Table of Contents

Introduction

Behavior-Driven Development

Behavior-Driven Development (BDD) is an Agile software development methodology that promotes collaboration between business stakeholders, developers, and testers to ensure the software meets user needs. BDD emphasizes defining system behavior through examples written in a shared, human-readable format.

Key Principles

- Define behavior through **collaborative conversations**
- Use a **shared language** (often Gherkin) to write scenarios
- Treat these scenarios as **living documentation** and **automated tests**

Typical BDD Process

1. **Collaborative Specification.** Stakeholders (e.g., product owners, developers, testers) discuss expected behaviors using real examples and write scenarios in Gherkin format.
2. **Automated Acceptance Testing.** Scenarios are automated using tools such as Cucumber and are integrated into the CI pipeline for continuous verification.
3. **Development.** Developers implement features defined in scenarios. Automated tests are run frequently to ensure behavior remains consistent and regressions are caught early.
4. **Review and Feedback.** Each iteration concludes with a stakeholder review. Scenarios may be updated to reflect changes or new understanding.
5. **Repeat.** The cycle continues in later iterations.

BDD integrates naturally with Scrum or other Agile frameworks.

References

Spring

Spring Context

The **Spring context** is a container that manages the lifecycle and dependencies of application components known as * *beans**. These beans are stored in memory and created, injected, and destroyed by Spring as needed.

Spring follows the **Inversion of Control (IoC)** principle, where the framework, not the developer, manages object creation and wiring. To benefit from Spring features, you need to tell Spring which objects to manage.

Adding Beans to the Spring Context

Beans can be added to the Spring context in three main ways:

1. @Bean Annotation

- Used in `@Configuration` classes.
- Provides full control over instance creation and configuration.
- Can add any object type, even ones not defined in your application.
- Allows adding multiple beans of the same type.
- Requires writing a separate method for each bean (more boilerplate).

```
@Configuration
public class ProjectConfig {

    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
```

```
    @Bean
    @Primary
    public Parrot parrot2() {
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }

    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

2. Stereotype Annotations (@Component, @Service, etc.)

- Less boilerplate than `@Bean`.
- Automatically picks up classes with stereotype annotations via `@ComponentScan`.
- Only works with classes defined in your app.
- Only one instance of each class can be added.
- Limited control over instantiation.

```
@Component
public class Parrot {
    private String name;

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    @PostConstruct
    public void init() {
        this.name = "Kiki";
    }
}

@Configuration
@ComponentScan(basePackages = "main")
public class ProjectConfig {
}
```

3. Programmatically (e.g., registerBean())

- Available since Spring 5.
- Allows dynamic, custom registration logic.
- Useful for conditionally or externally defined beans.

```
public class Main {
    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(ProjectConfig.class);

        Parrot x = new Parrot();
        x.setName("Kiki");

        Supplier<Parrot> supplier = () -> x;
        context.registerBean("parrot1", Parrot.class, supplier);

        Parrot p = context.getBean(Parrot.class);
        System.out.println(p.getName());
```

```
    }
}
```

Wiring Beans (Dependency Injection)

Spring uses **Dependency Injection (DI)** to wire beans together.

Manual Wiring (Method Calls)

```
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot());
        return p;
    }
}
```

Auto-Wiring (Method Parameters)

```
@Configuration
public class ProjectConfig {
    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
```

```
    @Bean
    public Person person(Parrot parrot) {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot);
        return p;
    }
}
```

Using @Autowired

- Inject via constructor (preferred), field (not recommended for production), or setter (rarely used).

```
@Component
public class Person {
    private String name = "Ella";
    private final Parrot parrot;

    @Autowired
    public Person(Parrot parrot) {
        this.parrot = parrot;
    }
}
```

Bean Scopes and Lifecycle

Spring manages bean creation and lifecycle using **scopes**:

- **singleton** (default): one instance per context.
- **prototype**: new instance every time requested.
- **request**: one instance per HTTP request (`@RequestScope`).

- **session**: one instance per HTTP session (`@SessionScope`).
- **application**: one instance per application (`@ApplicationScope`).

To use custom scopes:

```
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
@Bean
public Parrot parrot() {
    return new Parrot();
}
```

Aspect-Oriented Programming (AOP)

Aspects let you intercept method calls and inject logic (e.g., logging, security).

AOP Concepts

- **Aspect**: logic to apply.
- **Advice**: when to apply it (before, after, around).
- **Pointcut**: which methods to apply it to.

Spring uses **proxies** for weaving: instead of returning the original bean, Spring returns a proxy that wraps it with aspect logic.

Steps to Create an Aspect

1. Enable AOP:

```
@Configuration
@ComponentScan(basePackages = "services")
@EnableAspectJAutoProxy
public class ProjectConfig {
}
```

2. Create an Aspect:

