

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ
СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №3
з курсу
«Технології розроблення програмного забезпечення»

Виконавець:
Нікітченко Наталя Олегівна
студентка групи IA-33
залікова книжка № IA-3318

«07» 11 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

3. ЛАБОРАТОРНА РОБОТА № 3

Тема: Основи проектування розгортання.

Мета: Навчитися проєктувати діаграми розгортання та компонентів для системи що проєктується, а також розробляти діаграми взаємодії, а саме діаграми послідовностей, на основі сценаріїв зроблених в попередній лабораторній роботі.

Завдання:

- Ознайомитись з короткими теоретичними відомостями.
- Проаналізувати діаграми створені в попередній лабораторній роботі а також тему системи та спроєктувати діаграму розгортання використання відповідно до обраної теми лабораторного циклу.
- Розробити діаграму компонентів для проєктованої системи.
- Розробити діаграму розгортання для проєктованої системи.
- Розробити як мінімум дві діаграми послідовностей для сценаріїв прописаних в попередній лабораторній роботі.
- На основі спроєктованих діаграм розгортання та компонентів доопрацювати програмну частину системи. Реалізація системи, додатково до попередньої реалізації, повинна містити як мінімум дві візуальні форми. В системі вже повинен бути повністю реалізована архітектура (повний цикл роботи з даними від вводу на формі до збереження їх в БД і подальшій виборці з БД та відображенням на UI).
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму розгортання з описом, діаграму компонентів системи з описом, діаграми послідовностей, а також вихідний код системи, який було додано в цій лабораторній роботі.

Теоретичні відомості:

Діаграма компонентів (Component Diagram)

Діаграма компонентів описує логічну структуру програмної системи, тобто які модулі, підсистеми або компоненти її складають та як вони взаємодіють між собою. Вона показує залежності між частинами програми, а також інтерфейси, які ці компоненти надають або використовують. Такі діаграми відображають архітектуру високого рівня і демонструють, як програмні елементи об'єднані в компоненти та яким чином ці компоненти взаємозалежні.

Діаграма розгортання (Deployment Diagram)

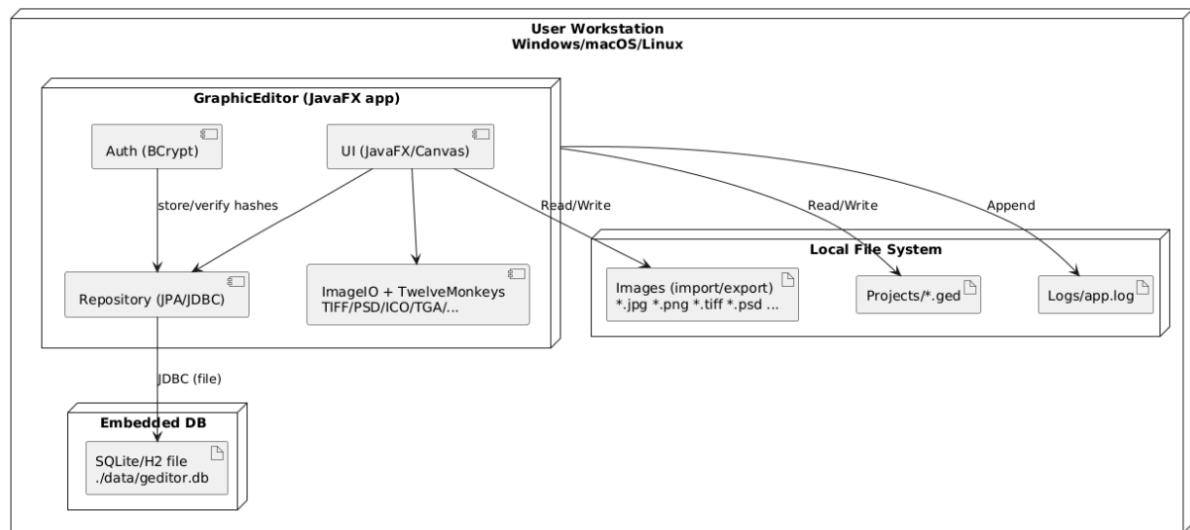
Діаграма розгортання описує фізичне розміщення системи: на яких апаратних або програмних вузлах вона працює та як ці вузли з'єднані між собою. Вона показує, де саме виконуються програмні компоненти — на сервері, ПК, мобільному пристройі, віртуальній машині тощо. Така діаграма дає уявлення про інфраструктуру, середовище виконання, мережеві з'єднання та розподіл компонентів між апаратними елементами.

