Hogeschool PXL

PXL-Digital

Java Expert

Author

Nele Custers

March 27, 2025

Contents

Chapter 1

Mockito

Learning goals
The junior-colleague

1. can describe the usage of mockito.
2. can describe what a mock object is.
3. can describe what stubbing is.
4. can implement unit tests with mockito.
5. can use MockMvc to test a controller's methods.

1.1 Mocking with Mockito

Mockito is a popular open source framework for mocking objects when testing software. Mocking an object means creating a lightweight, configurable imitation of a real object. In automated tests, mocks simulate the behavior of complex, real objects (like database connections, network services, or other external dependencies) when it's impractical or inefficient to use the real ones.

```java
package be.pxl;

public class MyService {

    private final OtherService otherService;

    public MyService(OtherService otherService) {
        this.otherService = otherService;
    }

    public int doCalculation(int value) {
        return value * otherService.getValue();
    }
}
```

```java
package be.pxl;
```

```
public interface OtherService {

    int getValue();
}
```

When writing a unit test for the method doCalculation() we need an object of the class OtherService. We don't need an object of an actual implementation of the interface. We can just mock the OtherService-object and tell it what it should return when it's called.

```
package be.pxl;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class MyServiceTest {

    @Mock
    private OtherService otherService;

    @InjectMocks
    private MyService myService;

    @Test
    public void doCalculationTest() {
        when(otherService.getValue()).thenReturn(5);
        assertEquals(500, myService.doCalculation(100));
    }
}
```

1.2 Creating fake (or dummy) objects

There are scenarios where mocking a data object might be useful, particularly when:

- **The object is difficult to instantiate**: If the object has a complex setup or requires dependencies that are not relevant to the test at hand.
- **The object has behavior that's being tested elsewhere**: If the object includes logic (e.g., methods that compute values based on its fields) and you want to isolate the unit test from this behavior.
- **The object's state is irrelevant to the test's outcome**: If you're testing

functionality that doesn't depend on the specific state of the object, and creating a real instance would distract from the intent of the test.

```
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoSettings;
import org.mockito.quality.Strictness;

@ExtendWith(MockitoExtension.class)
public class ClassroomTest {

    @Mock
    private Teacher mockTeacher;

    @Test
    public void testClassroomHasTeacher() {
        Classroom classroom = new Classroom()
        assertFalse(classroom.hasTeacher());

    }

    @Test
    public void testClassroomHasTeacher() {
        Classroom classroom = new Classroom()

        classroom.addTeacher(mockTeacher);
        assertTrue(classroom.hasTeacher());
    }

    @Test
    public void testClassroomGetTeacherName() {
        when(mockTeacher.getName()).thenReturn("Teacher Name");

        Classroom classroom = new Classroom();
        classroom.addTeacher(mockTeacher);

        assertEquals(classroom.getTeacherName(), "Teacher Name");
    }
}
```

We want to configure the mock and define what to do when specific methods of the mock are called. This is called **stubbing**.

Mockito offers two ways of stubbing. The first way is "when this method is called, then do something". This strategy uses Mockito's thenReturn call:

```
when(passwordEncoder.encode("1")).thenReturn("a");
```

In plain English: "When passwordEncoder.encode("1") is called, return an a."

The second way of stubbing reads more like "do something when this mock's method is called with the following arguments." This way of stubbing is harder to read as the cause is specified at the end. Consider:

```
doReturn("a").when(passwordEncoder).encode("1");
```

The snippet with this method of stubbing would read: "Return a when passwordEncoder's encode() method is called with an argument of 1."

For mocking void methods, there is no when-then option. We have to use do-when syntax.

1.3 Unit testing the service-layer methods

We can use Mockito to test the services of a spring boot application by mocking the repository.

```
@ExtendWith(SpringExtension.class)
public class BookServiceTest {

    @Mock
    private BookRepository bookRepository;

    @InjectMocks
    private BookService bookService;

    @Test
    public void testFindAllBooks() {
        // Arrange
        Book book1 = new Book("Book 1", "Author 1", 2019);
        Book book2 = new Book("Book 2", "Author 2", 2020);
        List<Book> books = Arrays.asList(book1, book2);
        when(bookRepository.findAll()).thenReturn(books);

        // Act
        List<BookDTO> result = bookService.findAllBooks();

        // Assert
        assertNotNull(result);
        assertEquals(2, result.size());
        // additional assertions about the retrieved DTO's
        verify(bookRepository).findAll();
    }
}
```

1.4 ArgumentCaptor

An ArgumentCaptor is used with Mockito to capture argument values for further assertions. They are particularly useful when we want to inspect complex objects passed to methods of mocked dependencies.

Below is an example demonstrating how to use an ArgumentCaptor in a unit test with Mockito. We'll test a UserService that depends on a UserRepository. The service method createUser takes a User object, does some processing, and then saves it using the repository. We want to ensure that the User object passed to the repository's save method has been processed correctly by the service.

First, we define a simple User class and a UserRepository interface.

```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int age;

    // Getters and setters
}



import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {

}
```

```java
import org.springframework.stereotype.Service;

@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User createUser(String name, int age) {
        User user = new User();
        user.setName(name);
        user.setAge(age);
        // Additional processing logic
        return userRepository.save(user);
    }
}
```

Now, we'll write a test for UserService using Mockito and an ArgumentCaptor to capture the User object passed to the save method of UserRepository.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.verify;

@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;
    @Captor
    private ArgumentCaptor<User> userArgumentCaptor;
    @InjectMocks
    private UserService userService;

    @Test
    public void createUserTest() {

        userService.createUser("John Doe", 30);

        // Capture the user object passed to save
        verify(userRepository).save(userArgumentCaptor.capture());

        // Retrieve the captured value
        User capturedUser = userArgumentCaptor.getValue();

        // Assert the state of the captured user
        assertEquals("John Doe", capturedUser.getName());
        assertEquals(30, capturedUser.getAge());
    }
}
```

After createUser is called on the UserService, verify(mockRepository).save(userArgumentCaptor.capture captures the User object passed to the save method. The captured object is then retrieved using userArgumentCaptor.getValue() for assertions.

This test ensures that the UserService correctly processes and passes a User object to the UserRepository. ArgumentCaptor is a powerful tool for asserting the indirect outputs of methods when direct return value assertions are not sufficient.

1.5 MockMvc

MockMvc is a core part of the Spring Framework's testing suite which allows you to test Spring MVC controllers without the need to start a full HTTP server. It provides a powerful way to quickly test MVC controllers by performing requests and asserting responses with a fluent API. By simulating HTTP requests and responses, developers can ensure their web layer conforms to specifications.

MockMvc is defined as the main entry point for server-side Spring MVC testing. Tests annotated with @WebMvcTest will auto-configure Spring Security and MockMvc. These tests will also auto-configure all classes relevant to MVC tests (e.g. @Controller, @ControllerAdvice, ...). Using the annotation @MockitoBean we can provide mock objects for the @Service components. By default, @WebMvcTest adds all @Controller beans to the application context. We can specify a subset of controllers by using the controllers attribute, as we have done in the examples discussed here. The method perform in the class MockMvc performs a request and returns a type that allows chaining further actions, such as asserting expectations, on the result. When implementing unit tests, we pass the path parameters using MockMvcRequestBuilders and verify the status response codes and response content using MockMvcResultMatchers and MockMvcResultHandlers.

Assuming a basic Book class and a BookService interface.

```
public class Book {
    private Long id;
    private String title;
    private String author;

    // Constructors, Getters, and Setters
}
```

```
public interface BookService {
    Book saveBook(Book book);
    Book getBookById(Long id);
    Book updateBook(Long id, Book book);
    void deleteBook(Long id);
}
```

Here is the BookController under test:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
```

```java
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/books")
public class BookController {

    private final BookService bookService;

    @Autowired
    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @PostMapping
    public ResponseEntity<Book> createBook(@RequestBody Book book) {
        Book savedBook = bookService.saveBook(book);
        return new ResponseEntity<>(savedBook, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id) {
        Book book = bookService.getBookById(id);
        return ResponseEntity.ok(book);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable Long id,
        ↪ @RequestBody Book book) {
        Book updatedBook = bookService.updateBook(id, book);
        return ResponseEntity.ok(updatedBook);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        bookService.deleteBook(id);
        return ResponseEntity.noContent().build();
    }
}
```

Let’s write the MockMvc tests for this BookController. We’ll mock the BookService since we want to isolate the controller layer for unit testing.

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.Test;
import org.mockito.ArgumentMatchers;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;


@WebMvcTest(BookController.class)
public class BookControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private BookService bookService;

    private ObjectMapper objectMapper = new ObjectMapper();

    @Test
    public void createBookTest() throws Exception {
        Book newBook = new Book("The Great Gatsby", "F. Scott Fitzgerald");
        Mockito.when(bookService.saveBook(ArgumentMatchers.any(Book.class))).thenReturn(r

        mockMvc.perform(MockMvcRequestBuilders.post("/books")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(newBook)))
                .andExpect(MockMvcResultMatchers.status().isCreated())
                .andExpect(MockMvcResultMatchers.jsonPath("$.title").value("The
                    ↪ Great Gatsby"));
    }

    @Test
    public void getBookByIdTest() throws Exception {
        Book book = new Book("The Great Gatsby", "F. Scott Fitzgerald");
        Mockito.when(bookService.getBookById(1L)).thenReturn(book);

        mockMvc.perform(MockMvcRequestBuilders.get("/books/{id}", 1))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.title").value("The
                    ↪ Great Gatsby"));
    }

    @Test
    public void updateBookTest() throws Exception {
        Book updatedBook = new Book("The Great Gatsby", "Fitzgerald");
        Mockito.when(bookService.updateBook(ArgumentMatchers.eq(1L),
            ↪ ArgumentMatchers.any(Book.class))).thenReturn(updatedBook);

        mockMvc.perform(MockMvcRequestBuilders.put("/books/{id}", 1)
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(updatedBook)))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.author").value("Fitzgerald"))
    }

    @Test
    public void deleteBookTest() throws Exception {
        Mockito.doNothing().when(bookService).deleteBook(1L);

        mockMvc.perform(MockMvcRequestBuilders.delete("/books/{id}", 1))
                .andExpect(MockMvcResultMatchers.status().isNoContent());
    }
}
```

In these tests, we use @WebMvcTest to bring up a context that focuses solely on the web layer (specifically, the BookController), and @MockitoBean to create and inject a mock BookService instance. This setup isolates the controller layer, allowing for focused unit testing of web interactions.

Chapter 2

Maven

Learning goals
The junior-colleague
1. can explain what Maven is.
2. can describe the 3 build lifecycles of Maven.
3. can explain that each build lifecycle is made up of phases.
4. can explain that each build phase is made up of plugin goals.
5. can identify the project coordinates.
6. can describe the Maven Standard Directory Layout.
7. can explain what a dependency is.
8. can describe the different dependency scopes.
9. can explain what a transitive dependency is.
10. can explain what a dependency tree is.
11. can execute Maven commands from CLI.
12. can manage dependencies with Maven.
13. understand the role of Maven in a Continuous Integration (CI) pipeline.
14. can configure Maven for automated builds and testing in a CI/CD workflow.
15. can implement a basic CI pipeline using GitHub Actions and Maven.

2.1 What is Maven?

Maven is a tool that can be used for building and managing Java-based projects. Every Java project requires certain dependencies, which are automatically downloaded when using Maven.

The main objectives of using Maven for developers are:

- Making the build process easy
- Providing a uniform build system
- Providing information about project quality (for example unit test reports)
- Encouraging better development practices

Maven is much more than a build tool. Maven offers support for automatic source generation, compiling the source code and test sources, packaging final product(s) for different environments, running health checks and reporting. Continuous builds, integration, and testing can be easily handled by using Maven. A tool commonly used for creating CI/CD pipelines (continuous integration and continuous delivery) is Jenkins. Jenkins cannot replace Maven or vice-versa. Jenkins can use Maven as its build tool. When Jenkins deploys artifacts to remote repositories, they are usually Maven repositories.

According to Jetbrain's 2023 Java ecosystem report [1] Maven is most popular build tool for the Java ecosystem.

2.2 Installation and Configuration

IntelliJ IDEA has in-built support for Maven. However we want to make Maven available in the command-line interface. Therefore we need to install Maven.

Maven download and documentation pages:
Download page: `https://maven.apache.org/download.cgi`
Installation instructions: `https://maven.apache.org/install.html`

Confirm that Maven is installed correctly by executing `$mvn -v` in a terminal window.

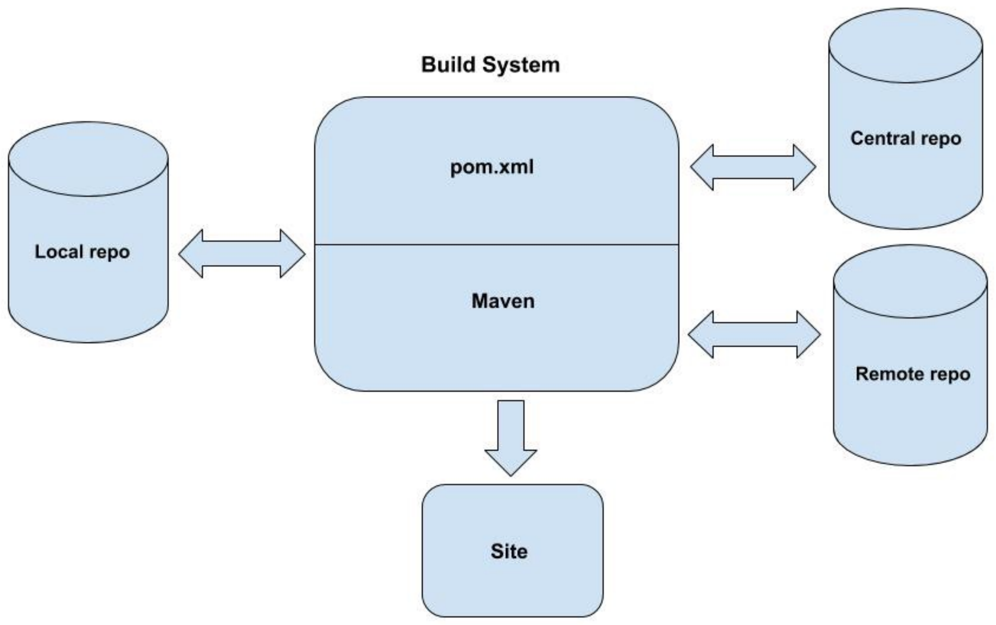
```
$ mvn -v
Apache Maven 3.8.2 (ea98e05a04480131370aa0c110b8c54cf726c06f)
Maven home: /usr/local/Cellar/maven/3.8.2/libexec
Java version: 17.0.1, vendor: Oracle Corporation, runtime:
    ↪ /Library/Java/JavaVirtualMachines/jdk-17.0.1.jdk/Contents/Home
Default locale: nl_BE, platform encoding: UTF-8
OS name: "mac os x", version: "11.5.2", arch: "x86_64", family: "mac"
```

2.3 Maven Standard Directory Layout

When you create a new Maven project, a directory which name matches the artifactId is generated. This directory is known as the project's base directory. Every Maven project has a file named **pom.xml** which is known as the **Project Object Model** (POM). This file describes the project, configures plugins, and declares dependencies. The project's source code and resources are placed in the folder **src/main**. The project's unit tests are located in **src/test**. The target folder is the Maven default output folder. You can delete all the target's folder content with the `$mvn clean` command.

[1] https://www.jetbrains.com/lp/devecosystem-2023/java/

2.4 Maven architecture



Central repository is provided by the Maven community. This Central repository contains a large number of common libraries. Whenever you specify a dependency in pom.xml, Maven will look for it in the local repository first. If the necessary dependency isn't included in the local repository, Maven will download it from the Central repository and copy it to your local repository. Remote repository is a place where developers (or Jenkins) can copy the final packages, so other developer can use these final packages as a dependency in their projects.

Acces to internet is recommended when using Maven, however it is possible to work offline if all dependencies are available in your local repository.

2.5 Maven built-in life cycles

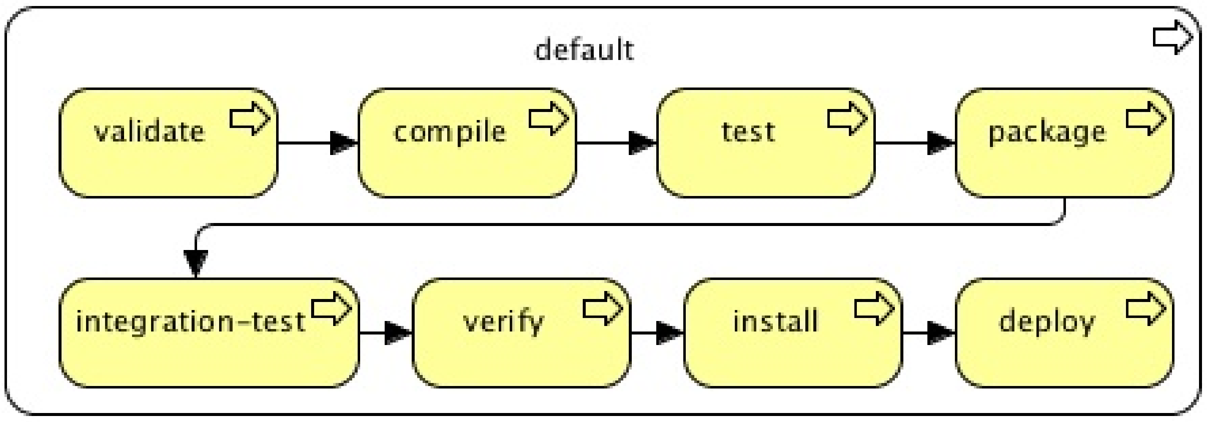
There are three built-in life cycles defined in Maven. The *default* build life cycle is the main build life cycle.

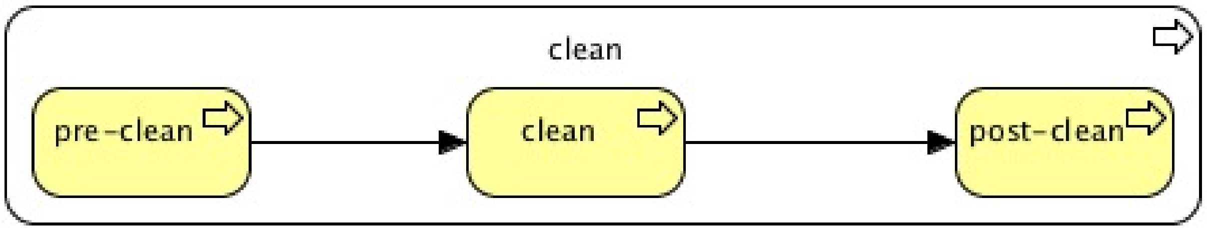
clean	handles the cleanup of directories and files generated during the build process	mvn clean
default	handles the build and distribution of the project	mvn [plugin:goal]* [phase]*
site	create project documentation	mvn site

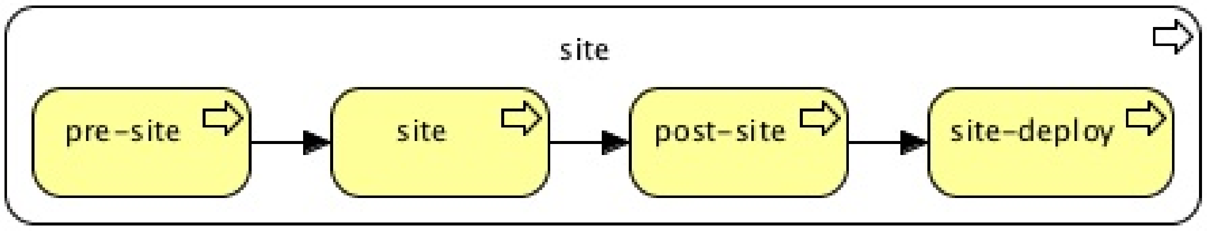
To run a specific goal, without executing the entire phase (and the preceding phases) the command `$mvn [plugin:goal]` can be used. Phases on the other hand are executed in a specific order and all the preceding phases are executed as well!

2.6 A Build Lifecycle is Made Up Of Phases

Each life cycle consists of a sequence of phases. The default build lifecycle consists of 23 phases. In the image below the 8 main phases are included. On the other hand, clean lifecycle consists of 3 phases, while the site lifecycle is made up of 4 phases.





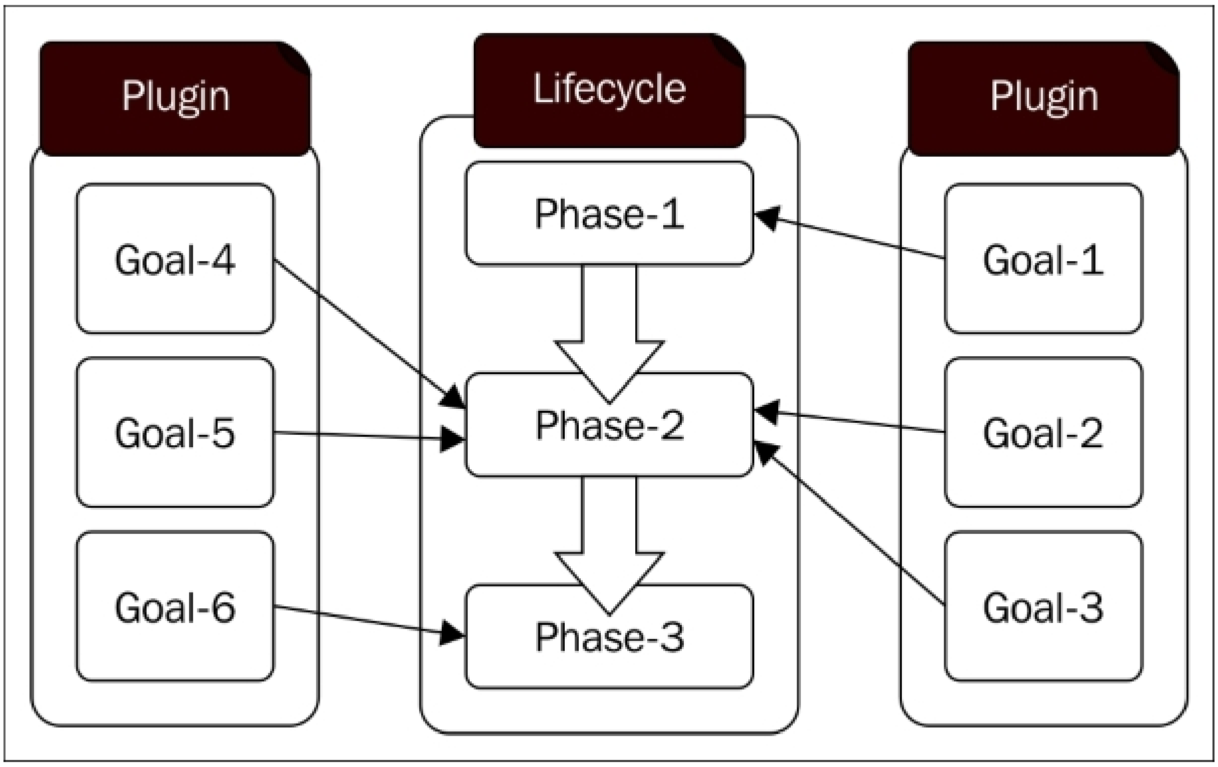


Each phase is responsible for a specific task. Here are the 8 most important phases in the default build lifecycle.

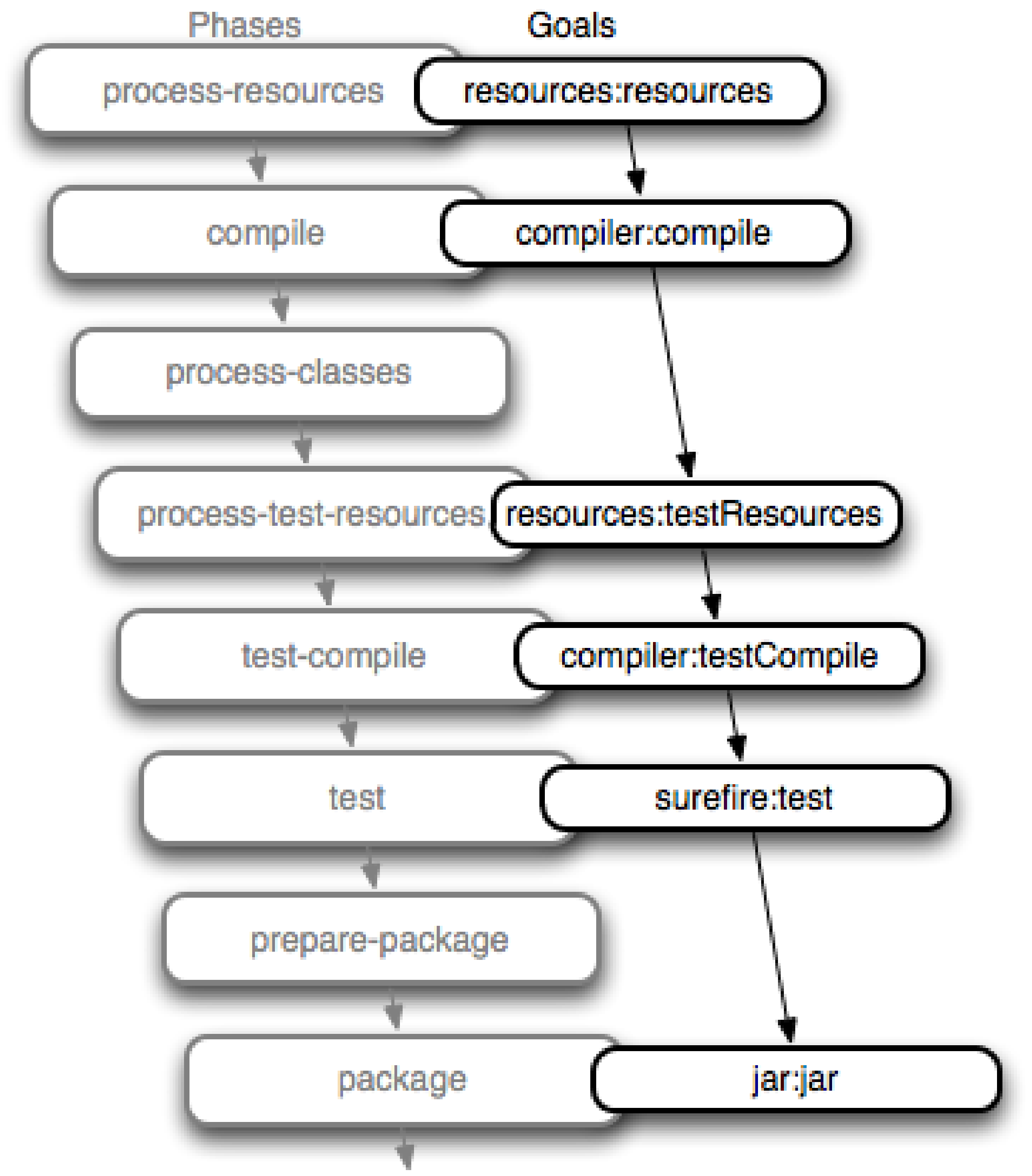
validate	check if all information necessary for the build is available
compile	compile the source code
test	run unit tests
package	package compiled source code into the distributable format, (jar, war, . . .)
integration-test	process and deploy the package if needed to run integration tests
verify	run any checks on results of integration tests to ensure quality criteria are met
install	install the package to a local repository
deploy	copy the package to the remote repository

2.7 Plugins and Goals

Maven is actually a plugin execution framework. Every task executed by Maven is actually done by a **goal**, where goals are grouped together in **plugins**. When we run a phase, all the goals bound to the phase are executed in order.



Several phases of the default built-in lifecycles have goals bounded to them. Due to this default configuration you're able to build a Java project without extra configuration.



Note: There are more phases than shown above, this is a partial list

The compile goal from the compiler plugin is bound to the compile phase and is responsible for compiling the source code. The test goal from the surefire plugin is bound to the test phase and is reponsible for running the unit tests.

You can generate an overview of all the phases and the specific goals bounded to these phases by running the command $\boxed{\text{\$mvn help:describe -Dcmd=PHASENAME}}$

```
$ cd superhero-backend/
$ mvn help:describe -Dcmd=compile
[INFO] Scanning for projects...
[INFO]
```

```
[INFO] ----------------------< be.pxl:superhero-backend
    ↪ >----------------------
[INFO] Building superhero-backend 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar
    ↪ ]---------------------------------
[INFO]
[INFO] --- maven-help-plugin:3.3.0:describe (default-cli) @
    ↪ superhero-backend ---
[INFO] 'compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle
    ↪ includes the following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources:
    ↪ org.apache.maven.plugins:maven-resources-plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources:
    ↪ org.apache.maven.plugins:maven-resources-plugin:2.6:testResources
* test-compile:
    ↪ org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy

[INFO]
    ↪ ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
    ↪ ------------------------------------------------------------------------
[INFO] Total time: 0.896 s
[INFO] Finished at: 2023-02-26T17:19:03+01:00
[INFO]
```

```
↪ ------------------------------------------------------------------
```

There are two types of plugins:

- **Build plugins**: The goals of build plugins are executed during the build process. If you want to execute additional goals during a phase of the build process, you need to configure the goal in the `<build>` element of the pom.xml.
- **Reporting plugins**: Reporting plugins are executed during the site generation process. These plugins are configured in the `<reporting>` element of the pom.xml.

2.8 Dependencies and scopes

A repository in Maven stores artifacts and dependencies of varying types. There are exactly two types of repositories: local and remote.

The **local repository** is a directory on the machine that runs Maven. By default Maven's local repository is located in the folder $user.home/.m2/repository. As you can see, this is a hidden folder. If you're unable to find the default .m2 folder, you can run the following command: `$mvn help:evaluate -Dexpression=settings.localRepository`

Remote repositories refer to any other type of repository, accessed by a variety of protocols such as file:// and https://.

Exercise 2.1. Locate and open the folder containing your local repository.

If you're looking for a dependency to add, you can use the Maven Central Repository Search [2].

There are two types of dependencies in Maven: direct and transitive. **Direct dependencies** are the dependencies that are defined in the pom.xml under the `<dependencies>` section. **Transitive dependencies** are dependencies of your direct dependencies. This means if your project needs dependency A and A depends on B, then your project needs both A and B. However, for Maven it suffices to add dependency A in the pom.xml. The transitive dependency B is included automatically.

You can generate the dependency tree with direct and transitive dependencies by running `$mvn dependency:tree`.

Sometimes, transitivity brings a very serious problem causing version mismatch issues at runtime. when multiple versions of the same artifact are encountered, Maven picks the "nearest definition". It uses the version of the closest dependency in the dependency tree.

