Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп’ютерних систем
Кафедра інформаційних систем

Федорова Вікторія Ігорівна,
студентка групи АІ-231

# КУРСОВА РОБОТА

Система обліку витрат домогосподарства

Спеціальність:
122 Комп’ютерні науки

Освітня програма: Комп’ютерні науки

Керівник-укладач:
Годовиченко Микола Анатолійович,

Одеса – 2025

# ЗМІСТ

# ВСТУП

У сучасних умовах ведення особистих фінансів стає все більш актуальним питанням. Автоматизація обліку витрат та доходів дозволяє користувачам ефективніше контролювати свої ресурси, приймати обґрунтовані фінансові рішення, аналізувати динаміку витрат і планувати бюджет.

Метою даної курсової роботи є розробка програмного забезпечення для обліку витрат домогосподарства з використанням сучасних веб-технологій. Застосунок дозволяє реєструвати користувачів, додавати записи витрат та доходів, формувати категорії, визначати бюджетні обмеження, а також генерувати звіти за різними критеріями.

У ході реалізації було використано стек технологій Java + Spring Boot, Spring Security, JWT, OAuth2, JPA/Hibernate, PostgreSQL. REST API створено згідно з сучасними стандартами веб-розробки.

Дана система може бути основою для реального веб-сервісу або мобільного додатку з функцією синхронізації та персоналізованої аналітики.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Опис предметної області системи обліку витрат домогосподарства

Система обліку витрат домогосподарства призначена для автоматизації процесу контролю особистих фінансів користувача. Основна мета – забезпечити інструмент для фіксації доходів і витрат, керування категоріями, встановлення бюджетних обмежень та формування фінансових звітів.

Користувач має змогу реєструвати свої витрати за категоріями, створювати доходи із зазначенням джерела, визначати обмеження витрат у межах бюджету на місяць, а також переглядати статистичні звіти. Система має дозволяти контролювати фінансову дисципліну та приймати обґрунтовані фінансові рішення. В рамках курсової роботи я також реалізуватиму реєстрацію користувача різними засобами, які збільшать надійність збереження та конфіденційності даних.

До основних функцій системи слід віднести наступні пункти:

- реєстрація користувачів та захищена автентифікація;
- додавання витрат, доходів та бюджету;
- формування звітів: сума витрат за місяць, найбільші витрати, перевищення бюджету;
- фільтрація витрат за датою;
- облік категорій (наприклад, «Їжа», «Транспорт», «Побутові послуги»);
- авторизація через логін/пароль або OAuth2 (наприклад, Google).

У центрі функціоналу знаходяться сутності:

- User – користувач, який створює всі інші записи;
- Expense – витрати користувача;
- Income – доходи користувача;
- Category – категорії витрат;
- Budget – бюджетне обмеження на місяць.

## 1.2 Детальний опис функціоналу бази даних

User (Користувач).

Ця сутність представляє особу, яка користується системою. Вона є відправною точкою для всіх інших сутностей, оскільки витрати, доходи, бюджети – все прив’язано саме до конкретного користувача. User ідентифікується унікальним email-адресом і може створювати, редагувати або видаляти пов’язані з ним записи.

Expense (Витрата).

Expense є записом про будь-які витрати користувача – це можуть бути покупки, оплата послуг, транспорт чи інші фінансові втрати. Витрата обов’язково належить певному користувачу і належить до певної категорії. Вона також містить інформацію про дату, суму та короткий опис.

Income (Дохід).

Сутність Income фіксує фінансові надходження користувача – зарплати, премії, стипендії тощо. Як і Expense, кожен дохід належить конкретному користувачу та зберігає дані про суму, дату та джерело походження коштів.

Category (Категорія).

Категорія слугує класифікатором для витрат або доходів. Наприклад, для витрат це може бути "Продукти", "Житло", "Транспорт", а для доходів – "Зарплата", "Стипендія". Кожна витрата або дохід прив’язується до певної категорії, що дозволяє здійснювати статистичний аналіз.

Budget (Бюджет).

Сутність Budget представляє місячне обмеження витрат для користувача. Вона визначає, скільки користувач планує витратити в межах конкретного місяця. Ця сутність є основою для розрахунків перевищення бюджету, залишку коштів і контролю фінансової дисципліни.

Усі сутності плануються до реалізації із використанням Spring Data JPA, зв’язки описані анотаціями: @ManyToOne, @OneToMany,

@Table(uniqueConstraints). Для зменшення шаблонного коду застосовано бібліотеку Lombok (@Getter, @Setter, @Builder тощо).

