.14.0

- io.cucumber:cucumber-junit-platform-engine:7.14.0

- org.junit.platform:junit-platform-suite:1.10.1

- Spring Boot Test framework

- AssertJ for assertions

2. TEST CONFIGURATION

@SpringBootTest

@ActiveProfiles("test")

@TestPropertySource(locations = "classpath:application-test.properties")

public class TestConfiguration {

// Test configuration class

}

3. APPLICATION TEST PROPERTIES

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=banking\_system\_test

aws.dynamodb.local.enabled=false

logging.level.com.bankingsystem=INFO

logging.level.org.springframework.data.mongodb=WARN

FEATURE FILES IMPLEMENTATION

----------------------------

1. TRANSFER FEATURE FILE

Feature: Money Transfer

As a banking system user

I want to transfer money between accounts

So that I can manage my finances effectively

Background:

Given the banking system is running

And there are existing customers and accounts

@banking

Scenario: Successful transfer between two accounts

Given customer "Rajesh Kumar" has account "ACC123456" with balance "₹50000"

And customer "Priya Sharma" has account "ACC123457" with balance "₹30000"

When I transfer "₹10000" from "ACC123456" to "ACC123457"

Then the transfer should be successful

And account "ACC123456" should have balance "₹40000"

And account "ACC123457" should have balance "₹40000"

And the transaction should be recorded in audit logs

@banking

Scenario: Transfer with insufficient funds

Given customer "Amit Patel" has account "ACC123458" with balance "₹5000"

And customer "Sneha Singh" has account "ACC123459" with balance "₹20000"

When I transfer "₹10000" from "ACC123458" to "ACC123459"

Then the transfer should fail with "Insufficient balance" error

And account "ACC123458" should have balance "₹5000"

And account "ACC123459" should have balance "₹20000"

@banking

Scenario: Transfer to same account

Given customer "Vikram Reddy" has account "ACC123460" with balance "₹25000"

When I transfer "₹5000" from "ACC123460" to "ACC123460"

Then the transfer should fail with "Cannot transfer to the same account" error

And account "ACC123460" should have balance "₹25000"

@banking

Scenario: Transfer with invalid account numbers

Given customer "Anita Desai" has account "ACC123461" with balance "₹15000"

When I transfer "₹5000" from "ACC123461" to "INVALID\_ACC"

Then the transfer should fail with "Destination account not found" error

And account "ACC123461" should have balance "₹15000"

@banking

Scenario: Transfer with zero amount

Given customer "Ravi Gupta" has account "ACC123462" with balance "₹20000"

When I transfer "₹0" from "ACC123462" to "ACC123463"

Then the transfer should fail with "Transfer amount must be greater than zero" error

@banking

Scenario: Transfer with negative amount

Given customer "Sunita Verma" has account "ACC123464" with balance "₹15000"

When I transfer "₹-1000" from "ACC123464" to "ACC123465"

Then the transfer should fail with "Transfer amount must be greater than zero" error

2. DEPOSIT FEATURE FILE

Feature: Money Deposit

As a banking system user

I want to deposit money into my account

So that I can add funds to my account

Background:

Given the banking system is running

And there are existing customers and accounts

@banking

Scenario: Successful deposit

Given customer "Rajesh Kumar" has account "ACC123456" with balance "₹50000"

When I deposit "₹10000" into account "ACC123456"

Then the deposit should be successful

And account "ACC123456" should have balance "₹60000"

And the transaction should be recorded in audit logs

@banking

Scenario: Deposit with invalid account

When I deposit "₹5000" into account "INVALID\_ACC"

Then the deposit should fail with "Account not found" error

@banking

Scenario: Deposit with zero amount

Given customer "Priya Sharma" has account "ACC123457" with balance "₹30000"

When I deposit "₹0" into account "ACC123457"

Then the deposit should fail with "Deposit amount must be greater than zero" error

3. WITHDRAWAL FEATURE FILE

Feature: Money Withdrawal

As a banking system user

I want to withdraw money from my account

So that I can access my funds

Background:

Given the banking system is running

And there are existing customers and accounts

@banking

Scenario: Successful withdrawal

Given customer "Amit Patel" has account "ACC123458" with balance "₹20000"

When I withdraw "₹5000" from account "ACC123458"