```
A
\-- B
    \-- C
        \-- D 2.0
\-- E
```

[2]https://search.maven.org

```
    \-- D 1.0
```

D 1.0 will be used when building project A because the path from A to D through E is shorter. You can explicitly add a dependency to D 2.0 in A to force the use of D 2.0.

```
A
 \-- B
    \-- C
       \-- D 2.0
 \-- E
    \-- D 1.0
 \-- D 2.0
```

There are 6 dependency scopes which are used to limit the transitivity of dependencies and determine when a dependency should be included in the classpath.

compile	This is the default scope. Depencencies with this scope are available on the classpath for all the build tasks.
provided	Dependencies that are provided at runtime by JDK or a container. The provided dependencies are available at compile-time and in the test classpath.
runtime	The dependencies with this scope are only required at runtime. They are not needed at compile-time and in the test classpath.
test	These dependencies are only needed for executing tests.
system	Similar to provided but specific jar is provided.
import	All dependencies listed in another pom are included.

2.9 Maven commands

Here is a brief list of usefull Maven commands.

Maven Command	**Description**
mvn -v	Prints out the version of Maven you are running.
mvn –version	Same as mvn -v
mvn clean	Clears the target directory into which Maven normally builds your project.
mvn package	Builds the project and packages the resulting JAR file into the target directory.
mvn package -DskipTests	Builds the project and packages the resulting JAR file into the target directory without running the unit tests during the build. You can also use -Dmaven.test.skip=true
mvn clean package	Clears the target directory, builds the project and packages the resulting JAR file into the target directory.
mvn install	Builds the project described by your Maven POM file and installs the resulting artifact (JAR) into your local Maven repository.
mvn -X package	Prints the maven version and runs the build in the debug mode.
mvn -o package	This command is used to run the maven build in the offline mode.
mvn -help	Prints the Maven usage and all the available options.
mvn dependency:tree	Generates the dependency tree of the Maven project.

2.10 Maven in the CI/CD Pipeline

Maven is a fundamental tool in modern Continuous Integration (CI) pipelines. It simplifies project builds, dependency management, and testing, making it an essential component of an automated software development workflow. In a CI environment, developers push code changes frequently, triggering automated builds and tests to ensure the code remains stable and functional. Maven streamlines this process by providing a standardized project structure and an efficient way to manage dependencies.

When integrated into a CI/CD pipeline, Maven ensures that the project is built, tested, and packaged in a consistent manner. It automates the execution of unit tests and code quality checks, reducing the risk of introducing defects into the main codebase. Moreover, Maven plugins enable additional functionalities such as static code analysis, reporting, and artifact deployment, further enhancing the CI/CD process.

2.11 Integrating Maven with GitHub Actions

GitHub Actions is a powerful automation tool that allows developers to define CI/CD workflows directly within their repositories. By integrating Maven with GitHub Actions, teams can automatically build and test their Java projects whenever new code is pushed to the repository.

A typical GitHub Actions workflow for a Maven-based project includes the following steps:

- Checkout the code: Retrieve the latest version of the repository.

- Set up Java: Define the required Java version using GitHub Actions setup.
- Cache dependencies: Speed up builds by caching Maven dependencies.
- Build and Test: Use Maven commands to compile, package, and run unit tests.
- Deploy Artifacts (optional): Deploy built artifacts to a repository such as GitHub Packages or an external artifact repository.

Here is a sample GitHub Actions workflow configuration for a Maven project:

```yaml
name: Run tests

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main


jobs:
  build:

    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
    - uses: actions/checkout@v4
    - name: Set up JDK 21
      uses: actions/setup-java@v4
      with:
        java-version: '21'
        distribution: 'temurin'
        cache: maven

    - name: Build with Maven
      run: mvn -B test
```

This workflow ensures that every commit and pull request triggers an automated Maven build and test cycle, helping teams detect issues early in the development process. The use of dependency caching also optimizes build times, making the CI process more efficient. The option -B or –batch-mode tells Maven to run in non-interactive mode, meaning it will not prompt for user input.

2.12 Deploying a Maven-based Java Application in a Docker Container

After building and testing your Java application using Maven, the next step is often to package and deploy it. One increasingly popular approach is to use Docker to containerize the application. This allows you to run your application in a consistent environment across different machines, reducing compatibility issues.

Once you have verified that your application passes all tests with Maven (mvn test), you can proceed to create a Docker image. This image encapsulates your Java application along with its runtime environment. A typical Dockerfile for a Maven-based Java application might look like this:

```
FROM openjdk:21-jdk-slim
COPY target/cargo-0.0.1.jar /app/myapp.jar
WORKDIR /app
CMD ["java", "-jar", "myapp.jar"]
```

This Dockerfile specifies:

- Using an official OpenJDK 21 image as the base.
- Copying the built JAR file (produced by Maven) into the container.
- Setting the working directory to /app.
- Specifying the command to run the application.

You can build the Docker image with:

```
docker build -t myapp .
```

And run the container with:

```
docker run -d -p 8080:8080 myapp
```

Chapter 3

Generics

Learning goals
The junior-colleague

1. can explain what generics is.
2. can describe type erasure and the consequences of type erasure
3. can use generic classes, methods and interfaces
4. can create generic classes, method and interfaces

Java Generics provides the ability to write generic code that is independent of a data type. When a developer uses a generic class, interface, or method, he will specify which data type will acutally be used. The Java Collections framework is entirely written generically. The moment you use an ArrayList, you need to indicate the data type for the elements of your list.

3.1 Before generics

Java is a strongly typed programming language. During the compilation of a program, you will be pointed out if you use the wrong data type.

```java
Object aReference = new Movie("Brother Bear");
Integer luckyNumber = aReference;
```

The second line of code results in a compilation error.

When Java was introduced, this datatype check was not implemented in the Java Collections framework. The following code compiles without errors. As soon as you execute the program, an exception will occur.

```java
import java.util.ArrayList;
import java.util.Iterator;

public class BeforeGenerics {

    public static void main(String[] args) {
        ArrayList objecten = new ArrayList();
        objecten.add(1);
        objecten.add(5.4);
        objecten.add(new Movie("Inception"));

        Iterator iterator = objecten.iterator();
        double total = 0;
        while (iterator.hasNext()) {
            total += (Double) iterator.next();
        }
        System.out.println(total);
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
cannot be cast to class java.lang.Double (java.lang.Integer and java.lang.Double
are in module java.base of loader 'bootstrap')
at be.pxl.ja.BeforeGenerics.main(BeforeGenerics.java:24)
```

Type-safety was introduced in the Java Collections framework since Java 5. When you open the java documentation for the class java.util.ArrayList, you will see the generic parameter $< E >$. E is the type-parameter and makes it possible to provide a datatype for the elements of the ArrayList.

```
/**
 * Resizable-array implementation of the {@code List} interface. Implements
 * all optional list operations, and permits all elements, including
 * {@code null}.
 * ...
 * @param <E> the type of elements in this list
 *
 * @author Josh Bloch
 * @author Neal Gafter
 * @see Collection
 * @see List
 * @see LinkedList
 * @see Vector
 * @since 1.2
 */
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    transient Object[] elementData;

    /**
     * Returns the element at the specified position in this list.
     *
     * @param index index of the element to return
     * @return the element at the specified position in this list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E get(int index) {
        Objects.checkIndex(index, size);
        return elementData(index);
    }

    E elementData(int index) {
        return (E) elementData[index];
    }

    ...
}
```

```
import java.util.ArrayList;
import java.util.Iterator;

public class SinceGenerics {

    public static void main(String[] args) {

        ArrayList<Double> objecten = new ArrayList<>();
        objecten.add((double) 1);
        objecten.add(5.4);
        //objecten.add(new Movie("Inception"));

        Iterator<Double> iterator = objecten.iterator();
        double total = 0;
        while (iterator.hasNext()) {
            total += iterator.next();
        }
        System.out.println(total);
    }
}
```

If you were to add a Movie-object to an ArrayList that accepts Double-objects, you get a compile error.

```
Error:(16, 30) java: incompatible types: be.pxl.ja.streamingservice.model.Movie
cannot be converted to java.lang.Double
```

When calling the constructor of a generic class, we use the diamond operator <>. The compiler can infer the datatype of the objects in the ArrayList (this is called type inference). We don't need to repeat the datatype when calling the constructor and use the diamond operator.

3.2 Writing generic classes

The generic class Duo contains a pair of objects of the generic datatype T. The class definition identifies this class as a generic class by using the type parameter T (<T>). In the Duo class, you can use T as the datatype for fields, parameters, and returntype.

```
public class Duo<T> {
    private T first;
    private T second;

    public Duo(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
}
```

At the moment you create objects of the class Duo, you will indicate which actual datatype the type parameter T should be replaced with. For a developer, it looks like the type parameter T is replaced by the chosen datatype.

```
public class DifferentDuos {

    public static void main(String[] args) {
        Duo<String> cocktail = new Duo<>("gin", "tonic");
        System.out.println(cocktail.getFirst());
        Duo<Actor> famousDuo = new Duo<>(new Actor("Ben","Stiller"), new
            ↪ Actor("Owen", "Wilson"));
        Duo<Integer> numbers = new Duo<>(5, 12);
    }

}
```

3.2.1 Naming conventions

To avoid confusion between actual classes and generic type parameters in Java, it is important to use correct naming conventions. We will always use a single uppercase letter to name a type parameter. According to the code conventions, you should never use a single uppercase letter as the name for a class; you must always use meaningful class names. Therefore, when you see a single uppercase letter in Java code, you may assume it is a generic parameter.

We adhere to the following guidelines:

- E – Element (used in the Java Collections Framework)
- K – Key (used in Map)
- N – Number
- T – Type
- V – Value (used in Map)
- S,U,V etc. – use when T already in use

3.3 Generic interfaces

3.3.1 Interface $Comparable < T >$

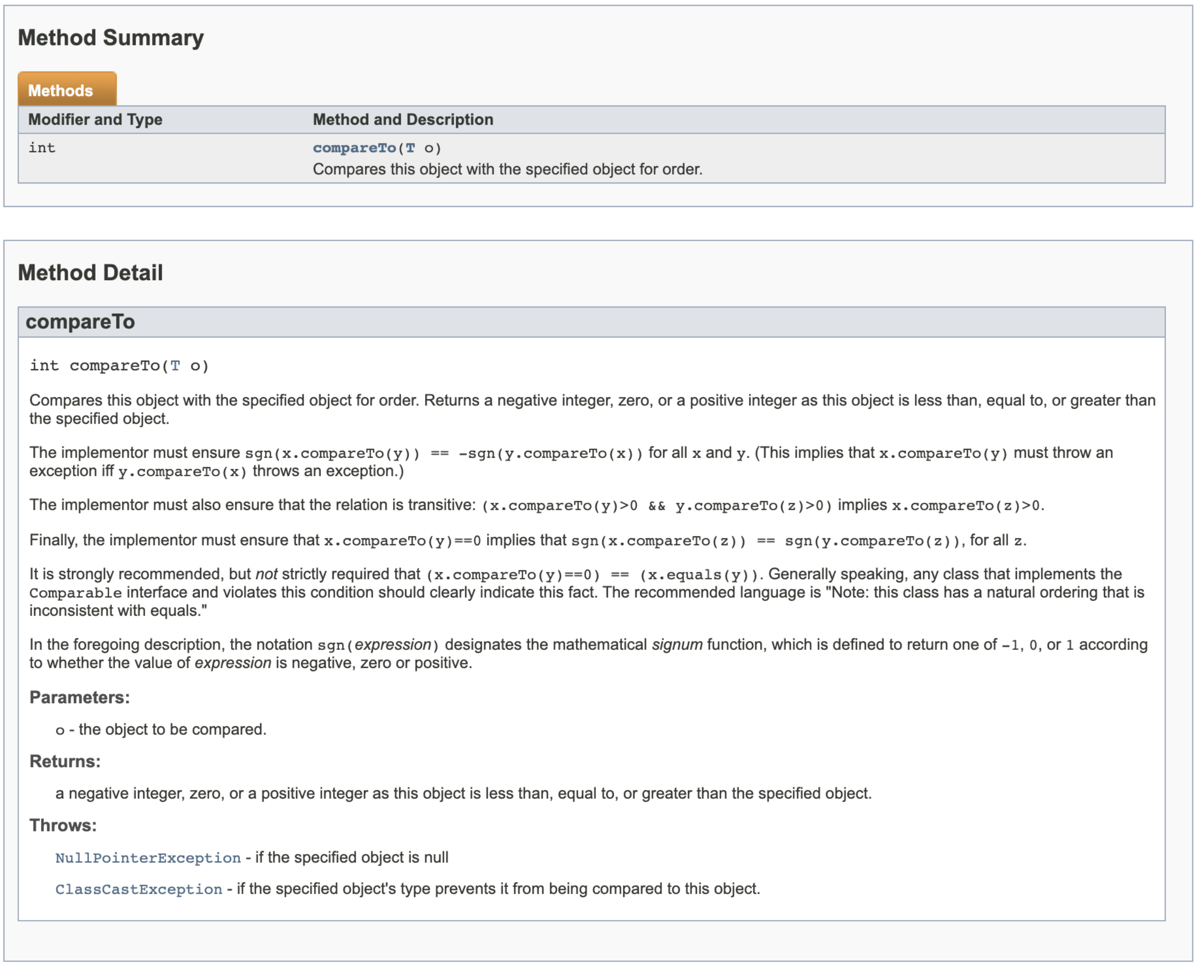
The generic interface $Comparable < T >$ makes it possible to compare objects.

Once a class implements this $Comparable < T >$ interface, you can use methods like Collections.sort(...) to sort a collection containing objects of this class. We call this ordering the natural order of the class.

When a class implements the $Comparable < T >$ interface, you fill in the type parameter T with the class itself. You must implement the method *int compareTo(T e)*. Through the implementation of this method, you can compare any object of the class with another object of the same class. The int compareTo(T e) method returns:

- 0 if both objects are equal.
- -1 or a negative number if this is less than the argument.
- 1 or a positive number if this is greater than the argument.

Always try to keep the natural order of a class consistent with the implementation of equals()-method. So, when e1.compareTo(e2) $==$ 0, e1.equals(e2) and e2.equals(e1) should also return true. Null does not belong to any class, so e1.compareTo(null) should always throw a NullPointerException.


Method Summary

Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Figure 3.1: Documentation for interface $Comparable < T >$

The class Person has a natural ordering based on the name of a Person-object. Objects of the class Person can be sorted alphabetically by name. When Person-objects share the same name, we sort by date of birth (younger to older).

```java
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.Objects;

public class Person implements Comparable<Person> {
    private String name;
    private LocalDate dateOfBirth;

    public Person(String name, LocalDate dateOfBirth) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }

    public int getAge() {
        if (dateOfBirth == null) {
            return 0;
        }
        return (int) ChronoUnit.YEARS.between(dateOfBirth,
            ↪ LocalDateTime.now());
    }

    @Override
    public int compareTo(Person other) {
        int nameCompare = this.name.compareTo(other.name);
        if (nameCompare == 0) {
            return other.dateOfBirth.compareTo(this.dateOfBirth);
        }
        return nameCompare;
    }

    public String getName() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        Person person = (Person) o;

        if (!Objects.equals(name, person.name)) {
            return false;
        }
        return Objects.equals(dateOfBirth, person.dateOfBirth);
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (dateOfBirth != null ?
            ↪ dateOfBirth.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age='" + getAge() + '\'' +
                '}';
    }
}
```

```
package be.pxl.demo;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SortingPersons {

    public static void main(String[] args) {
        List<Person> persons = new ArrayList<>();
        Person person1 = new Person("Erik", LocalDate.of(1998, 5, 2));
        Person person2 = new Person("Sam", LocalDate.of(2000, 5, 2));
        Person person3 = new Person("Ann", LocalDate.of(2005, 3, 1));
        Person person4 = new Person("Ann", LocalDate.of(2003, 4, 2));
        Person person5 = new Person("Ann", LocalDate.of(2003, 4, 2));

        persons.add(person1);
        persons.add(person2);
        persons.add(person3);
        persons.add(person4);
        persons.add(person5);

        System.out.println(person3.compareTo(person1));
        System.out.println(person4.compareTo(person3));
        System.out.println(person4.compareTo(person5));

        Collections.sort(persons);

        System.out.println(persons);
    }
}
```

```
-4
2
0
[Person{name='Ann', age='18'}, Person{name='Ann', age='20'}, Person{name='Ann', age='20'}
```

3.3.2 Writing a generic interface

In addition to generic classes, you can also develop generic interfaces and methods. Here is an example of a generic interface with two generic parameters T and U.

```java
public interface Service<T,U> {

    T execute(U arg);
}
```

The interface can have different parameter type and returntype, but this is not mandatory. You can, however, choose the same datatype for generic type T and U.

Here are two classes implementing the Service interface.

```java
public class CountService implements Service<Integer, String> {

    @Override
    public Integer execute(String arg) {
        return arg.length();
    }
}
```

```java
public class PersonConverterService implements Service<String, Person> {
    private static final DateTimeFormatter FORMATTER =
        ↪ DateTimeFormatter.ofPattern("dd/MM/yyyy");
    @Override
    public Person execute(String arg) {
        // Assuming the input format is "Name, Age"
        String[] parts = arg.split(",");
        if (parts.length != 2) {
            throw new IllegalArgumentException("Invalid input format: " +
                ↪ arg);
        }
        try {
            LocalDate dateOfBirth = LocalDate.parse(parts[1].trim(),
                ↪ FORMATTER);
            return new Person(parts[0].trim(), dateOfBirth);
        } catch (DateTimeParseException e) {
            throw new IllegalArgumentException("Invalid date format: " +
                ↪ arg);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Service<String, Person> personConverter = new
            ↪ PersonConverterService();
        Person person = personConverter.execute("John Doe, 12/03/2001");

        if (person != null) {
            System.out.println("Converted person: " + person);
        } else {
            System.out.println("Conversion failed.");
        }
    }
}
```

This program gives the following output:

```
Converted person: Person{name='John Doe', age='22'}
```

3.4 Generic methods

If you don't need a fully parameterized class, it's also possible to create generic methods. Both regular and static methods can contain one or more generic types. Even a constructor can use generic parameters.

Here's an example. The following static method *occursExactTimes* returns true if the specified item occurs exactly the given number of times in the List. If not, it returns false. Before the returntype of the method, you have to identify all the generic types that are used in the method.

```
import java.util.List;

public class OccurenceUtil {

    public static <T> boolean occursExactTimes(List<T> items, T item, int
        ↪ times) {
        int count = 0;
        for (T anItem : items) {
            if (anItem.equals(item)) {
                count++;
            }
        }
        return count == times;
    }

}
```

The method *occursExactTimes* can be used with different datatypes.

```
import java.util.Arrays;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(7, 15, 23, 12, 8, 7, 23, 13,
            ↪ 32, 7);
        System.out.println(OccurenceUtil.occursExactTimes(numbers, 7, 3));
        System.out.println(OccurenceUtil.occursExactTimes(numbers, 23, 5));
        List<String> animals = Arrays.asList("zebra", "elephant",
            ↪ "kangaroo", "cow", "kangaroo");
        System.out.println(OccurenceUtil.occursExactTimes(animals,
            ↪ "kangaroo", 2));
    }
}
```

Executing this program will give you the following output:

```
true
false
true
```

3.5 Bounded generics

Bounded means here ŕestricted; and we can restrict the datatypes that a method accepts. Given the following class Card.

```
public enum Suit {
    HEARTS,
    DIAMONDS,
    CLUBS,
    SPADES
}

public enum Rank {
    ACE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    TEN,
    JACK,
    QUEEN,
    KING
}

public class Card implements Comparable<Card> {
    private final Suit suit;
    private final Rank rank;
```

```java

    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public Suit getSuit() {
        return suit;
    }

    public Rank getRank() {
        return rank;
    }

    @Override
    public String toString() {
        return rank + " " + suit;
    }

    @Override
    public int compareTo(Card other) {
        return this.rank.compareTo(other.rank);
    }
}
```

We have a utility class with multiple methods to find the highest value.

```java
public class MyUtil {
    static int highest(int a, int b) {
        return Math.max(a, b);
    }
    static double highest(double a, double b) {
        return Math.max(a, b);
    }
    static Card highest(Card a, Card b) {
        return a.compareTo(b) > 0 ? a : b;
    }
}

public class Main {

    public static void main(String[] args) {
        System.out.println(MyUtil.highest(5, 9));
        System.out.println(MyUtil.highest(12.6, -4.6));
        System.out.println(MyUtil.highest(new Card(Suit.HEARTS, Rank.FIVE),
           ↪ new Card(Suit.HEARTS, Rank.SEVEN)));
    }
}
```

This program will produce the following output:

```
9
12.6
SEVEN HEARTS
```

Is it possible to write one generic method to decide on the highest value?

Let's rewrite our utility class. Integer, Double, and Card all implement the Comparable interface.

```java
public class MyUtil2 {

    static Integer highest(Integer a, Integer b) {
        return a.compareTo(b) > 0 ? a : b;
    }

    static Double highest(Double a, Double b) {
        return a.compareTo(b) > 0 ? a : b;
    }

    static Card highest(Card a, Card b) {
        return a.compareTo(b) > 0 ? a : b;
    }
}
```

This way we can write a generic method. The generic method allows for comparisons of any type that implements the Comparable interface.

```java
public class MyUtil2 {

    static <T extends Comparable<T>> T highest(T a, T b) {
        return a.compareTo(b) > 0 ? a : b;
    }
}
```

A ́boundŕ restriction is a constraint we impose on the generic type parameter. In a bound, you can include at most 1 class; there is no restriction on the number of interfaces. However, the class must always come first in the listing: *<T extends MyClass & MyFirstInterface & MySecondInterface & MyThirdInterface>*.

3.6 Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type.

3.6.1 Unbounded wildcards

```java
import java.util.Arrays;
import java.util.List;

public class Demo2 {

    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(1, 2, 3);
        List<String> list2 = Arrays.asList("one", "two", "three");
        MyUtil.printList(list1);
        MyUtil.printList(list2);
    }
}
```

How can you write a generic method to display the elements of a list. If you think the following method will do the job, feel free to try:

```java
import java.util.List;

public class MyUtil {

    public static void printList(List<Object> list) {
        for (Object element: list) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

However, this wil give a compilation error.

```
incompatible types: java.util.List<java.lang.Integer> cannot be converted to java.util.Lis

Required type: List<Object>
Provided: List<Integer>
```

Using an unbounded wildcard will do the job.

```java
import java.util.List;

public class MyUtil {

    public static void printList(List<?> list) {
        for (Object element: list) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Occasionally, a generic type parameter can be replaced with a wildcard. As shown in the example below, secondFunction will give a compile error. Essentially, as a rule of thumb, you may read data with an unknown data type (thus a wildcard), but you may not write the data.

```
public class Experiment {
    public static <E> void firstFunction(List<E> list) {
        list.add(list.get(0));
    }

    public static void secondFunction(List<?> list) {
        list.add(list.get(0)); // !!!!!!!!!!!!!! won't compile !!!!!!!!!
    }
}
```

3.6.2 Upper bound wildcards

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

The upper bounded wildcard matches Foo and any subtype of Foo.

3.6.3 Lower bound wildcards

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can hold Integer values.

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

3.7 Type Erasure and consequences

Generics exist only **at compile time**. During the compilation of Java code, various additional checks are performed on generic classes to ensure the correct use of data types. Then, when generating the bytecode, all information about the generic data type is simply removed. So, List<T> is, after some checks and the addition of extra code by the compiler, replaced by List<Object> and List<T extends Person> becomes List<Person>. This is what we call type erasure. In Java, there will always be only one compiled class (.class file) generated for a generic class.

```
List<Integer> list1 = new ArrayList<>();
List<Float> list2 = new ArrayList<>();
if (list1.getClass() == list2.getClass()) {
    System.out.println("Lists have same class.");
}
```

Due to type erasure, it is not possible to create generic static variables in a class. It is also not possible to call a constructor of a generic data type. Furthermore, it is not possible to use the generic data type with the instanceof operator.

```
class Box<T> {
   //compiler error
   private static T value;
   private T t;

   public Box() {
      // compiler error
      t = new T();
   }

   public void set(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public boolean test() {
       // compile error
      return t instanceof T;
   }
}
```

3.8 Exercise

Exercise 3.1. Create an abstract class Player with a member variable name. Provide a constructor with parameter name
Create 3 subclasses for this abstract class Player: BaseballPlayer, VolleyballPlayer, and SoccerPlayer.
Now create a class Team with the following properties:

- name (String)
- played (number of games played)
- won (number of games won)
- lost (number of games lost)
- tied (number of games drawn)
- members (collection of players)

Provide getters. In the constructor, you pass the name for the team.

Provide a method *addPlayer* to add a player to the team and a method *numberOfPlayers* to ask for the number of players in the team.
Can you add players of a different type (e.g., BaseballPlayer and SoccerPlayer) to one team? Test it! Make sure this is no longer possible.
Provide the method *matchResult(Team opponent, int ourScore, int theirScore)*. This method ensures that for the team for which the method is called and the opponent, the number of played, won, lost, and drawn matches is increased depending on the values for ourScore and theirScore.
Can you call this method (matchResult) for a team of volleyball players against a team of baseball players? Solve this if necessary, so that this is no longer possible.
Finally, add a method *ranking()*. This returns an integer where the team gets 3 points for each win and 1 point for a draw. Now make sure you can create a collection of Team objects and sort them based on the ranking.

Chapter 4

Java I/O

File handling is a fundamental aspect of programming, allowing developers to read from, write to, and manipulate files. Java provides two primary APIs for file handling—the traditional java.io package and the more recent java.nio package. Understanding these APIs enables Java developers to efficiently manage data stored in files.

4.1 Basics of File Handling

A file system manages access to both the content of files and the metadata about those files. Files are stored in directories. The files and directories are accessed by specifying a path. Paths are either absolute or relative:

- An absolute path is a path relative to the file system's root directory. It's expressed as the root directory symbol followed by a delimited hierarchy of directory names that ends in the target directory or file name.
- A relative path is a path relative to some other directory. It's expressed similarly to an absolute path but without the initial root directory symbol. In contrast, it's often prefixed with one or more delimited ".." character sequences, where each sequence refers to a parent directory.

Javadoc 1 (java.io.File)**.** https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/File.html

```java
package be.pxl.ja;

public class Demo01 {

    public static final String SEPARATOR =
        ↪ System.getProperty("file.separator");

    public static void main(String[] args) {
        System.out.println("Current operating system: " +
            ↪ System.getProperty("os.name"));
        System.out.println("File separator: " + SEPARATOR);
        System.out.println("User's home directory: " +
           ↪ System.getProperty("user.home"));
        System.out.println("Current working directory: " +
           ↪ System.getProperty("user.dir"));
    }
}
```

On some systems, Java can compensate for differences such as the direction of the file separator slashes in a pathname. For example, in the current implementation on Windows platforms, Java accepts paths with either forward slashes or backslashes. Thus, using forward slashes will make your application system independent.