```
@Aspect
@Component
public class LoggingAspect {
    @Around("execution(* services.*.*(..))")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        Object[] args = joinPoint.getArgs();

        // before method call
        Object result = joinPoint.proceed();
        // after method call

        return result;
    }
}
```

3. To define the order of multiple aspects:

```
@Aspect
@Order(1)
public class FirstAspect {
}

@Aspect
@Order(2)
public class SecondAspect {
}
```

Hibernate

Configuration

Hibernate is typically configured via an XML file named `hibernate.cfg.xml`, located in the classpath:

```
<hibernate-configuration>
    <session-factory>
        <property
name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:./db1</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"/>
        <property
name="dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">create-drop</property>
        <mapping class="chapter01.hibernate.Message"/>
    </session-factory>
</hibernate-configuration>
```

When using an application server or Spring, Hibernate can be integrated via resource injection or via configuration in the Spring context.

Integration Steps

To apply Hibernate to a project:

1. Identify POJOs with database representation.
2. Annotate POJOs to map fields to table columns.
3. Add Hibernate and database driver dependencies.
4. Configure `hibernate.cfg.xml` or use Spring configuration.
5. Create a `SessionFactory` and use `Session` objects for CRUD operations.

Connection Pooling

Hibernate includes a basic connection pool, not suitable for production. In production:

- Use HikariCP (default in Spring Boot).
- Configure external pool via classpath or JNDI.
- Example dependency for HikariCP:

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-hikaricp</artifactId>
    <version>${hibernate.core.version}</version>
</dependency>
```

Entity Requirements

- Must have a `public` or `protected` no-arg constructor.
- Should not be `final`.

JPA Annotations

- `@Entity`: declares a class as an entity.
- `@Id`: marks the primary key field.
- `@GeneratedValue`: configures ID generation strategy.
- `@Column`: customizes column mapping.

ID Generation Strategies

- `IDENTITY`: best for MySQL (auto-increment).

- `SEQUENCE`: preferred (e.g., PostgreSQL, Oracle).
- `TABLE`: slow, not recommended.
- `AUTO`: delegates to provider.
- `NONE`: requires manual assignment.

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

UUID Identifier

```
@Id
@GeneratedValue
private UUID courseId;
```

Custom Identifier Generator

Use `@GenericGenerator` and implement `IdentifierGenerator`:

```
@Id
@GeneratedValue(generator = "prod-generator")
@GenericGenerator(
        name = "prod-generator",
        strategy = "com.example.MyGenerator",
        parameters = @Parameter(name = "prefix", value = "prod")
)
private String prodId;
```

Composite Identifiers

@EmbeddedId

```
@Embeddable
public class OrderEntryPK implements Serializable {
```

```
    private long orderId;
    private long productId;
}

@Entity
public class OrderEntry {
    @EmbeddedId
    private OrderEntryPK entryId;
}
```

@IdClass

```
@Entity
@IdClass(OrderEntryPK.class)
public class OrderEntry {
    @Id
    private long orderId;
    @Id
    private long productId;
}
```

Derived Identifiers

Use `@MapsId` to derive one entity's ID from another:

```
@Entity
public class UserProfile {
    @Id
    private long profileId;

    @OneToOne
    @MapsId
    private User user;
}
```

Persistence Context States

- **Transient**: newly created, not associated with persistence context.
- **Persistent**: managed by session, tracked for changes.
- **Detached**: was persistent, but session closed or evicted.
- **Removed**: marked for deletion.

CRUD and Session Methods

- `save()`, `saveOrUpdate()`, `update()`
- `get()` (returns null if not found), `load()` (throws if not found)
- `merge()`, `refresh()`
- `delete()`, `flush()`
- `isDirty()`: check if session has unflushed changes

Transactions

A transaction groups operations into a single atomic unit:

- Call `commit()` to save changes.
- Use `rollback()` to discard changes.

Cascading

Cascade options allow operations on one entity to propagate to related entities:

- `PERSIST`
- `MERGE`
- `REFRESH`

- `REMOVE`
- `DETACH`
- `ALL`

Example:

```
@OneToOne(cascade = {CascadeType.REFRESH, CascadeType.MERGE})
private EntityType otherSide;
```

Lazy Loading

Hibernate loads collections and associations lazily by default unless configured otherwise.

