КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

РОЗРОБКА СИСТЕМИ ПІДТРИМКИ КОМУНІКАЦІЙ У НАВЧАЛЬНОМУ ПІДРОЗДІЛІ З МОБІЛЬНИМ КЛІЄНТОМ

Виконала студентка 4-го курсу:
Полянська Анастасія Вадимівна

(підпис)

Науковий керівник:
доцент, кандидат технічних наук
Ткаченко Олексій Миколайович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«____» _____ 202_ р.,
протокол № ___
Завідувач кафедри
Нікітченко М.С.

(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 45 сторінок, 15 ілюстрацій, 1 таблиця, 14 джерел посилань.
БАЗА ДАНИХ, МОБІЛЬНИЙ КЛІЄНТ, СЕРВЕР, СИСТЕМА ПІДТРИМКИ КОМУНІКАЦІЙ, ANDROID, SPRING.

Об’єкт дослідження: система комунікацій у навчальному закладі. Предмет: програмні системи підтримки комунікацій з мобільним клієнтом.

Мета роботи: розробка сервера та мобільного клієнта для підтримки комунікацій між викладачами та студентами.

Методи розроблення: системний аналіз, об’єктно-орієнтований аналіз, об’єктно-орієнтована та алгоритмічна декомпозиція, повторного використання коду (reuse), архітектурний підхід Model-View-Controller (MVC), метод порівняння. Інструменти розроблення: Android Studio – інтегроване середовище розробки (IDE) для платформи Android, IntelliJ IDEA – інтегроване середовище розробки програмного забезпечення, MySQL – вільна система керування реляційними базами даних, Spring – універсальний фреймворк для Java-платформи, мова програмування Java, Heroku – хмарна платформа, Postman та Swagger – інструмент тестування API.

Результати та їх новизна: виконано загальний огляд програм, які активно використовуються для дистанційного навчання, та створено клієнт-серверну систему підтримки комунікацій та навчального процесу з мобільним додатком. Запропоновано рекомендації щодо використання результатів роботи у навчальних закладах, у тому числі, за допомогою створення груп з відокремленим чатом.

Рекомендації щодо використання результатів роботи: додаток можна використовувати в навчальних закладах шляхом створення довільної кількості груп з чатом на кожну з них.

Висновки та пропозиції щодо розвитку об’єкта розроблення й доцільності продовження розробок: виконано аналіз подібних систем, огляд використаних технологій та інструментарію. Було розроблено систему підтримки комунікації у навчальних підрозділах з мобільним клієнтом. В майбутньому при подальшій розробці програмна система має варіанти розширення:

- додавання вкладки на головний екран актуального розкладу з можливістю переходити на відповідну відеоконференцію вибраного предмету та можливістю переходити в чат курсу безпосередньо через розклад;
- додавання інтеграції з системою КНУ імені Тараса Шевченка - Triton.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ОС – операційна система;

БД – база даних;

UML – Unified Modeling Language, уніфікована мова моделювання;

МК – мобільний клієнт;

SQL – Structured Query Language, мова структурованих запитів;

JSON – JavaScript Object Notation, текстовий формат обміну даними;

ORM – (Object-Relational Mapping) технологія, що пов’язує фізичну базу даних з базою, яка створюється віртуально під час виконання програми;

JPA – (Java Persistence API) технологія, яка реалізовує взаємодію з об’єктами фізичної бази даних;

СКБД – система керування базами даних;

MVC – (Model-View-Controller) шаблон архітектури програми;

URL – (Uniform Resource Locator) стандартизована адреса певного ресурсу;

HTML – (HyperText Markup Language) мова розмітки сторінки;

REST – (Representational State Transfer) підхід до архітектури мережевих протоколів;

XML – (eXtensible Markup Language) мова розмітки сторінки;

POJO – (Plain Old Java Object) простий Java об’єкт, який не наслідується від інших об’єктів, має поля та методи які працюють лише з полями даного класу;

DI – (Dependency injection) процесс надання зовнішньої залежності програмному компоненту;

DTO – (Data Transfer Object) шаблон проектування, який використовується для передачі даних;

API – (Application Programming Interface) програмний інтерфейс програми;

HTTP – (HyperText Transfer Protocol) протокол передачі даних.

ВСТУП

Оцінка сучасного стану об’єкта розробки: на сьогодні існує широкий спектр систем підтримки комунікацій, у тому числі і з реалізацією мобільного клієнта, та велика кількість інструментарію для розробки подібних систем. Найпоширенішими системами, які використовуються для підтримки навчального процесу на кафедрі ТТП, є Google Class, Zoom, Discord, Google Meet, Telegram. Завданням було створити подібну систему, яка б допомогла підтримувати комунікацію між викладачами та студентами.

Актуальність роботи та підстави для її виконання: перехід багатьох сфер діяльності в цифровий формат і використання онлайн-комунікацій є актуальним стійким трендом сучасного суспільства, зокрема, у сфері освіти. Особливо це затребувано в період пандемії.

Таблиця – Переваги та недоліки існуючих систем

Системи для підтримки комунікації	Переваги	Недоліки
Google Class	реалізує розбиття по курсам, викладення нових завдань	відсутня можливість створення чату
Zoom, Discord, Google Meet	забезпечують навчальний процес відеодзвінками	чат існує лише спільний для всіх і на одну сесію
Telegram	дозволяє створити безліч чатів з довільною кількістю користувачів	система занадто широкого спектру застосування

Найбільшою проблемою є відсутність єдиної навчальної системи, бо кожен викладач обирає для себе найбільш зручний спосіб взаємодії зі студентами. І тому у студента є декілька навчальних чатів у Telegram, декілька курсів у Google Class та безліч листів на пошті.

Мета й завдання роботи: створення системи для підтримки комунікації в навчальних підрозділах, за допомогою якої вчителі та студенти мали б доступ до актуальної інформації та могли оперативно комунікувати. Для досягнення цієї мети було поставлено наступні **задачі**:

- дослідити предметну область;
- оцінити переваги та недоліки онлайн-систем підтримки комунікацій;
- сформувати функціональні вимоги;
- спроектувати та розробити сервер;
- створити мобільний клієнт.

Об’єкт, методи й засоби розроблення: перед розробкою програм був проведений аналіз існуючих систем, що остаточно допомогло визначитися з функціоналом. Для розробки як серверу, так і мобільного клієнта було обрано мову Java, яка є строго типізованою та об’єктно-орієнтованою. Сервер розроблявся в інтегрованому середовищі IntelliJ IDEA, з використанням Spring фреймворку. Тестування API відбувалося за допомогою таких інструментів як Postman та Swagger. Мобільний додаток був створений в інтегрованому середовищі Android Studio.

Можливі сфери застосування: розроблена система рекомендується до використання у навчальних підрозділах.

РОЗДІЛ 1. ПРОЕКТУВАННЯ СИСТЕМИ

1.1 Перелік функціоналу системи відносно ролей

Система підтримує 3 ролі: Студент, Вчитель, Адміністратор. Кожна роль відрізняється набором функцій та рівнем доступу до методів.

Функціонал Студента:

- перегляд списку доступних чатів;
- перегляд повідомлень у чаті;
- перегляд учасників чату;
- написання повідомлення.

Функціонал Вчителя:

- перегляд списку доступних чатів;
- перегляд повідомлень у чаті;
- перегляд учасників чату;
- написання повідомлення;
- додавання посилання на заняття в інформацію про чат;
- додавання студентів в чат;
- додавання викладачів в чат;
- створення чату (для занять, курсових, дипломних, групових проектів);
- перегляд всіх користувачів системи (для додавання учасників до існуючого чату або до нового).

Функціонал Адміністратора:

- додавання нового користувача до системи;
- видалення користувачів із системи;
- створення/видалення чату;
- редагування навчальної групи студентів;

- створення/видалення навчальної групи (ТТП-3, ТТП-41);
- перезапуск (розвантаження) БД- видалення навчальних чатів, крім групових (ТТП-3), видалення всіх повідомлень.

1.2 Моделювання предметної області

Для кращого розуміння будови та принципу роботи системи, нижче наведенні UML діаграми та макет інтерфейсу користувача мобільного додатку для кожної з ролей.

