Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №9

Технологія розроблення програмного забезпечення

«Shell (total commander)»

Варіант 18

Виконав
студент групи ІА-13
Окаянченко Давид Олександрович

Перевірив:
Мягкий Михайло Юрійович

Київ 2023

Мета: Дослідити різні види взаємодії додатків: «Client-Server», «Peer-To-Peer», «Service-Oriented Architecture» та навчитися застосовувати один із них на практиці.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.

2. Реалізувати функціонал для роботи в розподіленому оточенні (логіку роботи).

3. Реалізувати взаємодію розподілених частин:

   А) Для клієнт-серверних варіантів: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient або .net-remoting на розсуд виконавця;

Варіант:

18. Shell (total commander) (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі - перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Хід роботи

Архітектура проєкту - це стратегічне планування та організація структури програмного забезпечення для досягнення певних цілей проєкту. Вона визначає взаємодію різних компонентів програми чи системи, їх функціональний розподіл та внутрішню організацію. Основні аспекти архітектури проєкту з точки зору взаємодії додатків включають:

- Розподіл функціональності:
  - *Опис:* Визначення та розподіл функцій між різними компонентами системи або додатків.
  - *Значення:* Цей аспект визначає, як різні частини системи співпрацюють, виконуючи свої визначені завдання.
- Взаємодія між компонентами:
  - *Опис:* Визначення протоколів та механізмів обміну даними між компонентами системи.
  - *Значення:* Це визначає, як інформація передається та обробляється

між різними частинами системи.

- Модульність та розширюваність:
  - *Опис:* Розбиття системи на невеликі, незалежні модулі, які можуть бути легко модифіковані чи розширені.
  - *Значення:* Це полегшує розробку, тестування та супровід системи, а також дозволяє легко впроваджувати нові функції.
- Масштабованість:
  - *Опис:* Здатність системи збільшувати обсяг роботи або кількість користувачів без втрати продуктивності.
  - *Значення:* Масштабованість гарантує, що система може адаптуватися до зростаючого обсягу даних та користувацького навантаження.
- Безпека та захист даних:
  - *Опис:* Визначення заходів для запобігання несанкціонованому доступу та збереження конфіденційності інформації.
  - *Значення:* Забезпечення високого рівня захисту даних та системи в цілому.
- Інтеграція з іншими системами:
  - *Опис:* Здатність системи взаємодіяти та обмінюватися даними з іншими додатками чи сервісами.
  - *Значення:* Інтеграція дозволяє системі взаємодіяти з існуючими рішеннями та використовувати зовнішні ресурси.
- Доступність та надійність:
  - *Опис:* Забезпечення стабільної та неперервної роботи системи в умовах різних викликів та можливих відмов.
  - *Значення:* Це важливо для забезпечення доступності сервісів та уникнення втрат продуктивності у випадку відмов.

Архітектура проєкту визначає цілісну структуру системи, спрощує розробку та підтримку, полегшує масштабування та дозволяє системі ефективно взаємодіяти з іншими компонентами та сервісами.

Клієнт-серверні додатки

Структура:

Опис

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). Розрізняють тонкі клієнти і товсті клієнти.

Тонкий клієнт - клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт - набір форм відображення і канал зв'язку з сервером.

У такому варіанті використання дуже велике навантаження лягає на сервер. Раніше такий варіант був доступний, оскільки обчислень було небагато, термінали були дорогими і як правило йшли з урізаним обчислювальним пристроєм. У нинішній час зі зростанням обчислювальних можливостей клієнтських машин має сенс частину обчислень перекладати на клієнтські комп'ютери.

Однак дана модель має сенс ще в захищених сценаріях - коли зайві дані не можна показувати клієнтським комп'ютерам в зв'язку з небезпекою взлому. Ще однією проблемою може бути множинний доступ - щоб уникнути конфліктності даних має сенс централізація обчислень в одному місці (на сервері) для визначення послідовності операцій та уникнення пошкодження даних.