```
File file = new File("java/introduction.txt");
```

```
import java.io.File;

public class PartitionSpace
{
   public static void main(String[] args)
   {
      File[] roots = File.listRoots();
      for (File root: roots)
      {
         System.out.println("Partition: " + root);
         System.out.println("Free space on this partition = " +
                            root.getFreeSpace());
         System.out.println("Usable space on this partition = " +
                            root.getUsableSpace());
         System.out.println("Total space on this partition = " +
                            root.getTotalSpace());
         System.out.println("***");
      }
   }
}
```

Like the legacy File class, Path also creates an object that may be used to locate a file in a file system.

Javadoc 2 (java.nio.file.Path). https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Path.html

The utility class *java.nio.files.Files* has a wide range of static method for file operations.

Javadoc 3 (java.nio.file.Files). https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Files.html

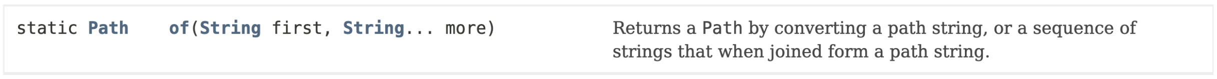
```
// java.io API
boolean fileExists = file.exists();
boolean fileIsFile = file.isFile();
boolean fileIsDir = file.isDirectory();
boolean fileReadable = file.canRead();
boolean fileWritable = file.canWrite();
boolean fileExecutable = file.canExecute();
boolean fileHidden = file.isHidden();

// java.nio API
boolean pathExists = Files.exists(path);
boolean pathIsFile = Files.isRegularFile(path);
boolean pathIsDir = Files.isDirectory(path);
boolean pathReadable = Files.isReadable(path);
boolean pathWritable = Files.isWritable(path);
boolean pathExecutable = Files.isExecutable(path);
boolean pathHidden = Files.isHidden(path);
```

4.2 Accessing directories

The following program is used to display the names of all the files in a given folder. It makes use of the File class.

```
 import java.io.File;

public class DisplayFiles {

    public static void main(String[] args) {
        // Check if a folder path has been provided as an argument
        if (args.length < 1) {
            System.out.println("Please provide a folder path as an
               ↪ argument.");
            return;
        }

        // Get the folder path from the command line arguments
        String folderPath = args[0];
        File folder = new File(folderPath);

        // Check if the folder exists and is indeed a directory
        if (!folder.exists() || !folder.isDirectory()) {
            System.out.println("The provided path does not exist or is not a
               ↪ directory.");
            return;
        }

        // List all files in the directory
        File[] listOfFiles = folder.listFiles();

        if (listOfFiles != null) {
            for (File file : listOfFiles) {
                if (file.isFile()) {
                    System.out.println(file.getName());
                }
            }
        } else {
            System.out.println("There was a problem reading the directory.");
        }
    }
}
```

A program with the same functionality can be written using the the java.nio classes. Pay special attention to the **try-with-resources statement.** The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.



static **Path**	**of**(**String** first, **String**... more)	Returns a Path by converting a path string, or a sequence of strings that when joined form a path string.

Figure 4.1: Static method in interface Path

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;
import java.io.IOException;
import java.util.stream.Stream;

public class DisplayFilesNIO {

    public static void main(String[] args) {
        // Check if a folder path has been provided as an argument
        if (args.length < 1) {
            System.out.println("Please provide a folder path as an
               ↪ argument.");
            return;
        }

        // Get the folder path from the command line arguments
        String folderPath = args[0];
        Path path = Paths.get(folderPath);

        // Check if the path exists and is a directory
        if (!Files.exists(path) || !Files.isDirectory(path)) {
            System.out.println("The provided path does not exist or is not a
                ↪ directory.");
            return;
        }

        // Use try-with-resources to ensure that the stream is closed
        try (Stream<Path> paths = Files.list(path)) {
            paths.forEach(filePath -> {
                if (Files.isRegularFile(filePath)) {
                    System.out.println(filePath.getFileName());
                }
            });
        } catch (IOException e) {
            System.out.println("An error occurred while reading the
                ↪ directory: " + e.getMessage());
        }
    }
}
```

Creating directories and files

The static method createDirectory() in the class Files can be used to create a new directory.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class Demo03CreatingDirectoriesAndFiles {

    public static void main(String[] args) {
        Path path = Path.of(System.getProperty("user.home"), "JavaAdvIO",
            ↪ "Opdracht3", "bijlage.txt");
        if (Files.notExists(path.getParent())) {
            try {
                Files.createDirectory(path.getParent());
            } catch (IOException e) {
                System.out.println("An error occured while creating
                    ↪ directory " + path.getParent());
            }
        }
        if (Files.notExists(path)) {
            try {
                Files.createFile(path);
            } catch (IOException e) {
                System.out.println("An error occured while creating file " +
                    ↪ path);
            }
        }
    }
}
```

Reading small files

For reading text files with a limited number of lines, you can use the static method readAllLines in the class Files.

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
import java.util.Random;

public class Demo04ReadingFiles {
    private static final Random RANDOM = new Random();

    public static void main(String[] args) {
        Path path = Paths.get("resources/small_file_with_text.txt");

        try {
            List<String> text = Files.readAllLines(path);
            System.out.println(text.get(RANDOM.nextInt(text.size())));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Copying files

Creating a copy of a file is easy using the static method copy in the class Files.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Demo05CopyFiles {

    public static void main(String[] args) {
        Path original = Paths.get("resources/small_file_with_text.txt");
        Path copy = Paths.get("resources", "copy_" +
            ↪ System.currentTimeMillis() + ".txt");
        System.out.println(Files.exists(copy));
        try {
            Files.copy(original, copy);
            System.out.println(Files.exists(copy));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.3 Byte Streams

A stream is an ordered sequence of bytes of an arbitrary length. Bytes flow over an output stream from an application to a destination and flow over an input stream from a source to an application. The java.io package provides several output stream and input stream classes that are descendants of its abstract *OutputStream* and *InputStream* classes.

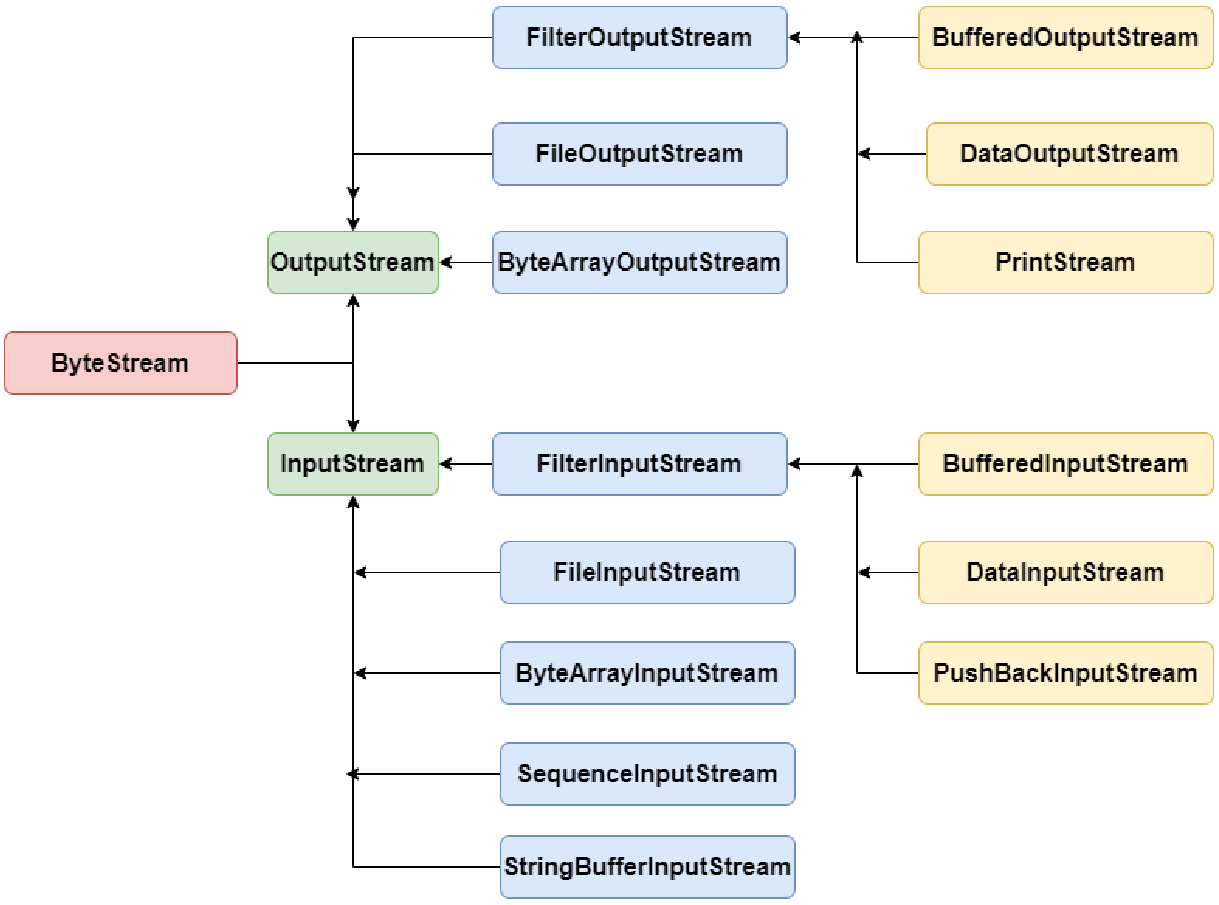


Figure 4.2: Byte Streams

```
import java.io.FileOutputStream;
import java.io.IOException;

public class HexDataWriter {
    public static void main(String[] args) {
        // Example hexadecimal data: Image header, random bytes, etc.
        byte[] hexData = new byte[]{(byte)0xBA, (byte)0xAD, (byte)0xF0,
           ↪ (byte)0x0D, (byte)0xDE, (byte)0xAD, (byte)0xBE, (byte)0xEF};

        try (FileOutputStream outputStream = new
           ↪ FileOutputStream("binarydata.bin")) {
            outputStream.write(hexData);
            System.out.println("Hexadecimal data has been written to the
               ↪ file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

```
import java.io.FileInputStream;
import java.io.IOException;

public class HexDataReader {
    public static void main(String[] args) {
        try (FileInputStream inputStream = new
           ↪ FileInputStream("binarydata.bin")) {
            int byteRead;
            while ((byteRead = inputStream.read()) != -1) {
                // Convert the byte to a hexadecimal string
                System.out.print(String.format("%02X ", byteRead));
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

4.4 Character Streams

A character set or charset is a set of characters and the encoding defines how these characters are stored in memory, using one or more bytes. If you need to stream characters, you should take advantage of Java's writer and reader classes, which were designed to support character I/O (they work with char instead of byte). Furthermore, the writer and reader classes take character encodings into account.

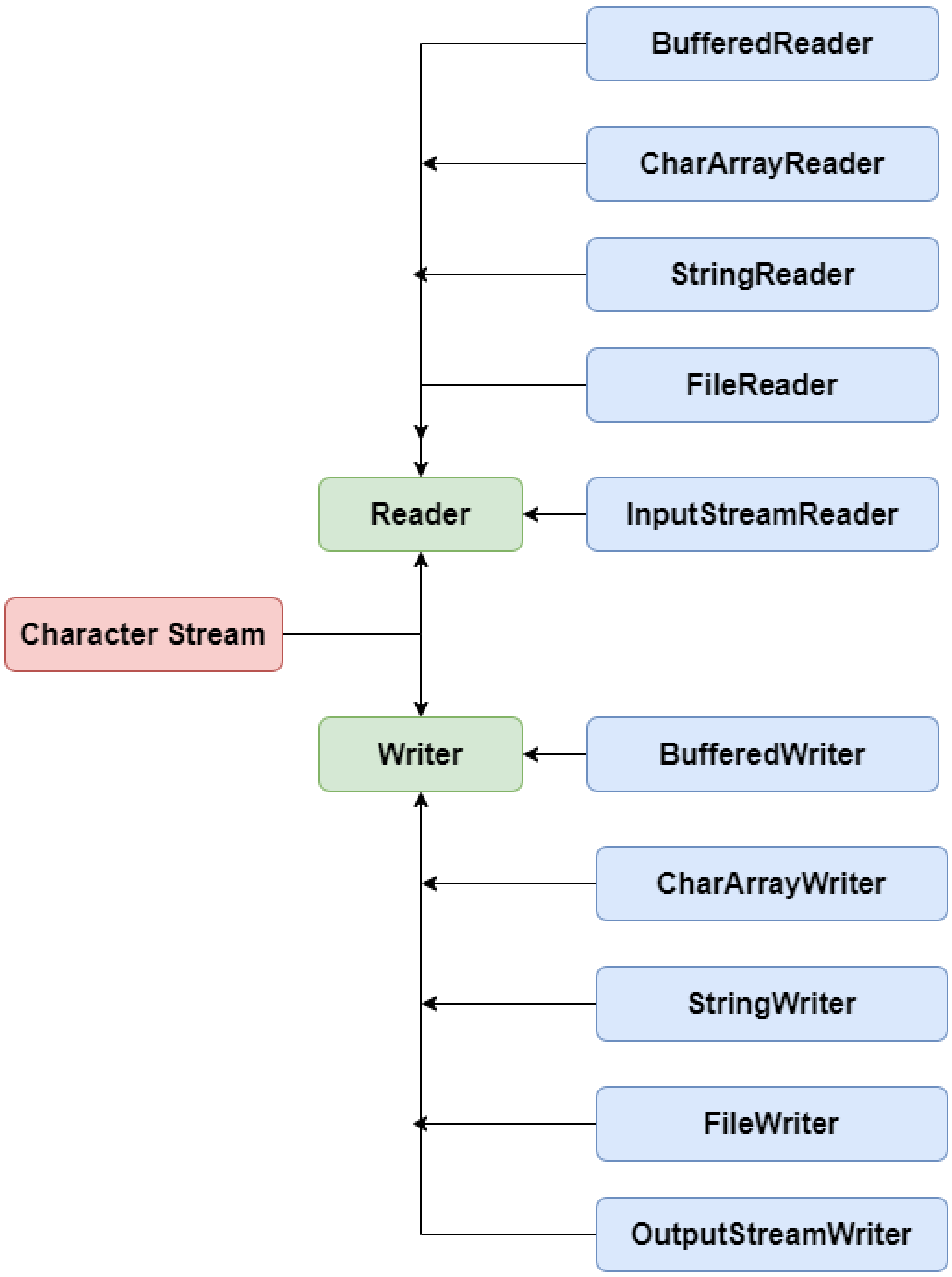


Figure 4.3: Byte Streams

4.4.1 BufferedReader

The BufferedReader class offers a simple and efficient way to read larger text files. It groups (buffers) characters from a file so that you can read the file line by line. The newBufferedReader() method from the Files class simplifies the creation of the BufferedReader. The end of the text file is reached when readLine() returns null.

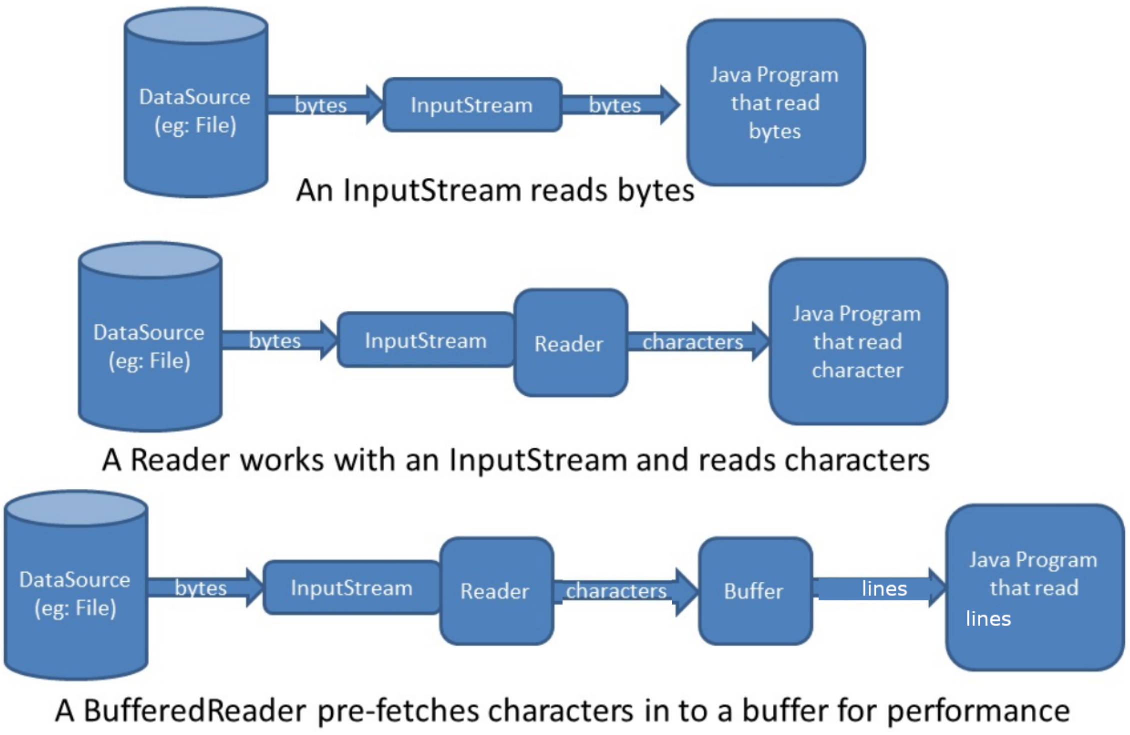

DataSource (eg: File)
bytes
InputStream
bytes
Java Program that read bytes
An InputStream reads bytes
DataSource (eg: File)
bytes
InputStream
Reader
characters
Java Program that read character
A Reader works with an InputStream and reads characters
DataSource (eg: File)
bytes
InputStream
Reader
characters
Buffer
lines
Java Program that read lines
A BufferedReader pre-fetches characters in to a buffer for performance

Figure 4.4: InputStream, InputStreamReader en BufferedReader

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class Demo06BufferedReader {

    public static void main(String[] args) {

        try (BufferedReader reader =
            ↪ Files.newBufferedReader(Paths.get("resources/big_file_with_text.txt")))
            ↪ {
            String line = null;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.4.2 BufferedWriter

Instead of writing to a file character by character, BufferedWriter writes larger pieces of text to the file at once.

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.io.BufferedWriter;
import java.io.IOException;

public class NIOBufferedWriterExample {
    public static void main(String[] args) {
        // Path to the file
        String filePath = "example_nio.txt";

        // Using try-with-resources to ensure the BufferedWriter is closed
            ↪ after use
        try (BufferedWriter writer =
            ↪ Files.newBufferedWriter(Paths.get(filePath),
                StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
            // Generating and writing 10 lines of text to the file
            for (int i = 1; i <= 10; i++) {
                String text = "Line " + i + ": This is an example of using
                    ↪ Files.createBufferedWriter in Java.\n";
                writer.write(text);
            }
            System.out.println("Text has been written to " + filePath);
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

The options StandardOpenOption.CREATE and StandardOpenOption.WRITE specify that if the file does not exist, it should be created. If the file already exists, it will be opened for writing.

4.4.3 Character sets

```
import java.nio.charset.Charset;
```

```java
public class DefaultCharset {

    public static void main(String[] args) {

        System.out.println(Charset.defaultCharset().displayName());
    }

}
```

A charset specifies how characters are encoded into bytes and decoded back into characters.

```java
Files.newBufferedWriter(Paths.get("resources/myfile.txt"),
    ↪ Charset.forName("UTF-8"));
```

In UTF-8 1 to 4 bytes are used to represent the characters.

Exercise 4.1. A text message has been encoded by replacing each character of the message with an integer. Each integer is an index into a key-phrase that contains all the lower case letters of the alphabet as well as the space character. The key-phrase may contain the same character in several locations. The encoded text is series of integers, like this:

```
35 10 10 33 9 24 3 17 41 8 3 20 51 16 38 44 47 32 33 10 19 38 35 28 49
```

To decode the message, use each integer as an index in the key-phrase and output the corresponding character. For example, say that the key-phrase is this (the index of each character has been written above it):

```
          1111111111222222222233333333334444444444455
01234567890123456789012345678901234567890123456789 01
six perfect quality black jewels amazed the governor
```

using each integer from the encoded text as an index into the phrase results in the decoded message:

```
attack the bridge at dawn
```

Write a program that decodes a secret message contained in a text file. The first line of the text file contains the key-phrase. Then the file contains a sequence of integers, each of which indexes the key-phrase. Find the character corresponding to each integer and output the secret message. Note if a character character such as ‘e’ occurs several places in the key-phrase it may be encoded as different integers in different parts of the secret message.
(The recipient of the secret message gets only the file of integers and must put the key-phrase at the top of the file.) For example, here is the contents of a secret message file ready for the program:

```
six perfect quality black jewels amazed the governor
35 10 10 33 9 24 3 17 41 8 3 20 51 16 38 44 47 32 33 10 19 38 35 28 49
```

Here is a sample run of the program:

```
C:\> java Decode < secretFile.txt

attack the bridge at dawn
```

Here is another secret message file, with key-phrase inserted, that you can use to

```
test your program:
six perfect quality black jewels amazed the governor
31 16 2 3 4 42 48 7 27 9 10 43 12 13 35 15 1 40 18 3
20 15 33 23 24 32 26 29 28 27 21 31 25 14 34 14 36
42 38 19 40 41 27 3 44 50 46 42 48 49 50 6
```

4.5 Program properties

It's a good idea to make Java programs easily configurable. To achieve this, parameters consisting of a key and a value are usually sufficient. In Java, you use the java.util.Properties class to read and write configuration files. You use the load method to read a configuration file and store to write the properties. With the getProperty method, you can request the corresponding value using a key.

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Properties;
import java.util.Scanner;

public class Demo08ProgramWithProperties {
    private static final String CONFIG_FILE = "resources/config.properties";

    public static void main(String[] args) {
        try(InputStream inputStream =
            ↪ Files.newInputStream(Path.of(CONFIG_FILE))) {
            Properties properties = new Properties();
            properties.load(inputStream);
            System.out.println("Welcome " +
                ↪ properties.getProperty("demo.name") + "!");
            System.out.println("You're mails will be sent to: " +
                ↪ properties.getProperty("demo.email"));
        } catch (IOException e) {
            createProperties();
        }
    }

    private static void createProperties() {
        Scanner scanner = new Scanner(System.in);
        System.out.println("You're using this program for the first time.");
        System.out.println("What's your name:");
        String name = scanner.nextLine();
        System.out.println("What's your company:");
        String company = scanner.nextLine();
        System.out.println("What's your email:");
        String email = scanner.nextLine();
        Properties properties = new Properties();
        properties.setProperty("demo.name", name);
        properties.setProperty("demo.company", company);
        properties.setProperty("demo.email", email);
        try(OutputStream outputStream =
           ↪ Files.newOutputStream(Path.of(CONFIG_FILE))) {
           properties.store(outputStream, "Demo program configuration");
        }
        catch (IOException e) {
           System.out.println("We were not able to save your
              ↪ configuration.");
        }
    }
}
```

The first time the program is executed, the config.properties file is not yet present. The user is then asked to enter the information.

```
You're using this program for the first time.
What's your name:
Nele
What's your company:
PXL
What's your email:
nele.custers@pxl.be
```

You will see that after running the program, a file *resources/config.properties* has been created. This file contains the following information.

```
#Demo program configuration
#Tue Nov 17 09:13:51 CET 2020
demo.name=Nele
demo.company=PXL
demo.email=nele.custers@pxl.be
```

When you restart the program, the .properties file is read, and you'll get the following output.

```
Welcome Nele!
You're mails will be sent to: nele.custers@pxl.be
```

4.6 Object serialization and deserialization

Object serialization in Java is the process of converting an object's state into a byte stream, thus enabling the object to be easily saved to a file or transmitted over a network. Deserialization, on the other hand, is the reverse process, where the byte stream is converted back into a replica of the original object, restoring its state. This mechanism allows for the persistent storage of objects and the exchange of objects between Java systems.

4.6.1 Object serialization

To serialize an object in Java, the class must implement the java.io.Serializable interface, which is a marker interface (it does not contain any method declarations) indicating that an object of this class can be serialized. Here is a simple example:

```java
import java.io.*;

public class User implements Serializable {
    private String name;
    private transient int age; // the transient keyword indicates that this
       ↪ field should not be serialized

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters and Setters
}

// Serialization process
try (ObjectOutputStream oos = new ObjectOutputStream(new
    ↪ FileOutputStream("user.ser"))) {
    User user = new User("John Doe", 30);
    oos.writeObject(user);
} catch (IOException e) {
    e.printStackTrace();
}
```

4.6.2 Object deserialization

Deserialization converts the byte stream back into an object, as shown in the following example:

```java
import java.io.*;

public class DeserializeExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new
            ↪ FileInputStream("user.ser"))) {
            User user = (User) ois.readObject();
            System.out.println("Name: " + user.getName() + ", Age: " +
                ↪ user.getAge());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In this example, the User class must implement the Serializable interface. Note the use of transient with the age variable; it indicates that this field should not be serialized. During deserialization, the age field of the deserialized User object will not be restored to its original value but will be set to the default value for its type (0 in this case).

4.7 Best practices

Here are some best practices for reading and writing to files using Java libraries:

- **Use try-with-resources**: When reading or writing to a file, it's important to properly handle the opening and closing of the file stream. Java 7 introduced the try-with-resources statement which automatically closes the file stream once the code block is executed.
- **Use buffered streams**: Reading or writing large files can be a performance bottleneck. Using buffered streams can help improve performance by reducing the number of I/O operations required.
- **Handle exceptions**: When reading or writing to a file, it's important to handle any exceptions that may occur. Common exceptions include FileNotFoundException and IOException.
- **Use appropriate encoding**: When reading or writing text files, it's important to use the appropriate encoding. The default encoding used by Java is usually UTF-8, but this may not be appropriate for all scenarios.
- **Use relative paths**: When working with files, it's best to use relative paths instead of absolute paths. This makes the code more portable and easier to maintain.
- **Check for file existence**: Before reading or writing to a file, it's important to check if the file exists. This can be done using the File.exists() method.
- **Use descriptive file names**: When creating new files, it's important to use descriptive file names that are easy to understand and maintain. This can help prevent naming conflicts and make it easier to find and manage files.

Exercise 4.2. **Exploring and Analyzing a Spring Boot Maven Project with Files.walk()**
Write a Java program that traverses a Spring Boot project directory using Files.walk(). Accept the project path from the user and validate that the directory exists.

```
java Explore C:\my-projects\secret-project
```

Extract project structure components:

- Locate **pom.xml** and count the dependencies.
- Identify application.properties.
- Find Java source files (*.java) and count their occurrences.
- Count the number of REST controller (@RestController)
- Detect hardcoded credentials in application.properties. Use regex to check for password=, secret=, or key=.

Example output:

```
Project Structure:
- Found pom.xml at: secret-project/pom.xml
- Found 5 dependencies
- Found 25 Java files.
- Found 5 controller classes.