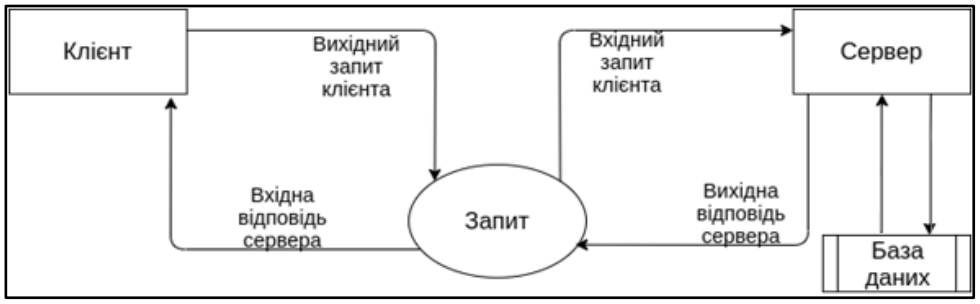

Рисунок 1.1 – Діаграма потоків даних

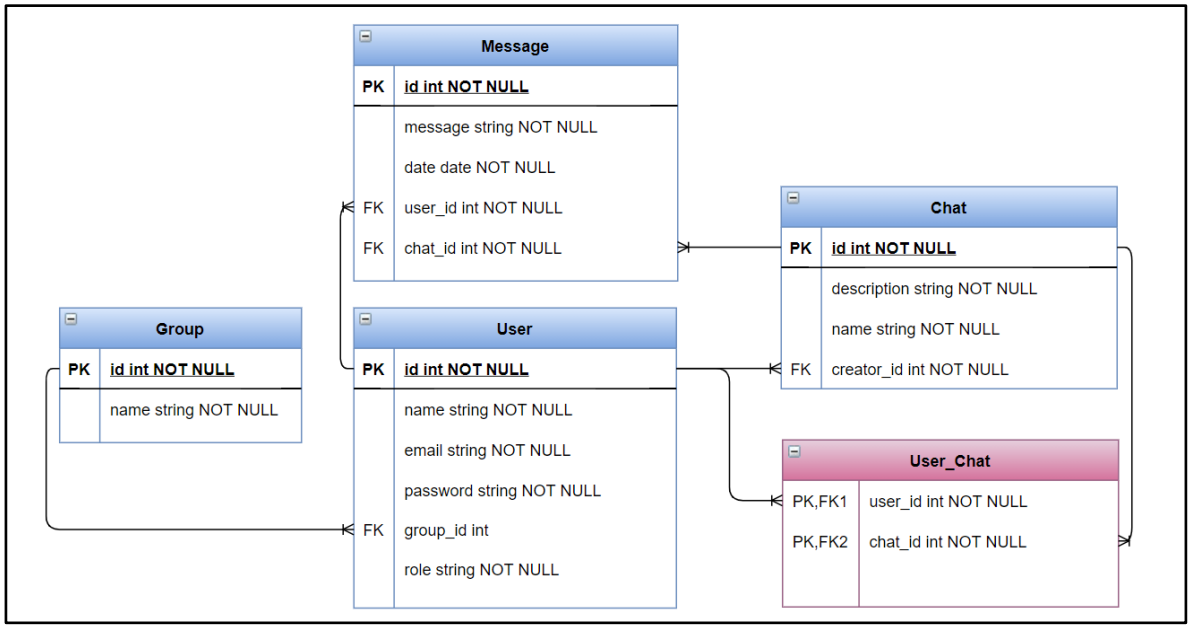

Рисунок 1.2 – Діаграма класів

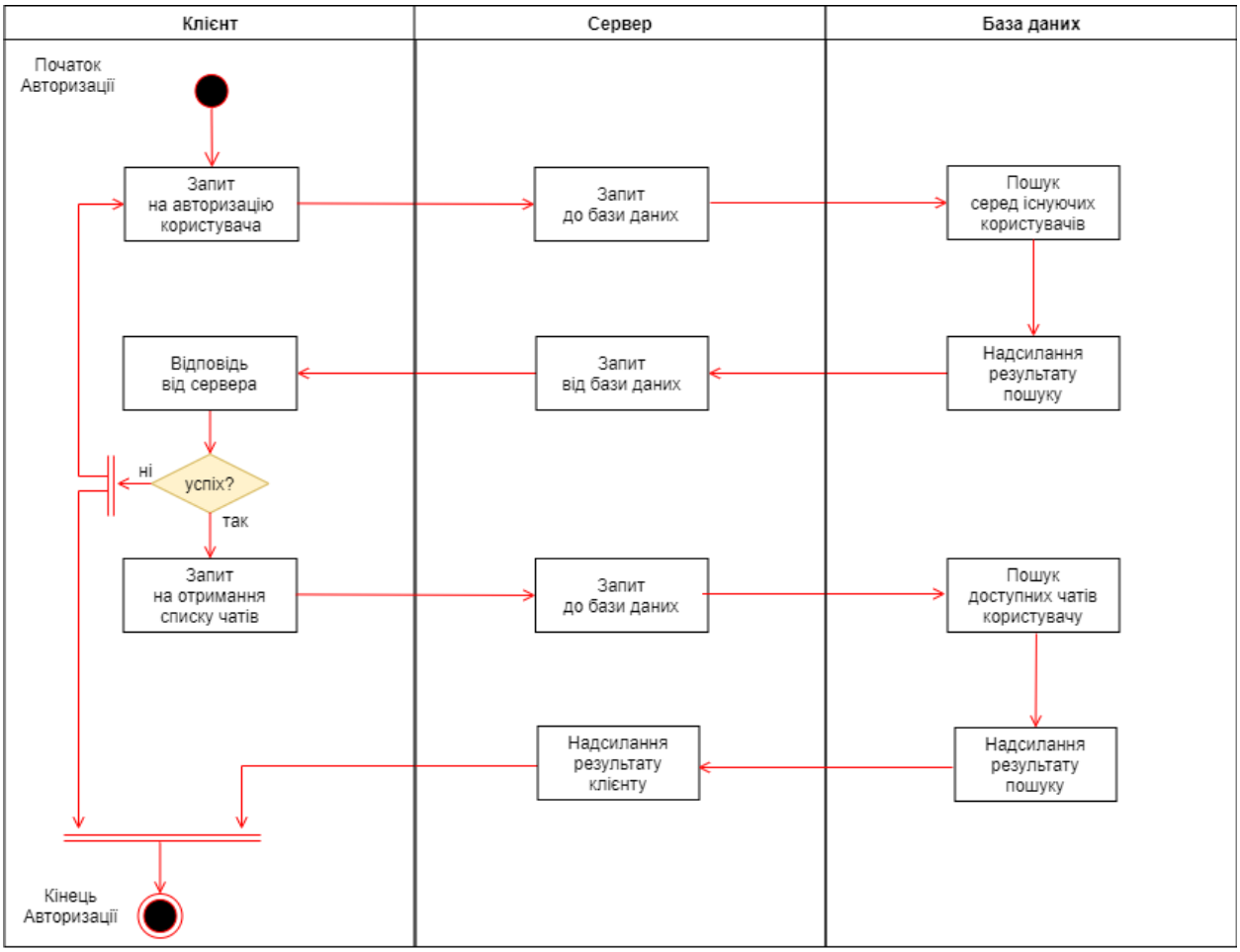

Рисунок 1.3 – Діаграма активності

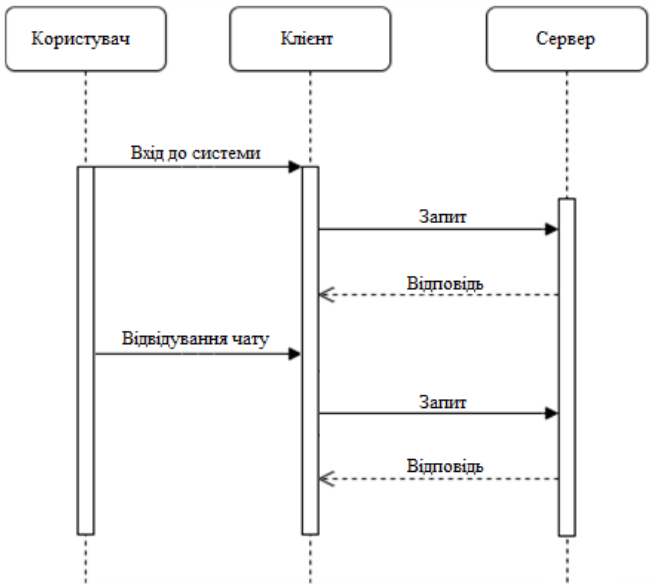

Рисунок 1.4 – Діаграма послідовності

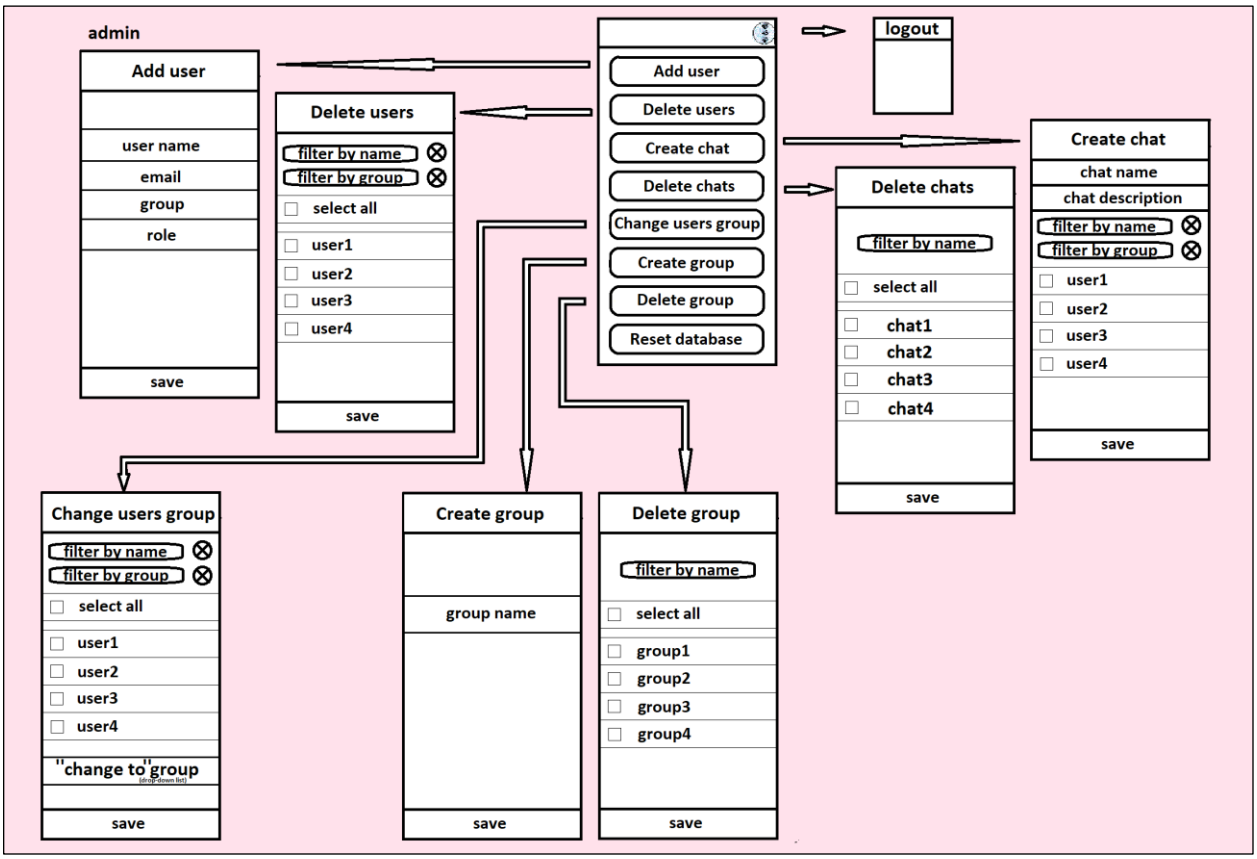

Рисунок 1.5 – Макет мобільного клієнта для адміністратора

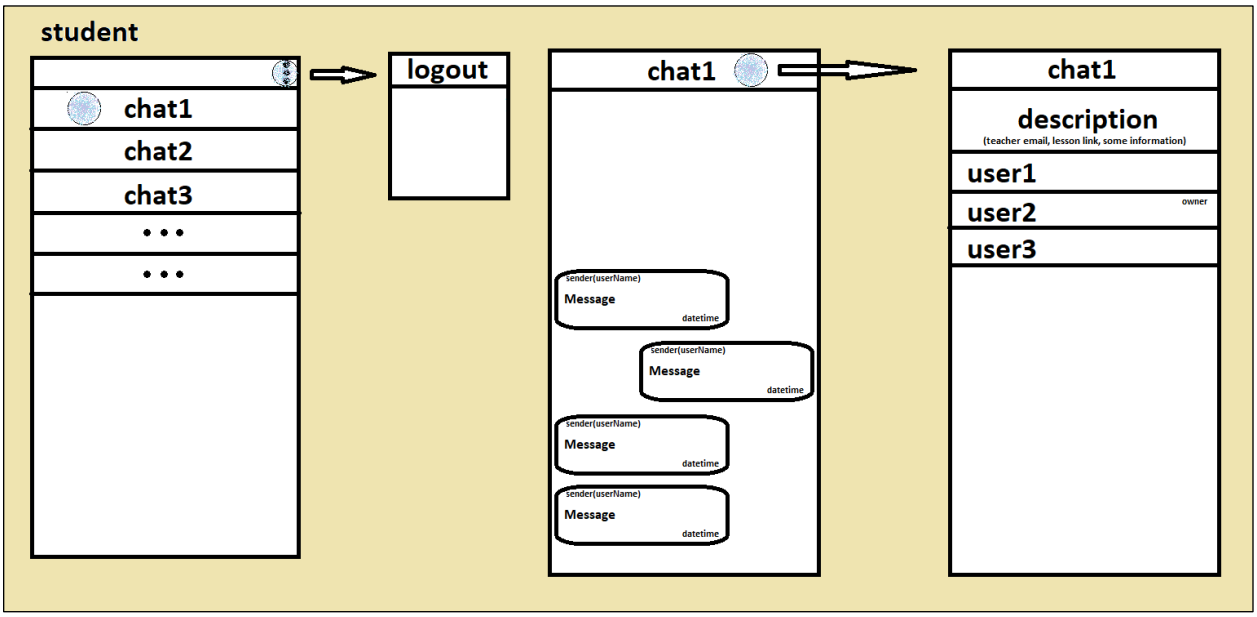

Рисунок 1.6 – Макет мобільного клієнта для студента

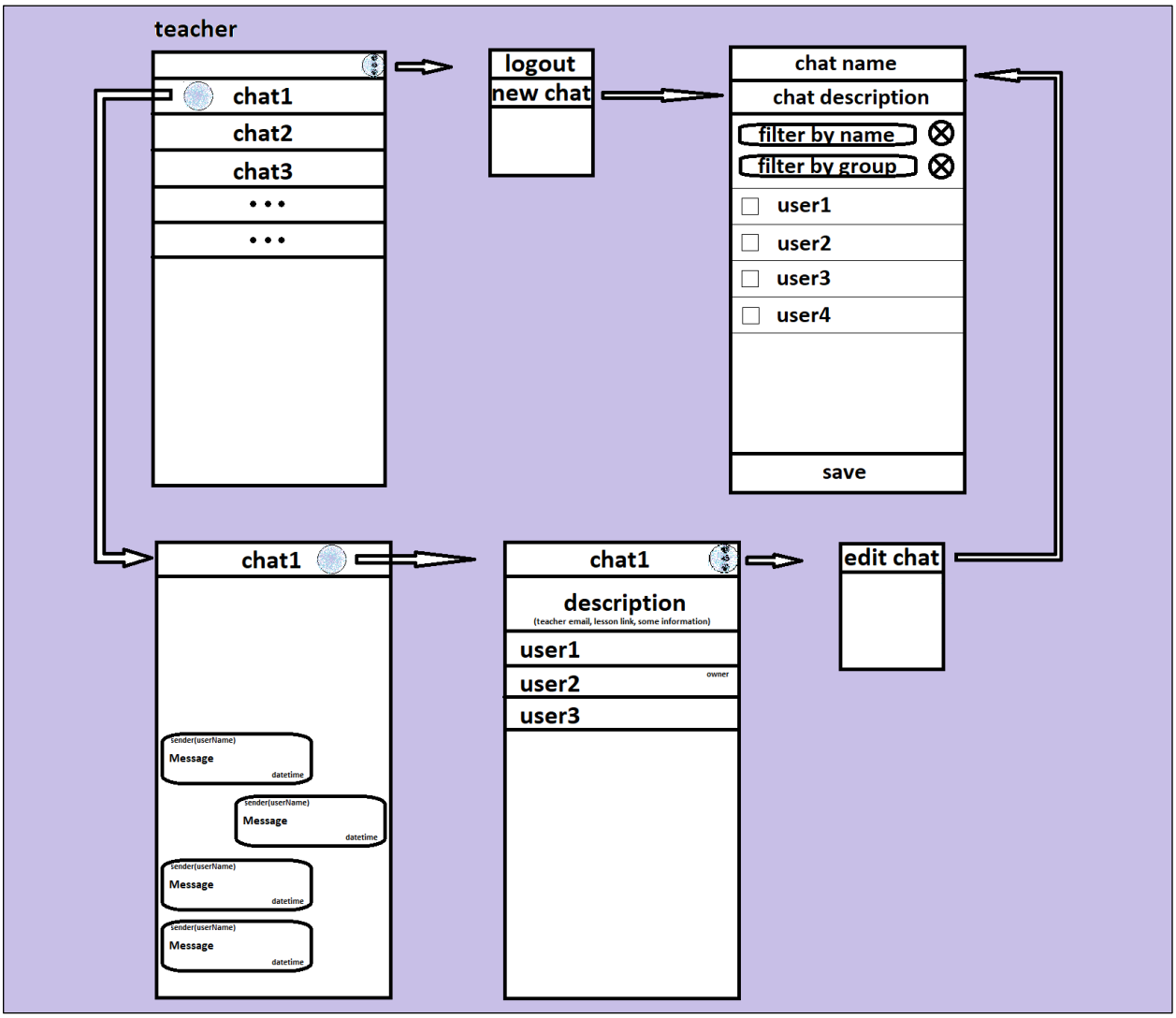

Рисунок 1.7 – Макет мобільного клієнта для вчителя

РОЗДІЛ 2. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ

2.1 Взаємодія з Базою даних

2.1.1 ORM

ORM (Object-Relational Mapping) - це технологія, що пов’язує фізичну БД з базою даних, яка створюється віртуально під час виконання програми. ORM дає змогу розробнику не писати велику кількість коду, часто одноманітного, де легко можна помилитися, для зв’язку з БД, запитами до таблиць, тим самим значно підвищуючи швидкість та якість розробки. Крім того, більшість сучасних реалізацій ORM дозволяють програмісту при необхідності самому прописати власні SQL-запити, які будуть використовуватися при тих чи інших діях (збереження в БД, видалення, пошук і т.д.).

2.1.2 JPA Repositories

JPA (Java Persistence API) - це технологія, яка реалізує взаємодію з об’єктами фізичної БД. [1] За допомогою неї в проекті можна швидко створити репозиторії в основному призначені для CRUD операцій:

- Create (створення);
- Read (читання);
- Update (оновлення);
- Delete (видалення).

При роботі з БД практично будь-яка програма має підтримувати ці операції. У базі даних MySQL цим операціям відповідають запити:

- INSERT (вставити);
- SELECT (вибрати);
- UPDATE (оновити);

- DELETE (видалити).

Якщо не вистачає стандартних запитів, то можна додати власні за допомогою анотації @Query. JPA модуль підключається шляхом додавання залежності spring-boot-starter-data-jpa до файлу конфігурації.

2.1.3 Hibernate

Hibernate - це бібліотека, яка реалізує технологію Object-Relational Mapping. Вона бере на себе задачу перетворення даних реляційного виду в об’єктний - для читання, і з об’єктного виду в реляційний - для запису. Крім того, бібліотека дозволяє легко налаштовувати підключення до СКБД і легко керувати транзакціями. [2]

Принцип роботи:

1. **Інтерфейс Session**