Товстий клієнт - антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу (наприклад, бази даних) або сполучна ланка з іншими клієнтськими комп'ютерами.

У моделі клієнт-серверної взаємодії також важливе місце займає модель «підписки / видачі». Комп'ютери клієнтів можуть «підписуватися» серверу на певний набір подій (поява нових користувачів, зміна даних тощо), а сервер в

свою чергу сповіщає клієнтські комп'ютери про походження цих подій. Даний спосіб взаємодії реалізується за допомогою шаблону «оглядач»; він несе велике навантаження на сервер (необхідно відстежувати хто на що підписався) і на канали зв'язку (багато повторюваних даних відправляються на різні точки в мережі). З іншого боку, це дуже зручно для клієнтських машин для організації оновлення даних без їх повторного перезапросу. Природно, інші механізми включають лініву ініціалізацію, перезапрос, запит за таймером, запит за потребою (після натискання кнопки «оновити» наприклад).

Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина.

Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.

Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.

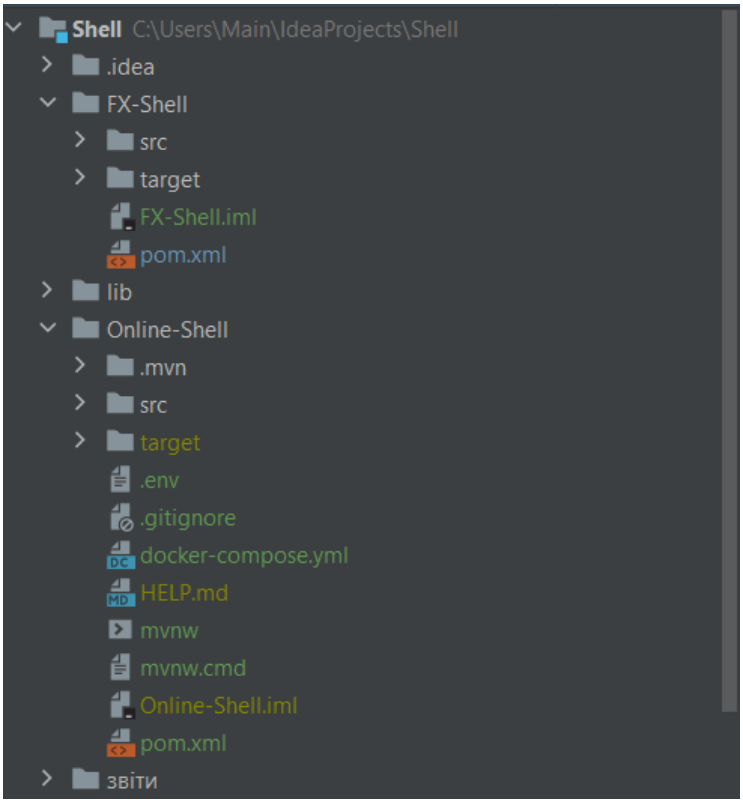
Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

Реалізація:

На стороні сервера (Spring Boot) створено контроллер AuthorizationController, який обробляє HTTP-запити для реєстрації, входу та виходу користувача. Для обробки бізнес-логіки використовується сервіс UserService.

На стороні клієнта (JavaFX) використовується HttpClient для взаємодії з сервером, відправляючи й отримуючи HTTP-запити. JavaFX використовується для створення графічного інтерфейсу, який дозволяє користувачеві взаємодіяти з додатком. Крім того, обробляються отримані від сервера відповіді для подальшого відображення інформації.

Стуктура проєкта:



Розділення проєкту на підпапки:

Проєкт було розбито на дві різні папки: "FX-Shell" і "Online-Shell".

FX-Shell:

Ця папка містить JavaFX застосунок або клієнтську частину. Тут розташовуються файли та папки, пов'язані з графічним інтерфейсом, контролерами та іншими ресурсами, які використовуються для взаємодії з користувачем.

Online-Shell:

Ця папка містить серверну частину додатка, розробленого за допомогою Spring Boot. Тут розташовуються файли та папки, пов'язані з логікою сервера, контролерами, сервісами та іншими компонентами, що обробляють запити від клієнта.