Діаграми послідовностей (Sequence Diagrams)

Діаграми послідовностей відображають динамічну поведінку системи, тобто порядок взаємодії між об'єктами під час виконання певного сценарію. Вони показують, які об'єкти беруть участь у процесі, які повідомлення або виклики методів передаються між ними, та у якій послідовності це відбувається. Такі діаграми дозволяють детально описати логіку роботи системи в часі та пов'язати її з конкретними варіантами використання.

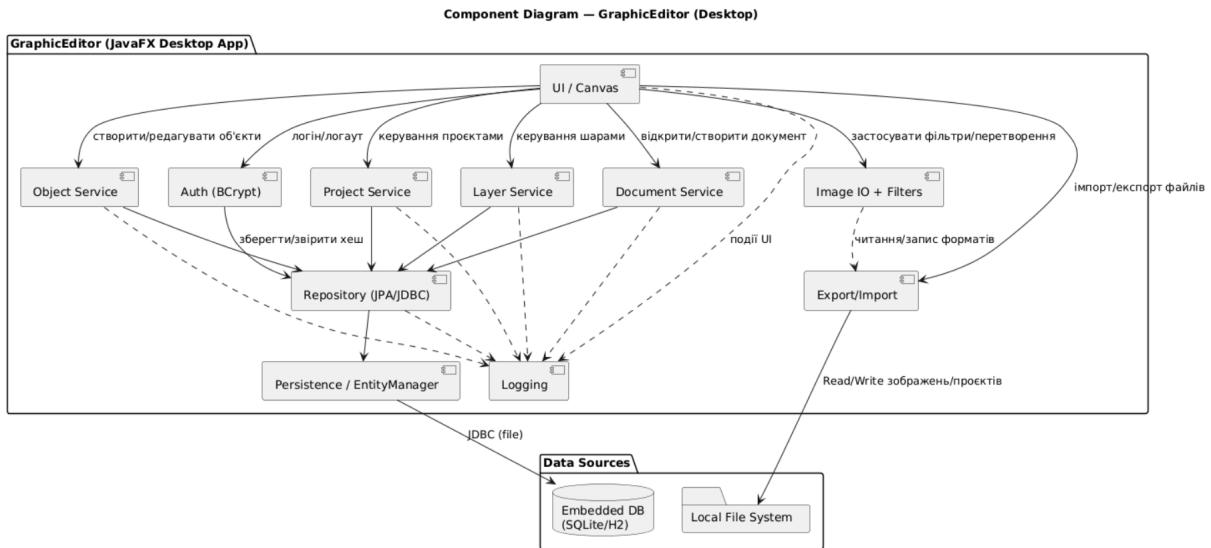
Хід роботи:

Проаналізувавши діаграми створені в попередній лабораторній роботі а також тему системи я спроектувала діаграму розгортання використання відповідно до обраної теми лабораторного циклу.



Діаграма розгортання відображає архітектуру настільного застосунку GraphicEditor, який працює локально на комп'ютері користувача та включає модулі авторизації (BCrypt), графічного інтерфейсу JavaFX, імпорту/експорту зображень (ImageIO + TwelveMonkeys) і доступу до даних через JPA/JDBC. Усі дані користувачів, проекти та параметри зображень зберігаються у вбудованій файловій базі даних (SQLite/H2), а графічні файли, експорт проектів і журнали подій розміщуються у локальній файловій системі. Така конфігурація забезпечує повністю автономну офлайн-роботу застосунку, підтримку численних графічних форматів і надійну обробку користувальських даних без необхідності серверної інфраструктури.

Діаграма компонентів для проєктованої системи:

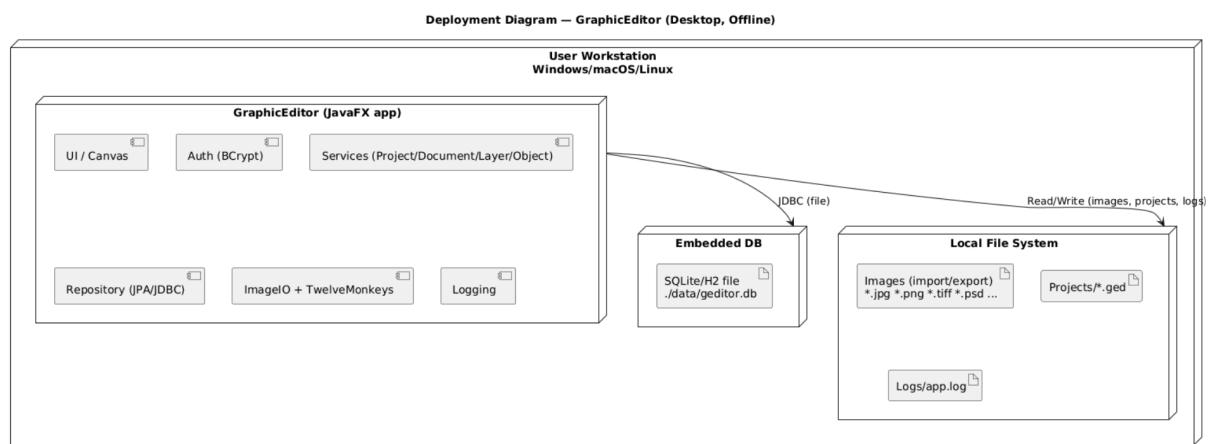


Діаграма компонентів відображає внутрішню архітектуру настільного графічного редактора GraphicEditor, побудованого за модульним підходом. Центральним елементом системи є компонент UI/Canvas, який забезпечує взаємодію користувача з інтерфейсом, обробку команд та відображення графічного полотна. Інтерфейс напряму використовує сервіси системи: *Project Service*, *Document Service*, *Layer Service* та *Object Service*, які відповідають за керування проектами, документами, шарами та графічними об'єктами відповідно. Авторизація в застосунку реалізована компонентом Auth (BCrypt), який здійснює хешування та перевірку паролів і працює через компонент Repository (JPA/JDBC), що забезпечує доступ до локальної бази даних.

Компонент Image IO + Filters відповідає за відкриття, збереження та обробку зображень різних форматів (JPEG, PNG, TIFF, PSD тощо) та передає ці операції компоненту Export/Import, який взаємодіє з локальною файловою системою для імпорту вихідних файлів та експорту результатів редагування. Для збереження даних застосунок використовує компонент Persistence/EntityManager, який працює через JDBC-файлове підключення до вбудованої бази даних *SQLite/H2*. Усі компоненти надсилають службові події до модуля Logging, який записує інформацію про роботу програми.

У такий спосіб діаграма демонструє чітку багаторівневу структуру: інтерфейс користувача → бізнес-логіка (сервіси) → доступ до даних → файлове та базове зберігання. Архітектура забезпечує розширеність, простоту модифікації та зручність локальної роботи без необхідності підключення до сервера.

Діаграма розгортання для проєктованої системи:

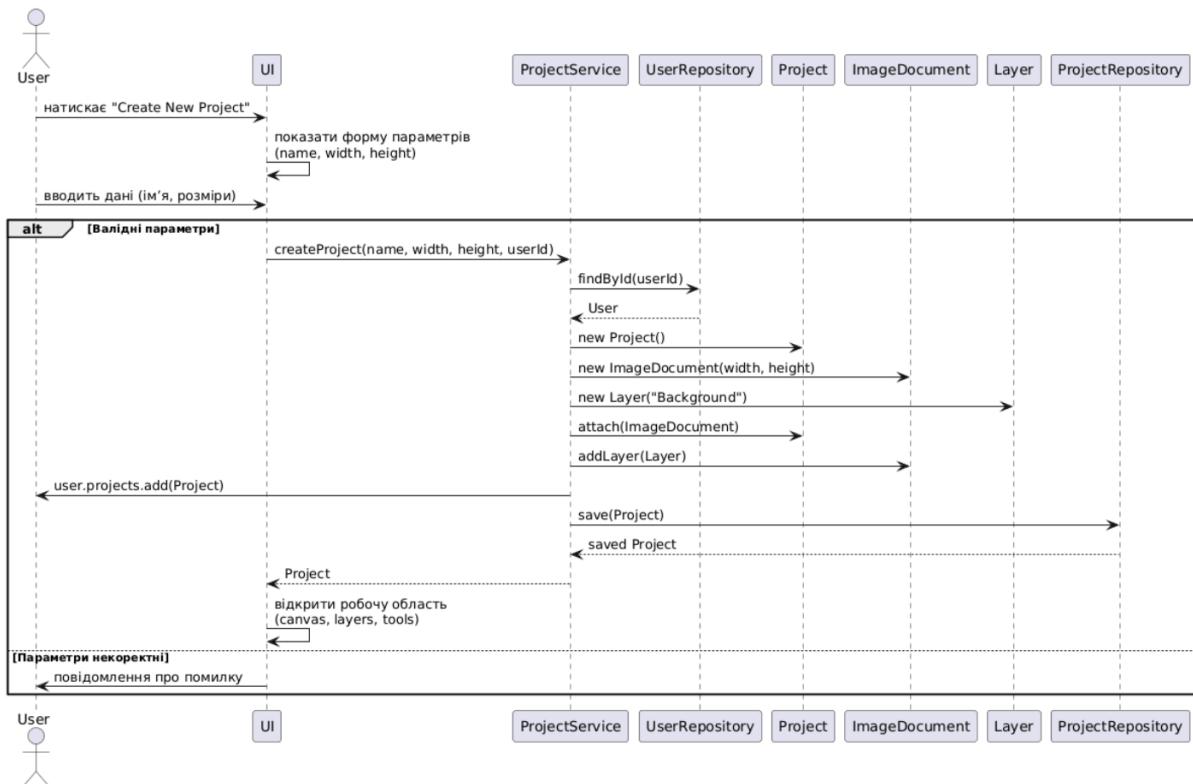


Діаграма розгортання демонструє структуру та фізичне розміщення компонентів настільного застосунку GraphicEditor, який працює локально на робочій станції користувача (Windows/macOS/Linux). Застосунок реалізовано як автономний JavaFX-додаток, що містить інтерфейс користувача (UI/Canvas), модуль авторизації (BCrypt), сервіси бізнес-логіки (керування проєктами, документами, шарами та графічними об'єктами), компонент для роботи з даними (Repository через JPA/JDBC), модуль обробки зображень (ImageIO + TwelveMonkeys) та підсистему журналювання (Logging).

Для зберігання внутрішніх даних система використовує будовану файлову базу даних SQLite або H2, до якої доступ здійснюється через JDBC у файловому режимі. Операції імпорту та експорту зображень, збереження

проектів у власному форматі (*.ged), а також ведення журналу подій виконуються через локальну файлову систему. Усі елементи застосунку функціонують на одному вузлі — робочому комп’ютері користувача, що забезпечує повністю офлайн-роботу, швидкий доступ до ресурсів і відсутність залежності від мережі чи серверної інфраструктури.

Діаграма послідовностей №1 — Створення нового проекту та полотна:



На поданій діаграмі послідовностей зображенено процес створення нового проекту в графічному редакторі. Сценарій починається з того, що користувач натискає кнопку «**Create New Project**», після чого інтерфейс (UI) показує форму введення параметрів документа — назви проекту та розмірів полотна. Користувач вводить необхідні дані, і якщо параметри коректні, UI передає їх у **ProjectService** методом `createProject(...)`.

ProjectService отримує дані та викликає **UserRepository** для пошуку користувача за його ідентифікатором. Після повернення об’єкта User сервіс