   У Hibernate робота з БД здійснюється через об’єкт типу org.hibernate.Session, який є мостом між програмою і Hibernate. Життєвий цикл сеансу обмежений початком і кінцем логічної транзакції. За допомогою сесій виконуються всі CRUD-операції з об’єктами-сутностями. Об’єкт типу Session отримують з екземпляра org.hibernate.SessionFactory, який повинен бути присутнім в програмі у вигляді singleton.

2. **Стани об’єктів**

   Об’єкт-сутність може перебувати в одному з 3-х станів:

   - Transient object. Об’єкт в даному статусі - це заповнений екземпляр класу-сутності. Може бути збережений до БД або не приєднаний до сесії. Поле Id має бути незаповненим, інакше об’єкт має статус detached;
   - Persistent object. Об’єкт в даному статусі - так звана збережена сутність, яка приєднана до конкретної сесії. Тільки в цьому статусі

об’єкт взаємодіє з базою даних. При роботі з об’єктом даного типу всі зміни об’єкта записуються в БД;

- Detached object. Об’єкт в даному статусі - це об’єкт, від’єднаний від сесії, який може існувати або не існувати в БД.

Будь-який об’єкт можна переводити з одного стану в інший через методи інтерфейсу Session.

3. **Експорт об’єктів з бази даних**

   Метод session.load() повертає proxy-object - це об’єкт-посередник, через який можна взаємодіяти з реальним об’єктом в базі даних. Він розширює функціонал об’єкта-сутності. Взаємодія з proxy-object повністю аналогічна взаємодії з об’єктом-сутністю. Proxy-object відрізняється від об’єкта-сутності тим, що при створенні proxy-object не виконується жодного запиту до БД. Однак перший викликаний get() або set() у proxy-object відразу ініціює запит select, і якщо об’єкта з даними Id немає в базі, то ми отримаємо ObjectNotFoundException. Основне призначення proxy-object - реалізація відкладеного завантаження.

4. **Збереження об’єктів**

   Збереження в базу даних можна робити тільки в рамках транзакції. Виклик session.openTransaction() відкриває для даної сесії нову транзакцію, а session.getTransaction().commit() її виконує. Якщо всередині транзакції щось змінимо в об’єкті статусу persistent або proxy, то виконається запит update.

2.1.4 Spring Data

Spring Data - механізм для взаємодії з сутностями бази даних, організації їх в репозиторії, вилучення та редагування даних. В деяких випадках для цього буде достатньо оголосити інтерфейс і методи в ньому, без імплементації.

Репозиторій - це основне поняття в Spring Data, що представляє собою кілька інтерфейсів, які використовують JPA Entity. В Spring Data існує два типи репозиторіїв, від яких можна наслідувати свої:

- CrudRepository;
- JpaRepository.