FX-Shell

Клас Main:

Клас Main використовується для організації центральної логіки в моєму додатку. Його функціональність охоплює обробку подій інтерфейсу користувача, створення об'єктів та використання шаблонів проектування, таких як Prototype, Template Method і Factory Method. Крім того, клас реалізує інтерпретатор виразів для фільтрації файлів та взаємодії з сервером через HTTP-запити, що додає новий рівень функціональності та взаємодії з іншими компонентами додатку.

```
package com.example.fxshell;

import com.example.fxshell.controllers.http.HttpController;
import com.example.fxshell.files.comboBoxes.ComboBoxTemplate;
import com.example.fxshell.files.comboBoxes.Impl.DiskComboBoxTemplate;
import com.example.fxshell.files.comboBoxes.Impl.ViewComboBoxTemplate;
import com.example.fxshell.files.filters.FileFilterExpression;
import com.example.fxshell.files.filters.Impl.ExtensionFilterExpression;
import com.example.fxshell.files.filters.Impl.FileFilter;
import com.example.fxshell.files.filters.Impl.NameFilterExpression;
import com.example.fxshell.files.views.FilesViewFactory;
import com.example.fxshell.files.views.FilesViewTab;
import com.example.fxshell.files.views.Impl.FilesView;
import com.example.fxshell.files.views.Impl.FilesViewType;
import com.example.fxshell.files.views.ListFilesViewFactory;
import com.example.fxshell.files.views.TableFilesViewFactory;
import com.example.fxshell.models.Disk;
import com.example.fxshell.models.User;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import org.kordamp.bootstrapfx.BootstrapFX;

import java.io.File;
```

```
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main extends javafx.application.Application {
    public static void main(String[] args) {
        launch();
    }

    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(Main.class.getResource("shell.fxml"));
        Scene scene = new Scene(fxmlLoader.load());
        scene.getStylesheets().add(BootstrapFX.bootstrapFXStylesheet());
        stage.setTitle("Shell");
        stage.setScene(scene);
        stage.show();

        // Test prototype pattern
        User user1 = new User("Davyd", "david@gmail.com", "123123");
        User user2 = user1.clone();
        user2.setName("Vitaliy");
        user2.setEmail("vitaliy@gmail.com");

        // Test Template and Factory method patterns
        List<File> testFiles = Arrays.asList(
                new File("Folder1"),
                new File("Folder2"),
                new File("File1.txt"),
                new File("File2.txt"),
                new File("File3.txt")
        );

        List<Disk> testDisks = Arrays.asList(
                new Disk("C"),
                new Disk("D"),
                new Disk("E")
        );

        ComboBoxTemplate diskComboBoxTemplate = new DiskComboBoxTemplate();

        ComboBox<String> diskComboBox = diskComboBoxTemplate.createMenu(
                testDisks.stream().map(Disk::getName).toList());

        Tab diskTab = new Tab("Disk Menu");
        diskTab.setContent(diskComboBox);


        ComboBoxTemplate viewComboBoxTemplate = new ViewComboBoxTemplate();

        ComboBox<String> viewComboBox = viewComboBoxTemplate.createMenu(
                Arrays.stream(FilesViewType.values())
                        .map(Enum::name)
                        .collect(Collectors.toList()));

        Tab viewTab = new Tab("View Menu");
        BorderPane pane = new BorderPane();
        pane.setTop(viewComboBox);

        viewComboBox.setOnAction(event -> {
            String selectedView = viewComboBox.getValue();
            FilesViewType viewType =
FilesViewType.valueOf(selectedView.toUpperCase());