References

- Hibernate 6 Documentation (https://hibernate.org/orm/documentation/6.0/)
- Spring Data JPA Reference (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)
- Baeldung: Hibernate Identifiers (https://www.baeldung.com/hibernate-identifiers)

Testing

Levels of Testing

Software testing is typically structured in the following levels:

Level	Purpose	Benefits	Key Characteristics
Unit Tests	Test individual software components in isolation	- Early bug detection - Safe refactoring - Encourages modular design	- Readable names - Fast execution - Isolated from external dependencies - Repeatable tests produce the same result every time they are executed
Integration Tests	Test interactions between software components to ensure they work together correctly	- Detects interface issues - Verifies data/control flow between components	- Built incrementally from unit-tested parts - Focused on interactions - Based on test plans
System Tests	Test the complete integrated system to ensure it meets its requirements and functions correctly	- Verifies overall functionality - Catches system-wide issues - Validates non-functional requirements	- End-to-end coverage - Tests external behavior - Driven by system specifications
Acceptance Tests	Confirm the system meets business or contractual requirements	- Validates business value - Final approval before release	- Performed by users or stakeholders - Based on acceptance criteria - Reflects real-world use

Testing Java Applications

JUnit 5

JUnit 5 is composed of three subprojects:

- **JUnit Platform** is responsible for launching testing frameworks on the JVM. It also provides a `TestEngine` API that can be used for implementing testing frameworks that can be used with the JUnit platform. It also provides a `ConsoleLauncher` that build tools like Gradle and Maven can use.
- **JUnit Jupiter** is a test framework for writing tests in JUnit 5 that implements the `TestEngine` API provided by the JUnit Platform. It also provides annotations, assertions, and other APIs for writing tests.
- **JUnit Vintage** is a test engine that provides support for running JUnit 3 and JUnit 4 tests on the JUnit 5 platform. It allows developers to migrate their existing JUnit 3 and JUnit 4 tests to JUnit 5 without the need for significant code changes.

Test methods are annotated with `org.junit.jupiter.api.Test`.

Test methods aren’t required to be public, can be package-private.

Test methods can have parameters. This is achieved by providing a `ParameterResolver`.

Test classes aren't required to have no-args constructors.

LifeCycle

Each test case is governed by a test life cycle which consists of the following three phases:

- **Setup phase**, where the test infrastructure is put in place. Two levels of setup:
  - class level:
    - Method annotated with `org.junit.jupiter.api.BeforeAll` where a costly object like database connection can be created for all the tests in a class.
    - It must be static and non-private.

    - individual test setup methods:
        - Method annotated with `org.junit.jupiter.api.BeforeEach` executes before each and every test, thus doing away with any side effects from other test executions.
        - A test case can have any number of methods marked with `BeforeEach`, but the execution order is not guaranteed.
- **Test execution phase**. Result verification is also part of the test execution phase. The execution result will signify a success or failure.
- **Cleanup phase**, where any cleanup required after posttest execution is performed. Two levels of cleanup:
    - `@AfterAll` performs a single time method invocation (i.e., post the execution of all test cases of a test class). It must be static and non-private.
    - `@AfterEach` - posttest execution cleanup. The method must not be static.

Optionally, we could extract out the `@BeforeAll` and `@AfterAll` methods to a superclass. This would clean up the duplicate code if the database is being used in more test cases.

JUnit Annotations

- `@Test`
- `@ParameterizedTest`
- `@ValueSource`, `@CsvSource`
- `@RepeatedTest`
- `@BeforeAll`
- `@AfterAll`
- `@BeforeEach`
- `@AfterEach`
- `@Disabled`
- `@DisplayName`

- `@Tag`
- `@Nested`

Assertions

Assertions are static methods from `org.junit.jupiter.api.Assertions`.

Assert Method	What It Does
assertTrue	Assert that condition is true
assertFalse	Assert that condition is false
assertNull	Assert that object is null
assertNotNull	Assert that object is not null
assertEquals	Assert that expected and actual are equal
assertNotEquals	Assert that expected and actual are not equal
assertArrayEqual	Assert that expected and actual arrays are equals
assertSame	Assert that expected and actual refer to the same obj
assertNotSame	Assert that expected and actual do not refer to same
assertThrow	Assert that an exception is thrown
assertAll	Group multiple assertions and evaluate all

Each `assertXXX` method has at least three overloaded methods:

- `assertNull(str);`

- `assertNull(str, "str should be null");`
- `assertNull(str, () -> "str should be null"); // lazy`

Most `assert` methods take:

- Expected value
- Actual value

AssertJ

AssertJ is a third-party library of assertions.

```
import static org.assertj.core.api.Assertions.assertThat;
```

- Basic assertions:
  - assertThat(actual).isEqualTo(expected);
  - assertThat(actual).isNotNull();
- String
  - assertThat(actual).startsWith(expected);
  - assertThat(actual).endsWith(expected);
  - assertThat(actual).isEqualToIgnoringCase(expected);
- Collections
  - assertThat(list).contains(expected);
  - assertThat(list).hasSize(size);
- Map
  - assertThat(map).containsKey(key);
  - assertThat(map).hasSize(size);
- Number

    - assertThat(value).isGreaterThan(expected);
    - assertThat(value).isBetween(lower, upper);
- Object
    - assertThat(obj).isInstanceOf(expected);
    - assertThat(obj).hasNoNullFieldsOrProperties();