Перший реалізує лише CRUD операції, другий наслідує перший та розширює свій функціонал сортуванням, динамічним розбиттям на сторінки з можливістю вказувати кількість елементів на сторінку, можливість збереження виконаних операцій в БД до завершення поточної транзакції (при налаштуваннях транзакції за умовчанням), можливістю збереження даних до бази одним пакетом (batch).

JPA Entity - це сутність, що представляє таблицю, яка зберігається в базі даних. Кожен екземпляр сутності є рядком у таблиці. Клас Entity є віртуальним представленням об'єкта з БД. Він має всі поля таких же типів що і таблиця з фізичної бази даних. Обов'язково мають бути анотації @Setter, @Getter через які ORM взаємодіє з JPA Entity класом. [1]

Також до класу Entity додаються наступні анотації:

- @Entity - вказує, що поточний клас представляє тип сутності Entity класу та слугує для взаємодії з об'єктами бази даних. (атрибут є обов'язковим);
- @Table - вказує з якою таблицею в базі даних зв'язується клас;
- @Id - вказує ідентифікатор об'єкта. (атрибут є обов'язковим);
- @GeneratedValue - вказує метод генерації значення ідентифікатора;
- @Column - вказує відповідність між атрибутом базової сутності Entity класу і стовпцем таблиці бази даних;
- @Enumerated - вказує, що атрибут entity представляє перелічувальний тип;
- @JoinColumn - вказує, що в даній колонці міститься ForeignKey іншої таблиці;

- @JoinTable - вказує на таблицю зв’язків між двома іншими таблицями бази даних;
- @OneToMany - вказує на відносини «один-до-багатьох» для об’єктів бази даних;
- @ManyToOne - вказує на відносини «багато-до-одного» для об’єктів бази даних;
- @ManyToMany - вказує на відносини «багато-до-багатьох» для об’єктів бази даних.

2.1.5 MySQL

MySQL - це реляційна система керування базами даних з відкритим вихідним кодом. MySQL є однією з найбільш популярних у використанні для розробок веб-програмах. Майже всі веб-фреймворки підтримують її на рівні базової конфігурації (без додаткових модулів). [3]

З переваг MySQL варто відзначити простоту використання, гнучкість, низьку вартість (щодо платних СКБД), та продуктивність. На відміну від MSSQL, розгортання MySQL на Heroku є безкоштовним.

MySQL дозволяє зберігати цілочисельні типи зі знаком і без, довжиною в 1, 2, 3, 4 і 8 байтів, працює з рядковими і текстовими даними фіксованої і змінної довжини, дозволяє здійснювати SQL-команди SELECT, DELETE, INSERT, REPLACE і UPDATE, забезпечує повну підтримку операторів і функцій в SELECT і WHERE частинах запитів, працює з GROUP BY і ORDER BY, підтримує групові функції COUNT(), AVG(), STD(), SUM(), MAX() і MIN(), дозволяє використовувати JOIN в запитах, в т.ч. LEFT і RIGHT OUTER JOIN, підтримує реплікацію, транзакції, роботу з зовнішніми ключами і каскадні зміни на їх основі.

2.2 Контролери

Контролери - спеціальні класи, які обробляють запит з клієнта. Розглянемо концепцію контролера в типовій архітектурі Spring MVC.

Основні обов'язки контролера:

- перехоплювати вхідні запити;
- приводити запит до вигляду, з яким може взаємодіяти програма;
- надсилати дані в Model для подальшої обробки;
- отримувати оброблені дані з Model та надсилати ці дані клієнту.

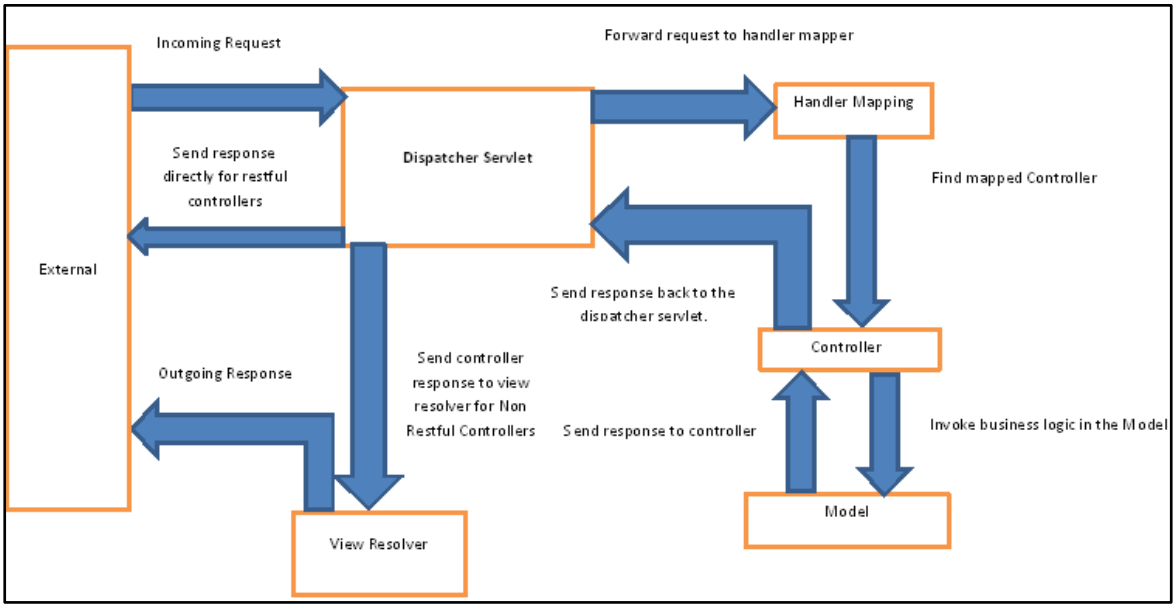

Рисунок 2.1 – Діаграма Spring MVC

1. Спершу на стороні клієнта формується запит та надсилається на сервер;
2. Всі вхідні запити потрапляють до Dispatcher Servlet, який передає їх до Handler Mapping;
3. Handler Mapping дивиться на URL, який записаний у запиті, відкидає від нього baseUrl та шукає в спеціальному словнику ендпойнт по всім контролерам;

4. Controller викликає відповідний до кінцевої точки метод, який реалізує певну бізнес логіку в моделі. На цьому етапі відбуваються запити до БД, певні обрахунки та інше;
5. Model, виконавши певні дії для обробки вхідного запиту, формує результуючий запит та передає його назад контролеру;
6. Controller надсилає запит до Dispatcher Servlet;
7. Далі можливі 2 варіанти:
   7.1. результуючий запит надсилається на RESTfull контролер клієнта;
   7.2. результуючий запит надсилається на View Resolver, для визначення потрібної HTML-сторінки, її створення (рендеру), а потім вже сформована сторінка надсилається клієнту.

На діаграмі контролером слід вважати Dispatcher Servlet, Handler Mapping та Controller (Рисунок 2.1). Діаграма представляє роботу як типових контролерів MVC, так і контролерів RESTful. [4]

У Spring існує 2 анотації для позначення різних типів контролерів:

- @Controller
- @RestController

Крім цього, клас контролера має містити анотацію @RequestMapping, в яку передається URL класу. Всі методи класу, які містять різновиди анотації @Mapping з вказаним URL, називаються ендпойнтами. Їхні URL додаються до URL класу, коли реєструються фреймворком. Існує 4 види анотації @Mapping для методів:

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping

Клас з анотацією @Controller може бути як звичайним MVC контролером, так і RESTful контролером. Звичайний контролер має або приймати

параметром об’єкт Model (ModelAndView) та вертати назву HTML-сторінки, або вертати ModelAndView. Розглянемо другий варіант. [4]

```
@Controller
@RequestMapping(value = "/test")
public class TestController {
	@GetMapping
	public ModelAndView getTestData() {
		ModelAndView mv = new ModelAndView();
		mv.setViewName("welcome");
		mv.getModel().put("data", "Welcome home man");
		return mv; }}
```

Тут контролер має URL «/test», а в метода URL відсутній, це означає, що він має такий самий URL як і клас. В методі створюється новий об’єкт ModelAndView, якому обов’язково має бути передана назва сторінки через метод setViewName(). Далі в модель (яка є просто словником) передаються дані у вигляді «ключ – значення». Ключ повинен називатись так само як відповідний параметр у заданому view.

RESTful контролер відрізняється від звичайного контролера тим, що він не має view та model. Всі методи цього контролера повертають необроблені дані, в основному у вигляді XML або JSON об'єктів, які записуються напряму в тіло відповіді HTTP. Для коректної роботи фреймворка ці об’єкти повинні мати анотацію @ResponseBody або повертати об’єкт шаблонного типу ResponseEntity. Анотація @ResponseBody вказує фреймворку, щоб обійти View Resolver і записати вихідні дані безпосередньо в тіло відповіді HTTP.

Розглянемо приклад такого класу:

```
@Controller
@RequestMapping(value = "/student")
public class RestController {
	@GetMapping(value = "{studentId}")
```

```
public @ResponseBody Student getTestData(@PathVariable Integer studentId){
        Student student = new Student();
        student.setName("Peter");
        student.setId(studentId);
        return student;}}
```

В цьому класі теж наявні вже описані анотації @Controller, @RequestMapping та @GetMapping. Метод приймає параметр studentId типу Integer, який має анотацію @PathVariable, яка буде описана далі. В тілі методу створюється простий Java-об’єкт (POJO) [5], який і повертається.

У випадку коли повертається ResponseEntity потрібно вказати тип об’єкта, який передається в ResponseEntity.

Приклад:

```
@Controller
@RequestMapping("/message")
public class MessageController {
    @Autowired
    private MessageMapper messageMapper;
    @Autowired
    private MessageService messageService;
    @GetMapping("{id}")
    ResponseEntity<MessageDto>
    getById(@PathVariable(name = "id")Integer id){
    return new ResponseEntity<>
    (messageMapper.mapToDomain(messageService.getById(id)),
    HttpStatus.OK); }}
```

Клас з анотацією @RestContoller являється, по суті, тим же RESTful контролером, який описаний вище, лише з тією різницею, що відсутня необхідність у додаванні анотації @ResponseBody до об’єкта, який повертається методом. [6]

Так виглядає попередній контролер з даною анотацією:

```
@RestController
@RequestMapping(value = "/student")
public class RestController { @GetMapping(value = "{studentId}")
    public Student getTestData(@PathVariable Integer studentId) {
        Student student = new Student();
        student.setName("Peter");
        student.setId(studentId);
        return student; }}
```

Для системи підтримки комунікації було обрано варіант з використанням RESTful контролерів, створених із анотацією @RestController. Всього в системі наявно 6 контролерів:

- AuthenticationController – керує операцією входу користувача
- ChatController – містить всі ендпойнти, пов’язані з чатами
- GroupController – містить всі ендпойнти, пов’язані з навчальними групами користувачів (наприклад, ТТП-3, ТТП-4)
- MessageController – містить всі ендпойнти, пов’язані з надісланими повідомленнями в чатах
- RestartDBController – містить ендпойнти, які запускають процес перевантаження бази даних (видалення тимчасових чатів, очищення постійних)
- UserController – містить всі ендпойнти, пов’язані з користувачами

Кожен контролер реалізований з дотриманням шаблону MVC, тому містить об’єкт відповідного сервісу, в якому реалізована логіка програми. Після отримання запиту контролер відразу передає його сервісу і отриману від нього відповідь відправляє клієнту. Для забезпечення шаблону DI (Dependency Injection) використовується анотація @Autowired, а поле об’єкта сервіса має тип інтерфейсу.