            FilesViewFactory filesViewFactory;

            if (viewType == FilesViewType.LIST) {
```

```
                filesViewFactory = new ListFilesViewFactory();
            } else {
                filesViewFactory = new TableFilesViewFactory();
            }

            FilesView filesView = filesViewFactory.create();
            FilesViewTab.setFilesView(filesView);
            pane.setCenter(FilesViewTab.getFilesView().getNode());
        });

        final FilesViewFactory filesViewFactory;

        filesViewFactory = new ListFilesViewFactory();
        FilesView listView = filesViewFactory.create();

        FilesViewTab.setFilesView(listView);
        FilesViewTab.setFiles(testFiles);

        pane.setCenter(FilesViewTab.getFilesView().getNode());
        viewTab.setContent(pane);

        // Test Interpreter pattern
        TextField extensionTextField = new TextField();
        extensionTextField.setPromptText("Enter Extension");

        // Кнопка для застосування фільтрації
        Button applyFilterButton = new Button("Apply Extension Filter");
        applyFilterButton.setOnAction(event -> {
            String extension = extensionTextField.getText();
            if (!extension.isEmpty()) {
                FileFilterExpression extensionFilter = new
ExtensionFilterExpression(extension);
                FileFilter.applyFilter((ListView<String>)
FilesViewTab.getFilesView().getNode(), testFiles, extensionFilter);
            } else {
                FilesViewTab.setFiles(testFiles);
            }
        });

        TextField nameTextField = new TextField();
        nameTextField.setPromptText("Enter File Name");

        // Кнопка для застосування фільтрації за іменем файлу
        Button applyNameFilterButton = new Button("Apply Name Filter");
        applyNameFilterButton.setOnAction(event -> {
            String targetName = nameTextField.getText();
            if (!targetName.isEmpty()) {
                FileFilterExpression nameFilter = new
NameFilterExpression(targetName);
                FileFilter.applyFilter((ListView<String>)
FilesViewTab.getFilesView().getNode(), testFiles, nameFilter);
            } else {
                FilesViewTab.setFiles(testFiles);
            }
        });

        VBox root = new VBox(extensionTextField, applyFilterButton, nameTextField,
applyNameFilterButton, FilesViewTab.getFilesView().getNode());
        viewTab.setContent(root);


        // Test Client-Server architecture
        HttpController httpController = new HttpController();
        Tab serverTab = new Tab("Server Menu");

        TextField loginField = new TextField();
        PasswordField passwordField = new PasswordField();
        Label messageLabel = new Label();
```

```
        Button registerButton = new Button("Register");
        registerButton.setOnAction(event ->
messageLabel.setText(httpController.register(loginField.getText(),
passwordField.getText())));
        Button loginButton = new Button("Login");
        loginButton.setOnAction(event ->
messageLabel.setText(httpController.login(loginField.getText(),
passwordField.getText())));

        VBox serverBox = new VBox(
                new Label("login:"),
                loginField,
                new Label("Password:"),
                passwordField,
                registerButton,
                loginButton,
                messageLabel
        );
        serverTab.setContent(serverBox);

        TabPane tabPane = new TabPane(diskTab, viewTab, serverTab);
        Scene testScene = new Scene(tabPane, 640, 480);
        stage.setScene(testScene);
        stage.show();
    }
}
```