Security Checks:
- Hardcoded password detected in application.properties: "password=admin123"
```

Chapter 5

JDBC

Learning goals
The junior-colleague

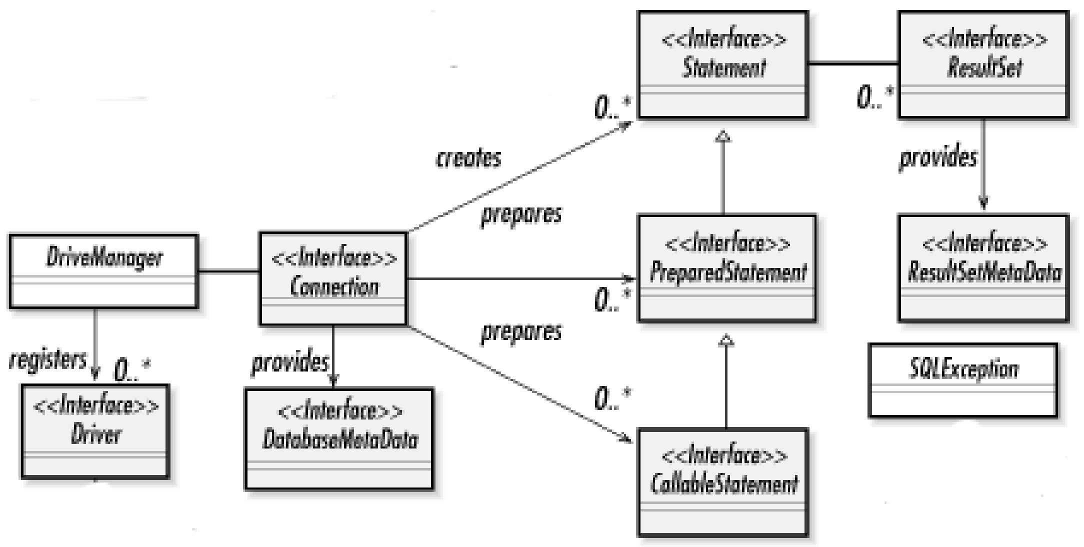
1. can describe what JDBC is.
2. can explain the benefits of JPA over JDBC.
3. can identify the base interfaces of JDBC.
4. can use JDBC to connect to a database.
5. can use JDBC to query a database.
6. can use JDBC to create a table.
7. can use JDBC to insert, update, and delete records in a table.
8. can use transactions in a JDBC application.
9. can describe what SQL injection is.
10. can use Prepared Statements to prevent SQL injection.
11. can explain the ACID-properties of a transaction.
12. can explain different isolation levels and the problems that can possibly occur.
13. can explain what connection pooling is.
14. can describe what HikariCP is.

5.1 What is JDBC?

The way every Java application connects to a database is through the JDBC API. JDBC or Java (Jakarta) Database Connectivity is Java's low-level API for making database connections and handling SQL queries and responses. It's a low-level API so you have to write quite some boilerplate code. JDBC is an adapter layer from Java to SQL. It gives Java developers a common interface for connecting to a database, issuing queries and commands, and managing responses.

5.2 The database

Use docker to supply the database in the development environment. In case anything goes wrong, you simply recreate your environment from scratch. There are some advantages of using docker on your development machine: there's less clutter on your machine and you can easily work on multiple projects with different databases and database versions.



5.3 JDBC Interfaces

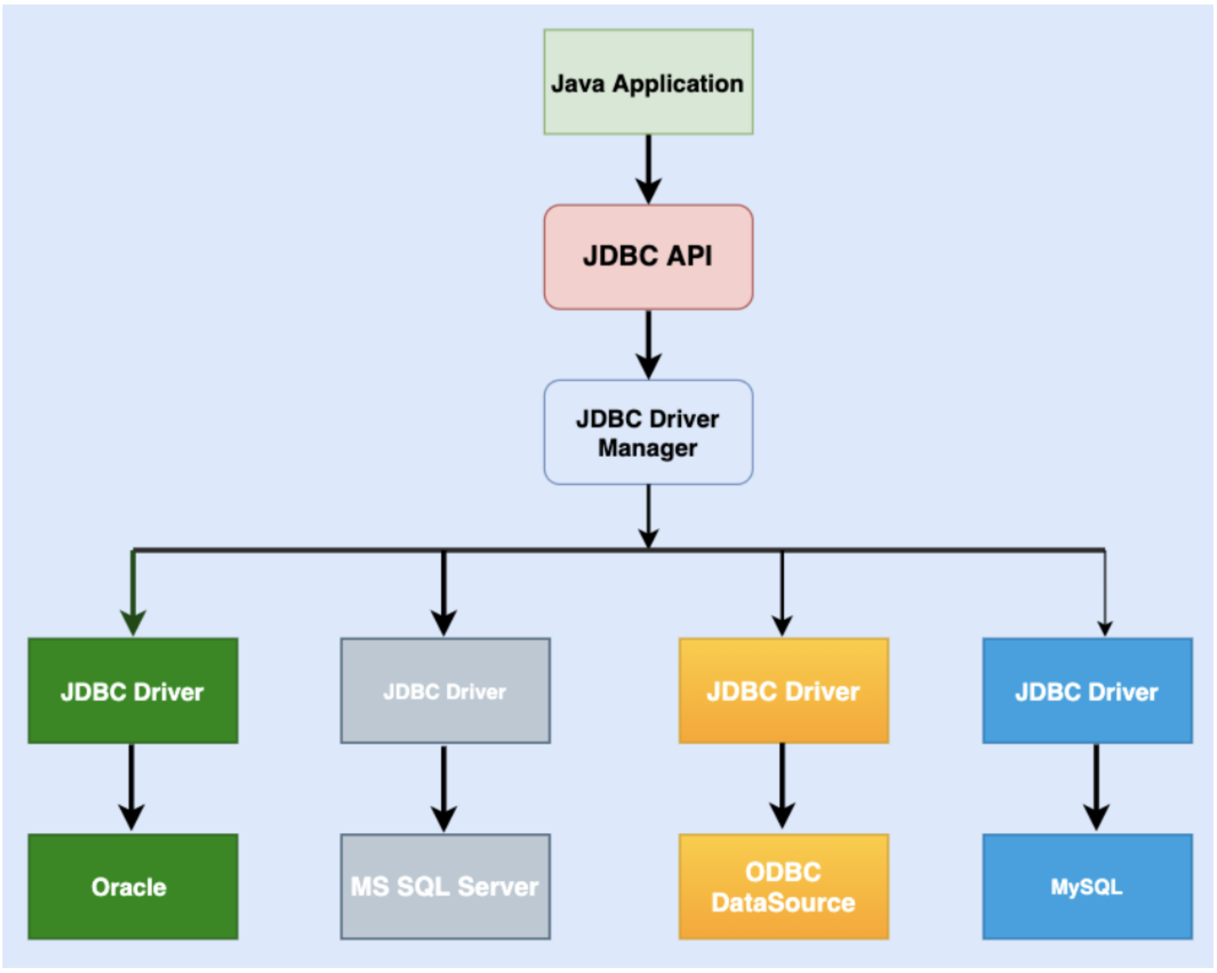
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;
```

Each of these imports provides access to a class that facilitates the standard Java database connection: Connection represents the connection to the database. DriverManager obtains the connection to the database. (Another option is DataSource, used for connection pooling.) ResultSet and Statement model the data result sets and SQL statements. SQLException handles SQL errors between the Java application and the database.

5.4 Database manipulation

5.4.1 Obtaining a database connection

```
try (Connection conn =
          DriverManager.getConnection("jdbc:mysql://localhost:3306/moviedb",
            ↪ "user", "password")) {
         LOGGER.info("Connnection established: " + conn.getCatalog());
         LOGGER.info("Connnection established: " +
           ↪ conn.getMetaData().getDriverName());
     } catch (SQLException e) {
         LOGGER.fatal("Something went wrong.", e);
     }
```



The try-with-resources block ensures automatic closing of the database connection, even if exceptions occur.

The call DriverManager.getConnection() attempts to establish a connection to the specified database. It leverages the provided connection URL and credentials to achieve this. The connection URL "jdbc:mysql://localhost:3306/moviedb" defines the connection details.

- jdbc:mysql - Identifies the database type, MySQL in this case.
- localhost - Specifies the hostname or IP address of the MySQL server
- 3306 - Denotes the default port number for MySQL connections
- moviedb - Represents the target database name within the MySQL server.
- Credentials: "user" and "password" represent the username and password for a valid MySQL account with access to the moviedb database.

If the connection is established successfully, the code proceeds to log informative messages:

- conn.getCatalog(): Retrieves and logs the name of the connected database catalog. In most cases, this corresponds to the database name specified in the connection

URL

- conn.getMetaData().getDriverName(): Obtains and logs information about the JDBC driver used to establish the connection. This can be helpful for debugging purposes.

The catch (SQLException e) block acts as a safety net. If any exceptions related to SQLException arise during the connection process, this block intercepts them.

5.4.2 Create a Table

```java
package be.pxl.jdbc;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class CreateTable {
    private static final Logger LOGGER =
        ↪ LogManager.getLogger(CreateTable.class);

    public static void main(String[] args) {
        try (Connection conn =
            ↪ DriverManager.getConnection("jdbc:mysql://localhost:3307/moviedb",
            ↪ "user", "password");
             Statement statement = conn.createStatement()) {
             statement.execute("CREATE TABLE employee (id INTEGER NOT NULL
                ↪ AUTO_INCREMENT, " +
                    "name TEXT, " +
                    "salary FLOAT, " +
                    "PRIMARY KEY (id))");
             LOGGER.info("Table 'employee' created.");
        } catch (SQLException e) {
             LOGGER.fatal("Something went wrong.", e);
        }
    }
}
```

The code showcases table creation using JDBC's Statement object and the CREATE TABLE SQL command.

5.4.3 SQL Injection

SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. It allows an attacker to insert arbitrary SQL code into a query that can be executed by the database, potentially compromising the security of the database and leaking information.

Given the movies database, let's consider a scenario where an application retrieves information about a movie based on user-provided input (e.g., movie name). The application might construct a SQL query by concatenating strings, including the user input. If the application fails to properly sanitize the user input, it can be exploited using SQL injection.

```java
package be.pxl.jdbc;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class SqlInjection {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(SqlInjection.class);

    public static void main(String[] args) {
        try (Connection conn =
            ↪ DriverManager.getConnection("jdbc:mysql://localhost:3307/moviedb",
            ↪ "user", "password")) {
            Statement statement = conn.createStatement();
            String movieName = "A CLOCKWORK ORANGE";
            String movieName2 = "A CLOCKWORK ORANGE' UNION SELECT *, '1',
                ↪ '1' FROM DIRECTOR where '1=1";
            String query = "SELECT * FROM MOVIES where MOV_TITLE = '" +
                ↪ movieName + "'";
            System.out.println(query);
            ResultSet resultSet = statement.executeQuery(query);
            final ResultSetMetaData meta = resultSet.getMetaData();
            final int columnCount = meta.getColumnCount();
            System.out.println("Column count: " + columnCount);
            while (resultSet.next()) {
                for (int column = 1; column <= columnCount; column++) {
                    Object value = resultSet.getObject(column);
                    System.out.print(value + " / ");
                }
                System.out.println();
            }
        } catch (SQLException e) {
            LOGGER.fatal("Something went wrong.", e);
        }
    }
}
```

If the application runs with user-provided movie name 'A clockwork orange' the following data is retrieved.

```
SELECT * FROM MOVIES where MOV_TITLE = 'A CLOCKWORK ORANGE'
Column count: 5
16 / A Clockwork Orange / 1971 / English / 2 /
```

SQL Injection Attack to Retrieve Director Information

An attacker could use the following input to not only retrieve movie information but also to steal the names and phone numbers of the directors: "A CLOCKWORK ORANGE' UNION SELECT *, '1', '1' FROM DIRECTOR where '1=1"

```
SELECT * FROM MOVIES where MOV_TITLE = 'A CLOCKWORK ORANGE' UNION SELECT *, '1', '1' FROM DIR
Column count: 5
16 / A Clockwork Orange / 1971 / English / 2 /
1 / Alfred Hitchcock / 555-0101 / 1 / 1 /
2 / Stanley Kubrick / 555-0102 / 1 / 1 /
3 / Christopher Nolan / 555-0103 / 1 / 1 /
4 / Paul Verhoeven / 555-0104 / 1 / 1 /
5 / George Sluizer / 555-0105 / 1 / 1 /
```

The malicious input ends the original query and adds a new query.

To prevent SQL injection, you should:

- Use prepared statements with parameterized queries. This ensures that user input is treated as data, not executable code.
- Employ ORM (Object Relational Mapping) libraries, which generally use parameterized queries.
- Validate and sanitize user input to ensure it conforms to expected formats.
- Apply least privilege access principles to your database operations. For instance, the application's database user should not have more privileges than it needs to perform its tasks.

5.5 CRUD

Here's an example demonstrating CRUD (Create, Read, Update, Delete) operations for tjhe "employee" table in a MySQL database using JDBC:

```
package be.pxl.jdbc;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Optional;

public class EmployeeCRUD {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(EmployeeCRUD.class);

    private static final String url = "jdbc:mysql://localhost:3307/moviedb";
    private static final String username = "user";
    private static final String password = "password";

    public static void createEmployee(Connection connection, String name,
        ↪ float salary) throws SQLException {
        String query = "INSERT INTO employee (name, salary) VALUES (?, ?)";
        PreparedStatement preparedStatement =
            ↪ connection.prepareStatement(query);
        preparedStatement.setString(1, name);
        preparedStatement.setFloat(2, salary);
        preparedStatement.executeUpdate();
        LOGGER.info("Employee '" + name + "' created.");
    }

    public static Optional<Employee> getEmployee(Connection connection, int
        ↪ id) throws SQLException {
        String query = "SELECT * FROM employee WHERE id = ?";
        PreparedStatement preparedStatement =
            ↪ connection.prepareStatement(query);
        preparedStatement.setInt(1, id);
        ResultSet resultSet = preparedStatement.executeQuery();

        if (resultSet.next()) {
            int retrievedId = resultSet.getInt("id");
            String retrievedName = resultSet.getString("name");
            double retrievedSalary = resultSet.getDouble("salary");
            Employee employee = new Employee(retrievedId, retrievedName);
            employee.setSalary(retrievedSalary);
            return Optional.of(employee);
        } else {
            LOGGER.info("Employee with ID " + id + " not found.");
            return Optional.empty();
        }
    }
```

```
public static void updateEmployee(Connection connection, int id, String
    ↪ name, float salary) throws SQLException {
    String query = "UPDATE employee SET name = ?, salary = ? WHERE id =
        ↪ ?";
    PreparedStatement preparedStatement =
        ↪ connection.prepareStatement(query);
    preparedStatement.setString(1, name);
    preparedStatement.setFloat(2, salary);
    preparedStatement.setInt(3, id);
    int rowsUpdated = preparedStatement.executeUpdate();
    if (rowsUpdated > 0) {
        LOGGER.info("Employee with ID " + id + " updated.");
    } else {
        LOGGER.info("Employee with ID " + id + " not found (update
            ↪ failed).");
    }
}

public static void deleteEmployee(Connection connection, int id) throws
    ↪ SQLException {
    String query = "DELETE FROM employee WHERE id = ?";
    PreparedStatement preparedStatement =
        ↪ connection.prepareStatement(query);
    preparedStatement.setInt(1, id);
    int rowsDeleted = preparedStatement.executeUpdate();
    if (rowsDeleted > 0) {
        LOGGER.info("Employee with ID " + id + " deleted.");
    } else {
        LOGGER.info("Employee with ID " + id + " not found (delete
            ↪ failed).");
    }
}

public static int countEmployees(Connection connection) throws
    ↪ SQLException {
    String query = "SELECT COUNT(*) FROM employee";
    PreparedStatement preparedStatement =
        ↪ connection.prepareStatement(query);
    ResultSet result = preparedStatement.executeQuery();
    if (result.next()) {
        return result.getInt(1);
    }
    return 0;
}

public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(url,
            ↪ username, password)) {
            createEmployee(connection, "John Doe", 75000.0f);
            createEmployee(connection, "Jane Smith", 82000.5f);

            getEmployee(connection, 1).ifPresent(System.out::println);
            updateEmployee(connection, 2, "Jane Doe", 85000.0f);
            getEmployee(connection, 2).ifPresent(System.out::println);
            System.out.println("Number of employees: " +
                ↪ countEmployees(connection));
            deleteEmployee(connection, 1);
            System.out.println("Number of employees: " +
                ↪ countEmployees(connection));

        } catch (SQLException e) {
            System.err.println("An error occurred. " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Executing the program after the employee table is created, will produce the following log messages.

```
11:57:25.825 [main] INFO  be.pxl.jdbc.EmployeeCRUD - Employee 'John Doe' created.
11:57:25.835 [main] INFO  be.pxl.jdbc.EmployeeCRUD - Employee 'Jane Smith' created.
Employee{id=1, name='John Doe', salary=75000.0}
11:57:25.868 [main] INFO  be.pxl.jdbc.EmployeeCRUD - Employee with ID 2 updated.
Employee{id=2, name='Jane Doe', salary=85000.0}
Number of employees: 2
11:57:25.885 [main] INFO  be.pxl.jdbc.EmployeeCRUD - Employee with ID 1 deleted.
Number of employees: 1
```

ResultSet represents the result set of a query that retrieves data from the database. It's a one-way cursor, meaning you can only iterate through the results once in a forward direction. Always check if resultSet.next() returns true before trying to access column data to avoid errors. resultSet.next() moves the cursor to the next result. It returns true if there's a next row, false if the end is reached. Retrieving data from a result in the ResultSet starts at column 1 (not 0). You can access column data using methods like getInt(columnIndex), getString(columnIndex), etc., specifying the desired column number.

5.6 Transactions

Transactions are a mechanism to group multiple database operations into a single unit. These operations are treated as all-or-nothing:

- Success: If all the operations in the transaction succeed, the changes are permanently applied to the database through a commit.
- Failure: If any operation fails due to errors (e.g. data integrity violation), the entire

transaction is rolled back using rollback, undoing all the changes attempted within that transaction.

Imagine a scenario where you want to give an employee a raise. This could involve updating two tables in your database:

- Employee Table: Update the employee's salary field.
- Salary History Table: Insert a new record reflecting the raise amount and date.

To ensure both updates happen successfully or neither happens at all, we can use a transaction.

```java
package be.pxl.jdbc;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class SalaryRaise {

    private static final DateTimeFormatter FORMAT =
       ↪ DateTimeFormatter.ofPattern("yyyy-MM-dd");
    private static final Logger LOGGER =
       ↪ LogManager.getLogger(EmployeeCRUD.class);
    private static final String URL = "jdbc:mysql://localhost:3307/moviedb";
    private static final String USERNAME = "user";
    private static final String PASSWORD = "password";

    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(URL,
           ↪ USERNAME, PASSWORD)) {
            // Turn off auto-commit to manage transactions manually
            connection.setAutoCommit(false);
            try {
                Statement statement = connection.createStatement();
                statement.executeUpdate("UPDATE employee SET salary = salary
                   ↪ * 1.1 WHERE id = 2");
                statement.executeUpdate("INSERT INTO salary_history
                   ↪ (employee_id, salary_raise, date) VALUES (2, 10, '" +
                   ↪ FORMAT.format(LocalDate.now()) + "')");
                // If both updates succeed, commit the transaction
                connection.commit();
                System.out.println("Salary raise successful!");
            } catch (SQLException e) {
                // If any error occurs, rollback the transaction (undo
                    ↪ changes)
                connection.rollback();
                System.out.println("Salary raise failed: " + e.getMessage());
            }
        } catch (SQLException e) {
            LOGGER.error("A database error occurred. " + e.getMessage());
        }

    }
}
```

We disable auto-commit using setAutoCommit(false). We perform the updates for both tables within a try block. If both updates succeed, commit finalizes the changes. If any SQLException occurs (e.g., insufficient privileges, data type mismatch), the catch block executes rollback to undo all changes within the transaction. This ensures that either both tables are updated reflecting the raise, or none are updated at all, maintaining data integrity.

ACID stands for Atomicity, Consistency, Isolation, and Durability. These are essential properties that guarantee data integrity and consistency in database transactions (like the salary raise example you saw).

- **Atomicity**: This ensures the transaction is treated as a single unit of work. Either all the operations within the transaction succeed, or none of them do.
- **Consistency**: This guarantees the transaction moves the database from one valid state to another. It enforces the database's defined rules and constraints.
- **Isolation**: This ensures concurrent transactions from multiple users don't interfere with each other.
- **Durability**: This guarantees that once a transaction commits, the changes are permanent and persist even in case of system failures (like power outages). The database system ensures the updates are written to stable storage (like a disk) to survive such incidents.

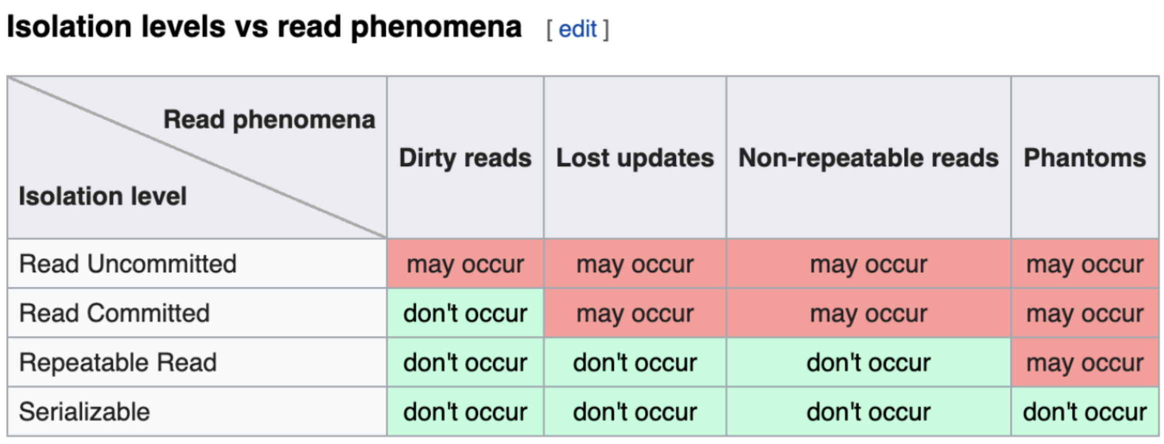
5.7 Transaction Isolation Level

Isolation levels in transactions play a crucial role in managing concurrency, which refers to how multiple transactions access and modify data simultaneously in a database. While concurrency improves performance by allowing multiple users to work with the database, it can also lead to data inconsistencies if not handled properly.

Isolation Levels define the visibility of uncommitted changes made by one transaction to other concurrent transactions. Different isolation levels offer varying degrees of data consistency at the expense of performance. Here are some common isolation levels and related phenomena:

- **Read Uncommitted**: This allows a transaction to read data even if it's not yet committed by another transaction. This can lead to **dirty reads**, where a transaction reads data that is later rolled back, resulting in seeing inconsistent data.

- **Read Committed**: This allows a transaction to read data only after it's committed by another transaction. This prevents dirty reads, but can introduce **non-repeatable reads**. Here, a transaction might read the same data twice and get different results if another transaction modifies the data in between those reads.
- **Repeatable Read**: This ensures a transaction can read the same data multiple times and get the same results, even if another transaction commits changes in between those reads. However, it can lead to **phantom reads**, where a transaction might not see data inserted by another transaction after it started its own read operation.
- **Serializable**: This is the most stringent level, ensuring transactions are serialized (executed one after another) as if there were no concurrency. It eliminates all the anomalies mentioned above (dirty reads, non-repeatable reads, phantom reads) but has the worst performance impact.



Isolation levels vs read phenomena [edit]

Read phenomena / Isolation level	Dirty reads	Lost updates	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur	may occur
Read Committed	don't occur	may occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Figure 5.1: Isolation Levels

Dirty Read

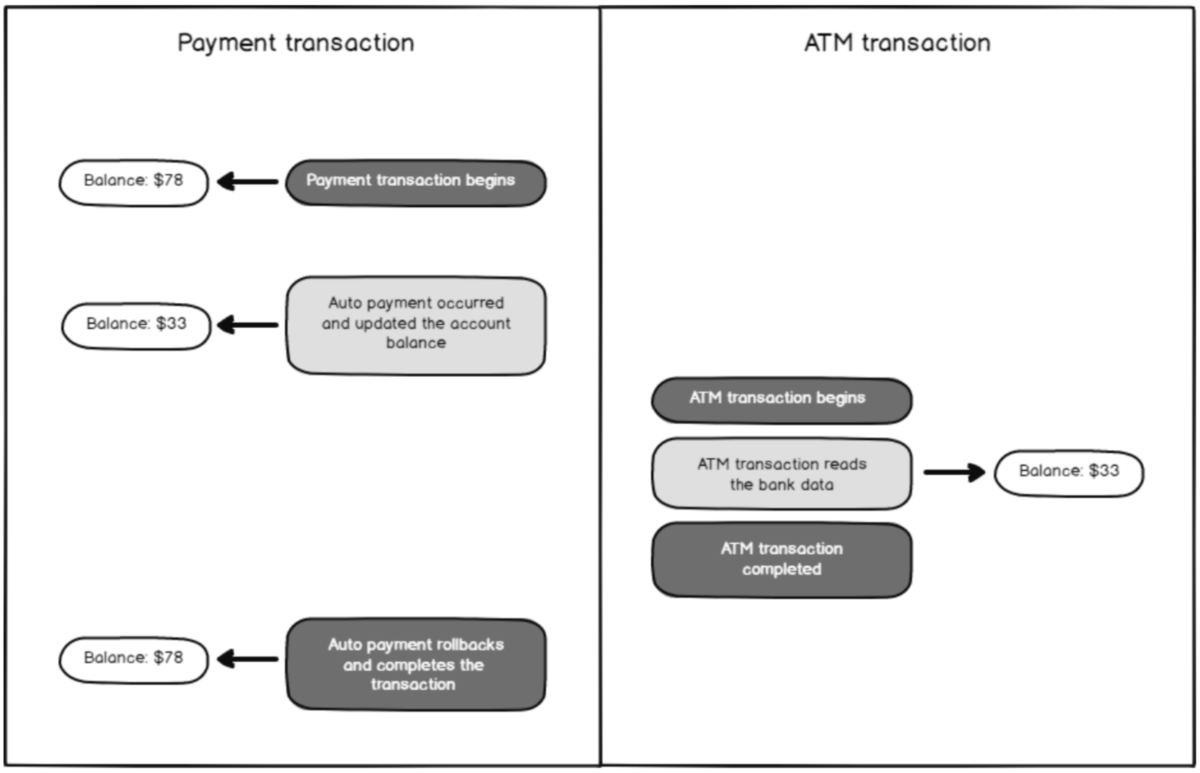

Payment transaction
ATM transaction
Balance: $78
Payment transaction begins
Balance: $33
Auto payment occurred and updated the account balance
ATM transaction begins
ATM transaction reads the bank data
Balance: $33
ATM transaction completed
Balance: $78
Auto payment rollbacks and completes the transaction

Figure 5.2: Dirty Read

A dirty read occurs when a transaction reads data that has been modified by another transaction, but that modification hasn't been committed yet. Imagine reading a friend's draft message before they send it - you see unfinalized information.

Non-Repeatable Read

Non-repeatable reads happen when a transaction reads the same data twice within its execution, and the data has changed in between due to another committed transaction.

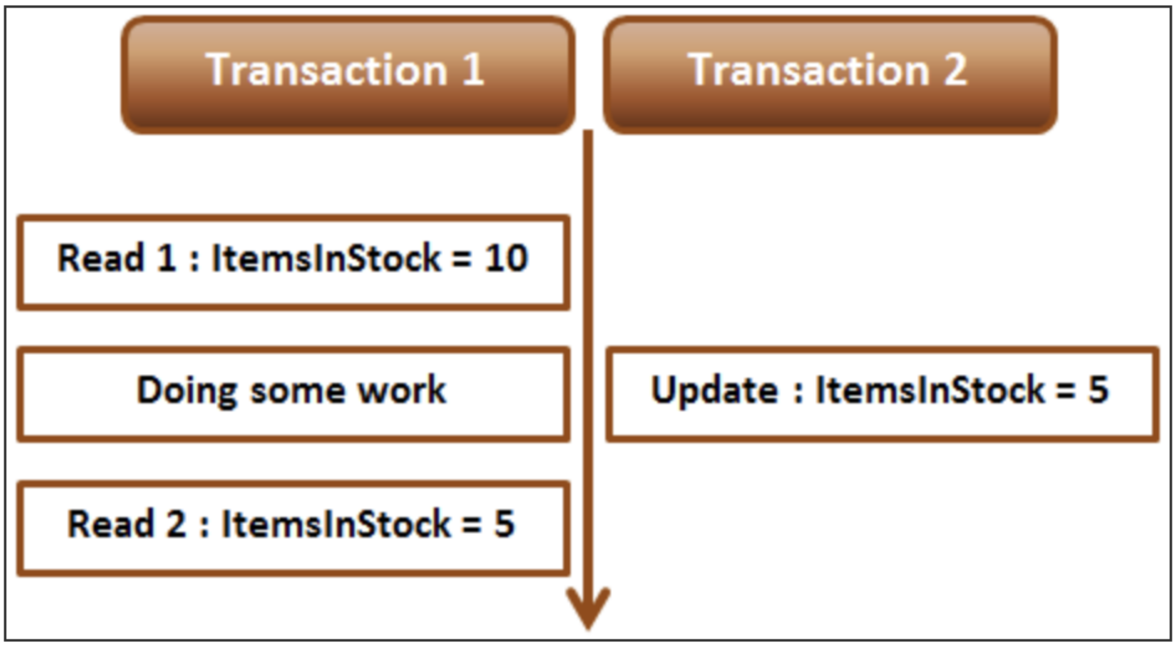

Transaction 1
Transaction 2
Read 1 : ItemsInStock = 10
Doing some work
Update : ItemsInStock = 5
Read 2 : ItemsInStock = 5

Figure 5.3: Non-Repeatable Read - Transaction 1 starts first, reads ItemsInStock and gets a value of 10. Transaction 1 is doing some work and at this point transaction 2 starts and updates ItemsInStock to 5. Transaction 1 then makes a new read. At this point transaction 1 gets a value of 5, resulting in non-repeatable read problem.

Lost update

Lost update isn't directly an isolation level issue, but a concurrency problem. It occurs when two transactions try to modify the same data piece simultaneously, and due to isolation levels, only one update succeeds, effectively "losing" the other update.

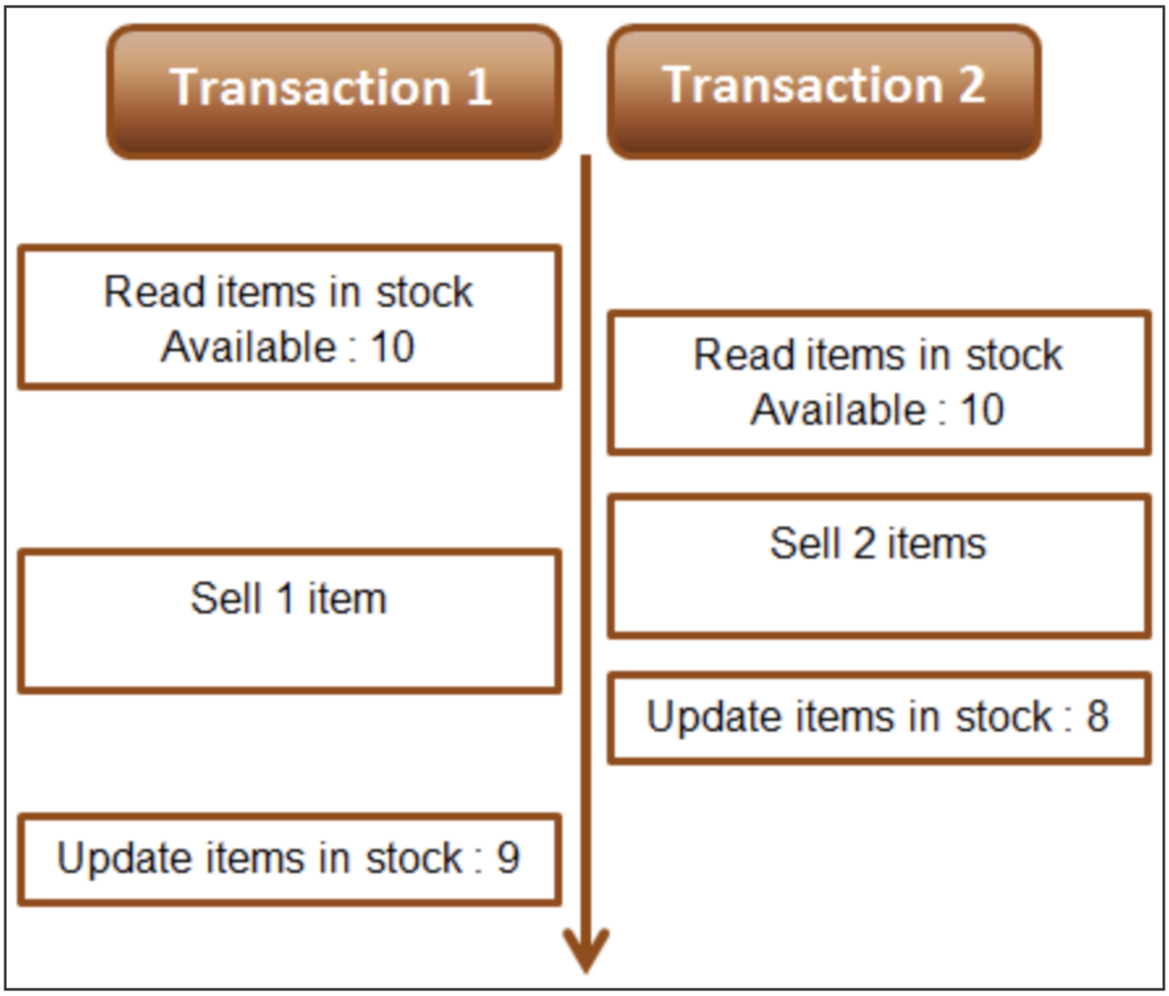

Transaction 1
Transaction 2
Read items in stock Available : 10
Read items in stock Available : 10
Sell 2 items
Sell 1 item
Update items in stock : 8
Update items in stock : 9

Figure 5.4: Lost Update - Transaction 1 silently overwrites the update of Transaction 2. This is called the lost update problem.

Phantom Read

Phantom Read occurs when a transaction reads data that wasn't there when it started its read operation, but another committed transaction inserted that data in between.

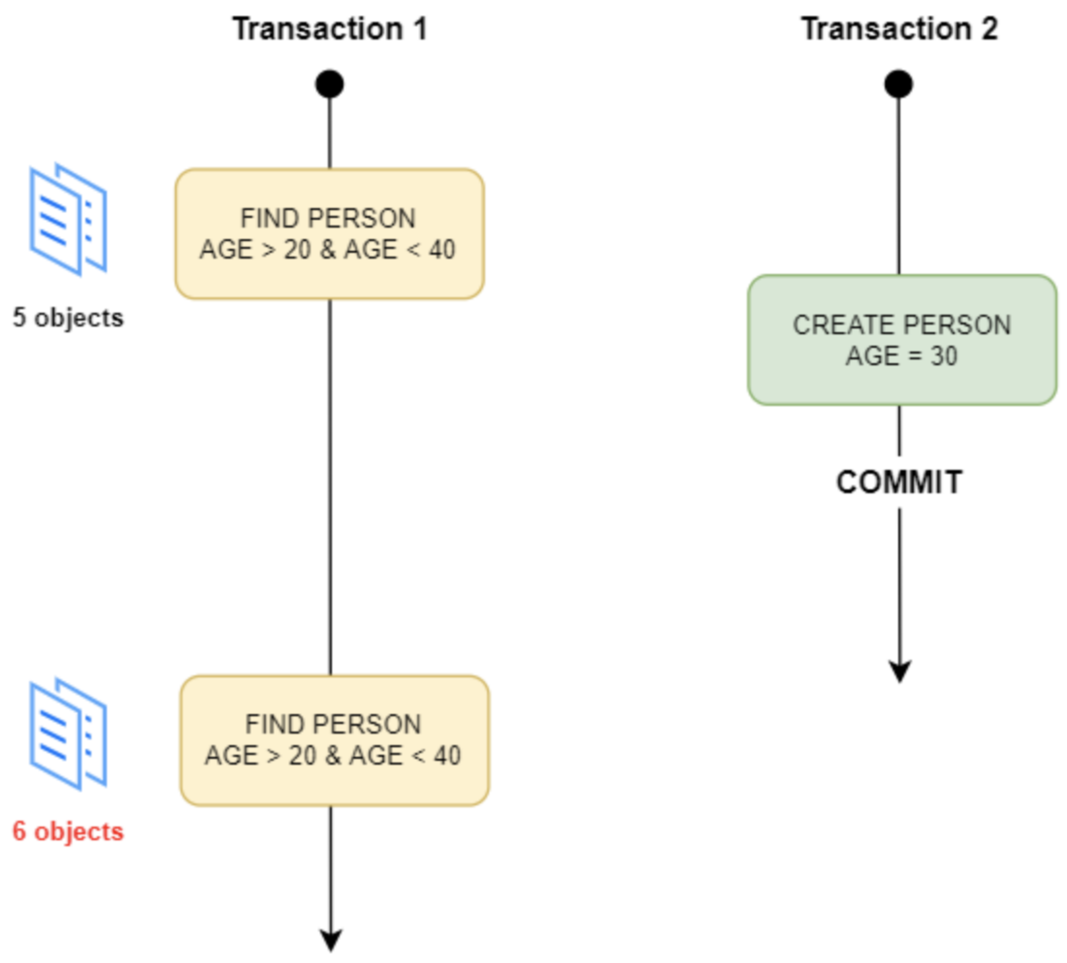


Figure 5.5: Phantom Read

Non-Repeatable Read deals with existing data being modified by another transaction after the first read while Phantom Read deals with entirely new data being inserted by another transaction that the first read hasn't seen.

Choosing the right isolation level depends on the specific needs of your application. Here's a general guideline:

- High Consistency: If data consistency is paramount and occasional performance slowdowns are acceptable, choose Read Committed or Repeatable Read.
- High Concurrency: If performance is crucial and some data inconsistencies are tolerable, consider Read Uncommitted (use with caution!).

Serializable offers the strongest consistency but at the cost of significant performance overhead. Use it when absolutely necessary.

5.7.1 Example

This program will update data in the database and may or may not commit the changes immediately, depending on the scenario you want to illustrate.