Mockito

```
@RunWith(MockitoJUnitRunner.class)
public class MyTest {
    @Mock
    private MyClass myMock;
}
```

- **Mocks**: Fake implementations
- **Two ways to mock**:
    - `Mockito.mock(SomeClass.class)`
    - `@Mock` annotation
- **Stubs**:

```
when(mockedList.get(0)).

thenReturn("first");
```

- **Spies**:

```
@Spy
private MyClass mySpy;
```

BDDMockito

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

given(seller.askForBread()).

willReturn(new Bread());
```

Argument Matchers

```
when(mockList.get(anyInt())).

thenReturn(42);
```

Custom matcher:

```
when(mockList.get(argThat(arg ->arg >=0&&arg <=9))).

thenReturn(42);
```

Verification

```
verify(myMock).

doSomething();

verify(myMock, times(2)).

doSomething();
```

- For void methods:

```
doThrow(RuntimeException .class).

when(repository).
```

```
delete(null);
```

- `doReturn`, `doThrow`, `doAnswer`, `doNothing`, `doCallRealMethod`

Testing Spring Applications

Spring TestContext Framework

Unit and integration testing in Spring is supported by the annotation-based **Spring TestContext Framework**, found in the `org.springframework.test.context` package.

It supports multiple test frameworks like **JUnit** and **TestNG**.

To improve performance, it reuses the same application context across tests instead of reloading it each time.

You configure the test context using annotations such as `@ContextConfiguration` and `@ActiveProfiles`.

Spring Boot Testing

Spring Boot provides test support through two main modules:

- `spring-boot-test` for core testing utilities
- `spring-boot-test-autoconfigure` for autoconfiguration in tests

Most projects use the `spring-boot-starter-test` dependency, which includes both modules and libraries like **JUnit Jupiter**, **AssertJ**, **Hamcrest**, **JSONassert**, and **JsonPath**.

To load the full Spring Boot context in tests, use the `@SpringBootTest` annotation. It replaces `@ContextConfiguration` and enables access to Spring Boot features. For narrower tests, other annotations are available (e.g. `@WebMvcTest`, `@DataJpaTest`).

- With **JUnit 4**, annotate the test class with `@RunWith(SpringRunner.class)`.
- With **JUnit 5**, no extra annotation is needed—`@SpringBootTest` and similar annotations already include `@ExtendWith(SpringExtension.class)`.

These annotations enable features like dependency injection and support for `@Autowired`, `@MockBean`, and others in test classes.

Annotations

Mocking

`@Mock` and `@InjectMocks` are Mockito annotations that used to create mock objects and inject them into the class under test.

`@MockBean` is a Spring Boot annotation that replaces a Spring-managed bean with a mock in tests that load the application context.

Auto-Configured Tests

Spring Boot's autoconfigured annotations help to load parts of the complete application and test specific layers of the codebase:

- `@DataMongoTest`. Test MongoDB applications. By default, it configures an in-memory embedded MongoDB if the driver is available through dependencies, configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories.
- `@DataRedisTest`. Test Redis applications. It scans for `@RedisHash` classes and configures Spring Data Redis repositories by default.
- `@RestClientTest`. Test REST clients. It autoconfigures different dependencies such as Jackson, GSON, and Jsonb support; configures a `RestTemplateBuilder`; and adds support for `MockRestServiceServer` by default.
- `@JsonTest`. Initializes the Spring application context only with those beans needed to test JSON serialization.

Other Annotations

- `@AutoConfigureMockMvc`. Used to autoconfigure a `MockMvc` instance for testing Spring MVC controllers.
- `@TestPropertySource`. Used to specify one or more properties files to load for testing.
- `@Transactional`. Used to indicate that a test method should be run within a transaction, which will be rolled back after the test completes.

- `@DirtiesContext`. Used to indicate that a test method modifies the application context and should cause the context to be recreated for subsequent tests.
- `@DatabaseSetup`. Allows you to populate a database with test data before running a test. This annotation is typically used in integration tests where you want to ensure that the database is in a specific state before the test starts.

References

- https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations (https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations)

Integration Tests

About

Integration tests focus on how multiple components work together.

Integration Testing in Spring Framework

The Spring Framework provides first-class support for integration testing through the `spring-test` module, which includes classes in the `org.springframework.test` package.

Comparison to Unit and End-to-End (E2E) Tests

Integration tests in Spring do not require an application server or external deployment environment. They are slower than unit tests but significantly faster than full end-to-end tests like those using Selenium.

While integration tests often avoid mocking, mocking is still appropriate when isolating external systems or avoiding dependencies such as third-party APIs or message brokers.

Because integration tests can be time-consuming and may require resources like a real or embedded database, they should be separated from unit tests. It is recommended to use a dedicated test profile and run them separately (e.g., using a `test` or `integration` Maven/Gradle task).

Common Use Cases

- Testing Spring MVC endpoints, including request-response flow, error handling, and security
- Verifying interaction between services and external dependencies (e.g., databases, messaging systems, web services)
- Testing Spring configuration: bean definitions, property sources, and active profiles
- Full-stack testing across the web, service, and persistence layers

Integration Testing in Spring Boot

@SpringBootTest

Used to load the complete application context for integration testing. Allows the use of `@Autowired` to inject any bean discovered through component scanning. Can be applied at the class level or to individual methods for customized test contexts.

MockMvc

The Spring MVC Test framework (`MockMvc`) supports testing MVC controllers without starting a real server.

Unlike unit tests of controllers, `MockMvc` tests involve a full request-processing pipeline—filters, interceptors, and controller logic—though still within the test environment.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMo
ckMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get
;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.conte
nt;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.statu
s;

@SpringBootTest
@AutoConfigureMockMvc
class MyMockMvcTests {
```

```
    @Test
    void testWithMockMvc(@Autowired MockMvc mvc) throws Exception {
        mvc.perform(get("/"))
                .andExpect(status().isOk())
                .andExpect(content().string("Hello World"));
    }
}
```