Утилітний клас Utils:

Допоміжний клас Utils містить константу SERVER_URL, яка представляє собою базовий URL для взаємодії із сервером додатку. Цей клас використовується для зручного доступу до цього URL та уникнення повторюваного введення адреси сервера в різних частинах програми.

```
package com.example.fxshell;

public class Utils {
    public static final String SERVER_URL = "http://localhost:8080/online-shell/";
}
```

Клас HttpController:

Клас HttpController відповідає за взаємодію із сервером додатку через HTTP-запити. Використовується для реєстрації користувача та входу, відправляючи відповідні POST-запити на сервер і отримуючи результат у формі текстового відгуку або повідомлення про помилку.

```
package com.example.fxshell.controllers.http;

import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.Map;

import com.example.fxshell.Utils;
import com.google.gson.Gson;

public class HttpController {
    Gson gson = new Gson();

    public String register(String login, String password) {
        HttpResponse<String> response = post("register", Map.of(
```

```
                "login", login,
                "password", password
        ));
        return response != null ? response.body() : "Помилка на сервері";
    }

    public String login(String login, String password) {
        HttpResponse<String> response = post("login", Map.of(
                "login", login,
                "password", password
        ));
        return response != null ? response.body() : "Помилка на сервері";
    }

    private HttpResponse<String> post(String endpoint, Map<Object, Object> data) {
        try {
            return HttpClient.newHttpClient().send(
                    HttpRequest.newBuilder()
                            .uri(URI.create(Utils.SERVER_URL + endpoint))
                            .header("Accept", "application/json")
                            .header("Content-Type", "application/json")

.POST(HttpRequest.BodyPublishers.ofString(gson.toJson(data)))
                            .build(),
                    HttpResponse.BodyHandlers.ofString()
            );
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Online-Shell

Клас OnlineShellApplication:

Клас OnlineShellApplication є головним класом додатку Spring Boot. Він містить метод main, який викликає SpringApplication.run для запуску додатку, і використовує анотації @SpringBootApplication, щоб позначити клас як основний клас додатку та автоматично конфігурувати Spring Boot.

```
package com.example.onlineshell;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class OnlineShellApplication {
    public static void main(String[] args) {
        SpringApplication.run(OnlineShellApplication.class, args);
    }
}
```

Клас AutorizationController:

Клас AuthorizationController є частиною серверної частини додатку і обробляє HTTP-запити для реєстрації, входу та виходу користувача. Використовується для обробки вхідних даних, взаємодії з моделями через сервіс користувачів (UserService) та повернення відповідей HTTP-статусу та повідомлень користувачеві.

```
package com.example.onlineshell.controllers;

import com.example.onlineshell.models.User;
import com.example.onlineshell.services.UserService;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;


@RestController
public class AuthorizationController {

    UserService userService;

    @Autowired
    public AuthorizationController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/register")
    protected ResponseEntity<?> register(@Valid @RequestBody User user,
                                         HttpServletRequest request) {
        request.getSession().invalidate();
        if (userService.exists(user.getLogin())) {
            return new ResponseEntity<>("Користувача з таким логіном вже було
зареєстровано.",
                    HttpStatus.BAD_REQUEST);
        }

        userService.create(user);
        request.getSession().setAttribute("user", user);

        return new ResponseEntity<>("Успішно зареєстровано.",
                HttpStatus.OK);
    }

    @PostMapping(value = "/login")
    protected ResponseEntity<?> login(@Valid @RequestBody User user,
                                      HttpServletRequest request) {
        User dbUser = userService.getByLogin(user.getLogin());
        if (dbUser == null) {
            return new ResponseEntity<>("Користувача з логіном: " + user.getLogin() +
", не існує.",
                    HttpStatus.NOT_FOUND);
        }
        if (!userService.checkPassword(dbUser, user.getPassword())) {
            return new ResponseEntity<>("Неправильний пароль.",
HttpStatus.UNAUTHORIZED);
        }

        request.getSession().setAttribute("user", user);
        return new ResponseEntity<>("Ви успішно ввійшли.", HttpStatus.OK);
    }
```

```
    @GetMapping("/logout")
    protected ResponseEntity<?> logout(HttpServletRequest request) {
        request.getSession().invalidate();
        return ResponseEntity.ok().build();
    }
}
```

Клас ErrorHandlingControllerAdvice:

Клас ErrorHandlingControllerAdvice є складовою частиною серверного застосунку і відповідає за обробку винятків, зокрема, випадку, коли аргумент методу не відповідає очікуваному формату (MethodArgumentNotValidException). При отриманні цього винятку сервер відповідає з HTTP-статусом "BAD_REQUEST" та повідомленням про помилку.

```
package com.example.onlineshell.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.Objects;

@ControllerAdvice
public class ErrorHandlingControllerAdvice {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    String onMethodArgumentNotValidException(MethodArgumentNotValidException e) {
        return
Objects.requireNonNull(e.getBindingResult().getFieldError()).getDefaultMessage();
    }
}
```

Клас User:

Клас User представляє модель користувача на серверному застосунку і використовується для взаємодії з базою даних. Він включає анотації JPA для мапінгу об'єкту на таблицю бази даних, а також валідаційні анотації для забезпечення коректності ведення даних.