2.2.1 DTO

Згідно з принципом MVC (якого вимагає Spring) контролери не повинні працювати напряму з моделлю. Тобто у нашому випадку, контролери не можуть обробляти класи Entity. Для цього існує спеціальний шаблон, який називається DTO (Data Transfer Object). DTO, на відміну від DAO (Data Access Object) не повинен містити ніякої поведінки. DTO можуть виглядати повністю або частково як Entity.

Для коректної передачі даних необхідний клас конвертації Entity в DTO. Можна використати уже готові бібліотеки, такі як ModelMapper або MapStruct, чи написати власну реалізацію. Основна вимога використання цих бібліотек: імена полів в класах для конвертації мають співпадати або різниця має бути відображена у відповідній анотації. Для MapStruct це виглядає так:

```
@Mapper
public interface EmployeeMapper {@Mappings({
        @Mapping(target="employeeId", source="entity.id"),
        @Mapping(target="employeeName", source="entity.name")   })
    EmployeeDTO employeeToEmployeeDTO(Employee entity);
    @Mappings({
        @Mapping(target="id", source="dto.employeeId"),
        @Mapping(target="name", source="dto.employeeName") })
    Employee employeeDTOtoEmployee(EmployeeDTO dto);}
```

Власна реалізація мапера має відповідати таким вимогам:

- конвертація в дві сторони;
- розділення на інтерфейс та реалізацію, для забезпечення коректної роботи DI;
- обов’язкова анотація для реалізації, яка ідентифікуватиме клас як JavaBean;
- перевірка вхідного об’єкта на null.

В системі використана власна реалізація.

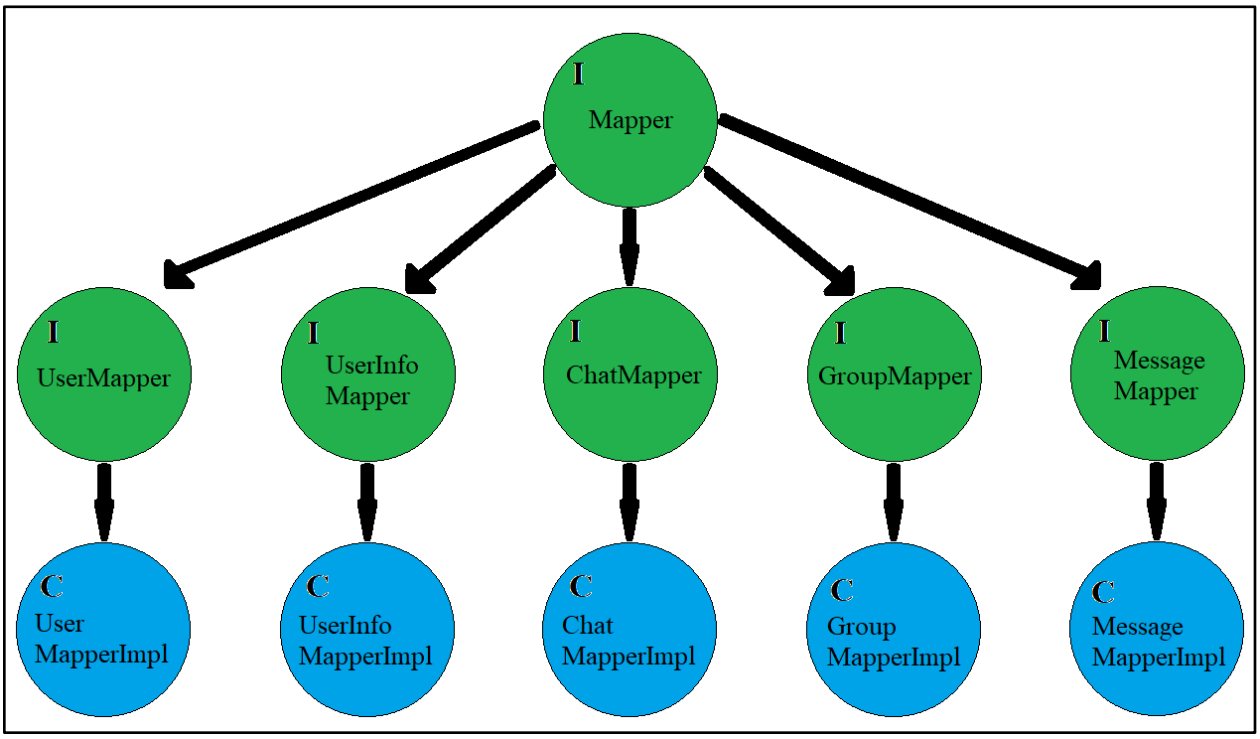

Рисунок 2.2 – Ієрархія інтерфейсів та класів Mapper

Головним елементом є інтерфейс Mapper.

```
public interface Mapper<D, E> {
        D mapToDomain(E entity);
        E mapToEntity(D domain); }
```

Це шаблонний клас, що параметризується двома типами: тип Entity та тип DTO. В інтерфейсі оголошені 2 функції: для конвертації з DTO в Entity та навпаки. Сигнатура функції залежить від шаблонних типів.

Для кожної пари Entity-DTO створено відповідний інтерфейс <Class>Mapper, який наслідує головний інтерфейс та передає йому відповідні типи.

Для кожного такого інтерфейсу створено клас реалізації, який позначено анотацією @Component, завдяки якій Spring визначає клас як JavaBean і поміщає його в контейнер бінів.

Для прикладу, реалізація мапера для об’єкта Group виглядає так:

Інтерфейс

```
public interface GroupMapper extends Mapper<GroupDto, Group> {}
```

Реалізація

```
@Component
public class GroupMapperImpl implements GroupMapper {
    @Autowired
    private UserInfoMapper userInfoMapper;
    @Override
    public GroupDto mapToDomain(Group entity) {
        if(entity == null){
        return null; }
        GroupDto groupDto = new GroupDto();
        groupDto.setId(entity.getId());
        groupDto.setName(entity.getName());
        groupDto.setUsers(entity.getUsers().stream()
                .map(userInfoMapper::mapToDomain)
                .collect(Collectors.toSet()));
        return groupDto; }
    @Override
    public Group mapToEntity(GroupDto domain) {
        if(domain == null){
                return null; }
        Group group = new Group();
        group.setId(domain.getId());
        group.setName(domain.getName());
        group.setUsers(domain.getUsers().stream()
                .map(userInfoMapper::mapToEntity)
                .collect(Collectors.toSet()));
        return group; }}
```

2.2.2 Ендпойнти

Кінцева точка - це один кінець каналу зв’язку. Коли API взаємодіє з іншою системою, точки цього зв'язку вважаються кінцевими точками. Для API кінцева точка може містити URL-адресу сервера або служби. Кожна кінцева точка - це місце, звідки API можуть отримати доступ до ресурсів, необхідних для виконання своїх функцій.

API працюють із використанням «запитів» та «відповідей». Коли API запитує інформацію від веб-програми або веб-сервера, він отримує відповідь. Місце, куди API надсилають запити, і де знаходиться ресурс, називається кінцевою точкою. [8]

Для перегляду всіх ендпойнтів сервера та надсилання запитів можна використовувати Postman або Swagger.

Сваггер має вбудовану підтримку у фреймворку Spring. Для додавання сваггера в свій проект необхідно включити дві залежності у файл pom.xml:

```
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.6.1</version>
</dependency>
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.6.1</version>
</dependency>
```