Then the withdrawal should be successful

And account "ACC123458" should have balance "₹15000"

And the transaction should be recorded in audit logs

@banking

Scenario: Withdrawal with insufficient funds

Given customer "Sneha Singh" has account "ACC123459" with balance "₹3000"

When I withdraw "₹5000" from account "ACC123459"

Then the withdrawal should fail with "Insufficient balance" error

And account "ACC123459" should have balance "₹3000"

STEP DEFINITIONS IMPLEMENTATION

-------------------------------

1. TRANSFER STEPS IMPLEMENTATION

@Component

public class TransferSteps {

@Autowired

private TransferService transferService;

@Autowired

private AccountRepository accountRepository;

@Autowired

private CustomerRepository customerRepository;

@Autowired

private AuditService auditService;

private Transaction lastTransaction;

private Exception lastException;

@Given("customer {string} has account {string} with balance {string}")

public void customerHasAccountWithBalance(String customerName, String accountNumber, String balance) {

// Create customer if not exists

Customer customer = customerRepository.findByFirstNameAndLastName(

customerName.split(" ")[0], customerName.split(" ")[1])

.orElseGet(() -> {

Customer newCustomer = new Customer();

newCustomer.setFirstName(customerName.split(" ")[0]);

newCustomer.setLastName(customerName.split(" ")[1]);

newCustomer.setEmail(customerName.toLowerCase().replace(" ", ".") + "@test.com");

newCustomer.setPhoneNumber("9876543210");

newCustomer.setAddress("Test Address");

newCustomer.setCity("Test City");

newCustomer.setState("Test State");

newCustomer.setPincode("123456");

return customerRepository.save(newCustomer);

});

// Create account if not exists

Account account = accountRepository.findByAccountNumber(accountNumber)

.orElseGet(() -> {

Account newAccount = new Account();

newAccount.setAccountNumber(accountNumber);

newAccount.setCustomer(customer);

newAccount.setAccountType(AccountType.SAVINGS);

newAccount.setBalance(BigDecimal.ZERO);

newAccount.setMinimumBalance(new BigDecimal("1000"));

newAccount.setStatus(AccountStatus.ACTIVE);

return accountRepository.save(newAccount);

});

// Set balance

account.setBalance(new BigDecimal(balance.replace("₹", "").replace(",", "")));

accountRepository.save(account);

}

@When("I transfer {string} from {string} to {string}")

public void iTransferFromTo(String amount, String sourceAccount, String destinationAccount) {

try {

BigDecimal transferAmount = new BigDecimal(amount.replace("₹", "").replace(",", ""));

lastTransaction = transferService.processTransfer(

sourceAccount, destinationAccount, transferAmount, "Test transfer", "TEST\_USER");

} catch (Exception e) {

lastException = e;

}

}

@Then("the transfer should be successful")

public void theTransferShouldBeSuccessful() {

assertThat(lastTransaction).isNotNull();

assertThat(lastTransaction.getStatus()).isEqualTo(TransactionStatus.COMPLETED);

assertThat(lastException).isNull();

}

@Then("the transfer should fail with {string} error")

public void theTransferShouldFailWithError(String expectedError) {

assertThat(lastException).isNotNull();

assertThat(lastException.getMessage()).contains(expectedError);

assertThat(lastTransaction).isNull();

}

@And("account {string} should have balance {string}")

public void accountShouldHaveBalance(String accountNumber, String expectedBalance) {

Account account = accountRepository.findByAccountNumber(accountNumber)

.orElseThrow(() -> new RuntimeException("Account not found: " + accountNumber));

BigDecimal expectedAmount = new BigDecimal(expectedBalance.replace("₹", "").replace(",", ""));

assertThat(account.getBalance()).isEqualTo(expectedAmount);

}

@And("the transaction should be recorded in audit logs")

public void theTransactionShouldBeRecordedInAuditLogs() {

// Verify audit log was created

List<AuditLog> auditLogs = auditService.getAuditLogsByActionType("TRANSFER");

assertThat(auditLogs).isNotEmpty();

AuditLog lastAuditLog = auditLogs.get(auditLogs.size() - 1);

assertThat(lastAuditLog.getStatus()).isEqualTo(AuditStatus.SUCCESS);

assertThat(lastAuditLog.getActionDetails()).contains("Transfer");