Функціональність REST API охоплює всі основні CRUD-операції, а також додаткові звіти та розрахунки, зокрема звіт по категоріях, перевищення бюджету, середня витрата на день, залишок бюджету, фільтрація по даті, OAuth2-вхід та автентифікація через JWT.

Ця модель даних має дозволяти повністю охопити всі завдання предметної області – від базового обліку витрат до комплексного аналізу фінансового стану користувача.

# 2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 2.1 Проектування структури даних (сутності, зв’язки, діаграми класів)

Кожна сутність відображає ключові аспекти фінансового обліку та має відповідні поля й анотації. Нижче наведено опис усіх сутностей та зв’язків між ними.

User (Користувач):

- id: Long – унікальний ідентифікатор;
- name: String – ім’я користувача;
- email: String – унікальна електронна пошта.

Користувач є центральною сутністю, з якою пов’язані всі інші (витрати, доходи, бюджети). Один користувач може мати багато витрат, доходів та бюджетів – відповідно зв’язок від одного до багатьох.

Expense (Витрата):

- id: Long – унікальний ідентифікатор витрати;
- user: User – зв’язок з користувачем (Many-to-One);
- amount: BigDecimal – сума витрати;
- date: LocalDate – дата витрати;
- category: Category – категорія витрати (Many-to-One);
- description: String – опис витрати.

Багато витрат можуть належати одному користувачу, а це відповідно від багатьох до одного. Також багато витрат можуть бути у межах однієї категорії (наприклад, «Їжа»), що також є зв’язком від багатьох до одного.

Income (Дохід):

- id: Long;
- user: User – зв’язок із користувачем (Many-to-One);
- amount: BigDecimal;
- date: LocalDate;
- source: String – джерело доходу (наприклад, «зарплата»).

Один користувач може мати багато доходів тобто зв’язок від багатьох до одного.

Category (Категорія):

- id: Long;
- name: String – назва категорії (наприклад, «Транспорт»);
- type: String – тип категорії: EXPENSE або INCOME.

Категорія може використовуватись у багатьох записах витрат. Тип категорії дозволяє обмежити її застосування (наприклад, «Продукти» як EXPENSE). Відповідно це зв’язок від одного до багатьох.

Budget (Бюджет):

- id: Long;
- user: User – зв’язок з користувачем (Many-to-One);
- month: String – місяць у форматі YYYY-MM;
- limit: BigDecimal – ліміт бюджету на цей місяць.

Передбачено обмеження унікальності комбінації user_id + month, щоб у кожного користувача був лише один бюджет на місяць.

Зв’язки між сутностями зображаються на UML-діаграмі класів (рис. 2.1.1), яка відображає структуру проєкту у вигляді таблиць та їхніх взаємозв’язків.