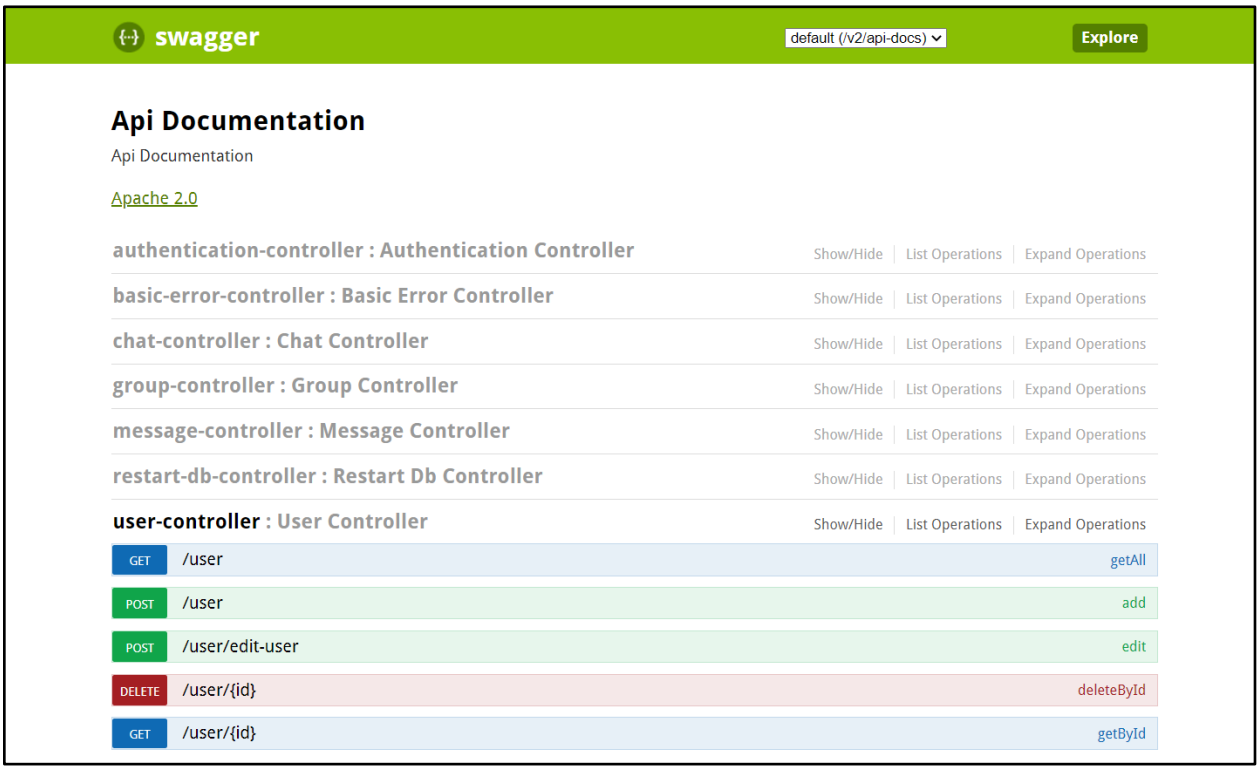

Рисунок 2.3 – Swagger

Далі потрібно створити клас, який буде мати анотації @Configuration, @EnableSwagger2 та реалізовувати інтерфейс WebMvcConfigurerAdapter і його метод addResourceHandlers(ResourceHandlerRegistry). Реалізація класу в системі:

```
@Configuration
@EnableSwagger2
public class Swagger2UiConfiguration extends WebMvcConfigurerAdapter {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build(); }
    @Override
```

```
public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("swagger-ui.html")
                .addResourceLocations("classpath:/META-INF/resources/");
        registry.addResourceHandler("/webjars/**").addResourceLocations
                ("classpath:/META-INF/resources/webjars/"); }}
```

В методі addResourceHandlers(ResourceHandlerRegistry) вказується відповідність між URL сваггера та місцем знаходження відповідних ресурсів в проекті. В заданому місці ресурси генеруються фреймворком автоматично, динамічно оновлюючись при кожному запуску програми, якщо присутні якісь зміни в контролерах або ендпойнтах.

Метод api() потрібен власне для формування сваггера, в ньому вказуються всі потрібні налаштування. Цей метод також має анотацію @Bean, яка означає, що об’єкт, який повертається цією функцією, є JavaBean, а тому керування життєвим циклом цього об’єкта покладається на Spring. При першому зверненні до такого об’єкта Spring створить його, помістить в контейнер бінів та повертатиме посилання на нього всюди, де це потрібно.

Також для коректної роботи сваггера кожен контролер позначений анотацією @CrossOrigin(origins = "*"). Вона потрібна для забезпечення міждоменного зв’язку, тому що хост для сваггера може відрізнятись від основного хоста сервера.

Для коректної роботи ендпойнтів у Spring використовуються такі анотації параметрів:

- @PathVariable – обов’язково має бути вказаний параметр name. Значення змінної береться з тієї частини URL, яка записана в анотації @Mapping методу, та обрамлена фігурними дужками. Назва в дужках має співпадати з параметром name;
- @RequestBody – значення змінної береться з тіла запиту шляхом десеріалізації об’єкта;

- @RequestParam – обов’язково має бути вказаний параметр name. Значення змінної береться з параметра в URL, назва якого вказана в анотації;
- @RequestHeader – обов’язково має бути вказана назва. Значення змінної береться з заголовка, назва якого вказана в анотації) запиту.

В системі використовуються анотації @PathVariable, @RequestBody та @RequestParam. Перша використовується коли достатньо прийняти і обробити id об’єкта, а друга - коли в запиті надсилається об’єкт, який десеріалізується в DTO. Отримані параметри передаються відповідному сервісу, але DTO додатково перетворюються у відповідний тип Entity, так як сервіс може обробляти тільки їх, @PathVariable та @RequestParam передаються без змін.

Для ендпойнтів та контролерів може бути налаштований доступ. Існує 3 типи доступу:

- permitAll() – доступ дозволено всім;
- authenticated() – доступ мають лише користувачі, які увійшли в систему;
- hasAnyAuthority() або hasRole() – доступ мають лише користувачі, роль яких відповідає заданій.

Обмеження доступу задаються у спеціальному класі, який має наслідувати клас WebSecurityConfigurerAdapter та повинен мати анотації @Configuration та @EnableWebSecurity, а також реалізовувати методи configure(). [7] Метод приймає параметр HttpSecurity, в якому і налаштовується доступ.

```
@Override
    protected void configure(final HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/", "/static/**", "/login", "/register-
        page").permitAll()
```

```
        .antMatchers("/profile").authenticated().antMatchers("/entrant/**")
    .hasAnyAuthority(RoleEntity.STUDENT.name())
        .antMatchers("/admin/**").hasAnyAuthority(
            RoleEntity.ADMIN.name())
        .and()
        .csrf()
        .disable();}
```

Доступ також може бути обмежений безпосередньо в самому методі або контролері. Spring Security надає кілька способів перевірки ролей користувачів:

- @PreAuthorize - застосовується до класу або методу, і приймає значення string value. @PreAuthorize("hasRole('ROLE_ADMIN')")
- Java-клас SecurityContext. За замовчуванням Spring Security використовує локальну копію цього класу. Це означає, що кожен запит у додатку має свій контекст безпеки, який містить деталі користувача, який робить запит.
- UserDetailsService. Перевага цього підходу полягає в тому, що можна перевіряти ролі будь-якого користувача, а не лише того, хто зробив запит.
- Servlet Request. За умови використання Spring MVC, можна перевірити ролі користувачів, використовуючи клас HttpServletRequest.
  public String getUsers(HttpServletRequest request){} [8]

2.2.3 Services

Розглянемо процес автентифікації. Для автентифікації створено окремий контролер AuthenticationController. Він містить метод login() з відповідним URL, який приймає 2 параметра: пошту та пароль та повертає клієнту об’єкт користувача, якщо автентифікація успішна. Вони необхідні для

фреймворка, щоб створити токен автентифікації та передати його відповідному сервісу.

```
@CrossOrigin(origins = "*")
@RestController
@RequestMapping("/")
public class AuthenticationController {
    @Autowired
    private UserMapper userMapper;
    @Autowired
    private UserService userService;
    @PostMapping("login")
    UserDto login(@RequestParam(name = "email") String email,
        @RequestParam(name = "password") String password){
            userService.authenticate(new
            UsernamePasswordAuthenticationToken(email, password));
            return userMapper.mapToDomain(
            userService.getByEmailAndPassword(email, password)); }}
```

Сервісом, який обробляє процес автентифікації є UserService.

Згідно стандарту фреймворка Spring сервіс, задля забезпечення роботи ін’єкції залежностей, має бути розділений на дві частини: інтерфейс та реалізація. Реалізація повинна мати анотацію @Service. Для того, щоб сервіс розпізнавався фреймворком як такий, що відповідає за автентифікацію, він має розширювати інтерфейс AuthenticationProvider та реалізовувати методи authenticate(Authentication) і supports(Class<?>). Вигляд інтерфейсу UserService:

```
public interface UserService extends AuthenticationProvider {
    List<User> getAll();
    User getById(Integer id);
    User save(User user);
    User getByEmailAndPassword(String email, String password);
```

```
    void deleteById(Integer id); }
```

Вигляд реалізації сервісу, клас UserServiceImpl:

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private GroupRepository groupRepository;
    /* опущена реалізація методів інтерфейсу UserService */
    @Override
    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        String email = authentication.getName();
        String password = (String) authentication.getCredentials();
        User user = getByEmailAndPassword(email, password);
        Authentication authenticate =
            new UsernamePasswordAuthenticationToken(
            user.getEmail(), user.getPassword(),
            singletonList(new
            SimpleGrantedAuthority(user.getRole().name())));
        authenticate.setAuthenticated(true);
    SecurityContextHolder.getContext().setAuthentication(authenticate);
    return authenticate; }
    @Override
    public boolean supports(Class<?> authentication) {
        return true; }}
```

Метод supports(Class<?>) потрібен для того, щоб вказати який саме тип автентифікації реалізує цей клас. В даному випадку підтримуються всі варіанти.

Метод authenticate(Authentication) приймає об’єкт типу Authentication, в якому записано логін, пароль, ролі користувача. Далі йде перевірка наявності

користувача з відповідними даними в базі. Якщо такого користувача не існує, то клієнту відправиться повідомлення про помилку. Якщо ж такий користувач знайдений, то він дістається з БД. Формується новий об’єкт Authentication, в який записується логін, пароль, ролі користувача та вказується що користувач авнетифікований. Об’єкт Authentication зберігається в SecurityContext, для подальшого використання в перевірці доступу користувача до ендпойнтів.

Після того, як контролер і сервіс створені, потрібно повідомити фреймворку, що процес автентифікації буде виконуватись з їх використанням. Для цього в класі, який наслідує WebSecurityConfigurerAdapter потрібно реалізувати функцію configure(AuthenticationManagerBuilder), а також додати в клас об’єкт типу UserService.