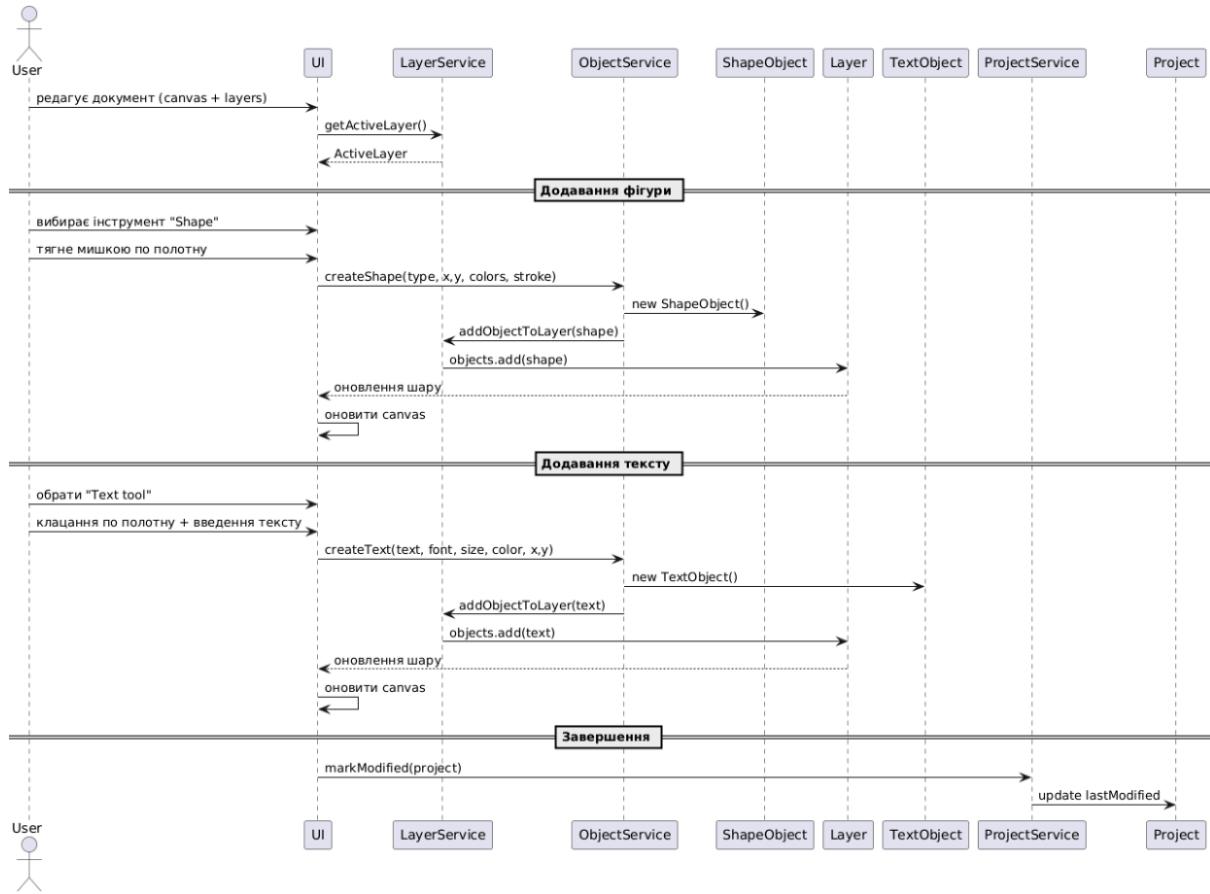
створює новий об'єкт **Project**, пов'язує з ним **ImageDocument** з указаними width/height та створює стартовий шар **Layer** (наприклад, "Background").

Створені модельні об'єкти зв'язуються між собою: документ прикріплюється до проєкту, шар додається до документа, а сам проєкт — у список проєктів користувача (user.projects.add(Project)).

Далі ProjectService зберігає проєкт через **ProjectRepository**, після чого повертає результат у UI, який відкриває робочу область редагування (canvas, layers, tools).

У випадку, якщо параметри документа некоректні, UI одразу повертає користувачеві повідомлення про помилку, і процес створення проєкту не продовжується.

Діаграма послідовностей №2 — Додавання фігури та тексту в активний шар:



На діаграмі подано процеси, що відбуваються під час редагування документа в графічному редакторі, коли користувач додає фігуру та текстовий об'єкт у выбраний шар. Сценарій починається з того, що користувач працює у вікні редагування (canvas + список шарів), після чого UI запитує в LayerService активний шар. Після вибору інструмента «Shape» користувач тягне мишкою по полотну, і UI передає параметри створення фігури в LayerService методом createShape(...). LayerService делегує створення нового графічного об'єкта в ObjectService, який ініціалізує новий ShapeObject і повертає його.

Сервіс шару додає нову фігуру до поточного шару (addObjectToLayer), а шар додає її у свій список об'єктів (objects.add(shape)), після чого UI оновлює відображення полотна.

Далі користувач вибирає інструмент «Text», клацає по полотну та вводить текст. UI викликає `createText(...)`, і `ObjectService` створює новий `TextObject`, який так само додається у список об'єктів шару. Після кожного доданого елемента UI повторно рендерить оновлене полотно.

У фіналі UI викликає `markModified(project)`, передаючи запит до `ProjectService`, який оновлює поле `lastModified` проекту, позначаючи зміни як несохранені. Таким чином, діаграма відображає повну взаємодію між користувачем, UI, сервісами, моделями об'єктів і проектом під час виконання операцій редагування шару.

На основі спроектованих діаграм розгортання та компонентів доопрацювано програмну частину системи.