```java
package be.pxl.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class UpdaterProgram {

    static final String DB_URL = "jdbc:mysql://localhost:3307/moviedb";
    static final String USER = "user";
    static final String PASS = "password";

    public static void main(String[] args) {

        try (Connection conn = DriverManager.getConnection(DB_URL, USER,
            ↪ PASS)) {
            conn.setAutoCommit(false);

            try (Statement stmt = conn.createStatement()) {
                String sqlUpdate = "UPDATE employee SET salary = salary +
                    ↪ 1000 WHERE id = " + args[0];
                stmt.executeUpdate(sqlUpdate);
                System.out.println("Data updated but not committed yet.
                    ↪ Sleeping for 10 seconds...");
                // Sleep to allow time for the ReaderProgram to run and
                    ↪ attempt to read the uncommitted data.
                Thread.sleep(10000);

                if ("commit".equals(args[1])) {
                    conn.commit();
                    System.out.println("Commit");
                } else {
                    conn.rollback();
                    System.out.println("Rollback");
                }
            }
        } catch (SQLException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

This program will attempt to read data from the database. Depending on the isolation level set and whether the Updater Program has committed its changes, the Reader Program may see different results.

```java
package be.pxl.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ReaderProgram {

    static final String URL = "jdbc:mysql://localhost:3307/moviedb";
    static final String USER = "user";
    static final String PASS = "password";

    public static void main(String[] args) {
        try (Connection conn = DriverManager.getConnection(URL, USER,
            ↪ PASS)) {
            // Set the isolation level here, for example:
            conn.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);

            try (Statement stmt = conn.createStatement()) {
                String sql = "SELECT * FROM employee WHERE id = " + args[0];
                ResultSet rs = stmt.executeQuery(sql);

                while (rs.next()) {
                    int id = rs.getInt("id");
                    String name = rs.getString("name");
                    double salary = rs.getDouble("salary");

                    System.out.println("ID: " + id + ", Name: " + name + ",
                        ↪ Salary: " + salary);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Start the UpdaterProgram. This program will update a row but will not commit immediately. It sleeps for 10 seconds, allowing you to start the ReaderProgram within this window.

Run the ReaderProgram. Depending on the isolation level you set in the Reader Program, it may or may not see the uncommitted changes made by the UpdaterProgram. You should run the Reader Program multiple times, each time setting a different isolation level to observe how each level affects the visibility of uncommitted changes.

- TRANSACTION_READ_UNCOMMITTED: The Reader Program will see the uncommitted changes.

- TRANSACTION_READ_COMMITTED (and higher levels): The Reader Program will not see the uncommitted changes.

5.8 Connection pooling

Opening and closing database connections like you did above with the DriverManager.getConnection method takes some time.

Especially in web applications, you do not want to open up a fresh database connection for every single user request, rather you want to have a small pool of connections that are always open and shared between users.

That's what JDBC connection pools are for. A connection pool keeps open a small number of database connections (think: 10) and instead of opening up database connections yourself, you'll ask the connection pool to give you one of these (10) connections.

HikariCP is the recommended connection pool to use in your Java application. However, you have plenty possibilities to choose from. HikariCP is also Spring Boot's default choice. Use Oracle's UCP, if you are working with Oracle databases.

All of these connection pools are solid, performant, and offer sane defaults and error handling.

Independent of the option you choose, you will then, in your JDBC code, not open up connections yourself through the DriverManager, but rather you will construct a connection pool, represented by the DataSource interface, and ask it to give you one of its connections.

```java
package be.pxl.jdbc;

import com.zaxxer.hikari.HikariDataSource;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Optional;

public class ConnectionPoolExample {
    private static final Logger LOGGER =
        ↪ LogManager.getLogger(ConnectionPoolExample.class);

    private static final String URL = "jdbc:mysql://localhost:3307/moviedb";
    private static final String USERNAME = "user";
    private static final String PASSWORD = "password";
    public static Optional<Employee> getEmployee(Connection connection, int
        ↪ id) throws SQLException {
        String query = "SELECT * FROM employee WHERE id = ?";
        PreparedStatement preparedStatement =
            ↪ connection.prepareStatement(query);
        preparedStatement.setInt(1, id);
        ResultSet resultSet = preparedStatement.executeQuery();

        if (resultSet.next()) {
            int retrievedId = resultSet.getInt("id");
            String retrievedName = resultSet.getString("name");
            double retrievedSalary = resultSet.getDouble("salary");
            Employee employee = new Employee(retrievedId, retrievedName);
            employee.setSalary(retrievedSalary);
            return Optional.of(employee);
        } else {
            LOGGER.info("Employee with ID " + id + " not found.");
            return Optional.empty();
        }
    }

    public static void main(String[] args) throws SQLException {
        DataSource dataSource = createDataSource();

        try (Connection connection = dataSource.getConnection()) {
            getEmployee(connection, 2).ifPresent(System.out::println);
        }
    }

    private static DataSource createDataSource() {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(URL);
        ds.setUsername(USERNAME);
        ds.setPassword(PASSWORD);
        return ds;
    }
}
```

Chapter 6

Spring Data JPA

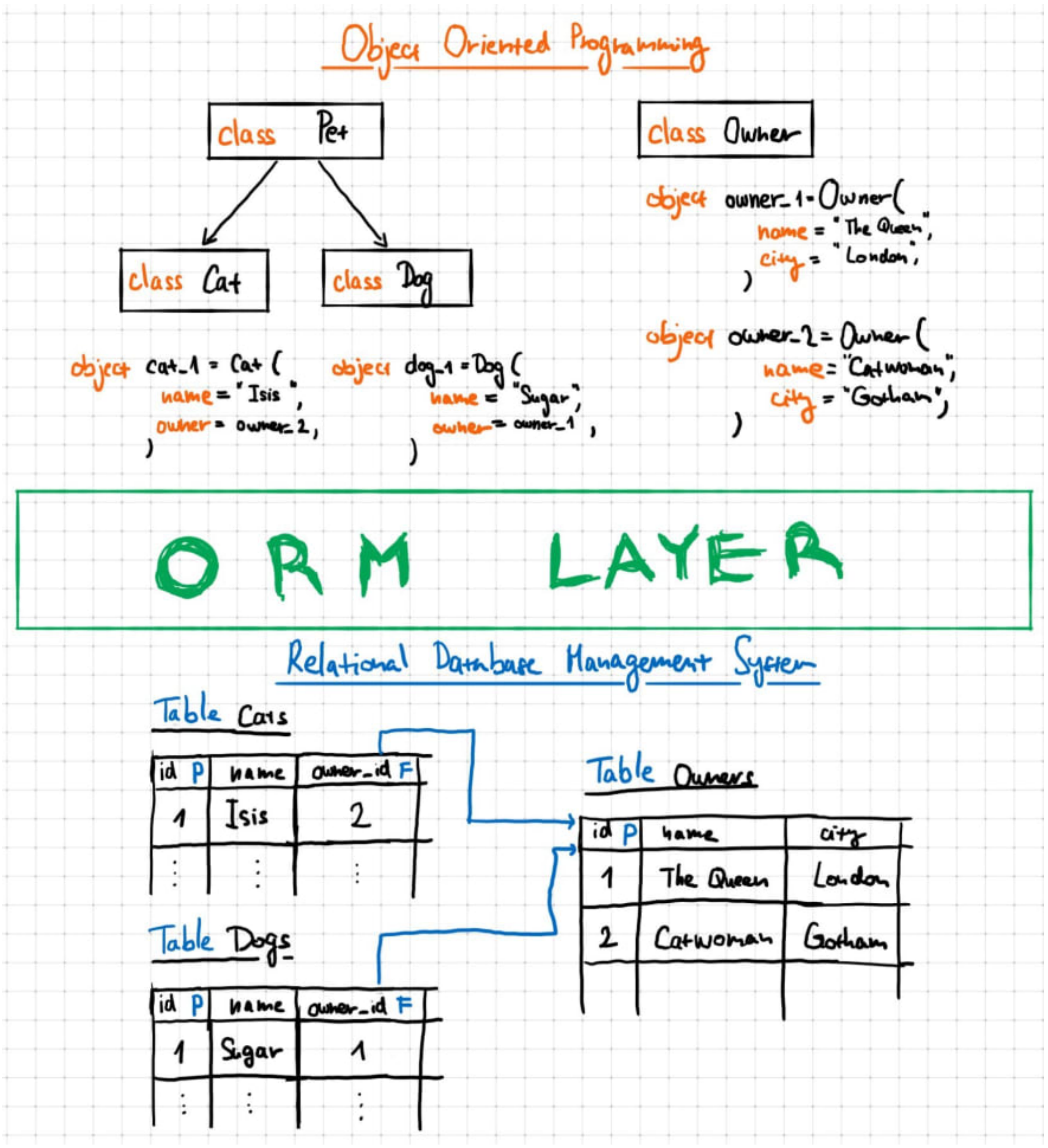
Learning goals
The junior-colleague

1. can describe the concept of ORM.
2. can explain what JPA is.
3. can explain the benefits of JPA over JDBC.
4. can list and describe different JPA providers.
5. can define what Hibernate is and its role within JPA.
6. can explain the benefits of using Spring Data JPA.
7. can describe what a persistence unit entails in JPA.
8. can outline and explain the fundamental interfaces of JPA.
9. can discuss what the persistence context is and its importance.
10. can implement and explain entity classes in JPA.
11. can use and explain the different annotations in JPA entity classes.
12. can describe the lifecycle of entity objects in JPA.
13. can implement and describe various types of relationships in entity classes.
14. can implement CRUD operations using JPA.
15. can implement queries using method names and annotations in Spring Data JPA.
16. can describe and implement transactions in JPA.
17. can explain transaction propagation.
18. can explain fetching strategy.
19. can describe and configure application properties for logging and debugging.
20. can describe and configure application properties for managing transactions.
21. can implement pagination and filtering in JPA applications.
22. can explain, identify, and solve the N + 1 query problem in JPA applications.
23. can implement unit tests using @DataJpaTest

Source for this chapter:
https://github.com/custersnele/demo_spring_data_jpa

6.1 What is ORM?

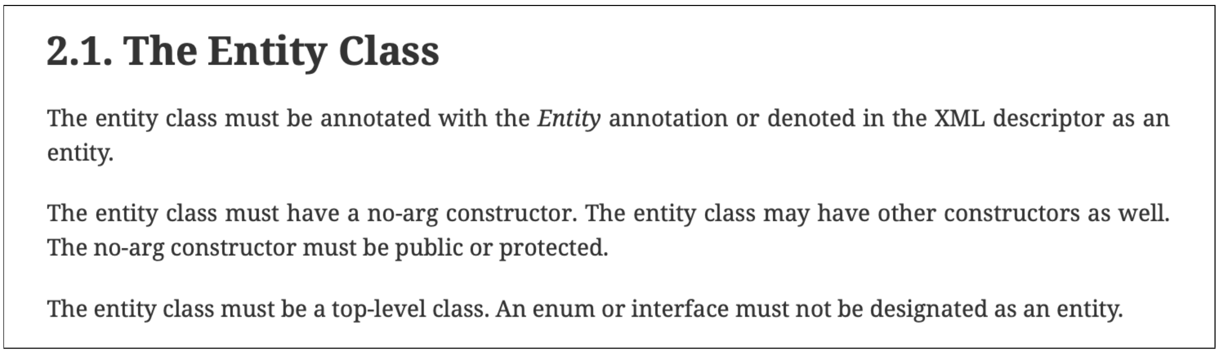
Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a relational database using an object-oriented programming language.


Object Oriented Programming
class Pet
class Owner
class Cat
class Dog
object owner_1 = Owner(name = "The Queen", city = "London",)
object cat_1 = Cat(name = "Isis", owner = owner_2,)
object dog_1 = Dog(name = "Sugar", owner = owner_1,)
object owner_2 = Owner(name = "Catwoman", city = "Gotham",)
ORM LAYER
Relational Database Management System
Table Cats
id P | name | owner_id F
1 | Isis | 2
Table Owners
id P | name | city
1 | The Queen | London
2 | Catwoman | Gotham
Table Dogs
id P | name | owner_id F
1 | Sugar | 1

6.2 What is JPA?

The Java Persistence API (or more correctly Jakarta Persistence API) is a specification that defines how to persist data in Java applications. JPA mainly focuses on the ORM layer and managing persistent objects.

JPA is a specification which means JPA consists of implementation guidelines for the Java ORM layer and the syntax. The specification only comes with interfaces, no actual implementation. A reference implementation can be provided but other companies can create and distribute their own implementation of the specification.



2.1. The Entity Class

The entity class must be annotated with the *Entity* annotation or denoted in the XML descriptor as an entity.

The entity class must have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor must be public or protected.

The entity class must be a top-level class. An enum or interface must not be designated as an entity.

Hibernate is the default JPA provider when building Spring Boot applications. [1].

Spring Data JPA is a part of the larger Spring Data family, aimed at simplifying data access and manipulation across a variety of database technologies. Spring Data JPA adds a layer on top of JPA. It uses all the features defined by the JPA specification, but adds no-code implementation of the repository pattern and the creation of database queries from method names. Within this framework, JpaRepository is a key interface that extends Spring Data's repository abstraction, specifically tailored for JPA technology. This interface is designed to streamline the storage and retrieval processes of entity instances, reducing the amount of boilerplate code developers need to write and offering a more intuitive interaction with the data layer. JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository.

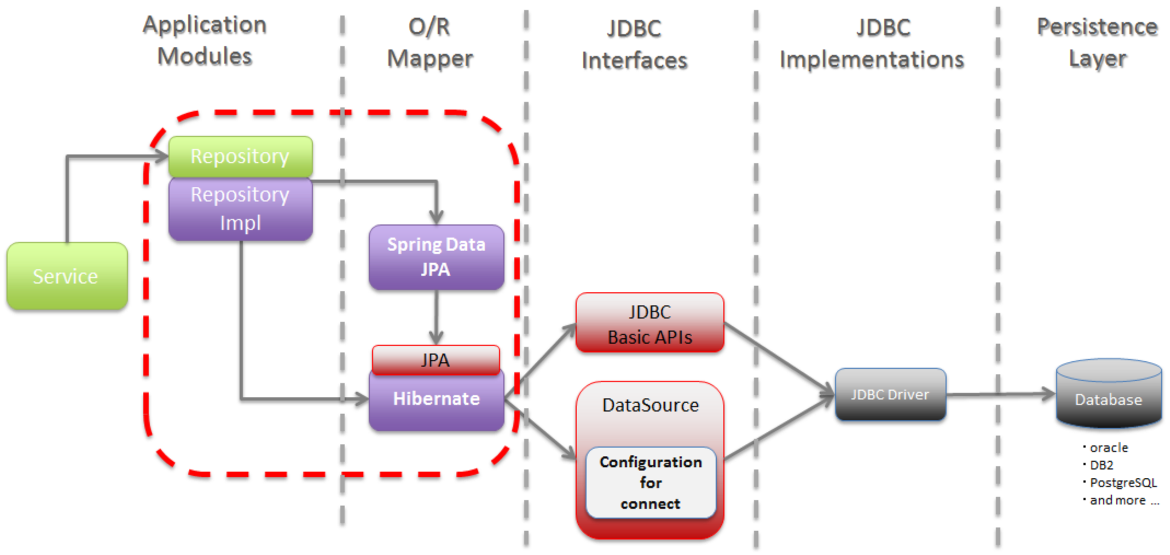
Their main functions are:

- CrudRepository mainly provides CRUD functions.
- PagingAndSortingRepository provides methods to do pagination and sorting records.
- JpaRepository provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.

6.3 Datasource and application properties

In Spring Boot a DataSource-object is the preferred means of getting a connection to a database. The interface jakarta.sql.DataSource is implemented by the database driver vendor.

[1] https://hibernate.org/andhttps://hibernate.org/orm/



The datasource can be specified in the application.properties file. Here are some common database properties:

spring.datasource.url	JDBC URL of the database.
spring.datasource.username	Login username of the database.
spring.datasource.password	Login password of the database.
spring.jpa.show-sql	Whether to enable logging of SQL statements. Default: false
spring.jpa.hibernate.ddl-auto	Possible values: none (production), create, create-drop, validate and update. [2]

Alternatively, the data source configuration can be done programmatically.

The appropriate JDBC driver for your database must be included in your project by declaring the driver as a dependency in your project's Maven configuration file, pom.xml. The dependency ensures that the JDBC driver is available during runtime, allowing Spring Data JPA to establish and manage database connections.

Here is an example of how to include a JDBC driver for MySQL in your pom.xml file:

```
<dependency>
<groupId>com.mysql</groupId>
<artifactId>mysql-connector-j</artifactId>
<scope>runtime</scope>
</dependency>
```

Similarly, to connect to a PostgreSQL database, you would use the PostgreSQL JDBC driver as shown below:

```
<dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<scope>runtime</scope>
</dependency>
```

Adjusting the driver version in your pom.xml may be necessary as you upgrade your database server or migrate to a newer version of Spring Data JPA.

6.3.1 Application properties

Configuring a Spring Boot application involves more than just setting up the datasource. In this section, we delve into application properties that assist in database initialization, enhance logging and debugging capabilities, and facilitate robust transaction management. Understanding these properties will enable you to fine-tune your application for better performance, easier maintenance, and more effective error handling.

Database initialization

The setting for Spring Data JPA's property *spring.jpa.hibernate.ddl-auto* is directly transferred to Hibernate as the property *hibernate.hbm2ddl.auto*. The permissible values — *create*, *create-drop*, *validate*, and *update* — determine how the schema tool management adjusts the database schema during application startup.

Logging and debugging

logging.level.ROOT is used for configuring the base level of logging across all components in the application unless overridden.

- **spring.jpa.show-sql**: When set to true, Hibernate will log all SQL statements, which is invaluable for debugging and understanding how JPA translates entity operations into SQL.
- **spring.jpa.properties.hibernate.format_sql**: Improves the readability of the SQL logged by Hibernate by formatting it.
- **logging.level.org.hibernate.type.descriptor.sql**: Adjusting the log level makes it possible to log JDBC parameters and the results coming from the database.

Transaction Management

Some properties are related to transaction management and connection handling, and configuring them properly can improve both performance and robustness, especially under load.

- **spring.datasource.hikari.auto-commit=false**
  This property applies to the HikariCP connection pool, which is the default in Spring Boot. Setting auto-commit to false means that each connection obtained from the pool does not automatically commit after each individual SQL statement. Instead, the application (or Spring's transaction manager) must explicitly call commit or rollback. In Spring Boot, this is handled for you via the **@Transactional** annotation. You rarely need to call commit() manually. This setting is crucial for ensuring consistent and predictable transaction behavior, especially in multi-statement transactions.
- **spring.jpa.properties.hibernate.connection.provider_disables_autocommit=true**
  This tells Hibernate that the connection pool is already managing auto-commit, so Hibernate should not attempt to change it. It avoids redundant JDBC calls, prevents

log warnings, and ensures that Hibernate respects the pooling layer's responsibility. This setting is particularly useful in Hibernate 5.4+ and Hibernate 6+, where connection management has become stricter.

- **spring.jpa.open-in-view=false**
  This setting controls whether a Hibernate Session (and JPA EntityManager) remains open during the entire HTTP request, including the response phase. By default, it is set to true, which allows lazy-loading of JPA relationships during view rendering. While this can be convenient in development, it introduces several issues:

  - Longer database connections remain open.
  - Unexpected database access can happen late in the request.
  - Lazy-loading exceptions are harder to detect early.
  - It violates separation of concerns, since the controller layer may trigger database operations.

  Setting this property to false ensures that the transaction is closed after the service layer has finished executing.

  This also applies when using @RestController, even if no HTML view is rendered. During JSON serialization (e.g. using Jackson), if a lazy-loaded relationship is accessed outside of the transaction, a LazyInitializationException may occur. Therefore, when open-in-view is disabled, it is important to fetch all required data in the service layer, and ideally return DTOs instead of JPA entities from REST controllers.

These settings is especially important for production environments, where performance, scalability, and architectural boundaries are critical.

For more info, see: `https://backendhance.com/en/blog/2023/open-session-in-view/`

6.4 The Entity class

Let's explore an entity class that uses key annotations to manipulate event records in a database.