Documentation: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework/server-setup-options.html (https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework/server-setup-options.html)

MockMvc Setup Options

- Programmatically configure Spring MVC to point directly to controllers
- Use Spring configuration that includes the full MVC and controller infrastructure

WebTestClient

`WebTestClient` is used for testing reactive applications built with Spring WebFlux.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import
org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.web.reactive.server.WebTestClient;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyRandomPortWebTestClientTests {

    @Test
    void exampleTest(@Autowired WebTestClient webClient) {
        webClient.get().uri("/")
                .exchange()
                .expectStatus().isOk()
                .expectBody(String.class).isEqualTo("Hello World");
```

```
    }
}
```

Note: This requires the following dependency:

```
implementation 'org.springframework.boot:spring-boot-starter-webflux'
```

Performing Requests with MockMvc

Documentation: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework/server-performing-requests.html (https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework/server-performing-requests.html)

Active Profiles

Use `@ActiveProfiles` to declare which profile should be active when loading the `ApplicationContext` for an integration test.

```
@ContextConfiguration
@ActiveProfiles("dev")
class DeveloperTest {
}
```

Example Integration Test Configuration

`application-integrationtest.properties`

```
spring.datasource.url=jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

End-to-End Tests

WebTestClient

WebTestClient documentation (https://docs.spring.io/spring-framework/reference/testing/webtestclient.html)

`WebTestClient` is an HTTP client used for performing end-to-end HTTP tests in Spring applications. It allows testing against:

- a mock server (without starting a real HTTP server)
- a running server (when bound to a `@SpringBootTest` with `RANDOM_PORT`)

Example use cases include verifying HTTP endpoints, request/response behavior, and full request processing logic in Spring WebFlux or Web MVC applications.

WebTestClient Setup Options

You can bind `WebTestClient` to:

- application context via mock infrastructure
- a real HTTP server using `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)`

Cucumber

Cucumber documentation (https://cucumber.io/)

Cucumber is a popular open-source BDD (Behavior-Driven Development) tool. It allows writing executable specifications in plain English using **Gherkin** syntax.

Key Concepts

- A **scenario** is a test case written in Gherkin.
- Each scenario consists of steps: **Given**, **When**, and **Then**.
- Steps are mapped to Java methods via **step definitions**.

- Scenarios are stored in `.feature` files under `src/test/resources`.

Scenarios are written before production code to define behavior. Once implemented, they serve as both **documentation** and **automated tests**.

Example file structure:

Software Requirements Specification

A **Software Requirements Specification (SRS)** is a technical document that defines what a software product must do and how it must be developed. It outlines the application's functionality, features, design constraints, limitations, and overall goals.

The SRS serves two primary purposes:

- **For clients** it defines expectations, deliverables, and scope.
- **For developers** it guides planning, effort estimation, technology stack selection, and project costing.

The structure and depth of an SRS can vary depending on project complexity and development methodology. However, all well-formed SRS documents typically include the following elements.

Purpose

A clear and concise statement defining the intent of the software. It outlines what the system must achieve once completed and what problem it solves.

Product Description

The definition of the system's expected behavior, including:

- The intended users
- The operational environment (e.g., web, mobile, desktop)
- Assumptions and constraints that may affect development or operation

Functional Requirements

Definition of the system's required behavior — what the system must do in response to specific inputs or conditions. Typically expressed as: `The system shall <perform a`

function>.