Було створено `JpaBaseRepository.java`.

```

1 package com.example.graphiceditor.repository;
2
3 import com.example.graphiceditor.persistence.JpaUtil;
4 import jakarta.persistence.EntityManager; import jakarta.persistence.EntityTransaction;
5 import java.util.List; import java.util.Optional;
6
7 @public abstract class JpaBaseRepository<T, ID> implements BaseRepository<T, ID> { 3 usages 3 inheritors new
8     private final Class<T> type; 5 usages
9     protected JpaBaseRepository(Class<T> type) { this.type = type; } 3 usages new *
0
1 @↑ public T save(T e) { 5 usages new *
2     EntityManager em = JpaUtil.emf().createEntityManager();
3     EntityTransaction tx = em.getTransaction();
4     try { tx.begin(); T m = em.merge(e); tx.commit(); return m; }
5     finally { if (tx.isActive()) tx.rollback(); em.close(); }
6 }
7 @↑ public Optional<T> findById(ID id) { no usages new *
8     EntityManager em = JpaUtil.emf().createEntityManager();
9     try { return Optional.ofNullable(em.find(type, id)); }
0     finally { em.close(); }
1 }
2 @↑ public List<T> findAll(int offset, int limit) { no usages new *
3     EntityManager em = JpaUtil.emf().createEntityManager();
4     try {
5         return em.createQuery(s: "select e from " + type.getSimpleName() + " e", type)
6             .setFirstResult(offset).setMaxResults(limit).getResultList();
7     } finally { em.close(); }
8 }
9 @↑ public void deleteById(ID id) { no usages new *
0     EntityManager em = JpaUtil.emf().createEntityManager();
1     EntityTransaction tx = em.getTransaction();
2     try { tx.begin(); T ref = em.find(type, id); if (ref!=null) em.remove(ref); tx.commit(); }

```

Вихідний код UI частини:

FXML-файли для двох форм:

- login.fxml
- projects.fxml

Java-контролери:

```
1 package com.example.graphiceditor.ui;
2
3 import com.example.graphiceditor.model.User;
4 import com.example.graphiceditor.repository.UserRepository;
5 import com.example.graphiceditor.service.AuthService;
6 import javafx.fxml.FXML;
7 import javafx.scene.control.Label;
8 import javafx.scene.control.TextField;
9 import javafx.scene.Scene;
10 import javafx.fxml.FXMLLoader;
11 import javafx.stage.Stage;
12
13 public class LoginController { 1 usage new *
14
15     @FXML private TextField usernameField;
16     @FXML private Label errorLabel;
17
18     private final AuthService authService = new AuthService();  no usages
19     private final UserRepository userRepo = new UserRepository();  2 usages
20
21     @FXML new *
22     private void onLogin() {
23         String name = usernameField.getText() == null ? "" : usernameField.getText().trim();
24         if (name.isEmpty()) {
25             errorLabel.setText("Введіть ім'я користувача");
26             return;
27         }
28         try {
29             User u = userRepo.findByUsername(name).orElseGet(() -> {
30                 User nu = new User();
31                 nu.setUsername(name);
32                 nu.setPasswordHash("");      // для демо
33
34                 nu.setAuthorized(true);
35                 return userRepo.save(nu);
36             });
37
38             goToProjects(u);
39         } catch (Exception ex) {
40             errorLabel.setText("Помилка входу: " + ex.getMessage());
41         }
42
43     private void goToProjects(User current) throws Exception { 1 usage new *
44         FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/ui/projects.fxml"));
45         Scene scene = new Scene(loader.load());
46         ProjectsController ctrl = loader.getController();
47         ctrl.setCurrentUser(current);      // передаємо користувача
48         ctrl.initData();                  // завантаження таблиці
49
50         Stage stage = (Stage) usernameField.getScene().getWindow();
51         stage.setTitle("Projects - " + current.getUsername());
52         stage.setScene(scene);
53         stage.centerOnScreen();
54     }
55 }
```

```

package com.example.graphiceditor.ui;

import com.example.graphiceditor.model.Project;
import com.example.graphiceditor.model.User;
import com.example.graphiceditor.repository.ProjectRepository;
import com.example.graphiceditor.service.EditorService;
import javafx.collections.FXCollections;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.util.StringConverter;

import java.time.format.DateTimeFormatter;
import java.util.List;

public class ProjectsController { 2 usages new *

</>    @FXML private TableView<Project> table;
</>    @FXML private TableColumn<Project, String> nameCol;
</>    @FXML private TableColumn<Project, String> sizeCol;
</>    @FXML private TableColumn<Project, String> modifiedCol;
</>    @FXML private Label statusLabel;

private final ProjectRepository projectRepo = new ProjectRepository(); 1 usage
private final EditorService editor = new EditorService(); 1 usage

private User currentUser; 4 usages

public void setCurrentUser(User u) { this.currentUser = u; } 1 usage new *
```

```

@FXML new *
private void initialize() {
    DateTimeFormatter fmt = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    nameCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
        .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() ->
            cd.getValue().getName())));

    sizeCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
        .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() -> {
            var doc = cd.getValue().getDocument();
            return (doc == null) ? "-" : (doc.getWidth() + " x " + doc.getHeight());
        })));

    modifiedCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
        .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() ->
            cd.getValue().getLastModified() == null ? "-" : fmt.format(cd.getValue().getLastModified())))
    })
}
```

```
public void initData() { 1 usage new *
    refresh();
}

@FXML 3 usages new *
public void refresh() {
    if (currentUser == null) return;
    List<Project> list = projectRepo.findById(currentUser.getId());
    table.setItems(FXCollections.observableList(list));
    statusLabel.setText("Loaded " + list.size() + " projects");
}

@FXML new *
private void onNewProject() {
    Dialog<ProjectParams> dlg = buildNewProjectDialog();
    dlg.showAndWait().ifPresent(params -> {
        try {
            Project p = editor.createProjectWithDocument(currentUser, params.name, params.w, params.h);
            statusLabel.setText("Created project: " + p.getName());
            refresh(); // перечитати з БД
        } catch (Exception ex) {
            new Alert(Alert.AlertType.ERROR, s: "Помилка створення: " + ex.getMessage()).showAndWait();
        }
    });
}
}
```

```
@FXML new *
private void onOpenProject() {
    Project sel = table.getSelectionModel().getSelectedItem();
    if (sel == null) { new Alert(Alert.AlertType.INFORMATION, s: "Оберіть проект").showAndWait(); return; }
    new Alert(Alert.AlertType.INFORMATION, s: "Відкриваємо: " + sel.getName()).showAndWait();
}

@FXML new *
private void onDeleteProject() {
    Project sel = table.getSelectionModel().getSelectedItem();
    if (sel == null) { new Alert(Alert.AlertType.INFORMATION, s: "Оберіть проект").showAndWait(); return; }
    new Alert(Alert.AlertType.INFORMATION, s: "(демо) Видалення ще не реалізовано").showAndWait();
}
```

```
private Dialog<ProjectParams> buildNewProjectDialog() { 1 usage  new *
    Dialog<ProjectParams> d = new Dialog<>();
    d.setTitle("New Project");

    Label nameL = new Label(s: "Name:");
    TextField nameF = new TextField(s: "My Project");

    Label wL = new Label(s: "Width:");
    Spinner<Integer> wSp = new Spinner<>(i: 1, i1: 10000, i2: 1280);

    Label hL = new Label(s: "Height:");
    Spinner<Integer> hSp = new Spinner<>(i: 1, i1: 10000, i2: 720);

    GridPane gp = new GridPane();
    gp.setHgap(10); gp.setVgap(10);
    gp.addRow(i: 0, nameL, nameF);
    gp.addRow(i: 1, wL, wSp);
    gp.addRow(i: 2, hL, hSp);
    d.getDialogPane().setContent(gp);

    d.getDialogPane().getButtonTypes().addAll(ButtonType.OK, ButtonType.CANCEL);
    d.setResultConverter(bt -> bt == ButtonType.OK
        ? new ProjectParams(nameF.getText(), wSp.getValue(), hSp.getValue())
        : null);
    return d;
}

private record ProjectParams(String name, int w, int h) {} 4 usages  new *
}
```

```
1 package com.example.graphiceditor.ui;
2
3 import javafx.fxml.FXMLLoader;
4 import javafx.scene.Parent;
5 import javafx.scene.Scene;
6 import javafx.stage.Stage;
7
8 public class ViewRouter { 4 usages new*
9
10    private static Stage primaryStage; 7 usages
11
12    public static void setPrimaryStage(Stage stage) { 1 usage new*
13        primaryStage = stage;
14    }
15
16    public static void show(String fxmlPath, String title) { 1 usage new*
17        try {
18            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPa
19            Parent root = loader.load();
20
21            primaryStage.setTitle(title);
22            primaryStage.setScene(new Scene(root));
23            primaryStage.show();
24        } catch (Exception ex) {
25            ex.printStackTrace();
26            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
27        }
28    }