```
package com.example.onlineshell.models;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.Data;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.ToString;

import java.util.UUID;

@Entity
@Getter
@NoArgsConstructor
@ToString
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = {
```

```
                "login"
        })
})
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @NotBlank(message = "Логін не може бути пустим")
    @Size(max = 100)
    @Email(message = "Некоректний логін")
    private String login;

    @NotBlank(message = "Пароль не може бути пустим")
    @Size(max = 100)
    private String password;

    public User(String login, String password) {
        this.login = login;
        this.password = password;
    }
}
```

Клас UserRepository:

Клас UserRepository є інтерфейсом, який використовує Spring Data JPA для взаємодії з базою даних для сутності User. Він надає методи для виконання операцій читання, запису та оновлення даних користувачів у базі даних.

```
package com.example.onlineshell.repository;

import com.example.onlineshell.models.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Integer> {
    User getByLogin(String login);
}
```

Клас FactoryRepository:

Клас FactoryRepository є інтерфейсом, який оголошує метод getUserRepository(), призначений для створення та повернення екземпляра інтерфейсу UserRepository. Це може використовуватися для створення реалізацій репозиторію користувачів в рамках патерна фабричного методу для роботи з базою даних у Spring-застосунках.

```
package com.example.onlineshell.repository.factory;


import com.example.onlineshell.repository.UserRepository;

public interface FactoryRepository {
    UserRepository getUserRepository();
}
```

Клас FactoryRepositoryImpl:

Клас FactoryRepositoryImpl є реалізацією інтерфейсу FactoryRepository та є компонентом Spring. Він містить поле userRepository, яке ініціалізується через конструктор за допомогою механізму впровадження залежностей. Метод getUserRepository() повертає екземпляр репозиторію користувачів, що може використовуватися для доступу до бази даних в Spring-застосунках.

```
package com.example.onlineshell.repository.factory;

import com.example.onlineshell.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class FactoryRepositoryImpl implements FactoryRepository {
    UserRepository userRepository;

    @Autowired
    public FactoryRepositoryImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserRepository getUserRepository() {
        return userRepository;
    }
}
```

Клас UserService:

Інтерфейс UserService є компонентом Spring та визначає методи, які пов'язані із роботою з користувачами в системі. Методи включають отримання користувача за логіном, перевірку існування користувача за логіном, перевірку пароля та створення нового користувача. Цей інтерфейс дозволяє реалізувати сервісні функції, пов'язані із управлінням користувачами у веб-застосунках на основі Spring.

```
package com.example.onlineshell.services;

import com.example.onlineshell.models.User;
import org.springframework.stereotype.Component;

@Component
public interface UserService {
    User getByLogin(String login);

    boolean exists(String login);

    boolean checkPassword(User user, String password);

    void create(User user);
}
```

Клас UserServiceImpl:

Клас UserServiceImpl є реалізацією інтерфейсу UserService у контексті Spring. Він використовує FactoryRepository для отримання доступу до репозиторію користувачів (UserRepository). Клас надає реалізацію методів для отримання, перевірки існування, перевірки пароля та створення користувача в системі. Також, він використовує BCryptPasswordEncoder для безпечного зберігання та порівняння хешів паролів. Цей сервісний клас відповідає за логіку роботи з об'єктами користувачів в системі та використовує підходи від Spring для ін'єкції залежностей та управління життєвим циклом компоненту.