Non-Functional Requirements

Definition of the system's required qualities — such as performance, reliability, scalability, usability, and security. Typically expressed as: `The system shall <meet a quality requirement>.`

User Stories

A structured way to describe expected system behavior from the user's perspective. Each story includes:

- **Title** – A concise description of the feature or capability
- **Narrative** – `As a <type of user>, I want <some goal>, so that <some reason>`
- **Acceptance Criteria** – Scenarios written in Gherkin syntax, using:
  - *Given* (initial state)
  - *When* (triggering event or action)
  - *Then* (expected result)

Test Cases

Used to verify that a specific function or feature of the software behaves as required. Follows the **AAA pattern**:

- **Arrange** – Set up the necessary data and environment
- **Act** – Execute the function or feature being tested
- **Assert** – Check that the result matches the expected outcome

YAML

YAML stands for **"YAML Ain't Markup Language"**. It is a human-readable data serialization format commonly used for configuration files and data exchange between languages. YAML is widely used in modern software tools and infrastructure. Files typically have `.yaml` or `.yml` extensions.

Basic Data Types

YAML supports several standard data types:

- **Integers**: `15`, `123`
- **Strings**: `"15"`, `'Hello, YAML!'` (use single or double quotes)
- **Floats**: `15.033`
- **Booleans**: `true`, `false`
- **Null**: `null`

YAML tries to auto-detect types. To explicitly declare a type, use `!!`. Example: `!!str yes` forces YAML to treat `yes` as a string.

Maps (Key–Value Pairs)

YAML maps are structured as key-value pairs.

```yaml
object: Book

metadata:
  name: Three Men in a Boat
  author: Jerome K Jerome
  genre: humorous account

published:
```

```
  year: 1889
  country: United Kingdom
```

- The `---` line is optional unless you're defining multiple documents.
- Keys and values are separated by a colon `:`, followed by a space.
- Use **spaces** for indentation, **not tabs**.

Lists (Sequences)

YAML supports block-style and flow-style lists.

Block Style:

```
animals:
  - cat
  - dog
  - bird
```

Each list item starts with a `-` followed by a space.

Flow Style:

```
animals: [ cat, dog, bird ]
```

Combining Maps and Lists

YAML allows nesting of maps and lists.

```
weekend:
  saturday:
    - order cleaning
    - order a pizza
    - watch new series
  sunday:
```

```
    - go to yoga
    - hang out with a friend
```

You can nest lists inside maps, maps inside lists, and so on.

Multi-line Strings

Use `|` to preserve newlines in multi-line strings.

```
saturday: |
  order cleaning
  order a pizza
  watch new series
```

Comments

Use `#` to add comments.

```
# This is a comment
metadata: # inline comment
  name: Three Men in a Boat
  author: Jerome K Jerome
  genre: humorous account
```

Keycloak

OAuth 2.0

OAuth 2.0 defines four key roles:

- **Resource Owner**: The end user who owns the data.
- **Resource Server**: Hosts the protected resources (e.g., REST APIs).
- **Client**: The application accessing resources.
- **Authorization Server**: Issues access tokens (e.g., Keycloak).

OAuth 2.0 Flows:

- **Client Credentials Flow**: For machine-to-machine communication where the application accesses resources on its own behalf.
- **Device Flow**: For devices without browsers (e.g., smart TVs).
- **Authorization Code Flow**: Default for most web/mobile applications.

Client Types:

- **Confidential Clients**: Can securely store credentials (e.g., backend services).
- **Public Clients**: Cannot store credentials securely (e.g., single-page apps).

Safeguards for public clients:

- Use pre-configured redirect URIs.
- Use **PKCE** (Proof Key for Code Exchange) to prevent token interception.

Tokens:

- **Access Token**: Short-lived; passed in requests to access resources.

- **Refresh Token**: Long-lived; allows refreshing access tokens without re-authentication.
- **ID Token**: JWT containing user identity; unlike opaque OAuth tokens, it is inspectable.

OpenID Connect

OpenID Connect is an authentication layer built on top of OAuth 2.0.

Key Roles:

- **End User**: The person authenticating.
- **Relying Party (RP)**: The application requesting user authentication.
- **OpenID Provider (OP)**: The identity provider (e.g., Keycloak).

OpenID Connect Flows:

- **Authorization Code Flow**: Returns ID token, access token, and refresh token.
- **Hybrid Flow**: Returns ID token and authorization code immediately.

OpenID Connect requires `scope=openid` to initiate authentication.

Additional Specs:

- **Discovery**: Dynamic provider configuration.
- **Dynamic Registration**: Clients can register themselves dynamically.
- **Session Management**, **Front-Channel Logout**, **Back-Channel Logout**: For logout and session lifecycle handling.

Keycloak Overview

Keycloak acts as an identity and authorization server.

Endpoints:

- **Frontend**: User login, logout, and consent.

- **Backend**: Token issuance and validation.
- **Admin**: Realm and client management.