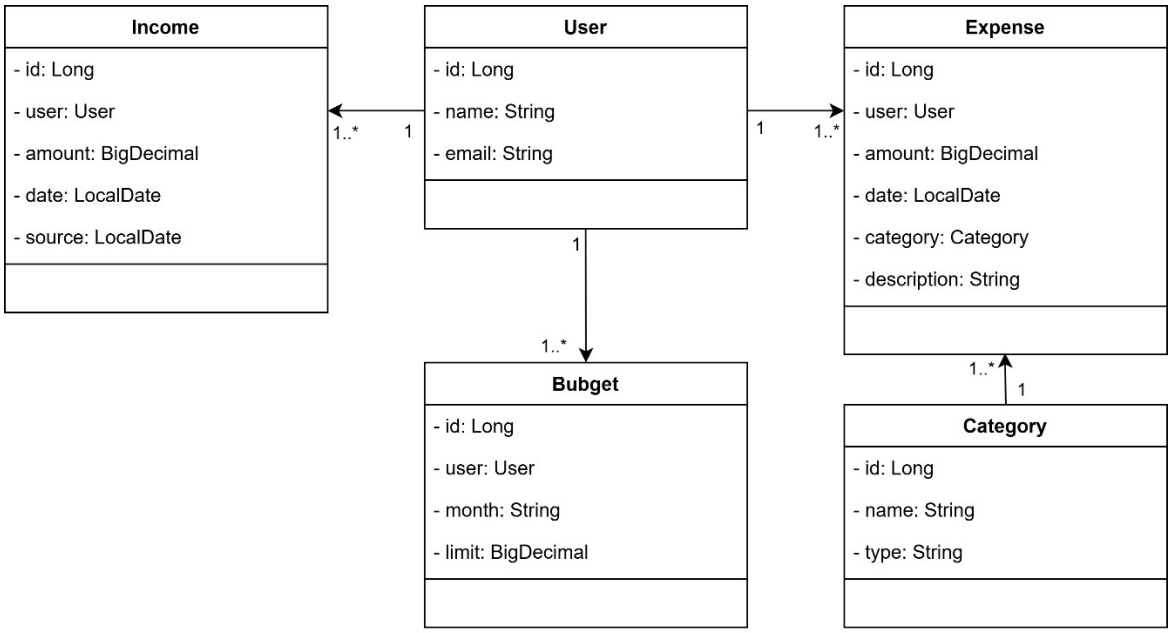


Рисунок 2.1.1 – Діаграма класів системи обліку витрат господарства

## 2.2 Опис архітектури застосунку (рівні Controller–Service–Repository, залежності між компонентами)

Для реалізації системи обліку витрат домогосподарства застосовано багаторівневу архітектуру, що відповідає принципам розділення обов'язків (Separation of Concerns). Це дозволяє легко масштабувати систему, проводити модульне тестування та змінювати окремі компоненти без впливу на інші.

Архітектура умовно поділяється на три основні рівні: Controller, Service, Repository, а також окремо виділено рівень безпеки Security.

Контролери відповідають за обробку HTTP-запитів з боку клієнта (наприклад, від браузера або мобільного застосунку). Вони приймають вхідні параметри, передають дані до відповідних сервісів, і повертають результати у вигляді JSON-об'єктів.

Кожен контролер відповідає за окрему логічну частину:

- UserController – облік користувачів;
- ExpenseController – робота з витратами;
- IncomeController – облік доходів;
- BudgetController – бюджетування;
- CategoryController – управління категоріями;
- AuthController, OAuth2Controller – автентифікація (JWT та OAuth2).

Сервіси реалізують бізнес-логіку застосунку. Вони перевіряють коректність вхідних даних, взаємодіють із репозиторіями, виконують обчислення (наприклад, загальна сума витрат, перевищення бюджету) та реалізують сценарії, необхідні для контролерів.

Наприклад ExpenseService виконує фільтрацію витрат по місяцю/даті, розрахунок суми витрат, а BudgetService перевіряє унікальність місячного бюджету, обчислює залишок та перевищення.

Репозиторії є інтерфейсами до бази даних і забезпечують доступ до сутностей. Вони реалізовані з використанням Spring Data JPA, що дозволяє уникнути написання SQL-запитів. CRUD-операції реалізуються автоматично.

Приклади:

- ExpenseRepository – з методами findByUserId(), findByUserIdAndDateBetween();
- BudgetRepository – містить метод findByUserIdAndMonth() для унікального бюджету;
- UserRepository, CategoryRepository, IncomeRepository – базові JPA-інтерфейси.

Рівень безпеки включає в свою чергу JWT-аутентифікацію: генерує та перевіряє токени (JwtUtil, JwtFilter), OAuth2-вхід через Google (CustomOAuth2UserService, OAuth2LoginSuccessHandler). Авжеж включає рівень безпеки і реєстрацію та вхід користувача (AuthController, AppUserRepository) і AppUser як окрему сутність для обліку зареєстрованих користувачів системи (це не сутність User за предметною областю завдання).

Залежності між компонентами:

- контролери взаємодіють з відповідними сервісами через ін’єкцію залежностей (@RequiredArgsConstructor);
- сервіси звертаються до репозиторіїв, щоб отримати або зберегти дані;
- JWT та OAuth2 інтегруються з механізмами Spring Security для контролю доступу;
- усі шари розроблені як незалежні модулі, що спрощує тестування та розширення.

## 2.3 Опис REST API

REST API реалізує інтерфейс взаємодії користувача із серверною частиною системи. Усі запити побудовані за принципами REST (Representational State Transfer) і працюють із форматами JSON.

Кожен ресурс (користувач, витрати, доходи, бюджети тощо) має свій контролер та унікальний шлях. Далі наведено повний перелік REST-запитів, їхнє призначення та приклади використання.

Розглянемо контролер сутності «Користувачі».