```java
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.persistence.Column;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Embedded;
import jakarta.persistence.Embeddable;

@Entity
@Table(name = "events")
public class Event {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", nullable = false, length = 200)
    private String title;

    @Enumerated(value = EnumType.STRING)
    private Category category;

    @Embedded
    private EventDetails details;

    // Constructors, getters, and setters
}

@Embeddable
class EventDetails {
    private LocalDate date;
    @Column(length = 512)
    private String location;

    // Constructors, getters, and setters
}
```

A JPA entity class is a POJO (Plain Old Java Object) class that is annotated as having the ability to represent objects in the database. The **@Entity** annotation is used to declare that a class is an entity, so JPA will manage it and map it to a database table.

@Table specifies the table in the database with which the entity is mapped.

The **@Id** annotation marks a field as a primary key field.

The **@GeneratedValue** annotation is used to configure the primary key generation strategy for an entity. This annotation is usually combined with @Id to specify that the persistence provider should automatically generate a unique identifier for the entity objects. There are several strategies defined in the GenerationType enumeration that can be used with @GeneratedValue. Here's an overview:

- **GenerationType.AUTO**

  This is the default strategy and the persistence provider will choose the generation strategy based on the specific database capabilities and dialect.

- **GenerationType.IDENTITY**

  Indicates that the persistence provider must assign primary keys using the database identity column. This is typically supported by SQL databases with an auto-increment column.

- **GenerationType.SEQUENCE**

  Specifies that the primary key values will be generated using a database sequence.

This is a special database object that generates numbers in sequential order. Not all databases support sequences. This strategy is useful for databases supporting sequences, like Oracle, PostgreSQL, etc. It's beneficial for high-volume applications due to better performance compared to IDENTITY, as the sequence generation can be more efficiently managed.

- **GenerationType.TABLE**

  Uses a database table to simulate a sequence. This strategy uses a table to hold the next id incrementally. It's a portable solution but not as efficient as using database-specific features like sequences or identity columns.

The **@Column** annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **name**: specify the name of the column.
- **length**: specify the size of the column, particularly for a String value.
- **nullable**: mark the column to be NOT NULL when the database schema is generated.
- **unique**: mark the column to contain only unique values.

The **@Embeddable** annotation marks a class to be embedded within another entity. In this example, the EventDetails does not need its own table; instead, its properties are incorporated into the table of the entity that embeds it.

The annotation **@Embedded** is used to denote a class whose instances are stored as an intrinsic part of an owning entity. All the fields of EventDetails class are embedded within the events table, avoiding the need for a separate table.

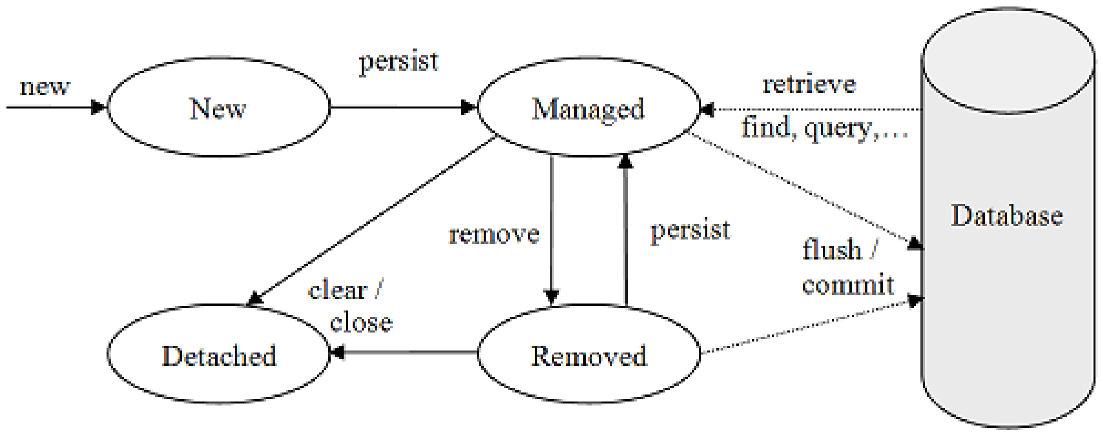
The **@Enumerated** annotation is used to specify how an enum type should be mapped to a database column when using JPA. This annotation can take one of two values: EnumType.STRING or EnumType.ORDINAL.

Using EnumType.STRING as the value for the @Enumerated annotation means that the enum values are stored in the database as their corresponding string names. This approach is often preferred because it makes the database values clear and readable, and it ensures that changes to the order of the enum values in the code do not affect the stored data.

6.5 Entity lifecycle

The EntityManager is a core interface of JPA. An instance of EntityManager is used to create and remove entity instances, to find entities by their primary key, and to query over entities. In fact, an instance of the EntityManager is used to interact with the persistence context.

The persistence context is one of the main concepts in JPA. It is a set of all entity object that you are currently using or used recently. You can think of the persistence context as some kind of first-level cache. Each entity object in the persistence context represents a record in the database. The persistence context manages these entity objects and their lifecycle. Each entity object has one of 4 states: new, managed, removed, and detached.



A newly instantiated entity object is in state **new** or **transient**. The entity object hasn't been persisted yet, so there's no database record yet. The persistence context does not know about the entity object yet.

All entity objects that are managed by the persistence context are in the lifecycle state **managed**. An entity object becomes managed when it is persisted to the database. Entity object retrieved from the database are also in the managed state. If a managed entity object is modified (updated) within an active transaction, the change is detected by the persistence context and passed on to the database.

When a managed entity is removed within an active transaction, the state changes from managed to removed, and the record is physically deleted from the database (when the transaction commits).

An entity gets detached when it is no longer managed by the persistence context but still represents a record in the database. Detached objects are limited in functionality.

Programmatically we need to use the entitymanager to change the state of the entity object in the persistence context which results in changes or updates in our database.

When using Spring Data JPA you don't have to implement the interaction with the entitymanager. When you create a Repository, the SimpleJpaRepository class provided by Spring Data JPA provides the default implementation. SimpleJpaRepository class internally uses JPA entitymanager.

Exercise 6.1. Open the class SimpleJpaRepository and take a look at its implementation (e.g. save()-method).

6.6 Relationships

6.6.1 OneToOne relationship

A OneToOne relationship in JPA is used when one entity instance is associated with exactly one instance of another entity. In this example a user has exactly one passport.

In a **uni-directional** OneToOne relationship, only one side of the relationship maintains knowledge of the other side's existence. Consider a scenario where each Person has exactly one Passport. The Person entity holds a reference to the Passport, but the Passport does not hold any reference to the Person.

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "passport_id", referencedColumnName = "id")
    private Passport passport;

    // Standard getters and setters
}

@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String number;

    // Standard getters and setters
}
```

In this example, the Person entity contains a Passport field annotated with @OneToOne. The @JoinColumn annotation specifies the column in the Person table that joins to the primary key of the Passport entity.

The concept of c̈ascade types̈defines how persistence operations such as save, delete, and update performed on one entity should be propagated (or cascaded) to its associated entity.

When using the @OneToOne annotation in JPA, you can specify a cascade type to automate the persistence lifecycle management of the associated entity. Here are the most common cascade types used in JPA:

- **CascadeType.PERSIST**: When persisting an entity, also persist the associated entity. For example, when saving a Person object, also save its associated Passport.
- **CascadeType.MERGE**: When merging the state of an entity into the current persistence context, also merge the entity held in this field.
- **CascadeType.REMOVE**: When deleting an entity, also delete the associated entity. This is particularly useful when the associated entity no longer makes sense to exist independently.

- **CascadeType.REFRESH**: When refreshing an entity, also refresh the entities held in this field. This means reloading the content of the associated entity from the database, which can be useful if the database might be changed by other processes.
- **CascadeType.DETACH**: When an entity is detached from the persistence context, also detach the entities held in this field.
- **CascadeType.ALL**: A convenience that cascades all the above operations (PERSIST, MERGE, REMOVE, REFRESH, DETACH). Using CascadeType.ALL is common for @OneToOne and @OneToMany associations.

In a **bi-directional** OneToOne relationship, both entities are aware of each other. This relationship allows navigation from either side. Using the same example as above, both the Person and Passport entities can have references to each other.

```java
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
    private Passport passport;

    // Standard getters and setters
}

@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String number;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

    // Standard getters and setters
}
```

In this bi-directional setup, the Person entity uses the mappedBy attribute in the @OneToOne annotation to indicate that the Person is not the owner of the relationship and that the Passport entity contains the foreign key.

6.6.2 ManyToOne relationship

A ManyToOne relationship in JPA is commonly used when one entity is related to multiple instances of another entity. For instance, in a blog system, many posts may belong to one category.

Uni-directional ManyToOne Relationship In a uni-directional ManyToOne relationship, only the 'many' side of the relationship is aware of the 'one' side. This setup is often seen when the 'one' side doesn't need to directly access the 'many' side.

Example: Books and Authors Suppose each book can have one author, but each author can write many books. Here, the relationship from books to authors is a typical example of a uni-directional ManyToOne relationship.

```java
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id", nullable = false)
    private Author author;

    // Standard getters and setters
}

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // Standard getters and setters, no reference back to Book
}
```

In this model, each Book entity holds a reference to its Author, which is annotated with @ManyToOne. The @JoinColumn annotation specifies the foreign key column in the Book table that links to the Author.

Repository Configuration Repositories for each entity can be defined as follows:

```java
public interface BookRepository extends JpaRepository<Book, Long> {
}

public interface AuthorRepository extends JpaRepository<Author, Long> {
}
```

Bi-directional ManyToOne Relationship In a bi-directional relationship, both sides of the relationship are aware of each other. This is useful when you want to navigate the relationship from either side.

Example: Books and Authors Continued Expanding on the previous example, suppose now we want authors to also be aware of the books they've written.

Entity Definition

```java
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id", nullable = false)
    private Author author;

    // Standard getters and setters
}

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")
    private Set<Book> books = new HashSet<>();

    // Standard getters and setters
}
```

In this bi-directional setup, the Author class includes a Set<Book> to hold the collection of books. The @OneToMany annotation in Author uses the mappedBy attribute to indicate that the Author entity is not the owner of the relationship and that the Book entity contains the foreign key.

6.6.3 ManyToMany relationship

A ManyToMany relationship is used when multiple instances of one entity are associated with multiple instances of another entity. Using the example of books and authors, a ManyToMany relationship would be appropriate if each book could have multiple authors and each author could write multiple books.

In a uni-directional ManyToMany relationship, only one entity maintains the relationship information.

Imagine a scenario where each book can have multiple authors, but we are only interested in navigating from books to authors and not the other way around.

```java
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToMany
    @JoinTable(
        name = "book_author",
        joinColumns = @JoinColumn(name = "book_id"),
        inverseJoinColumns = @JoinColumn(name = "author_id")
    )
    private Set<Author> authors = new HashSet<>();

    // Standard getters and setters
}

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // Standard getters and setters, no reference back to Books
}
```

In this example, the Book entity defines a ManyToMany relationship to the Author entity. The @JoinTable annotation specifies the table that maps books to authors, including columns for each foreign key.

In a bi-directional ManyToMany relationship, both entities are aware of each other, and navigation is possible from either side.

This time, both books and authors are aware of each other and can navigate the relationship.

```java
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToMany
    @JoinTable(
        name = "book_author",
        joinColumns = @JoinColumn(name = "book_id"),
        inverseJoinColumns = @JoinColumn(name = "author_id")
    )
    private Set<Author> authors = new HashSet<>();

    // Standard getters and setters
}

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<>();

    // Standard getters and setters
}
```

In the bi-directional arrangement, the Author entity includes a Set<Book> to hold the collection of books. The mappedBy attribute indicates that the Author is not the owner of the relationship, and the mapping details are managed by the Book entity.

6.7 Fetching strategy

JPA provides two main fetching strategies to control how related entities are loaded from the database:

- **Eager Fetching**: With eager fetching, related entities are loaded immediately with the main entity, whether they are accessed in the application or not. This can lead to performance issues if the relationships involve large sets of data. Eager fetching is often used when the data sets are small or always used together with the main entity.
- **Lazy Fetching**: Lazy fetching loads the related entities only when they are explicitly accessed in the application. This can improve performance by reducing the initial load time and the amount of memory consumed. However, it requires careful management of the persistence context to avoid LazyInitializationException.

In the case of ManyToMany relationships, the default fetching strategy is lazy because eager fetching can result in very large numbers of joins and thus severe performance degradation.

```
@ManyToMany(fetch = FetchType.LAZY)
private Set<Author> authors = new HashSet<>();
```

Understanding and choosing the right fetching strategy based on the specific use case and data access patterns is crucial for developing efficient, scalable applications.

Consider the following method findBooksByAuthor() in a BookService. When we look closely at the logging of the SQL-queries. We see that first, the author is retrieved and later, the author's books are retrieved on the moment we call author.getBooks(), not sooner. This is lazy loading or lazy fetching. When dealing with one-to-many or many-to-many relationships, lazy fetching is mostly the best solution in terms of performance.

Make sure your method is running in a transaction. Otherwise you will encounter a LazyInitializationException.

```
org.hibernate.LazyInitializationException:
failed to lazily initialize a collection of role: be.pxl.domain.Author.books:
could not initialize proxy - no Session
```

```
@Service
public class BookService {

    private static final Logger LOGGER =
       ↪ LogManager.getLogger(BookService.class);

    private final AuthorRepository authorRepository;

    public BookService(AuthorRepository authorRepository) {
        this.authorRepository = authorRepository;
    }

    @Transactional
    public List<String> getBooksByAuthor(String authorName) {
        LOGGER.info("Starting getBooksByAuthor");
        Author author =
           ↪ authorRepository.findAuthorByName(authorName).orElseThrow(()
           ↪ -> new NotFoundException(""));
        LOGGER.info("Author retrieved...");
        return author.getBooks().stream().map(Book::getTitle).toList();
    }
}
```

```
2024-04-23T20:23:48.883+02:00  INFO 26656 --- [nio-8080-exec-8]
be.pxl.service.BookService      : Starting getBooksByAuthor
Hibernate:
    select
        a1_0.id,
        a1_0.name
    from
        authors a1_0
    where
        a1_0.name=?
2024-04-23T20:23:48.946+02:00  INFO 26656 --- [nio-8080-exec-8]
be.pxl.service.BookService      : Author retrieved...
Hibernate:
    select
        b1_0.author_id,
        b1_1.id,
        b1_1.title
    from
        book_authors b1_0
    join
        books b1_1
            on b1_1.id=b1_0.book_id
    where
        b1_0.author_id=?
```

If we change the relationship between Author and Books to eager fetching, the books are retrieved on the moment we search the author by its name.

```java
@Entity
@Table(name = "authors")
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "authors", fetch = FetchType.EAGER)
    private Set<Book> books = new HashSet<>();

    // Standard getters and setters

    public void addBook(Book book) {
        books.add(book);
    }

    public Set<Book> getBooks() {
        return books;
    }
}
```

```
2024-04-23T20:21:02.145+02:00  INFO 19956 --- [nio-8080-exec-6]
be.pxl.service.BookService      : Starting getBooksByAuthor
Hibernate:
    select
```

```
        a1_0.id,
        a1_0.name
    from
        authors a1_0
    where
        a1_0.name=?
Hibernate:
    select
        b1_0.author_id,
        b1_1.id,
        b1_1.title
    from
        book_authors b1_0
    join
        books b1_1
            on b1_1.id=b1_0.book_id
    where
        b1_0.author_id=?
2024-04-23T20:21:02.216+02:00  INFO 19956 --- [nio-8080-exec-6]
be.pxl.service.BookService      : Author retrieved...
```

Always be very carefull with bi-directional relationships and eager fetching. The performance cost may be underestimated. In this case, if an author has written many books, it may be beneficial, to only maintain the owning part of the relationship and search for in author's books with a query in the BookRepository.

6.8 Transactions

The **@Transactional** annotation in Spring Boot is used to declare that a method or a class should be executed within a transactional context.

A transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction Management ensures that an application maintains data integrity and consistency in scenarios involving multiple transaction operations.

Newly created entities should always be saved to the database, however changes to managed entities are persisted automatically when the transaction is committed.

6.8.1 Usage of @Transactional

When placed above a method, @Transactional ensures the method executes within a transaction. If a transaction is already running, the method by default runs within that transaction.

If @Transactional is placed at the class level, it applies to all the public methods of that class.

Transaction propagation behaviour defines how transactions relate to each other. Common propagation types include:

- REQUIRED: Use the current transaction or start a new one if none exists.
- REQUIRES_NEW: Always start a new transaction, suspending the current one if it exists.
- SUPPORTS: Run within the current transaction if it exists; otherwise, run non-transactionally.
- NEVER: Ensure no current transaction exists, throwing an exception if one exists.

Suppose a method annotated with @Transactional is called in a Spring Boot application. The transaction manager checks if a current transaction exists. Depending on the propagation setting:

- The current transaction is used, or a new transaction is created.
- The current transaction is suspended, and a new transaction is created.

Then the method executes and database operations are performed as part of the current transaction. If the method completes successfully, the transaction is committed. If an exception occurs, the transaction is rolled back.

Let's consider an example with Book and Author entities. We want to ensure that adding a book linked to an author is transactional (either both the book and the author are saved, or neither).

```java
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    // constructors, getters, and setters
}

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    @ManyToOne
    private Author author;
    // constructors, getters, and setters
}

@Service
public class BookService {
    @Autowired
    private BookRepository bookRepository;
    @Autowired
    private AuthorRepository authorRepository;
    @Transactional
    public void addBookAndAuthor(Book book, Author author) {
        authorRepository.save(author); // Save author
        book.setAuthor(author);
        bookRepository.save(book); // Save book
        // Both saves are part of the same transaction
    }
}
```

6.8.2 Transaction propagation

In the context of transaction management, transaction propagation behaviours define how transactions are handled when one transactional method calls another. The common types of transaction propagation behaviours with are:

- **REQUIRED** - Use the existing transaction if available; start a new one if none exists. This is the default value.
- **SUPPORTS** - Execute within the existing transaction if one is already present; execute non-transactionally if there is no transaction.
- **MANDATORY** - Require an existing transaction; throw an exception if there is no existing transaction.
- **REQUIRES_NEW** - Suspend the current transaction if one exists, and start a new transaction.
- **NOT_SUPPORTED** - Suspend the existing transaction and execute non-transactionally.
- **NEVER** - Execute non-transactionally; throw an exception if there is an existing transaction.

We will discuss these transaction propagation behaviours with example code.

```
public record JoinOrganisationDto(String organisation, String employee) {
}

@RestController
@RequestMapping("/organisation")
public class OrganisationController {

    private final OrganisationService organisationService;

    public OrganisationController(OrganisationService organisationService) {
        this.organisationService = organisationService;
    }

    @PostMapping("/join")
    public void joinOrganisation(@RequestBody JoinOrganisationDto
        ↪ joinOrganisationDto) {
```

```
        organisationService.joinOrganisation(joinOrganisationDto.organisation(),
            ↪ joinOrganisationDto.employee());
    }
}

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    private Organisation organisation;


    private String name;

    public String getName() {
        return name;
    }

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

    public void setOrganisation(Organisation organisation) {
        this.organisation = organisation;
    }
}

@Entity
public class Organisation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

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

    public String getName() {
        return name;
    }
}

@Service
public class OrganisationService {
    private final OrganizationRepository organizationRepository;
    private final EmployeeService employeeService;


    public OrganisationService(OrganizationRepository
        ↪ organizationRepository, EmployeeService employeeService) {
        this.organizationRepository = organizationRepository;
        this.employeeService = employeeService;
    }

    public void joinOrganisation(String organisationName, String
        ↪ employeeName) {
        Organisation organisation =
            ↪ organizationRepository.findByName(organisationName).orElseThrow(()
            ↪ -> new NotFoundException("Not found"));
        Employee employee = employeeService.insertEmployee(employeeName);
        employee.setOrganisation(organisation);
    }
}

@Service
public class EmployeeService {

    public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
    }
}

public interface OrganizationRepository extends
    ↪ JpaRepository<Organisation, Long> {
    Optional<Organisation> findByName(String name);
}
```

```
POST http://localhost:8080/organisations/join
{
"organisation" : "Happy Organisation",
"employee": "Tamara"
}
```

The employee data is persisted, but when something goes wrong in the method joinOrganisation, we end up with inconsistent data. The employee is saved without an organisation.

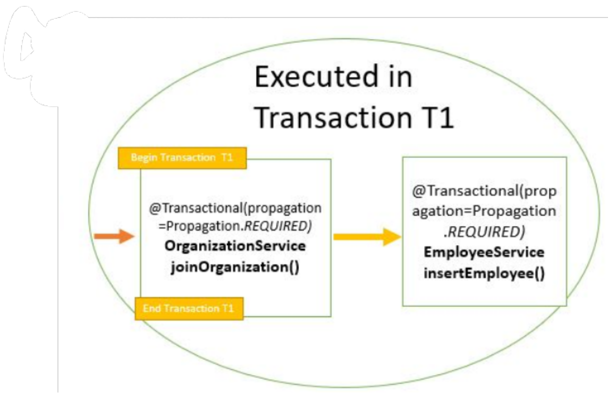
```
 public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
}

public void joinOrganisation(String organisationName, String employeeName)
    ↪ {
        Organisation organisation =
            ↪ organizationRepository.findByName(organisationName).orElseThrow(()
            ↪ -> new NotFoundException("Not found"));
        Employee employee = employeeService.insertEmployee(employeeName);
        employee.setOrganisation(organisation);
        throw new RuntimeException("Intentional error");
}
```

Propagation.REQUIRED

Let's add transaction management and discuss transaction propagation. When you use the annotation @Transactional, the default value for transaction propagation is REQUIRED: @Transactional(propagation = Propagation.REQUIRED).

```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
    ↪ {
        Organisation organisation =
            ↪ organizationRepository.findByName(organisationName).orElseThrow(()
            ↪ -> new NotFoundException("Not found"));
        Employee employee = employeeService.insertEmployee(employeeName);
        employee.setOrganisation(organisation);
        //throw new RuntimeException("Intentional error");
}

@Transactional(propagation = Propagation.REQUIRED)
 public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
}
```

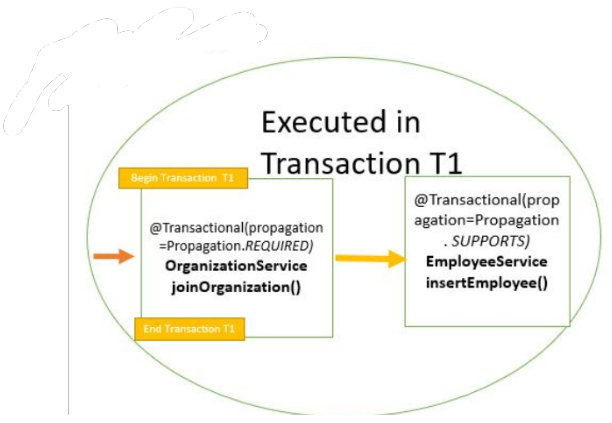
When calling joinOrganisation a new transaction is created. This transaction is used when calling the method insertEmployee. This results in persisting an employee with his/her organisation. If an error occurs, the transaction is rolled back, and the employee is not persisted in the database.



Propagation.SUPPORTS

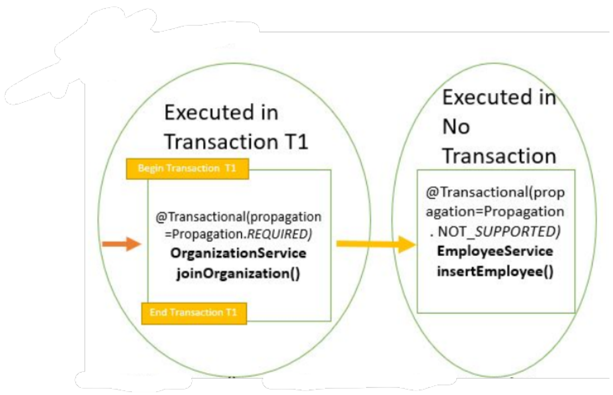
```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
   ↪ {
       Organisation organisation =
          ↪ organizationRepository.findByName(organisationName).orElseThrow(()
          ↪ -> new NotFoundException("Not found"));
       Employee employee = employeeService.insertEmployee(employeeName);
       employee.setOrganisation(organisation);
       //throw new RuntimeException("Intentional error");
}

@Transactional(propagation = Propagation.SUPPORTS)
 public Employee insertEmployee(String name) {
       Employee employee = new Employee();
       employee.setName(name);
       employee = employeeRepository.save(employee);
       return employee;
}
```

The method insertEmployee uses the transaction that is created when calling the joinOrganisation method. However, if the insertEmployee is called directly from the controller, no transaction will be created.



Propagation.NOT_SUPPORTED

```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
   ↪ {
       Organisation organisation =
          ↪ organizationRepository.findByName(organisationName).orElseThrow(()
          ↪ -> new NotFoundException("Not found"));
       Employee employee = employeeService.insertEmployee(employeeName);
       employee.setOrganisation(organisation);
       //throw new RuntimeException("Intentional error");
}

@Transactional(propagation = Propagation.NOT_SUPPORTED)
 public Employee insertEmployee(String name) {
       Employee employee = new Employee();
       employee.setName(name);
       employee = employeeRepository.save(employee);
       return employee;
}
```

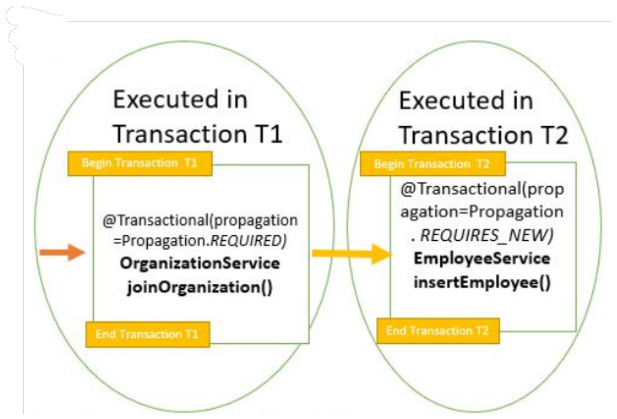


The method insertEmployee runs without transaction. It doesn't use the existing transaction, nor creates a new transaction. The employee is persisted in the database, without a link to its organisation.

Propagation.REQUIRES_NEW

```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
   ↪ {
       Organisation organisation =
          ↪ organizationRepository.findByName(organisationName).orElseThrow(()
          ↪ -> new NotFoundException("Not found"));
       Employee employee = employeeService.insertEmployee(employeeName);
       employee.setOrganisation(organisation);
       //throw new RuntimeException("Intentional error");
}
```

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
 public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
}
```


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.REQUIRED)
OrganizationService
joinOrganization()
End Transaction T1
Executed in Transaction T2
Begin Transaction T2
@Transactional(propagation=Propagation.REQUIRES_NEW)
EmployeeService
insertEmployee()
End Transaction T2

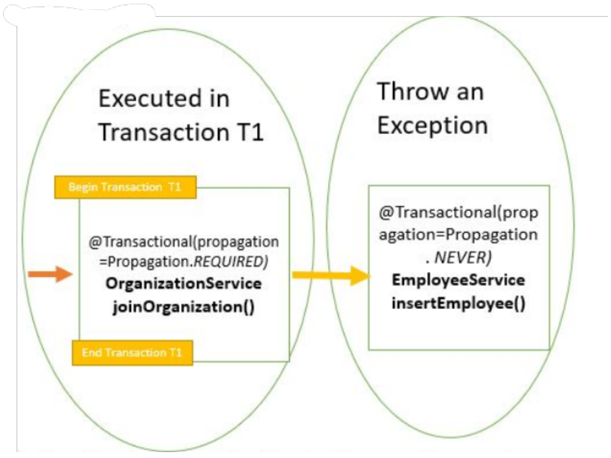
The method insertEmployee runs in its own, newly created transaction. If something goes wrong in the method joinOrganisation, only the sql statements from its own transaction are rolled back. Persisting the employee was already committed and will therefore not rollback.

Propagation.NEVER

```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
    ↪ {
        Organisation organisation =
            ↪ organizationRepository.findByName(organisationName).orElseThrow(()
            ↪ -> new NotFoundException("Not found"));
        Employee employee = employeeService.insertEmployee(employeeName);
        employee.setOrganisation(organisation);
        //throw new RuntimeException("Intentional error");
}

@Transactional(propagation = Propagation.NEVER)
 public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
}
```

The method insertEmployee never runs in a transaction. If the calling method runs in a transaction, an exception will occur.

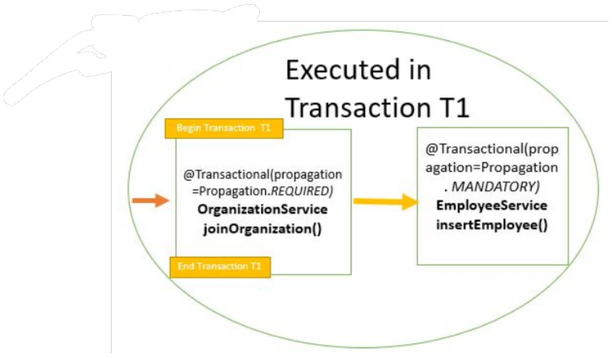
```
org.springframework.transaction.IllegalTransactionStateException: Existing transaction fo
```



Propagation.MANDATORY

```
@Transactional(propagation = Propagation.REQUIRED)
public void joinOrganisation(String organisationName, String employeeName)
   ↪ {
        Organisation organisation =
           ↪ organizationRepository.findByName(organisationName).orElseThrow(()
           ↪ -> new NotFoundException("Not found"));
        Employee employee = employeeService.insertEmployee(employeeName);
        employee.setOrganisation(organisation);
        //throw new RuntimeException("Intentional error");
}

@Transactional(propagation = Propagation.MANDATORY)
 public Employee insertEmployee(String name) {
        Employee employee = new Employee();
        employee.setName(name);
        employee = employeeRepository.save(employee);
        return employee;
}
```

The method insertEmployee uses the existing transaction.


Executed in Transaction T1
Begin Transaction T1
@Transactional(propagation =Propagation.REQUIRED)
OrganizationService
joinOrganization()
End Transaction T1
@Transactional(propagation=Propagation. MANDATORY)
EmployeeService
insertEmployee()

If there is no current transaction that can be used, an error will be thrown.