Roles:

- **Realm Roles**: Assigned at realm level (organization-wide).
- **Client Roles**: Specific to a client.
- **Composite Roles**: Combine multiple roles.

> ⚠ Prefer **groups** over composite roles for assigning multiple roles to users.

Groups:

- Not included in tokens by default.
- Add a **protocol mapper** to include group info in tokens.

Authorization Services:

- Define policies and resource access rules.
- Manage via Keycloak Admin UI or REST API.

Spring Boot Integration

Spring Boot Client (Web App)

Dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Configuration (`application.yaml`):

```
spring:
  security:
    oauth2:
      client:
        registration:
          myfrontend:
            provider: keycloak
            client-id: mywebapp
            client-secret: CLIENT_SECRET
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/"
            scope: openid
        provider:
          keycloak:
            authorization-uri:
http://localhost:8180/auth/realms/myrealm/protocol/openid-connect/auth
            token-uri:
http://localhost:8180/auth/realms/myrealm/protocol/openid-connect/token
            jwk-set-uri:
http://localhost:8180/auth/realms/myrealm/protocol/openid-connect/certs
```

Replace `CLIENT_SECRET` with the value from Keycloak.

Spring Boot Resource Server (Backend API)

Dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Configuration (`application.yaml`):

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8180/auth/realms/myrealm
```

Get Access Token (Resource Owner Password Flow):

```
export access_token=$(curl -X POST \
  http://localhost:8180/auth/realms/myrealm/protocol/openid-
connect/token \
  --user mybackend:CLIENT_SECRET \
  -H 'content-type: application/x-www-form-urlencoded' \
  -d 'username=alice&password=alice&grant_type=password' \
  | jq --raw-output '.access_token')
```

Use Token to Access API:

```
curl -X GET http://localhost:8080 \
  -H "Authorization: Bearer $access_token"
```

Running Keycloak with Docker

Run Keycloak Dev Mode:

```
docker run -p 28080:8080 \
  -e KEYCLOAK_ADMIN=admin \
  -e KEYCLOAK_ADMIN_PASSWORD=admin \
  quay.io/keycloak/keycloak:18.0.0 start-dev
```

Admin Console: http://localhost:28080/admin/ (http://localhost:28080/admin/)

Get Started with Docker: https://www.keycloak.org/getting-started/getting-started-docker (https://www.keycloak.org/getting-started/getting-started-docker)

OpenSSH

Reference: ssh.com/academy/ssh (https://www.ssh.com/academy/ssh)

OpenSSH is a suite of secure networking tools based on the SSH protocol. It includes both client and server components, and supports secure remote login, file transfer, and tunneling.

Key OpenSSH Tools

- `sshd`: OpenSSH server daemon.
- `ssh`: Establishes a secure shell session on a remote system.
- `scp`: Secure file transfer between local and remote systems.
- `ssh-copy-id`: Installs your public key on a remote server's `authorized_keys` file.
- `ssh-keyscan`: Collects public host keys from remote hosts.
- `ssh-keygen`: Creates and manages authentication keys.
- `ssh-agent`: Caches private key passphrases to enable automatic authentication.
- `ssh-add`: Adds private keys to the `ssh-agent`.
- `sshfs`: Mounts a remote filesystem via SSH.

Basic Usage

```
ssh username@host -p port
```

SSH Server Configuration

Check if SSH Server is Running

```
systemctl status sshd
```

Allow SSH Port Through Firewall

```
sudo ufw allow 22/tcp
```

Modify SSH Daemon Configuration

Edit the file:

```
sudo nano /etc/ssh/sshd_config
```

To reload the configuration:

```
sudo systemctl reload sshd.service
```

Enable strict mode to check file and directory permissions:

```
StrictModes yes
```

Validate SSH Configuration

```
sudo sshd -t
```

(Silent if no errors)

Regenerate Host Keys

```
sudo rm /etc/ssh/ssh_host*
sudo ssh-keygen -A
sudo chmod 400 /etc/ssh/ssh_host*
sudo chmod 644 /etc/ssh/ssh_host*.pub
```

> ⚠ SSH host keys should **not** be password-protected.

SSH Client Configuration

Create or edit the file `~/.ssh/config`:

```
Host qwerty
  Hostname 10.0.0.1
  User root
  Port 2222
```

Then connect using:

```
ssh qwerty
```

Generate a New Key

```
ssh-keygen -t ed25519
```

> ⚠ The `ed25519` algorithm is preferred over `rsa` for better security.

Send Public Key to Server

```
ssh-copy-id -i ~/.ssh/id_rsa.pub user@host
```

This adds the public key to the server's `~/.ssh/authorized_keys`.

Useful Commands

Check Failed Login Attempts

```
cat /var/log/secure | grep "Failed password for"
```

Send File Over SSH

```
scp ~/test.txt user@host:documents
```

Run GUI Applications Remotely

1. On the server: ensure `X11Forwarding yes` in `/etc/ssh/sshd_config`.

2. Then run:

```
ssh -XC user@remotehost "eclipse"
```

Run Local Script on Remote Host

```
ssh user@host 'bash -s' < script.sh
```

Docker

Docker Engine

Docker Engine is the runtime that builds and runs Docker containers. It is composed of several key stages:

Build

Developers create containerized applications by packaging code, dependencies, and environment configurations into read-only **Docker images**.