Зареєструвати користувача – POST /users. Тіло запиту:

```
{
  "name": "Іван Іванов",
  "email": "ivan@example.com"
}
```

Тіло відповіді:

```
{
    "id": 1,
    "name": "Олексій",
    "email": "oleksii@example.com"
}
```

Отримати профіль користувача – GET /users/{id}. Тіло відповіді:

```
{
    "id": 1,
    "name": "Олексій",
    "email": "oleksii@example.com"
}
```

Оновити профіль користувача – PUT /users/{id}. Тіло запиту:

```
{
  "name": "Олексій Новий",
  "email": "new@example.com"
}
```

Тіло відповіді:

```
{
    "id": 1,
    "name": "Олексій Новий",
    "email": "new@example.com"
}
```

Видалити користувача – DELETE /users/{id}. Тіло відповіді за замовчуванням є пустим для команд видалення, оскільки немає чого повернути, об’єкт же видаляється. Тому пустий рядок коду 200 OK є навпаки коректною

відповіддю, на відміну від повних рядків помилок, які повернуться в разі некоректності запиту.

Розглянемо контролер сутності «Витрати».

Додати витрату – POST /expenses. Тіло запиту:

```
{
  "user": { "id": 1 },
  "amount": 120.50,
  "date": "2025-06-10",
  "category": { "id": 2 },
  "description": "Купівля продуктів"
}
```

Тіло відповіді:

```
{
    "id": 2,
    "user": {
        "id": 1,
        "name": "Олексій Новий",
        "email": "new@example.com"
    },
    "amount": 120.00,
    "date": "2025-06-11",
    "category": {
        "id": 1,
        "name": "Їжа",
        "type": "EXPENSE"
    },
    "description": "Купівля овочів"
}
```

Отримати список витрат користувача – GET /expenses/user/{userId}. Тіло відповіді:

```
[
    {
        "id": 1,
        "user": {
```

```
            "id": 1,
            "name": "Олексій Новий",
            "email": "new@example.com"
        },
        "amount": 150.00,
        "date": "2025-06-11",
        "category": {
            "id": 1,
            "name": "Їжа",
            "type": "EXPENSE"
        },
        "description": "Купівля фруктів"
    },
    …
]
```


Оновити запис витрати – PUT /expenses/{id}. Тіло запиту:


```
{
  "amount": 130.00,
  "date": "2025-06-10",
  "category": { "id": 1 },
  "description": "Оновлена покупка"
}
```


Тіло відповіді:


```
{
    "id": 1,
    "user": {
        "id": 1,
        "name": "Олексій Новий",
        "email": "new@example.com"
    },
    "amount": 150.00,
    "date": "2025-06-11",
    "category": {
        "id": 1,
        "name": "Їжа",
```

```
        "type": "EXPENSE"
    },
    "description": "Оновлена покупка"
}
```

Видалити витрату – DELETE /expenses/{id}. Тіло відповіді за замовчуванням є пустим для команд видалення, оскільки немає чого повернути, об’єкт же видаляється. Тому пустий рядок коду 200 OK є навпаки коректною відповіддю, на відміну від повних рядків помилок, які повернуться в разі некоректності запиту.

Розглянемо контролер сутності «Категорії».

Додати категорію витрат – POST /categories. Тіло запиту:

```
{
  "name": "Транспорт",
  "type": "EXPENSE"
}
```

Тіло відповіді:

```
{
    "id": 1,
    "name": "Транспорт",
    "type": "EXPENSE"
}
```

Отримати категорії – GET /categories. Тіло відповіді:

```
[
    {
        "id": 1,
        "name": "Їжа",
        "type": "EXPENSE"
    },
    {
        "id": 2,
        "name": "Продукти",
        "type": "EXPENSE"
    }
]
```

Оновити категорію – PUT /categories/{id}. Тіло запиту:

```
{
  "name": "Їжа",
  "type": "EXPENSE"
}
```

Тіло відповіді:

```
{
    "id": 2,
    "name": "Їжа",
    "type": "EXPENSE"
}
```

Видалити категорію – DELETE /categories/{id}. Тіло відповіді за замовчуванням є пустим для команд видалення, оскільки немає чого повернути, об’єкт же видаляється. Тому пустий рядок коду 200 OK є навпаки коректною відповіддю, на відміну від повних рядків помилок, які повернуться в разі некоректності запиту.

Розглянемо контролер сутності «Доходи».