```
org.springframework.transaction.IllegalTransactionStateException: No existing transactio
```

6.9 Queries

Basic CRUD queries are in Spring Data JPA available in the repositories. But there are multiple ways of creating custom queries in a Spring boot application. Let's discuss the different types of queries.

6.9.1 Query methods

Spring Data JPA can create queries based on method names. You can use keywords like *findBy*, *findAllBy*, *findBy...And...*, ...

An overview of the keywords can be found in spring documentation https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html.

```java
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Query method with parameters for finding products by name and price
    List<Product> findByNameAndPrice(String name, double price);
}
```

6.9.2 JPQL queries

JPQL, or Java Persistence Query Language, is a query language designed to abstract database details from the developer, allowing queries to be written based on entity model classes rather than on database tables.

```java
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.age >= :minAge")
    List<User> findByAgeGreaterThan(@Param("minAge") int minAge);
}
```

6.9.3 Native SQL queries

```java
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(value = "SELECT * FROM users WHERE age >= :minAge", nativeQuery
        ↪ = true)
    List<User> findByAgeGreaterThanOrEqualNative(@Param("minAge") int
        ↪ minAge);
}
```

6.10 Unit testing a repository

Spring Data JPA offers an annotation @DataJpaTest which makes repository testing possible with a minimum of configuration.

Add the h2 dependency with scope test in your pom.xml. This way the unit test for your repository will use the h2 database to test your queries.

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

The builder pattern is used in this example to create entity objects for testing purpose. These entity objects are stored in the in-memory database. There is an IntelliJ IDEA plugin that generates the builder-classes for you (Generate Builder plugin).

```
public final class AuthorBuilder {
    private String name;
    private Set<Book> books;

    private AuthorBuilder() {
    }

    public static AuthorBuilder anAuthor() {
        return new AuthorBuilder();
    }

    public AuthorBuilder withName(String name) {
        this.name = name;
        return this;
    }

    public AuthorBuilder withBooks(Set<Book> books) {
        this.books = books;
        return this;
    }

    public Author build() {
        Author author = new Author();
        author.setName(name);
        author.setBooks(books);
        return author;
    }
}
```

The @DataJpaTest annotation is used to test JPA repositories in Spring Boot applications. It's a specialized test annotation that provides a minimal Spring context for testing the persistence layer.

By default, each test method annotated with @DataJpaTest runs within a transactional boundary. This ensures that changes made to the database are automatically rolled back at the end of the test, leaving a clean slate for the next test.

The test class uses the entity managers, which is injected in the test with the @PersistenceContext annotation. The flush() forces to synchronize the persistence context to the database. The clear() empties the persistence context. Therefore, all entities are detached and can be fetched from the database.

```
package be.pxl.repository;

import be.pxl.builder.AuthorBuilder;
import be.pxl.domain.Author;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import java.util.Arrays;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

@DataJpaTest
public class AuthorRepositoryTest {

    @Autowired
    private TestEntityManager testEntityManager;

    @Autowired
    private AuthorRepository authorRepository;

    private final Author author1 =
       ↪ AuthorBuilder.anAuthor().withName("Famous Author").build();


    private final Author author2 = AuthorBuilder.anAuthor().withName("Not
       ↪ So Famous Author").build();

    @BeforeEach
    public void init() {
        authorRepository.saveAll(Arrays.asList(author1, author2));
        testEntityManager.flush();
        testEntityManager.clear();
    }

    @Test
    public void returnsAuthorByName() {
        Optional<Author> author = authorRepository.findAuthorByName("Famous
            ↪ Author");

        assertTrue(author.isPresent());
        assertEquals("Famous Author", author.get().getName());
    }

    @Test
    public void returnsEmptyOptionalWhenAuthorNotFoundByName() {
        Optional<Author> author =
            ↪ authorRepository.findAuthorByName("Bestseller Author");

        assertTrue(author.isEmpty());
    }
}
```

6.11 Pagination

Pageable is an interface provided by Spring Data that contains information about the pagination and sorting of data. It helps in structuring the way data is retrieved from the database in manageable chunks or pages, rather than pulling large quantities of data all at once, which can be inefficient and slow.

```
@RestController
@RequestMapping("books")
public class BookController {

    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("search")
    public List<BookDto> searchBooks(@RequestBody FilterDto filter) {
        return bookService.searchBooks(filter);
    }

    @GetMapping
    public Page<BookDto> getBooks(Pageable pageable) {
        return bookService.findAllBooks(pageable);
    }
}
```

The parameter Pageable is automatically populated by Spring when the method is called. The Pageable object typically includes:

- page: The current page number (0-indexed, where 0 is the first page).

- size: The number of records per page.
- sort: Criteria for sorting the data (optional).

For example, a request URL might look like this: */books?page=0&size=10&sort=title,asc*. This tells the application to get the first page of books, with 10 books per page, sorted by title in ascending order.

```
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

```
@Service
public class BookService {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(BookService.class);

    private final AuthorRepository authorRepository;
    private final BookRepository bookRepository;

    public BookService(AuthorRepository authorRepository, BookRepository
        ↪ bookRepository) {
        this.authorRepository = authorRepository;
        this.bookRepository = bookRepository;
    }

    public Page<BookDto> findAllBooks(Pageable pageable) {
        Page<Book> books = bookRepository.findAll(pageable);
        return books.map(this::mapToBookDto);
    }


    private BookDto mapToBookDto(Book book) {
        return new BookDto(book.getTitle(), book.getCategory(),
            ↪ book.getAuthors().stream().map(Author::getName).toList());
    }
}
```

The Pageable parameter is passed to the BookService's method findAllBooks. This method is expected to interact with the repository layer that extends Spring Data's PagingAndSortingRepository or JpaRepository. These repositories natively support Pageable for pagination and sorting. The repository method uses the Pageable object to construct a database query that fetches only the specified slice of data based on the page number, page size, and sorting criteria.

The method returns a Page<BookDto>, which is a specialized implementation of the Slice interface. It provides not only the list of data for the current page but also additional information about the total number of pages, the total number of elements, whether there are more pages available, and so forth.

Always use DTOs in your REST controllers rather than exposing JPA entities directly. This helps avoid lazy-loading problems and keeps your API decoupled from internal data structures.

```
{
"content": [
{
"title": "Whispers of the Ancient",
"category": "HISTORY",
"authors": [
"Tessa Fairwind",
"Elara Thornwood"
]
},
{
"title": "The Last Ember",
"category": "DETECTIVE",
"authors": [
"Milo Ventris"
]
},
{
"title": "Beneath the Starless Sky",
"category": "NON-FICTION",
"authors": [
"Tessa Fairwind"
]
},
{
"title": "Echoes of the Forgotten",
"category": "ROMANTIC",
"authors": [
"Quentin Ashlore"
]
},
{
"title": "The Glass Fortress",
"category": "HISTORY",
"authors": [
"Nora Spellbound",
"Quentin Ashlore"
]
}
],
"pageable": {
"pageNumber": 0,
"pageSize": 5,
"sort": {
"empty": true,
"sorted": false,
"unsorted": true
},
"offset": 0,
"paged": true,
"unpaged": false
},
"last": false,
"totalPages": 2,
"totalElements": 10,
"size": 5,
"number": 0,
"sort": {
"empty": true,
"sorted": false,
"unsorted": true
},
"numberOfElements": 5,
"first": true,
"empty": false
}
```

Only the requested slice of data is fetched from the database, which can be crucial for performance when dealing with large datasets. Paging and sorting offers flexibility: clients of your API can specify how many records they want per page and how the data should be sorted. Spring Data handles much of the heavy lifting, it's easy to implement robust pagination and sorting without much boilerplate code.

6.12 Searching and filtering data

The JpaSpecificationExecutor interface in Spring Data JPA is an extension used to allow the execution of database queries using the Specification criteria API, which adds a layer of abstraction over string-based criteria queries. This interface is particularly useful for building complex or dynamic queries based on conditions that are determined at runtime.

```
public interface BookRepository extends JpaRepository<Book, Long>,
    ↪ JpaSpecificationExecutor<Book> {
}
```

```
public record FilterDto(String authorName, BookCategory category, String
    ↪ titlePart) {
}
```

```
@RestController
@RequestMapping("books")
public class BookController {
```

```java

    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("search")
    public List<BookDto> searchBooks(@RequestBody FilterDto filter) {
        return bookService.searchBooks(filter);
    }
}
```

```java
@Service
public class BookService {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(BookService.class);

    private final AuthorRepository authorRepository;
    private final BookRepository bookRepository;

    public BookService(AuthorRepository authorRepository, BookRepository
        ↪ bookRepository) {
        this.authorRepository = authorRepository;
        this.bookRepository = bookRepository;
    }


    public List<BookDto> searchBooks(FilterDto filter) {
        return bookRepository.findAll(createBookSpecification(filter))
                .stream()
                .map(this::mapToBookDto)
                .toList();
    }

    private Specification<Book> createBookSpecification(FilterDto filter) {
        return new Specification<Book>() {
            @Override
            public Predicate toPredicate(Root<Book> root, CriteriaQuery<?>
                ↪ query, CriteriaBuilder builder) {
                List<Predicate> allPredicates = new ArrayList<>();
                if (filter.titlePart() != null) {
                    allPredicates.add(builder.like(builder.lower(root.get("title")),
                        ↪ "%" + filter.titlePart().toLowerCase() + "%"));
                }
                if (filter.category() != null) {
                    allPredicates.add(builder.equal(root.get("category"),
                        ↪ filter.category()));
                }

                if (filter.authorName() != null) {
                    Join<Book, Author> authors = root.join("authors",
                        ↪ JoinType.LEFT);
                    allPredicates.add(builder.like(builder.lower(authors.get("name")),
                        ↪ "%" + filter.authorName().toLowerCase() + "%"));
                }

                query.distinct(true);

                return builder.and(allPredicates.toArray(new Predicate[0]));
            }
        };
    }

    private BookDto mapToBookDto(Book book) {
        return new BookDto(book.getTitle(), book.getCategory(),
            ↪ book.getAuthors().stream().map(Author::getName).toList());
    }
}
```

The method createBookSpecification is an example of how to implement a custom Specification<Book> using Spring Data JPA's Criteria API. This method constructs a dynamic query specification based on filtering criteria encapsulated within a FilterDto object. The specification is used to generate a query that can fetch Book entities from the database according to the provided filter parameters.

The URL *http://localhost:8080/books/search* can be called with different filters.

```
{
"category": "HISTORY",
"titlePart": "ro"
}
```

The above filter will retrieve all the books with category HISTORY that contain ro in the title.

```
[
{
"title": "Ironheart",
"category": "HISTORY",
"authors": [
"Tessa Fairwind"
]
},
{
"title": "The Crown of Thorns",
"category": "HISTORY",
"authors": [
"Quentin Ashlore"
]
}
]
```

6.13 N + 1 query problem

Suppose we have the entity-class Post and the entity-class PostComment. There is a unidirectional many-to-one relationship between PostComment and Post. A PostComment belongs to exactly one Post however for one Post there may exist multiple PostComments.

```java
package be.pxl.demo.domain;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@Entity
public class Post {
    @Id
    @GeneratedValue
    private Long id;
    private String title;

    public Post() {
        // JPA only
    }

    public Post(String title) {
        this.title = title;
    }

    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }
}
```

```java
package be.pxl.demo.domain;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;

@Entity
public class PostComment {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    private Post post;
    private String review;

    public PostComment() {
        // JPA only
    }

    public PostComment(Post post, String review) {
        this.post = post;
        this.review = review;
    }

    public Long getId() {
        return id;
    }

    public Post getPost() {
        return post;
    }

    public String getReview() {
        return review;
    }
}
```

We have a JpaRepository for both entity-classes. The PostService implements a method to create a post, a method for creating comments, and a method for retrieving all comments from the database.

```java
package be.pxl.demo.service;

import be.pxl.demo.api.dto.PostCommentDTO;
import be.pxl.demo.domain.Post;
import be.pxl.demo.domain.PostComment;
import be.pxl.demo.exception.ResourceNotFoundException;
import be.pxl.demo.repository.PostCommentRepository;
import be.pxl.demo.repository.PostRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PostService {

    private final PostCommentRepository postCommentRepository;
    private final PostRepository postRepository;

    public PostService(PostCommentRepository postCommentRepository,
                       PostRepository postRepository) {
        this.postCommentRepository = postCommentRepository;
        this.postRepository = postRepository;
    }

    public long createPost(String title) {
        return postRepository.save(new Post(title)).getId();
    }

    public void createPostComment(long postId, String review) {
        Post post = postRepository.findById(postId).orElseThrow(() -> new
            ↪ ResourceNotFoundException("Post", "id", postId));
        PostComment postComment = new PostComment(post, review);
        postCommentRepository.save(postComment);

    }

    public List<PostCommentDTO> findAll() {
        List<PostComment> allComments = postCommentRepository.findAll();
        return allComments.stream().map(pc -> new
            ↪ PostCommentDTO(pc.getPost().getTitle(),
            ↪ pc.getReview())).toList();
    }
}
```

Then we have the PostController where we implement two endpoints. One endpoint for generating testdata and another endpoint for retrieving all the post comments.

```
package be.pxl.demo.api;

import be.pxl.demo.api.dto.PostCommentDTO;
import be.pxl.demo.service.PostService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
```

```java
@RequestMapping("posts")
public class PostController {

    private final PostService postService;

    public PostController(PostService postService) {
        this.postService = postService;
    }

    @GetMapping("init")
    public void init() {
        long post1Id = postService.createPost("PXL'er Ward Lemmelijn kroont
            ↪ zich tot wereldkampioen indoorroeien.");
        long post2Id = postService.createPost("PXL naar finale in
            ↪ Cybersecurity challenge.");
        long post3Id = postService.createPost("Luister naar PXLRadio!!");

        postService.createPostComment(post1Id, "Schitterend ***");
        postService.createPostComment(post1Id, "Proficiat!");
        postService.createPostComment(post2Id, "Ik ben zeker van de
            ↪ partij!");
        postService.createPostComment(post2Id, "Ik hou van uitdagingen!");
        postService.createPostComment(post3Id, "Leuke muziek!");
        postService.createPostComment(post3Id, "Toffe radiozender!");
        postService.createPostComment(post3Id, "Zet die plaat af.");
    }

    @GetMapping
    public List<PostCommentDTO> getAllComments() {
        return postService.findAll();
    }
}
```

In application.properties file make sure to enable show-sql.

```
spring.jpa.show-sql=true
# Following property can be used to format the sql shown
# spring.jpa.properties.hibernate.format_sql=true
```

After we call the endpoint to generate testdata, we retrieve all comments from the database. In the log-files you will find the following log messages.

```
Hibernate: select p1_0.id,p1_0.post_id,p1_0.review from post_comment p1_0
Hibernate: select p1_0.id,p1_0.title from post p1_0 where p1_0.id=?
Hibernate: select p1_0.id,p1_0.title from post p1_0 where p1_0.id=?
Hibernate: select p1_0.id,p1_0.title from post p1_0 where p1_0.id=?
```

After retrieving the comments from the database, every post that's involved in the comments is retrieved as well. So, if you have N posts involved, N+1-queries will be executed. That's not efficient!

You should tackle this problem by re-writing the query to retrieve all post comments.

```java
package be.pxl.demo.repository;

import be.pxl.demo.domain.PostComment;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface PostCommentRepository extends JpaRepository<PostComment,
   ↪ Long> {

    @Override
    @EntityGraph(
            type = EntityGraph.EntityGraphType.FETCH,
            attributePaths = {
                    "post"
            }
    )
    List<PostComment> findAll();
}
```

When retrieving comments using PostCommentRepository.findAll(), the @EntityGraph annotation causes Spring Data JPA and Hibernate to fetch the associated entities included in attributePaths.

Only one query is executed.

```
Hibernate: select p1_0.id,p2_0.id,p2_0.title,p1_0.review from post_comment
   ↪ p1_0 left join post p2_0 on p2_0.id=p1_0.post_id
```

Being able to see what Hibernate is actually doing with the database is very important. It's good practice to enable SQL output when working on a Spring Boot project, just as a sanity check. You will definitely find problems you were unaware of by examining the SQL output.

An extended example of the N+1 query problem and the solution can be found in this post: https://tech.asimio.net/2020/11/06/Preventing-N-plus-1-select-problem-using-Spring-Data
html.

Chapter 7

Spring security

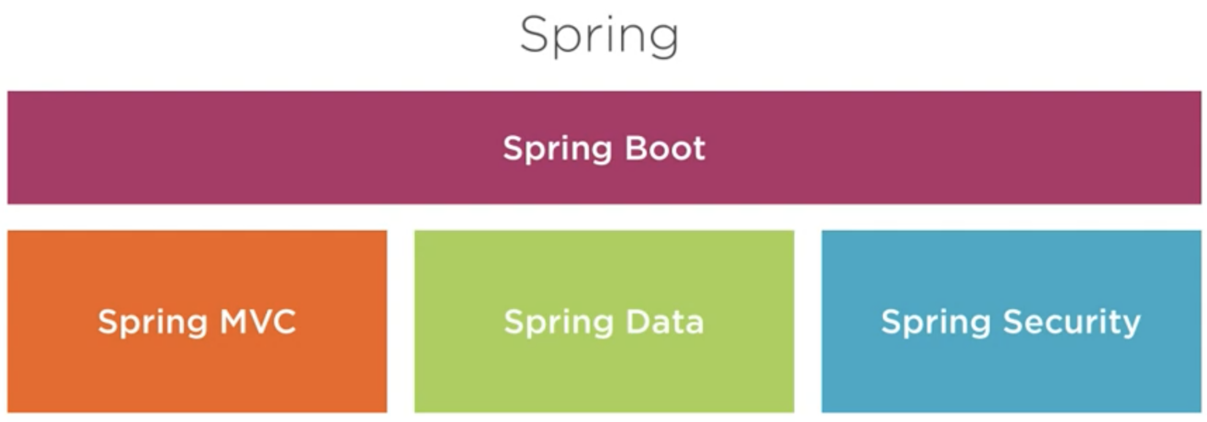
WARNING: this chapter is currently being updated.

Learning goals
The junior-colleague

1. Understand JWT and its role in authentication
2. Can configure stateless authentication with Spring Security
3. Can implement and test method-level security with annotations like @PreAuthorize
4. Can test secured endpoints using MockMvc and SpringBootTest annotations

Source for this chapter:
https://github.com/custersnele/spring_security_jwt_example

7.1 What is spring security



In today's digital landscape, securing web applications and protecting sensitive data is more critical than ever. Spring Security is a comprehensive framework built on the Spring ecosystem that provides a wide range of security services in a developer-friendly and flexible manner. Although it is widely used to secure web applications, its modular design also allows it to be integrated into stand-alone applications.

Key Features and Benefits

Spring Security empowers developers with:

- **Advanced Web Security:** Safeguard your applications against common vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking.
- **Resource and URL Protection:** Restrict access to specific endpoints and enforce secure communications (e.g., HTTPS) to protect critical resources.
- **Role-Based Access Control:** Easily manage and enforce permissions through role-based authentication and authorization.
- **Flexible Authentication Mechanisms:** Seamlessly integrate various authentication strategies—whether it's JWT, OAuth2, Basic Authentication, or session-based authentication—to meet your application's requirements.
- **Domain Model Security:** Secure your business data by controlling access to domain objects, ensuring that only authorized users can perform sensitive operations.

Integration with External Identity Providers

While Spring Security allows you to handle authentication within your application (for example, by generating and managing JWT tokens internally), it is often beneficial to delegate authentication to external identity providers in production environments. These providers offer:

- **Centralized User Management:** Streamlined administration and enhanced security features such as single sign-on (SSO) and multi-factor authentication (MFA).
- **Robust Security and Compliance:** Built-in support for security standards and protocols that reduce the risk of vulnerabilities.

Some commonly used external authentication providers include:

- **Keycloak:** An open-source identity and access management solution by Red Hat, supporting OAuth2, OpenID Connect, and SAML.
- **AWS Cognito:** A cloud-based service designed for handling authentication, authorization, and federation with both social and enterprise identity providers.
- **Okta:** A cloud-based identity management provider known for its adaptive MFA and comprehensive API security.
- **Auth0:** A developer-friendly platform that offers extensive customization options for OAuth2, OpenID Connect, and SAML-based authentication.

In this chapter, we will focus on implementing JWT token generation and management with Spring Security, exploring how these concepts can be applied to build a secure, stateless authentication mechanism. Working samples are available at `https://github.com/spring-projects/spring-security-samples`.

7.2 JWT Authentication

In this section, we will setup a secure JWT-based authentication system using Spring Boot 3 with Spring Security 6.

JWT (JSON Web Token) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs are commonly used for authentication and authorization in web applications and APIs.

JWTs are digitally signed, not encrypted. The information they contain can be decoded by anyone with access to the token, but the signature ensures its integrity. Storing sensitive data in the payload is not recommended, as the payload can be easily decoded.

To prevent tampering, it's important to use strong and secure algorithms for signing the tokens. Secrets used for signing should be kept secret.

Environment Setup

Begin by creating a PostgreSQL database using Docker Compose.

```
services:
  spring-security-db:
    image: postgres
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: demo
      POSTGRES_DB: demo
    ports:
      - "5432:5432"
```

Project Dependencies

Create a new Spring Boot project with the following dependencies:

- Spring Security
- Spring Web
- PostgreSQL driver
- Spring Data JPA

Next, add the needed dependencies in pom.xml files. The following api's from io.jsonwebtoken are used: *jjwt-api* defines how JWTs work, *jjwt-impl* provides the actual implementation, and *jjwt-jackson* ensures correct JSON handling, making it easier to work with JWT payloads.

```
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-api</artifactId>
            <version>0.12.6</version>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-impl</artifactId>
            <version>0.12.6</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-jackson</artifactId>
            <version>0.12.6</version>
            <scope>runtime</scope>
        </dependency>
```

Application Configuration

Next, configure the `application.properties` file to set up the database connection, JPA settings, server configuration, and JWT properties:

```
spring.application.name=SpringSecurityExample
## DB Configuration ##
spring.datasource.url=jdbc:postgresql://localhost:5432/demo
spring.datasource.username=demo
spring.datasource.password=demo
## JPA Configuration ##
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.generate-ddl=true
spring.jpa.open-in-view=false
spring.datasource.hikari.auto-commit=false
spring.jpa.properties.hibernate.connection.provider_disables_autocommit=true
## Server Configuration ##
server.servlet.context-path=/api
server.port=8080
## JWT Configuration (demo values)
# Access token validity: 2 minutes
jwt.jwtExpirationTime=120000
# Refresh token validity: 5 minutes
jwt.jwtRefreshTokenExpirationTime=300000
app.admin.password=ElfLordsOnly!42
```

REST Endpoints Overview

The sample application exposes the following REST endpoints:

Method	URL	Action
POST	/api/auth/register	Registers a new account
POST	/api/auth/authenticate	Authenticates a user and issues tokens.
POST	/api/auth/refresh-token	Exchanges a refresh token for a new access token.
GET	/api/users	Retrieves all users
GET	/api/public/welcome	Serves public content.
GET	/api/dashboard	Displays a user-only dashboard.
GET	/api/admin/hello	Provides admin-only content
PUT	/api/admin/users/{id}/promote	Allows admin to promote a user.

User and Role Models

Define user roles and the user entity. Start with a simple enumeration for roles:

```
package be.pxl.demo.domain;

public enum Role {
    USER, ADMIN
}
```

Next, create the `User` entity. This class implements Spring Security's `UserDetails` interface, ensuring it can be used seamlessly within the security framework:

```
package be.pxl.demo.domain;

import jakarta.persistence.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.io.Serializable;
import java.util.Collection;
import java.util.Collections;
import java.util.Objects;

@Entity
@Table(name = "customUsers")
public class User implements Serializable, UserDetails {
    private static final String ROLE_PREFIX = "ROLE_";
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String email;
    private String password;
```

```java
    @Enumerated(EnumType.STRING)
    private Role role;

    public User(String email, String password, Role role) {
        this.email = email;
        this.password = password;
        this.role = role;
    }

    public User() {
        // JPA only
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singletonList(new
            ↪ SimpleGrantedAuthority(ROLE_PREFIX + role.name()));
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return this.email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
    // Getters and setters are omitted for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return email != null && email.equals(user.email);
    }

    @Override
    public int hashCode() {
        return Objects.hash(email);
    }
}
```

User Repository and Custom User Details Service

Create a repository interface to handle user persistence:

```java
package be.pxl.demo.repository;

import be.pxl.demo.domain.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {

    boolean existsByEmail(String email);
    Optional<User> findByEmail(String email);
}
```

Implement a custom `UserDetailsService` to load user details during authentication:

```java
package be.pxl.demo.security;

import be.pxl.demo.domain.User;
import be.pxl.demo.repository.UserRepository;
import org.springframework.security.core.userdetails.UserDetailsService;
import
    ↪ org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Component;

@Component
public class CustomUserDetailsService implements UserDetailsService {
    private final UserRepository userRepository ;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public User loadUserByUsername(String email) throws
        ↪ UsernameNotFoundException {
        return userRepository.findByEmail(email).orElseThrow(()-> new
            ↪ UsernameNotFoundException("User not found !"));
    }

}
```

7.3 Implementing JWT Token Handling and Security

Now that the basic application and user management are in place, we move on to securing our application using JSON Web Tokens (JWTs). This section explains how JWTs are generated, validated, and integrated into Spring Security. We'll look at key classes that enable stateless authentication, authorization, and role-based access control using tokens.

Overview of the Security Components

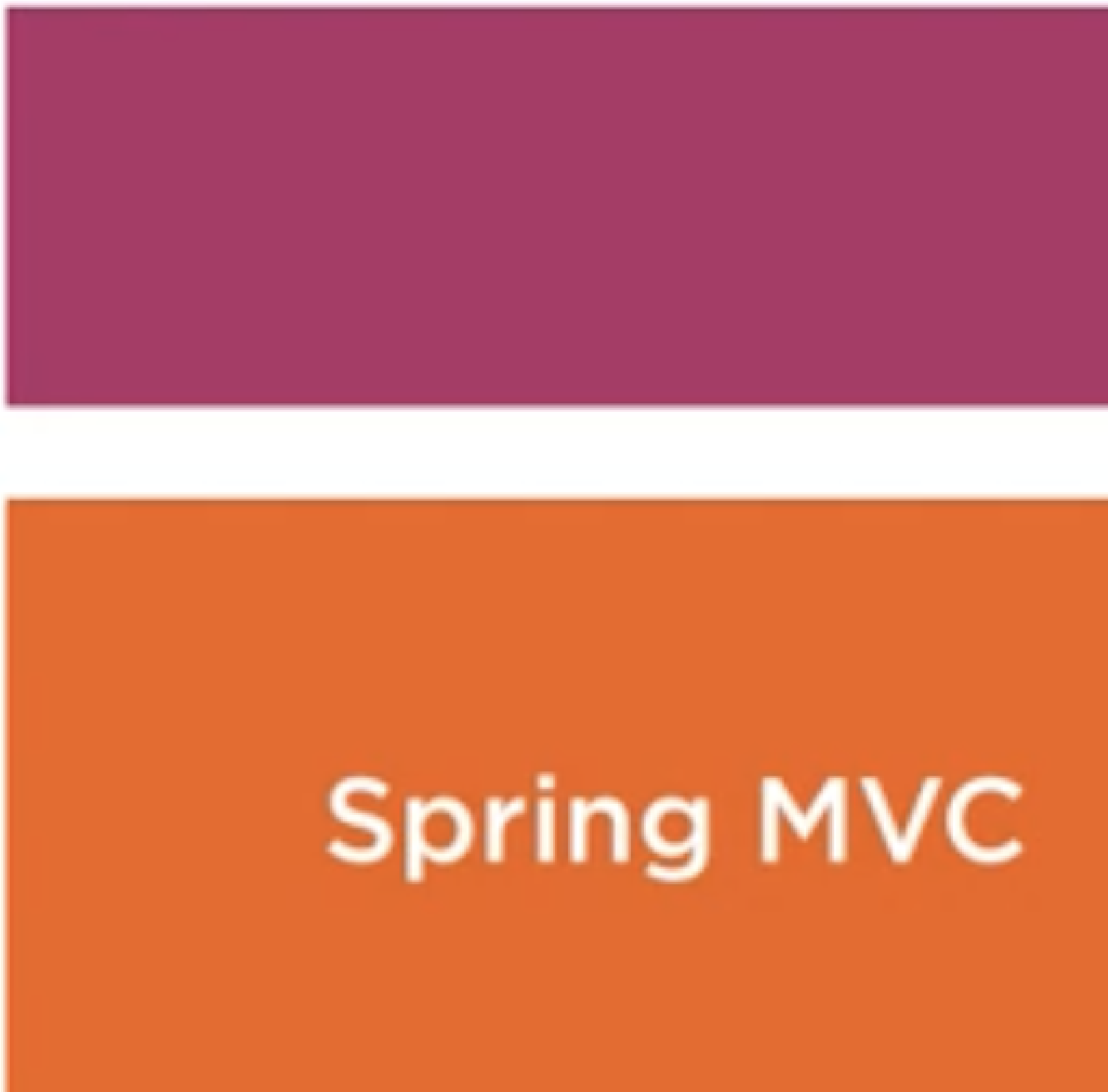
We will discuss the following components.