```
@Override
    protected void configure(final AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(provider); }}
```

Саме у цій функції вказується, що процес автентифікації виконується в спеціально створеному сервісі.

Вказувати, що використовується власний контролер для ендпойнта автентифікації, не потрібно, оскільки його URL збігається зі стандартним, тому Spring буде використовувати реалізацію з нашої системи.

РОЗДІЛ 3. КЛІЄНТСЬКА ЧАСТИНА

3.1 Популярні мобільні системи

Android - мобільна операційна система заснована на ядрі Linux та віртуальної машині Java. Належить корпорації Google. Офіційно перша версія ОС вийшла 23 вересня 2008 року. Відкритий первинний код Android дозволяє виробникам пристроїв розробляти та адаптувати ОС до своїх телефонів і планшетів, а розробникам програм завантажувати власні додатки до Google Play Store. У даний час Android є найпопулярнішою платформою для смартфонів у світі і використовується багатьма виробниками телефонів. [9]

iOS (до 24 червня 2010 року iPhone OS) - друга найпопулярніша мобільна операційна система, яка базується на основі Darwin компонентів, розроблена американською компанією Apple. Перша iPhone OS була презентована 9 січня 2007 року сумісно з першим смартфоном від компанії Apple iPhone 2G. Первинний код системи закритий. Щоб завантажити сторонній додаток до App Store розробникам необхідно підписати спеціальний сертифікат, випущений компанією Apple, що робить операційну систему найбільш безпечною. Розробники можуть отримати тимчасовий сертифікат для встановлення програм на обмеженому числі пристроїв. [10]

3.2 Мобільна операційна система Android

Android базується на ядрі Linux, що забезпечує систему такими функціями як багатопоточність, керування пам’яттю низького рівня та функціями безпеки. Розробляється альянсом Open Handset Alliance і комерційно спонсорується Google. [11] Дана ОС головним чином розроблена для мобільних пристроїв із сенсорним екраном, таких як смартфони та планшети. Це безкоштовне програмне забезпечення з відкритим первинний кодом, відомим як Android Open Source Project (AOSP). Більшість пристроїв

Android постачаються з попередньо встановленим додатковим програмним забезпеченням мобільних служб Google, який включає основні додатки, такі як Google Chrome, платформа цифрового розповсюдження Google Play та пов’язана з ним платформа розвитку Google Play Services. Близько 70 відсотків смартфонів Android керують екосистемою Google. Завдяки відкритому первинному коду Android можна не лише встановлювати власні створені додатки, а й міняти версію системи.

3.3 Retrofit

Retrofit - це відома серед Android-розробників бібліотека для мережевої взаємодії, REST клієнт для Java і Android. Він дозволяє легко отримувати і завантажувати JSON (або інші структуровані дані) через веб-сервіс на основі REST. Зазвичай для серіалізації даних JSON використовується Gson конвертер, але також є можливим додавання власних конвертерів для обробки XML або інших протоколів. У Retrofit використовується бібліотека OkHttp для HTTP-запитів. [12]

Для роботи з Retrofit необхідні наступні класи:

- Model клас, який використовується як модель JSON;
- інтерфейси, які визначають можливі HTTP операції;
- клас Retrofit.Builder - екземпляр, який використовує інтерфейс і API Builder, щоб задати визначення кінцевої точки URL для операцій HTTP.

Кожен метод інтерфейсу являє собою один з можливих викликів API. Він повинен мати HTTP анотацію (GET, POST, PUT, DELETE), щоб вказати тип запиту і відносний URL. Значення, що повертається завершує відповідь в Call-об’єкті з типом очікуваного результату. [13]

Для кращого розуміння розділимо роботу з Retrofit на наступні задачі:

Завдання перше. Model

Враховуючи структуру відповіді серверу у вигляді JSON (або інших форматів), необхідно створити на її основі Java-клас у вигляді класу Model. Зручніше створювати за допомогою готових веб-сервісів в автоматичному режимі або можна самостійно створити клас, якщо структура є достатньо не складною. Список анотацій залежить від типу конвертера який використовується. [13]

Завдання друге. Інтерфейс

Наступним кроком необхідно створити інтерфейс і вказати ім’я методу, додати необхідні параметри, якщо вони потрібні. В інтерфейсі задаються команди-запити для сервера. Команда комбінується з базовою адресою сервера baseUrl. Таким чином формується повний шлях. Запити розміщуються в класі Call із вказаним бажаним типом. У більшості випадків повертається об’єкт Call<T> з потрібним типом. Якщо тип відповіді є неважливим, то можна вказати Call<Response>. Тут використовуються бібліотечні анотації, за допомогою яких вказуються веб-команди, а потім Java-метод.

@GET GET-запит для базової адреси. Також можна вказати параметри в дужках;

@POST POST-запит для базової адреси. Також можна вказати параметри в дужках;

@Query анотація для запитів з параметрами. Припустимо, необхідно передати додатковий параметр до запиту, який виводить список елементів у відсортованому вигляді: https://server.herokuapp.com/users?sort=desc.

Запит з параметром виглядатиме так:

```
@GET("users?sort=desc")
Call<Users>getAllUsers();
```

@Path анотація для запиту що містить змінні частини шляху. У таких випадках використовують фігурні дужки в запиті, в самому методі через анотацію @Path вказується ім’я, яке буде підставлятися в шлях. [13]

Запит матиме такий вигляд:

```
@GET ("/users/{id}")
Call getUser (@Path ("id") String userName);
```

Завдання третє. Retrofit

Найпростіший об’єкт Retrofit для запиту до сервера створюється наступним чином:

```
public static final String BASE_URL = "http://api.example.com/";
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build();
```

Після цього буде отримано об’єкт Retrofit, що містить базовий URL і здатний перетворювати JSON-дані в об’єкти за допомогою зазначеного конвертера Gson. Потім потрібно створити об’єкт, з якого будуть здійснюватись запити, за допомогою функції create(), в якій потрібно вказати клас інтерфейсу.

Далі ми отримуємо об’єкт Call для відповідного запиту. Якщо метою стоїть виконати синхронний запит то слід використати метод Call.execute(), якщо асинхронний - метод Call.enqueue(). В Android програмах заборонено робити синхронний запит з головного потоку (головним є потік, в якому обробляються активіті. Це зроблено для того, щоб рендер інтерфейсу не зупинявся для очікування відповіді від сервера). Якщо було використано функцію enqueue(), запит буде виконаний в окремому потоці, а результат прийде в Callback в потік, з якого було викликано функцію enqueue().

В результаті Retrofit зробить запит, отримає відповідь і запише дані у відповідні об’єкти для подальшої обробки даних. Основна частина роботи

відбувається в методі onResponse(), помилки виводяться в onFailure() (неправильна адреса сервера, некоректний формат даних, неправильний формат класу-моделі тощо). Клас Response має зручний метод isSuccessful() для успішної обробки запиту (коди 200xx). Якщо при виконанні запиту трапилась помилка, то її можна обробити в методі errorBody() класу ResponseBody. [13]

3.3 UI та опис роботи з мобільним клієнтом

Запускаючи вперше додаток користувач одразу потрапляє на сторінку логіна (Рисунок 3.1), де йому необхідно ввести пошту та пароль. Після успішної автентифікації МК зберігає введені дані користувача на пристрої до постійного сховища SharedPreferences. [14] Якщо програму буде закрито, а потім повторно запущено, то спершу додаток перевірить постійне сховище на наявність даних поточного користувача. Якщо дані наявні, тобто якщо раніше відбувся успішний вхід в систему, то з цими даними автоматично формується запит входу в систему, який надсилається серверу. Інакше користувач потрапляє на сторінку логіна. При виході з системи за допомогою опції Logout дані з постійного сховища будуть видалені і користувач буде переадресовано на сторінку логіна.

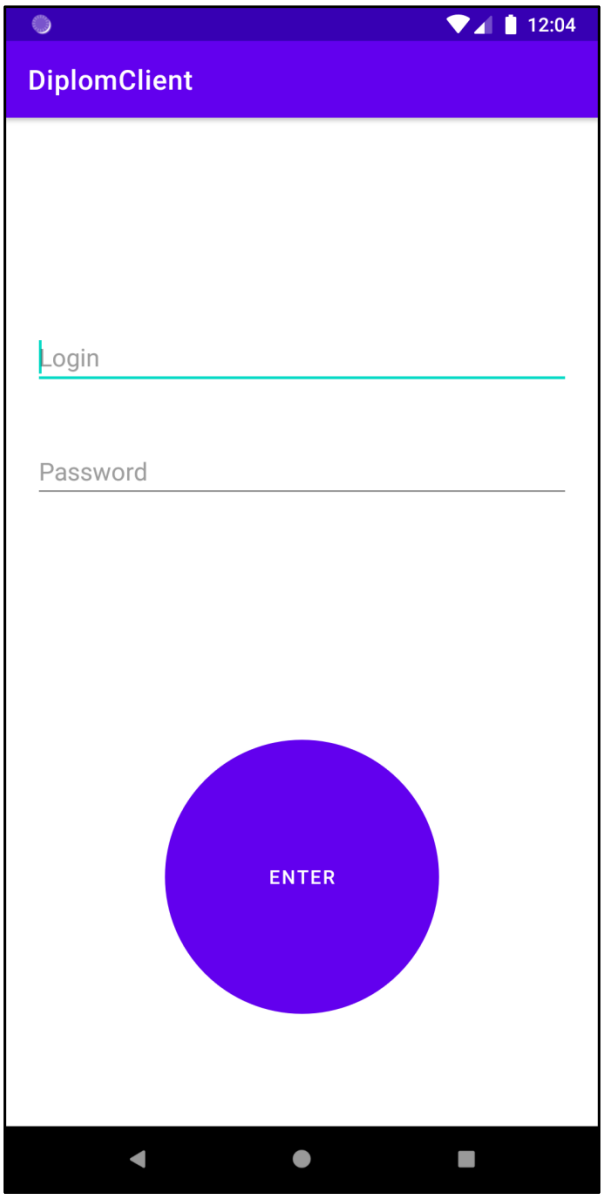

Рисунок 3.1 – Сторінка логіна

Для автентифікованих користувачів системи головною сторінкою є сторінка чатів (Рисунок 3.2 а), на якій відображаються всі чати в яких користувач є учасником, або власником.

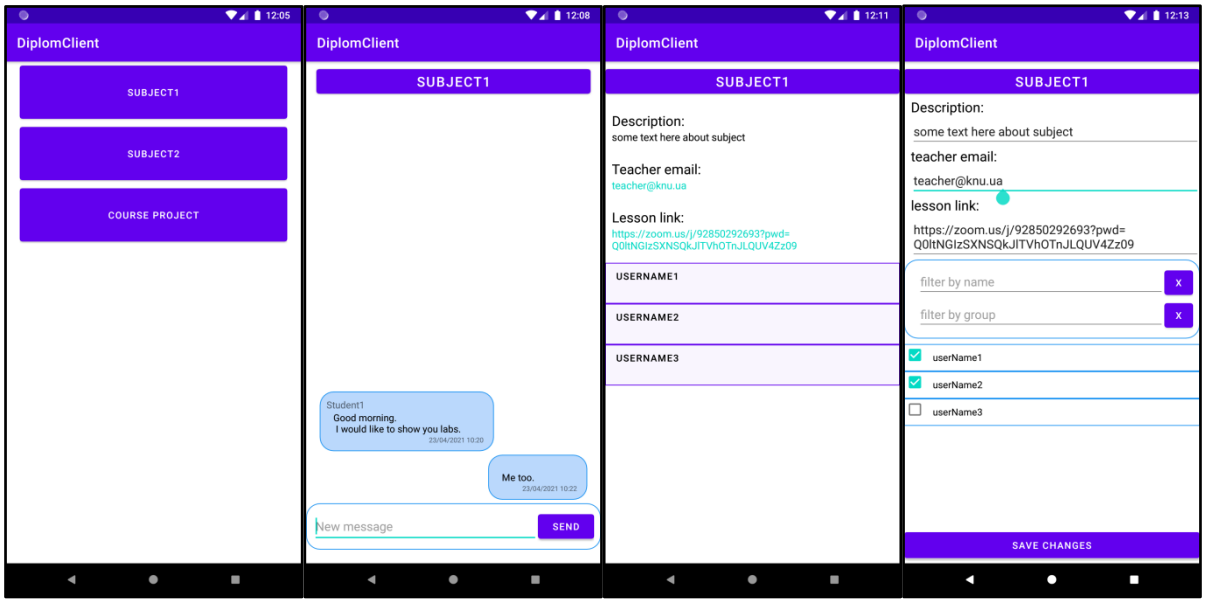

Рисунок 3.2 – UI студента та вчителя: а – сторінка чатів; б – сторінка чату; в – сторінка опису чату; г – сторінка редагування/створення чату

Перейшовши в певний чат (Рисунок 3.2 б) користувач може переглядати повідомлення та надсилати повідомлення всім учасникам групи.

Щоб перейти на сторінку з інформацією про чат (Рисунок 3.2 в): з описом, поштою викладача, посиланням на заняття та списком учасників треба натиснути кнопку з назвою чату вгорі сторінки чату (Рисунок 3.2 б).

Сторінка редагування/створення чату (Рисунок 3.2 г) недоступна студентам і є доступною викладачу. Додавання учасників до чату можливе з застосуванням фільтрів по імені та по групі. Поля: опис, пошта вчителя та посилання на заняття не є обов’язковими.

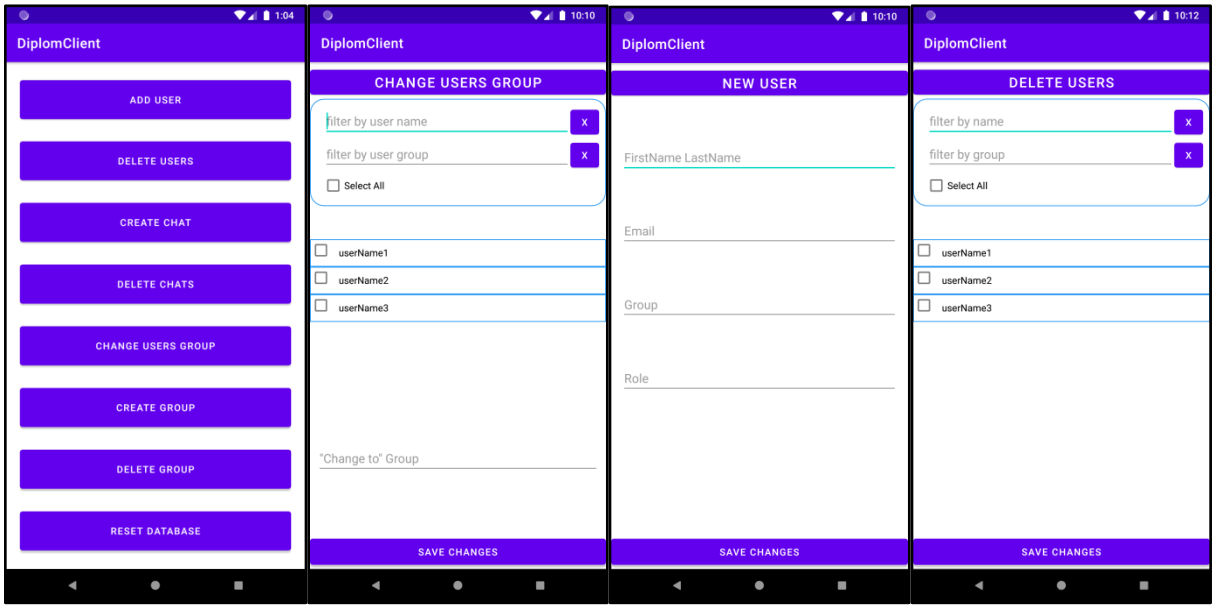

Рисунок 3.3 – UI адміна: а – головна сторінка; б – сторінка зміни групи користувачам; в – сторінка додавання нового користувача; г – сторінка видалення користувачів

Після входу в систему адміністратору відкривається головна сторінка (Рисунок 3.3 а) з вісьмома доступними функціями:

1. В кінці навчального року, коли студенти переходять на наступний курс, адміністратор має змінити їм групу (наприклад з ТТП-2 на ТТП-3). Для пошуку користувачів, яким треба встановити нову групу (Рисунок 3.3 б), можна застосувати фільтри: по імені студента та по назві групи. В нижній частині сторінки є поле з випадаючим списком назв груп, які можна обрати. Зміни застосуються лише до вибраних користувачів після натискання кнопки зберегти.
2. Для додавання нового користувача в систему (Рисунок 3.3 в) необхідно заповнити поля імені користувача, пошти, групи(для студента) та ролі.
3. На сторінці видалення користувачів із системи (Рисунок 3.3 г) є два фільтри: по імені студента та по назві групи. Видаляються лише ті користувачі, які були вибрані.

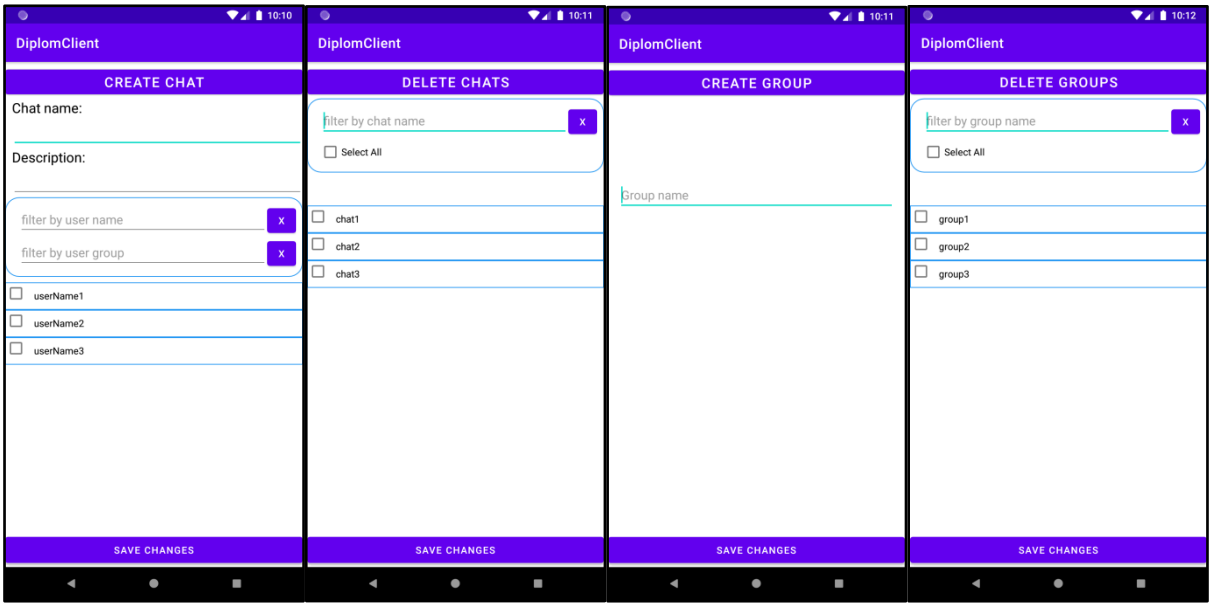

Рисунок 3.4 – UI адміністратора: а – сторінка створення чату; б – сторінка видалення чатів; в – сторінка створення групи; г – сторінка видалення груп

4. Щоб створити чат групи (Рисунок 3.4 а) треба заповнити поля назви групи, опис (необов'язково) та додати учасників. Для пошуку потрібних користувачів можна застосувати фільтри: по імені та по групі.
5. Для видалення одного або декількох чатів (Рисунок 3.4 б) їх необхідно вибрати зі списку. Для пришвидшення пошуку можна скористатися фільтром по назві чату.
6. Іноді виникає потреба створити нову групу (Рисунок 3.4 в). Наприклад якщо було зараховано більше студентів ніж минулого року. На сторінці створення групи треба ввести лише назву нової групи.
7. Щоб видалити групи (Рисунок 3.4 г), назви яких більше не використовуються, треба їх вибрати із списку та натиснути кнопку зберегти.
8. Перезапуск ми даних рекомендується робити в кінці кожного року. Ця опція видаляє всі повідомлення з бази даних та видаляє всі чати (окрім групових).

ВИСНОВКИ

Проведено дослідження предметної області та аналіз онлайн-систем підтримки комунікацій. Виконано огляд технологій, інструментарію для розробки серверної та клієнтської частин системи, сформовано функціональні вимоги. Розроблено систему підтримки комунікації у навчальних підрозділах закритого типу з мобільним додатком.

В майбутньому дана система може бути розширена для викладачів та студентів функцією перегляду розкладу, з можливістю переходити за посиланням на заняття та одразу в чат цього ж предмету прямо з сторінки розкладу. При подальшій розробці можна реалізувати інтеграцією з системою КНУ імені Шевченка - Triton. Програма має перспективи перерости у повноцінну єдину навчальну систему з месенджером.

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАННЯ

1. Spring Data JPA [Електронний ресурс] – Режим доступу до ресурсу: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#preface.
2. Hibernate [Електронний ресурс] – Режим доступу до ресурсу: https://hibernate.org/orm/.
3. MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://dev.mysql.com/doc/.
4. Spring Controllers [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/spring-controllers.
5. Блінов І. Java. Методы программирования / І. Блінов, В. Романчик. – Мінськ: «ЧЕТЫРЕ ЧЕТВЕРТИ», 2013. – 896 с.
6. Spring REST Controllers [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/spring-rest-http-headers.
7. Spring Security [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/spring-security-check-user-role.
8. API Endpoints [Електронний ресурс] – Режим доступу до ресурсу: https://smartbear.com/learn/performance-monitoring/api-endpoints/.
9. Офіційний сайт Android [Електронний ресурс] – Режим доступу до ресурсу: https://www.android.com/intl/.
10. Android [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/.
11. iOS (operating system) [Електронний ресурс] – Режим доступу до ресурсу: https://www.apple.com/ru/ios/app-store/.
12. Murphy M. The Busy Coder's Guide to Android Development / Mark Murphy., 2009. – 468 с.

13. Климов О. Retrofit [Електронний ресурс] / Олександр Климов – Режим доступу до ресурсу: http://developer.alexanderklimov.ru/android/library/retrofit.php.
14. SharedPreferences [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/reference/android/content/SharedPreferences.