}

}

2. DEPOSIT STEPS IMPLEMENTATION

@Component

public class DepositSteps {

@Autowired

private DepositService depositService;

@Autowired

private AccountRepository accountRepository;

private Transaction lastTransaction;

private Exception lastException;

@When("I deposit {string} into account {string}")

public void iDepositIntoAccount(String amount, String accountNumber) {

try {

BigDecimal depositAmount = new BigDecimal(amount.replace("₹", "").replace(",", ""));

lastTransaction = depositService.processDeposit(

accountNumber, depositAmount, "Test deposit", "TEST\_USER");

} catch (Exception e) {

lastException = e;

}

}

@Then("the deposit should be successful")

public void theDepositShouldBeSuccessful() {

assertThat(lastTransaction).isNotNull();

assertThat(lastTransaction.getStatus()).isEqualTo(TransactionStatus.COMPLETED);

assertThat(lastException).isNull();

}

@Then("the deposit should fail with {string} error")

public void theDepositShouldFailWithError(String expectedError) {

assertThat(lastException).isNotNull();

assertThat(lastException.getMessage()).contains(expectedError);

assertThat(lastTransaction).isNull();

}

}

3. WITHDRAWAL STEPS IMPLEMENTATION

@Component

public class WithdrawalSteps {

@Autowired

private WithdrawService withdrawService;

@Autowired

private AccountRepository accountRepository;

private Transaction lastTransaction;

private Exception lastException;

@When("I withdraw {string} from account {string}")

public void iWithdrawFromAccount(String amount, String accountNumber) {

try {

BigDecimal withdrawalAmount = new BigDecimal(amount.replace("₹", "").replace(",", ""));

lastTransaction = withdrawService.processWithdrawal(

accountNumber, withdrawalAmount, "Test withdrawal", "TEST\_USER");

} catch (Exception e) {

lastException = e;

}

}

@Then("the withdrawal should be successful")

public void theWithdrawalShouldBeSuccessful() {

assertThat(lastTransaction).isNotNull();

assertThat(lastTransaction.getStatus()).isEqualTo(TransactionStatus.COMPLETED);

assertThat(lastException).isNull();

}

@Then("the withdrawal should fail with {string} error")

public void theWithdrawalShouldFailWithError(String expectedError) {

assertThat(lastException).isNotNull();

assertThat(lastException.getMessage()).contains(expectedError);

assertThat(lastTransaction).isNull();

}

}

TEST RUNNER CONFIGURATION

-------------------------

1. CUCUMBER TEST RUNNER

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features",

glue = "com.bankingsystem.stepdefs",

plugin = {

"pretty",

"html:target/cucumber-reports",

"json:target/cucumber-reports/Cucumber.json",

"junit:target/cucumber-reports/Cucumber.xml"

},

tags = "@banking"

)

public class CucumberTestRunner {

// Test runner class

}

2. JUNIT 5 TEST SUITE

@Suite

@SuiteDisplayName("Banking System BDD Test Suite")

@SelectPackages("com.bankingsystem.stepdefs")

@IncludeEngines("cucumber")

public class BankingSystemBDDTestSuite {

// BDD test suite

}

3. SPRING BOOT TEST CONFIGURATION

@SpringBootTest

@TestPropertySource(locations = "classpath:application-test.properties")

@ActiveProfiles("test")

public class BDDTestConfiguration {

@Autowired

private TestEntityManager entityManager;

@BeforeEach

void setUp() {

// Clean up test data before each test

entityManager.getEntityManager().createQuery("DELETE FROM Transaction").executeUpdate();

entityManager.getEntityManager().createQuery("DELETE FROM Account").executeUpdate();

entityManager.getEntityManager().createQuery("DELETE FROM Customer").executeUpdate();

entityManager.getEntityManager().createQuery("DELETE FROM AuditLog").executeUpdate();

}

}

MOCK IMPLEMENTATIONS

--------------------

1. MOCK SERVICES FOR TESTING

@TestConfiguration

public class MockTestConfiguration {

@Bean

@Primary

public DynamoDBService mockDynamoDBService() {

return Mockito.mock(DynamoDBService.class);

}

@Bean

@Primary

public AuditService mockAuditService() {

AuditService mockAuditService = Mockito.mock(AuditService.class);