- **JwtUtilities:** Core JWT helper class.
- **JwtAuthenticationFilter:** Intercepts requests to validate tokens.
- **SpringSecurityConfig:** Sets up Spring Security with JWT support.
- **AuthenticationService:** Encapsulates registration, login, and refresh logic.

JwtUtilities: Token Management

This utility class encapsulates JWT logic to:

- Generate access and refresh tokens.
- Extract claims (like username and token ID).
- Validate token integrity and expiration.
- Retrieve tokens from HTTP headers.


Spring MVC

In this demo, the token signing keys are generated in memory. The class uses *Jwts.SIG.RS256.keyPair().*
to generate a private-public key pair. The private key signs the token (signWith), the public key verifies the signature (verifyWith). In production environments, store keys securely using a keystore, environment variables, or a secrets manager.

The refresh token generation contains a unique JWT ID (jti - Token ID) which will be stored in the databased and used to verify refresh token legitimacy and detect reuse.

The claimsCache caches claims to avoid redundant parsing per request.

```java
package be.pxl.demo.config;

import io.jsonwebtoken.*;
import jakarta.servlet.http.HttpServletRequest;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;

@Component
public class JwtUtilities {
    private static final Logger LOGGER =
       ↪ LogManager.getLogger(JwtUtilities.class);

    @Value("${jwt.jwtExpirationTime}")
    private Long jwtExpirationTime;

    private final KeyPair keyPair = Jwts.SIG.RS256.keyPair().build();
    private final Map<String, Claims> claimsCache = new
       ↪ ConcurrentHashMap<>();

    private PrivateKey getPrivateKey() {
        return keyPair.getPrivate();
    }

    private PublicKey getPublicKey() {
```

```
    return keyPair.getPublic();
}

public String extractUsername(String token) {
    return extractClaim(token, Claims::getSubject);
}

public Claims extractAllClaims(String token) {
    return claimsCache.computeIfAbsent(token, t ->
            Jwts.parser()
                    .verifyWith(getPublicKey())
                    .build()
                    .parseSignedClaims(t)
                    .getPayload()
    );
}

public <T> T extractClaim(String token, Function<Claims, T>
    ↪ claimsResolver) {
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
}

public String generateToken(String email, List<String> roles) {
    return Jwts.builder()
            .subject(email)
            .claim("role", roles)
            .issuedAt(new Date(System.currentTimeMillis()))
            .expiration(Date.from(Instant.now().plus(jwtExpirationTime,
                ↪ ChronoUnit.MILLIS)))
            .signWith(getPrivateKey())
            .compact();
}

public String generateRefreshToken(String email, String jti, Instant
    ↪ expirationDate) {
    return Jwts.builder()
            .subject(email)
            .id(jti)
            .issuedAt(new Date())
            .expiration(Date.from(expirationDate)) // long-lived
            .signWith(getPrivateKey())
            .compact();
}

public void validateToken(String token) {
    try {
        Jwts.parser().verifyWith(getPublicKey()).build().parseSignedClaims(token);
        } catch (SecurityException | MalformedJwtException e) {
            LOGGER.info("Invalid JWT token.", e);
            throw e;
        } catch (ExpiredJwtException e) {
            LOGGER.info("Expired JWT token.", e);
            throw e;
        } catch (UnsupportedJwtException e) {
            LOGGER.info("Unsupported JWT token.", e);
            throw e;
        } catch (IllegalArgumentException e) {
            LOGGER.info("JWT token compact of handler are invalid.", e);
            throw e;
        }
    }

    public UUID extractTokenId(String token) {
        return UUID.fromString(extractClaim(token, Claims::getId)); // ID
            ↪ == jti
    }

    public String getToken(HttpServletRequest httpServletRequest) {
        final String bearerToken =
            ↪ httpServletRequest.getHeader("Authorization");
        if (StringUtils.hasText(bearerToken) &&
            ↪ bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7);
        }
        return null;
    }
}
```

The configuration class **SpringSecurityConfig** is responsible for setting up Spring Security in a Spring Boot application. It:

- Defines security filters
- Configures authorization rules
- Enables method-level security (@EnableMethodSecurity)
- Implements JWT-based authentication (stateless sessions)
- Defines an authentication manager and password encoder

The following annotations are important:

@Configuration	This annotation tells Spring that
@EnableWebSecurity	This annotation enables Spring S
@EnableMethodSecurity	This annotation enables method-

```
package be.pxl.demo.config;

import be.pxl.demo.domain.Role;
import be.pxl.demo.domain.User;
import be.pxl.demo.repository.UserRepository;
import be.pxl.demo.security.CustomUserDetailsService;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.ProviderManager;
import
   ↪ org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
   ↪ org.springframework.security.config.annotation.method.configuration.EnableMethodSe
import
   ↪ org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
   ↪ org.springframework.security.config.annotation.web.configuration.EnableWebSecurit
import
   ↪ org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigu
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
   ↪ org.springframework.security.web.authentication.UsernamePasswordAuthenticationFil

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SpringSecurityConfig {

    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final CustomUserDetailsService customerUserDetailsService;

    @Value("${app.admin.password}")
    private String adminPassword;

    public SpringSecurityConfig(JwtAuthenticationFilter
        ↪ jwtAuthenticationFilter, CustomUserDetailsService
        ↪ customerUserDetailsService) {
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
        this.customerUserDetailsService = customerUserDetailsService;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
        ↪ Exception {
```

```
    http.csrf(AbstractHttpConfigurer::disable)
           .sessionManagement(session ->
              ↪ session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
           .authorizeHttpRequests(auth ->
                  auth.requestMatchers("/public/**", "/users/**",
                     ↪ "auth/**", "/error").permitAll()
                         .requestMatchers("/admin/**").hasRole("ADMIN")
                         .requestMatchers("/dashboard").authenticated());
    http.addFilterBefore(jwtAuthenticationFilter,
       ↪ UsernamePasswordAuthenticationFilter.class);
    http
           .exceptionHandling(customizer -> customizer
                  .authenticationEntryPoint((request, response,
                     ↪ authException) ->
                     ↪ response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                     ↪ "Unauthorized"))
                  .accessDeniedHandler((request, response,
                     ↪ accessDeniedException) ->
                     ↪ response.sendError(HttpServletResponse.SC_FORBIDDEN,
                     ↪ "Access Denied"))
           );
    return http.build();
}

@Bean
public AuthenticationManager authenticationManager() {
    DaoAuthenticationProvider authProvider = new
       ↪ DaoAuthenticationProvider();
    authProvider.setUserDetailsService(customerUserDetailsService);
    authProvider.setPasswordEncoder(passwordEncoder());
    return new ProviderManager(authProvider);
}

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

@Bean
public CommandLineRunner createDefaultAdmin(UserRepository
    ↪ userRepository, PasswordEncoder encoder) {
    return args -> {
        if
           ↪ (userRepository.findByEmail("elrond.doesnt.play@rivendell.biz").isEmpty
           ↪ {
           User admin = new User();
           admin.setEmail("elrond.doesnt.play@rivendell.biz");
           admin.setPassword(encoder.encode(adminPassword));
```

```
                admin.setFirstName("Legitimus");
                admin.setLastName("Elrondson");
                admin.setRole(Role.ADMIN);
                userRepository.save(admin);
            }
        };
    }
}
```

Let's break down the security features of the filter chain:

Feature	Code
CSRF Protection Disabled	http.csrf(Abstr
Stateless Authentication	sessionManage
Public Access	auth.requestMa
Restricted Access	auth..requestM
Admin Access Restriction	auth.requestMa
JWT Authentication Filter	http.addFilterB
Custom Exception Handling	http.exceptionI

The class SpringSecurityConfig also provides an AuthenticationManager bean that Spring Security can use for authentication.

And finally, BCryptPasswordEncoder is used for hashing passwords. BCrypt is a strong one-way hashing algorithm, making it a secure choice for password storage.

JwtAuthenticationFilter is a Spring Security filter that runs before any secured endpoint is accessed. It performs the following functions:

- Extracts JWT from the request header (Authorization: Bearer <token>).
- Validates the JWT using JwtUtilities.
- Retrieves user details from the token and authenticates the user.
- Handles expired or invalid JWTs by returning appropriate HTTP status codes.

```
package be.pxl.demo.config;

import be.pxl.demo.security.CustomUserDetailsService;
import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.JwtException;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.lang.NonNull;
import
   ↪ org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;
import java.io.PrintWriter;

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private static final Logger LOGGER =
       ↪ LogManager.getLogger(JwtAuthenticationFilter.class);
    private final JwtUtilities jwtUtilities;
    private final CustomUserDetailsService customerUserDetailsService;

    public JwtAuthenticationFilter(JwtUtilities jwtUtilities,
       ↪ CustomUserDetailsService customerUserDetailsService) {
        this.jwtUtilities = jwtUtilities;
        this.customerUserDetailsService = customerUserDetailsService;
    }

    @Override
    protected void doFilterInternal(@NonNull HttpServletRequest request,
       ↪ @NonNull HttpServletResponse response,
                                    @NonNull FilterChain filterChain)
            throws ServletException, IOException {
        String token = jwtUtilities.getToken(request);
        try {
            if (token != null) {
                jwtUtilities.validateToken(token);
                String email = jwtUtilities.extractUsername(token);
                UserDetails userDetails =
                   ↪ customerUserDetailsService.loadUserByUsername(email);
                if (userDetails != null) {
                    UsernamePasswordAuthenticationToken authentication =
                            new
                               ↪ UsernamePasswordAuthenticationToken(userDetails.getUsern
                               ↪ null, userDetails.getAuthorities());
                    LOGGER.info("authenticated user with email :{}", email);
                    SecurityContextHolder.getContext().setAuthentication(authentication);
                }
            }
        } catch (ExpiredJwtException e) { // Handle Expired Token
            sendErrorResponse(response, HttpServletResponse.SC_FORBIDDEN,
               ↪ "Authorization header is expired.");
            return;
        } catch (JwtException e) { // Handle Other JWT Issues
            sendErrorResponse(response, HttpServletResponse.SC_UNAUTHORIZED,
               ↪ "Authorization header is invalid.");
            return;
        }
        filterChain.doFilter(request, response);
    }

    private void sendErrorResponse(HttpServletResponse response, int
        ↪ status, String message) throws IOException {
        response.setStatus(status);
        response.setContentType("application/json");
        PrintWriter writer = response.getWriter();
        writer.write("{\"error\": \"" + message + "\"}");
    }
}
```

The **AuthenticationService** and it's implementation define the logic for registering and authenticating a user. It also handles the functionality for exchanging a refresh token (see later).

```
package be.pxl.demo.service;

import be.pxl.demo.api.dto.LoginDto;
import be.pxl.demo.api.dto.RegisterDto;
import be.pxl.demo.api.dto.TokenDto;

public interface AuthenticationService {
        TokenDto authenticate(LoginDto loginDto);
        void register (RegisterDto registerDto);
        TokenDto refreshToken(String refreshToken);
}
```

```
package be.pxl.demo.service.impl;

import be.pxl.demo.api.dto.LoginDto;
import be.pxl.demo.api.dto.RegisterDto;
import be.pxl.demo.api.dto.TokenDto;
import be.pxl.demo.config.JwtUtilities;
import be.pxl.demo.domain.RefreshToken;
import be.pxl.demo.domain.User;
import be.pxl.demo.exception.DuplicateEmailException;
import be.pxl.demo.exception.RefreshTokenExpiredException;
import be.pxl.demo.exception.RefreshTokenNotFoundException;
import be.pxl.demo.repository.RefreshTokenRepository;
import be.pxl.demo.repository.UserRepository;
import be.pxl.demo.service.AuthenticationService;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.authentication.AuthenticationManager;
import
```

```
    ↪ org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetailsService;
import
    ↪ org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Collections;
import java.util.UUID;

@Service
public class DefaultAuthenticationService implements AuthenticationService
    ↪ {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(DefaultAuthenticationService.class);

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;
    private final JwtUtilities jwtUtilities;
    private final UserDetailsService userDetailsService;
    private final RefreshTokenRepository refreshTokenRepository;

    @Value("${jwt.jwtRefreshTokenExpirationTime}")
    private Long refreshTokenExpirationTime;

    public DefaultAuthenticationService(UserRepository userRepository,
        ↪ PasswordEncoder passwordEncoder, AuthenticationManager
        ↪ authenticationManager, JwtUtilities jwtUtilities,
        ↪ UserDetailsService userDetailsService, RefreshTokenRepository
        ↪ refreshTokenRepository) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
        this.authenticationManager = authenticationManager;
        this.jwtUtilities = jwtUtilities;
        this.userDetailsService = userDetailsService;
        this.refreshTokenRepository = refreshTokenRepository;
    }
```

```
public void register(RegisterDto registerDto) {
    if (userRepository.existsByEmail(registerDto.email())) {
        throw new DuplicateEmailException();
    } else {
        User user = new User();
        user.setEmail(registerDto.email());
        user.setFirstName(registerDto.firstName());
        user.setLastName(registerDto.lastName());
        user.setPassword(passwordEncoder.encode(registerDto.password()));
        user.setRole(registerDto.userRole());
        userRepository.save(user);
    }
}


@Transactional
public TokenDto refreshToken(String refreshToken) {
    jwtUtilities.validateToken(refreshToken);
    String email = jwtUtilities.extractUsername(refreshToken);
    UUID jti = jwtUtilities.extractTokenId(refreshToken);

    User user = (User) userDetailsService.loadUserByUsername(email);

    RefreshToken storedRefreshToken =
        ↪ refreshTokenRepository.findByUuidAndUser(jti,
        ↪ user).orElseThrow(RefreshTokenNotFoundException::new);
    if (storedRefreshToken.getExpirationTime().isBefore(Instant.now()))
        ↪ {
        refreshTokenRepository.delete(storedRefreshToken);
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("The refresh token is expired.");
        }
        throw new RefreshTokenExpiredException();
    }

    refreshTokenRepository.delete(storedRefreshToken);

    String newAccessToken =
        ↪ jwtUtilities.generateToken(user.getUsername(),
            user.getAuthorities().stream()
                    .map(GrantedAuthority::getAuthority)
                    .toList());

    String newRefreshToken = createAndStoreRefreshToken(user);
    return new TokenDto(newAccessToken, newRefreshToken);
}
```

```
    @Transactional
    public TokenDto authenticate(LoginDto loginDto) {
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        loginDto.email(),
                        loginDto.password()
                )
        );
        SecurityContextHolder.getContext().setAuthentication(authentication);
        User user =
            ↪ userRepository.findByEmail(authentication.getName()).orElseThrow(()
            ↪ -> new UsernameNotFoundException("User not found"));
        String token = jwtUtilities.generateToken(user.getUsername(),
            ↪ Collections.singletonList(user.getRole().name()));
        String refreshToken = createAndStoreRefreshToken(user);

        return new TokenDto(token, refreshToken);
    }

    @Transactional(propagation = Propagation.REQUIRED)
    public String createAndStoreRefreshToken(User user) {
        refreshTokenRepository.deleteByUser(user);
        UUID uuid = UUID.randomUUID();
        Instant expirationDate =
            ↪ Instant.now().plus(refreshTokenExpirationTime,
            ↪ ChronoUnit.MILLIS);
        String token =
            ↪ jwtUtilities.generateRefreshToken(user.getUsername(),
            ↪ uuid.toString(), expirationDate);
        RefreshToken refreshToken = new RefreshToken(uuid, token,
            ↪ expirationDate, user);
        refreshTokenRepository.save(refreshToken);
        return token;
    }
}
```

The **UserService** and it's implementation offer a method to retrieve all users. Here the usage is **@PreAuthorize** is used to demonstrate securing an endpoint. @PreAuthorize is a Spring Security annotation used to apply method-level security. It ensures that only users with specific roles or permissions can execute a method. This annotation works with Spring Expression Language (SpEL) to define security constraints.

@PreAuthorize("hasRole('ADMIN')") ensures that only users with the role ROLE_ADMIN can access the findAllUsers method. If a user without the ROLE_ADMIN role tries to invoke findAllUsers, Spring Security will throw an AccessDeniedException.

```
package be.pxl.demo.service;

import be.pxl.demo.api.dto.UserDto;
import org.springframework.security.access.prepost.PreAuthorize;

import java.util.List;

public interface UserService {
    @PreAuthorize("hasRole('ADMIN')")
    List<UserDto> findAllUsers();

    @PreAuthorize("hasRole('ADMIN')")
    void promoteToAdmin(Long id);
}
```

```
package be.pxl.demo.service.impl;

import be.pxl.demo.api.dto.UserDto;
import be.pxl.demo.repository.UserRepository;
import be.pxl.demo.service.UserService;
import be.pxl.demo.service.mapper.UserMapper;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class DefaultUserService implements UserService {

    private final UserMapper userMapper;
    private final UserRepository userRepository;

    public DefaultUserService(UserRepository userRepository, UserMapper
        ↪ userMapper) {
        this.userRepository = userRepository;
        this.userMapper = userMapper;
    }

    @Override
    public List<UserDto> findAllUsers() {
        return
            ↪ userRepository.findAll().stream().map(userMapper::toDto).toList();
    }
}
```

The RestControllers describe the endpoints of the application.

```
package be.pxl.demo.api;

import be.pxl.demo.api.dto.*;
import be.pxl.demo.service.AuthenticationService;
import be.pxl.demo.service.UserService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/auth")
public class AuthenticationController {
    private final UserService userService;
    private final AuthenticationService authenticationService;

    public AuthenticationController(UserService userService,
       ↪ AuthenticationService authenticationService) {
        this.userService = userService;
        this.authenticationService = authenticationService;
    }

    @PostMapping("/register")
    public void register(@RequestBody RegisterDto registerDto) {
        authenticationService.register(registerDto);
    }

    @PostMapping("/refresh-token")
    public TokenDto refreshToken(@RequestBody RefreshTokenRequest request) {
        return authenticationService.refreshToken(request.refreshToken());
    }

    @PostMapping("/authenticate")
    public TokenDto authenticate(@RequestBody LoginDto loginDto) {
        return authenticationService.authenticate(loginDto);
    }
}
```

```
package be.pxl.demo.api;

import be.pxl.demo.api.dto.UserDto;
import be.pxl.demo.service.UserService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;

    public UserController(UserService userService) {
```

```java
        this.userService = userService;
    }

    @GetMapping
    public List<UserDto> findAllUsers() {
        return userService.findAllUsers();
    }
}
```

```java
package be.pxl.demo.api;

import be.pxl.demo.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/admin")
public class AdminController {

    private final UserService userService;

    public AdminController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/hello")
    @PreAuthorize("hasRole('ADMIN')") // Additional layer of security
    public String sayHello(Authentication authentication) {
        return "Welcome " + authentication.getPrincipal() +" you are
            ↪ authenticated as Admin!";
    }

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @PutMapping("/users/{id}/promote")
    public ResponseEntity<?> promoteToAdmin(@PathVariable Long id) {
        userService.promoteToAdmin(id);
        return ResponseEntity.ok().build();
    }
}
```

```java
package be.pxl.demo.api;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/public")
public class PublicController {

    @GetMapping("/welcome")
    public ResponseEntity<String> welcome() {
        return ResponseEntity.ok("Welcome! This is a public content!");
    }
}
```

```java
package be.pxl.demo.api;

import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/dashboard")
public class DashboardController {

    @GetMapping
    public String getAdminDashboard(Authentication authentication) {
        return "Dashboard - Welcome " + authentication.getName();
    }
}
```

7.3.1 Usage

Now it's time to start the application and test all the endpoints. The Authentication-Controller offers an endpoint to create new users. A user's email must be unique. New user's have the role USER.

```
POST http://localhost:8080/api/auth/register
{
  "firstName": "Gandalf",
  "lastName": "TheGrey",
  "email": "admin@gondor.com",
  "password": "youShallNotPass123"
}
```

```
POST http://localhost:8080/api/auth/register
{
  "firstName": "Samwise",
  "lastName": "Gamgee",
  "email": "sam@shire.org",
  "password": "secondBreakfast"
}
```

Only an existing admin can promote users.

```
PUT http://localhost:8080/api/admin/users/3/promote
> Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJzdWIi....CDA1lbWsuvEEpX3hYA
```

Normally, a http status 200 OK is returned after creating a new user, unless the email is already taken, then a 400 Bad Request with error Ëmail already taken.ïs returned.

The public endpoint is accessible without authentication.

```
GET http://localhost:8080/api/public/welcome
```

However, if you try to access the dashboard without authentication, you will receive a 4001 Unauthorized.

```
GET http://localhost:8080/api/dashboard
```

Samwise Gamgee will authenticate and request an access token and refresh token.

```
POST http://localhost:8080/api/auth/authenticate
{

  "email": "sam@shire.org",
  "password": "secondBreakfast"
}
```

When the email and password are valid a 200 OK is returned with the accessToken and refreshToken in the response.

```
{
"accessToken": "eyJhbGciOiJSUzI1NiJ9.eyJzdWIi....CDA1lbWsuvEEpX3hYA",
"refreshToken": "eyJhbGciOiJSUzI1Ni...5c6zBNTqccs1YONXjWUBitXJQ"
}
```

When the email and/or password are not valid a 401 Unauthorized is returned.

For demo, the accessToken is only valid for x minutes, after this time the refreshToken can be used once to retrieve a new accessToken before the refreshToken is expired.

Samwise Gamegee can access the dashboard:

```
GET http://localhost:8080/api/dashboard
> Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJzdWIi....CDA1lbWsuvEEpX3hYA
```

```
200 OK
Dashboard - Welcome sam@shire.org
```

As soon as the accessToken is expired. Samwise Gamegee will see

```
403 Forbidden
{
"error": "Authorization header is expired."
}
```

Now the refresh token can be exchanged for a new accessToken.

```
POST http://localhost:8080/api/auth/refresh-token
{
"refreshToken": "eyJhbGciOiJSUzI1NiJ9.eyJzd...t1d8gwSfN-w"
}
```

If the refreshToken is already used, the following response will be returned.

```
403 Forbidden
{
"timeStamp": "2025-03-26T10:29:28.562962",
"error": "Refresh token not found or already used.",
"status": "FORBIDDEN"
}
```

When the refreshToken is valid, a new accessToken en refreshToken are returned.

Samwise Gamegee (user) will not be able to retrieve all the users.

```
GET http://localhost:8080/api/users
> Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJzdWIi....CDA1lbWsuvEEpX3hYA
```

```
403 Forbidden
{
"timeStamp": "2025-03-26T10:37:13.168673",
"error": "Access Denied",
"status": "FORBIDDEN"
}
```

However, the admin user gets access to retrieve all users.

```
[
{
"firstName": "Samwise",
"lastName": "Gamgee",
"email": "sam@shire.org",
"userRole": "USER"
},
{
"firstName": "Gandalf",
"lastName": "TheGrey",
"email": "admin@gondor.com",
"userRole": "ADMIN"
}
]
```

7.4 Refresh tokens

7.5 Unit tests

```
package be.pxl.demo.api;
```

```
import be.pxl.demo.config.JwtAuthenticationFilter;
import be.pxl.demo.config.JwtUtilities;
import be.pxl.demo.config.SpringSecurityConfig;
import be.pxl.demo.domain.Role;
import be.pxl.demo.domain.User;
import be.pxl.demo.security.CustomUserDetailsService;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.context.annotation.Import;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Collections;

import static
    ↪ org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(AdminController.class) // Loads only the AdminController with
    ↪ MockMvc
@Import({SpringSecurityConfig.class, JwtUtilities.class})
public class AdminControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private JwtUtilities jwtUtilities;

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @MockitoBean
    private CustomUserDetailsService customUserDetailsService;

    @Test
    void testSayHello_WithAdminUser_ShouldReturnWelcomeMessage() throws
        ↪ Exception {
        String jwtToken = jwtUtilities.generateToken("admin",
            ↪ Collections.singletonList("ADMIN"));
        User user = new User();
        user.setRole(Role.ADMIN);
        user.setEmail("admin");
        Mockito.when(customUserDetailsService.loadUserByUsername("admin")).thenReturn(use

        mockMvc.perform(get("/admin/hello").header("Authorization", "Bearer
            ↪ " + jwtToken))
                .andExpect(status().isOk()) // Expect 200 OK
                .andExpect(content().string("Welcome admin you are
                    ↪ authenticated as Admin!"));
    }

    @Test
    void testSayHello_WithUser_ShouldReturnAccessDenied() throws Exception {
        String jwtToken = jwtUtilities.generateToken("eve",
            ↪ Collections.singletonList("USER"));
        User user = new User();
        user.setRole(Role.USER);
        user.setEmail("eve");
        Mockito.when(customUserDetailsService.loadUserByUsername("eve")).thenReturn(user)

        mockMvc.perform(get("/admin/hello").header("Authorization", "Bearer
            ↪ " + jwtToken))
                .andExpect(status().is(403));
    }

    @Test
    void testSayHello_WithoutAuthentication_ShouldReturnUnauthorized()
        ↪ throws Exception {
        mockMvc.perform(get("/admin/hello"))
                .andExpect(status().isUnauthorized()); // Expect 401
                    ↪ Unauthorized
    }
}
```

With the @SpringBootTest annotation, Spring Boot provides a convenient way to start up an application context to be used in a test.

```
package be.pxl.demo.service.impl;

import be.pxl.demo.api.dto.UserDto;
import be.pxl.demo.builder.UserBuilder;
import be.pxl.demo.domain.User;
import be.pxl.demo.repository.UserRepository;
import be.pxl.demo.service.UserService;
import be.pxl.demo.service.mapper.UserMapper;
import be.pxl.demo.service.mapper.UserMapperImpl;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.access.AccessDeniedException;
import
    ↪ org.springframework.security.authentication.AuthenticationCredentialsNotFoundExce
import
    ↪ org.springframework.security.config.annotation.method.configuration.EnableMethodS
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.context.bean.override.mockito.MockitoBean;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

@SpringBootTest(classes = {UserService.class, DefaultUserService.class,
    ↪ UserMapperImpl.class})
@EnableMethodSecurity
class UserServiceTest {

    @MockitoBean
    private UserRepository userRepository;

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private UserService userService;

    @BeforeEach
    void setUp() {
        List<User> users = Arrays.asList(UserBuilder.anUser().build(),
            ↪ UserBuilder.anUser().build());
        when(userRepository.findAll()).thenReturn(users);
    }

    @Test
    @WithMockUser(roles = "ADMIN") // Simulate a user with ADMIN role
    void findAllUsers_ShouldReturnUsers_WhenUserIsAdmin() {
        List<UserDto> users = userService.findAllUsers();
        assertNotNull(users);
        assertEquals(2, users.size());
    }

    @Test
    @WithMockUser(roles = "USER") // Simulate a user with USER role
    void findAllUsers_ShouldThrowException_WhenUserIsNotAdmin() {
        assertThrows(AccessDeniedException.class, () ->
            ↪ userService.findAllUsers());
    }

    @Test
    void findAllUsers_ShouldThrowException_WhenUserIsNotAuthenticated() {
        assertThrows(AuthenticationCredentialsNotFoundException.class, ()
            ↪ -> userService.findAllUsers());
    }
}
```

When using @SpringBootTest annotation to test controllers with Spring Security, it's necessary to explicitly configure the filter chain when setting up MockMvc.

Using the static springSecurity method provided by SecurityMockMvcConfigurer is the preferred way to do this.

The annotation @WithMockUser is used to mock a logged-in user.