```
package com.example.onlineshell.services;

import com.example.onlineshell.models.User;
import com.example.onlineshell.repository.factory.FactoryRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {
    private final FactoryRepository fr;
    private final BCryptPasswordEncoder passwordEncoder = new
BCryptPasswordEncoder();

    @Autowired
    public UserServiceImpl(FactoryRepository factoryRepository) {
        this.fr = factoryRepository;
    }

    @Override
    public User getByLogin(String login) {
        return fr.getUserRepository().getByLogin(login);
    }

    @Override
    public boolean exists(String login) {
        return getByLogin(login) != null;
    }

    @Override
    public boolean checkPassword(User user, String password) {
        return passwordEncoder.matches(password, user.getPassword());
    }

    @Override
    public void create(User user) {
        fr.getUserRepository().save(new User(
                user.getLogin(),
                passwordEncoder.encode(user.getPassword())
        ));
    }
}
```

Конфігураційний файл application.properties:

Цей конфігураційний файл містить параметри конфігурації для Spring Boot додатку та бази даних, який використовує MySQL. Деякі ключові налаштування включають порт сервера (8080), включення компресії, контекстний шлях додатку (/online-shell), дані для підключення до бази даних MySQL, шлях до файлу конфігурації Docker Compose, налаштування Hibernate (наприклад, автоматичне оновлення схеми бази даних при старті додатку), та інші параметри для логування та ініціалізації даних.

```
## Server Properties
server.port=8080
server.compression.enabled=true
server.servlet.context-path=/online-shell
server.error.include-stacktrace=never
logging.pattern.dateformat=
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url=jdbc:mysql://localhost:3306/online-shell
spring.datasource.username=admin
spring.datasource.password=root
## Spring Docker Properties
spring.docker.compose.file=./Online-Shell/docker-compose.yml
## Hibernate Properties
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.format_sql=true
# Initialize the datasource with available DDL and DML scripts
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
```

Конфігураційний файл docker-compose.yml:

Файл docker-compose.yml представляє конфігурацію для Docker Compose, яка визначає контейнер з образом MySQL та його параметри. Він встановлює середовищні змінні для користувацького імені, пароля та назви бази даних MySQL, а також вказує порти для зовнішнього доступу до служби MySQL.

```
version: "3.1"
services:
  ### mysql ###############################################
  mysql:
    image: mysql
    container_name: ${DOCKER_PROJECT_NAME}-mysql
    environment:
      - MYSQL_USER=${DB_USERNAME}
      - MYSQL_PASSWORD=${DB_PASSWORD}
      - MYSQL_DATABASE=${DB_DATABASE}
      - MYSQL_ALLOW_EMPTY_PASSWORD=yes
    ports:
      - ${DOCKER_MYSQL_PORT}:${DB_PORT}
```

Файл .env:

Цей файл містить змінні оточення для використання в проекті, зокрема, параметри підключення до бази даних та конфігураційні параметри для Docker Compose. Змінні визначають різні параметри, такі як ім'я користувача та пароль бази даних, а також налаштування Docker-контейнера для MySQL, що використовується в проекті.

```
# DB #
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=online-shell
DB_USERNAME=admin
DB_PASSWORD=root

# DOCKER #
DOCKER_PROJECT_NAME=online-shell
DOCKER_MYSQL_PORT=3306
```

Файл pom.xml:

Цей файл представляє собою файл конфігурації проекту Maven для застосунку Spring Boot, названого "Online-Shell". Він містить важливу інформацію про залежності, версії та інші налаштування, необхідні для збірки та виконання проекту. Файл POM (Project Object Model) визначає структуру проекту, його залежності, плагіни та інші параметри. Зокрема, вибрані залежності вказують на використання Spring Boot, бази даних MySQL, інструментів розробки та інших бібліотек, необхідних для роботи застосунку.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>Online-Shell</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Online-Shell</name>
    <description>Online-Shell</description>
    <properties>
        <java.version>21</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-docker-compose</artifactId>
```

```
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.30</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>jakarta.persistence</groupId>
            <artifactId>jakarta.persistence-api</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-crypto</artifactId>
            <version>6.0.2</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Висновок: У ході виконання лабораторної роботи було проведено ознайомлення з теоретичними відомостями та реалізовано взаємодію програм у архітектурі «Client-Server». Окрім того, підготовлений звіт включає всі необхідні компоненти, що відображають структуру розробленої системи.