```

```
28    }
29
30    public static <T> T showAndGetController(String fxmlPath, String title) { no usages new*
31        try {
32            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPath));
33            Parent root = loader.load();
34
35            primaryStage.setTitle(title);
36            primaryStage.setScene(new Scene(root));
37            primaryStage.show();
38
39            return loader.getController();
40        } catch (Exception ex) {
41            ex.printStackTrace();
42            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
43        }
44    }
45 }
```

Висновки:

У ході виконання лабораторної роботи було реалізовано повноцінну програмну частину клієнтської частини системи GraphicEditor, яка включає роботу з даними, реалізацію бізнес-логіки, репозиторіїв, сервісів та побудову графічного інтерфейсу через JavaFX.

Було створено та налаштовано архітектуру застосунку, що охоплює всі рівні: модель, сховище даних (JPA/Hibernate), сервер логіки (сервіси) та користувачький інтерфейс. Застосунок забезпечує повний цикл роботи з даними: створення користувача, створення проекту з графічним документом, додавання шарів, збереження цих даних у базі, а також подальше їх отримання та відображення в UI.

Було реалізовано дві візуальні форми відповідно до вимог:

- LoginForm для входу користувача;
- ProjectsForm для перегляду, створення та керування проектами авторизованого користувача.

Для навігації між формами створено компонент ViewRouter, який інкапсулює логіку перемикання сцен та дозволяє контролерам відкривати нові вікна без дублювання коду.

Усі діаграми — розгортання, компонентів, послідовностей — використані як основа для побудови системи, а структура проекту відповідає сучасним принципам модульності, перевикористання та розділення відповільностей.

Реалізована система демонструє коректну взаємодію UI з бізнес-логікою та нормальну функціонування як повноцінний застосунок.

Контрольні питання:

- 1.**Діаграма розгортання — це UML-діаграма, яка відображає фізичне розміщення програмних компонентів системи на апаратних та програмних вузлах. Вона показує, на яких пристроях працює система, які середовища виконання використовуються, а також як ці вузли з'єднані між собою.
- 2.**На діаграмі розгортання розрізняють два основні види вузлів: апаратні вузли (наприклад, ПК, сервер, мобільний пристрій) та програмні або віртуальні вузли (контейнери виконання, операційні системи, JVM, Docker-контейнери).
- 3.**На діаграмі розгортання використовують два основні типи зв'язків: комунікаційні зв'язки, які показують мережеву взаємодію між вузлами, та зв'язки розгортання, які відображають, які компоненти встановлені або працюють на конкретному вузлі.
- 4.**На діаграмі компонентів присутні самі компоненти системи, їхні інтерфейси, реалізації, залежності, а також допоміжні артефакти, бібліотеки чи модулі, від яких залежить система.
- 5.**Зв'язки на діаграмі компонентів представляють собою залежності між компонентами. Вони показують, який компонент використовує інший, які інтерфейси надаються або вимагаються, а також схему взаємодії між частинами системи.
- 6.**До діаграм взаємодії у UML належать діаграми послідовностей, діаграми комунікації (колаборації), діаграми часу та оглядові діаграми взаємодії.

7.Діаграма послідовностей призначена для відображення порядку взаємодії між об'єктами у часі. Вона показує, які об'єкти беруть участь у виконанні сценарію та які повідомлення надсилаються між ними.

8.На діаграмі послідовностей можуть бути такі ключові елементи: об'єкти та їхні життєві лінії, повідомлення між об'єктами, повернення результатів, альтернативні гілки, цикли, опціональні блоки, а також області активності.

9.Діаграми послідовностей тісно пов'язані з діаграмами варіантів використання, оскільки для кожного варіанта використання зазвичай створюється відповідна діаграма послідовностей. Вона деталізує, як саме реалізується конкретний сценарій, описаний у діаграмі варіантів використання.

10.Діаграми послідовностей пов'язані з діаграмами класів тим, що об'єкти, які фігурують у діаграмах послідовностей, є екземплярами класів із діаграмами класів. Виклики методів у діаграмі послідовностей підтверджують наявність цих методів у класах, а взаємодія між об'єктами відповідає зв'язкам між класами.