```
docker image pull ubuntu:latest
```

Ship

Docker images are stored and shared using Docker registries such as **Docker Hub** (default public registry).

```
docker search alpine --filter "is-official=true"
```

Run

Docker **containers** are runtime instances of images.

```
docker container run -it --restart always -p 80:8080 ubuntu:latest
/bin/bash
docker container run -d --name c1 -p 80:8080 web:latest
```

Common docker run options

Option	Purpose
`-t`	Allocate a pseudo-TTY
`-d`	Run container in background
`--name`	Name the container
`--rm`	Remove container after exit
`-p`	Map container port to host
`-e`	Set environment variables
`-m`	Set memory limit

```
docker run --name hello -d --rm hello-java
```

> ⚠ A container must have a main process to stay alive.

```
docker container exec -it vigilant_borg bash
Ctrl+P+Q    # Detach from container shell
docker container stop vigilant_borg
docker container rm vigilant_borg
docker container rm $(docker container ls -aq) -f
```

Dockerfile

A Dockerfile defines instructions to build an image.

```
FROM alpine
LABEL maintainer="example@example.com"
COPY . /src
```

```
WORKDIR /src
RUN apk add --update nodejs npm
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "app.js"]
```

Command	Purpose
`FROM`	Base image
`COPY`	Copy files into image
`ADD`	Like COPY, but supports remote URLs and tar unpacking
`ENV`	Set environment variable
`RUN`	Execute command during build
`CMD`	Default command when running container
`EXPOSE`	Document listening ports
`VOLUME`	Define mount point
`WORKDIR`	Set working directory
`LABEL`	Add metadata

⚠ Use `.dockerignore` to exclude files from build context.

Images, Volumes, and Networks

```
docker image rm <id>           # Remove image
docker image prune -a          # Remove all unused images
docker volume prune            # Remove unused volumes
docker network ls              # List networks
docker volume ls               # List volumes
```

Build and Push Image

```
docker image build -t username/app:latest .
docker tag <image-id> username/app:latest
docker login
docker push username/app:latest
```

Docker Compose

Tool for defining and running multi-container applications.

`docker-compose.yml` example:

```yaml
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - type: volume
        source: counter-vol
        target: /code
    networks:
      - counter-net

  redis:
    image: redis:alpine
    networks:
      - counter-net
```

```
networks:
  counter-net:

volumes:
  counter-vol:
```

```
docker-compose up -d
docker-compose down
docker-compose ps
docker-compose stop
docker-compose restart
docker-compose rm
```

⚠ Restart policies: `always`, `unless-stopped`, `on-failure`

Kubernetes Overview

Kubernetes manages containerized applications across clusters of machines.

Concepts

- **Pod**: Smallest deployable unit; holds one or more containers.
- **Node**: A worker machine (VM or physical).
- **Cluster**: Set of nodes managed by master nodes.
- **Service**: Abstraction for exposing pods.
- **ReplicationController**: Ensures a specific number of pod replicas are running.
- **Label**: Key-value metadata for selection and grouping.
- **Kubelet**: Agent that runs on each node.

Pod Definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: couchbase-pod
spec:
  containers:
    - name: couchbase
      image: couchbase
      ports:
        - containerPort: 8091
```

ReplicationController

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase-controller
spec:
  replicas: 2
  selector:
    app: couchbase-rc-pod
  template:
    metadata:
      labels:
        app: couchbase-rc-pod
    spec:
      containers:
        - name: couchbase
          image: couchbase
          ports:
            - containerPort: 8091
```

Service Definition

```yaml
apiVersion: v1
kind: Service
metadata:
  name: couchbase-service
  labels:
    app: couchbase-service-pod
spec:
  ports:
    - port: 8091
  selector:
    app: couchbase-rc-pod
```

MERN Stack with Docker

Example to run MongoDB:

```
docker run -d -p 27017-27019:27017-27019 --name mongodb-service
mongo:4.0.4
```

Project structure:

- `server`: Node.js (Express + Socket.IO)
- `client`: React app
- `worker`: Libraries or background jobs

Resources

- Play with Docker (https://labs.play-with-docker.com/)
- Dockerfile Best Practices (https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
- Dockerfile Reference (https://docs.docker.com/engine/reference/builder/)

- Alpine Docker Image (https://hub.docker.com/_/alpine/)