Mockito.doNothing().when(mockAuditService).logSuccess(any(), any(), any(), any(), any());

Mockito.doNothing().when(mockAuditService).logFailure(any(), any(), any(), any(), any(), any());

return mockAuditService;

}

}

2. TEST DATA BUILDER

@Component

public class TestDataBuilder {

public Customer createTestCustomer(String firstName, String lastName) {

Customer customer = new Customer();

customer.setFirstName(firstName);

customer.setLastName(lastName);

customer.setEmail(firstName.toLowerCase() + "." + lastName.toLowerCase() + "@test.com");

customer.setPhoneNumber("9876543210");

customer.setAddress("Test Address");

customer.setCity("Test City");

customer.setState("Test State");

customer.setPincode("123456");

customer.setStatus(CustomerStatus.ACTIVE);

return customer;

}

public Account createTestAccount(String accountNumber, Customer customer, BigDecimal balance) {

Account account = new Account();

account.setAccountNumber(accountNumber);

account.setCustomer(customer);

account.setAccountType(AccountType.SAVINGS);

account.setBalance(balance);

account.setMinimumBalance(new BigDecimal("1000"));

account.setStatus(AccountStatus.ACTIVE);

return account;

}

}

TEST REPORTING

--------------

1. CUCUMBER REPORTING CONFIGURATION

@Configuration

public class CucumberReportingConfig {

@Bean

public CucumberReportingService cucumberReportingService() {

return new CucumberReportingService();

}

}

2. HTML REPORT GENERATION

public class CucumberReportingService {

public void generateHtmlReport() {

try {

File reportOutputDirectory = new File("target/cucumber-reports");

List<String> jsonFiles = Arrays.asList("target/cucumber-reports/Cucumber.json");

String buildNumber = "1";

String projectName = "Banking System";

boolean runWithJenkins = false;

boolean parallelTesting = false;

Configuration configuration = new Configuration(reportOutputDirectory, projectName);

configuration.setBuildNumber(buildNumber);

configuration.addClassifications("Platform", "Windows");

configuration.addClassifications("Browser", "Chrome");

configuration.addClassifications("Branch", "master");

ReportBuilder reportBuilder = new ReportBuilder(jsonFiles, configuration);

reportBuilder.generateReports();

} catch (Exception e) {

log.error("Failed to generate Cucumber HTML report", e);

}

}

}

INTEGRATION WITH CI/CD

----------------------

1. MAVEN TEST EXECUTION

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/CucumberTestRunner.java</include>

</includes>

</configuration>

</plugin>

2. JENKINS PIPELINE INTEGRATION

pipeline {

agent any

stages {

stage('BDD Tests') {

steps {

sh 'mvn test -Dtest=CucumberTestRunner'

}

post {

always {

publishHTML([

allowMissing: false,

alwaysLinkToLastBuild: true,

keepAll: true,

reportDir: 'target/cucumber-reports',

reportFiles: 'index.html',

reportName: 'Cucumber Report'

])

}

}

}

}

}

PERFORMANCE TESTING

-------------------

1. LOAD TESTING SCENARIOS

@Component

public class LoadTestingSteps {

@Autowired

private TransferService transferService;

@When("I perform {int} concurrent transfers")

public void iPerformConcurrentTransfers(int numberOfTransfers) {

ExecutorService executor = Executors.newFixedThreadPool(10);

List<CompletableFuture<Transaction>> futures = new ArrayList<>();

for (int i = 0; i < numberOfTransfers; i++) {

CompletableFuture<Transaction> future = CompletableFuture.supplyAsync(() -> {

return transferService.processTransfer(

"ACC123456", "ACC123457", new BigDecimal("100"), "Load test", "TEST\_USER");

}, executor);

futures.add(future);

}

CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

executor.shutdown();

}

}

2. PERFORMANCE ASSERTIONS

@Then("all transfers should complete within {int} seconds")

public void allTransfersShouldCompleteWithinSeconds(int maxSeconds) {

// Performance assertion logic

long startTime = System.currentTimeMillis();

// Execute transfers

long endTime = System.currentTimeMillis();

long duration = (endTime - startTime) / 1000;

assertThat(duration).isLessThan(maxSeconds);

}

This BDD testing implementation provides comprehensive test coverage with clear, readable scenarios and robust step definitions for the banking system.