Додати запис доходу – POST /incomes. Тіло запиту:

```
{
  "user": { "id": 1 },
  "amount": 8000.00,
  "date": "2025-06-01",
  "source": "Зарплата"
}
```

Тіло відповіді:

```
{
    "id": 1,
    "user": {
        "id": 1,
        "name": null,
        "email": null
    },
    "amount": 8000.00,
    "date": "2025-06-01",
    "source": "Зарплата"
```

```
}
```

Отримати доходи користувача – GET /incomes/user/{userId}. Тіло відповіді:

```
[
    {
        "id": 1,
        "user": {
            "id": 1,
            "name": "Олексій Новий",
            "email": "new@example.com"
        },
        "amount": 10000.00,
        "date": "2025-06-01",
        "source": "Зарплата"
    },
    …
]
```

Видалити дохід – DELETE /incomes/{id}. Тіло відповіді за замовчуванням є пустим для команд видалення, оскільки немає чого повернути, об’єкт же видаляється. Тому пустий рядок коду 200 OK є навпаки коректною відповіддю, на відміну від повних рядків помилок, які повернуться в разі некоректності запиту.

Розглянемо контролер сутності «Бюджети».

Додати місячний бюджет – POST /budget. Тіло запиту:

```
{
  "user": { "id": 1 },
  "month": "2025-06",
  "limit": 5000.00
}
```

Тіло відповіді:

```
{
    "id": 7,
    "user": {
        "id": 3,
        "name": null,
        "email": null
```

```
    },
    "month": "2025-06",
    "limit": 5000.00
}
```


Отримати бюджети користувача – GET /budgets/user/{userId}. Тіло відповіді:


```
[
    {
        "id": 1,
        "user": {
            "id": 1,
            "name": "Олексій Новий",
            "email": "new@example.com"
        },
        "month": "2025-06",
        "limit": 5500.00
    }
]
```


Оновити бюджет – PUT /budgets/{id}. Тіло запиту:


```
{
  "user": { "id": 1 },
  "month": "2025-06",
  "limit": 6000.00
}
```


Тіло відповіді:


```
{
    "id": 1,
    "user": {
        "id": 1,
        "name": "Олексій Новий",
        "email": "new@example.com"
    },
    "month": "2025-06",
    "limit": 6000.00
}
```

Отримати перевищення бюджету – GET /budgets/exceeded/{userId}/{month}. Приклад відповіді:

```
520.00
```

Згенерувати звіт по категоріях – GET /budgets/report/{userId}/{month}. Приклад відповіді:

```
[
  "Їжа: 1200.00",
  "Транспорт: 800.00"
]
```

Отримати залишок бюджету – GET /budgets/left/{userId}/{month}. Приклад відповіді:

```
3000.00
```

Отримати суму витрат за місяць – GET /expenses/user/{userId}/month/{month}. Приклад відповіді:

```
270.00
```

Отримати середню витрату на день – GET /expenses/user/{userId}/month/{month}/average. Приклад відповіді:

```
{
    "id": 1,
    "user": {
        "id": 1,
        "name": "Олексій Новий",
        "email": "new@example.com"
    },
    "amount": 150.00,
    "date": "2025-06-11",
    "category": {
        "id": 1,
        "name": "Їжа",
        "type": "EXPENSE"
    },
    "description": "Купівля фруктів"
}
```

Отримати найбільші витрати місяця – GET /expenses/user/{userId}/month/{month}/max. Приклад відповіді:

```
5730.00
```

Отримати витрати по даті – GET /expenses/user/{userId}/date/{date}. Приклад відповіді:

```
[
    {
        "id": 1,
        "user": {
            …
        },
        "amount": 150.00,
        "date": "2025-06-11",
        "category": {
            …
        },
        "description": "Купівля фруктів"
    },
    …
]
```

Аутентифікація (JWT). Реєстрація користувача – POST /auth/register Тіло:

```
{
  "username": "user1",
  "password": "pass123"
}
```

Вхід у систему (JWT-токен) – POST /auth/login. Відповідь:

```
Bearer eyJhbGciOiJIUzI1...
```

Аутентифікація через OAuth2 через Google після вдалої реєстрації також виводить Bearer токен на кшталт «Bearer eyJhbGciOi...».

Більшість відповідей мають формат JSON. Обробка помилок реалізована через стандартні HTTP-статуси (400, 404, 401), з повідомленнями у полі message.

# 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

## 3.1 Моделі

У цьому підрозділі описано реалізацію моделей (entity-класів), які відповідають структурі бази даних проєкту. Усі класи розміщено в пакеті ua.opnu.practice1_template.model, окремо винесено сутність користувача для аутентифікації в пакет security.entity.

Для зменшення кількості шаблонного коду (гетери, сетери, конструктори, builder) застосовано бібліотеку Lombok. Анотації JPA використовуються для визначення зв’язків між сутностями та обмежень.

Клас User:

```
@Entity
@Table(name = "users", uniqueConstraints =
@UniqueConstraint(columnNames = "email"))
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class User {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @Column(nullable = false)
  private String email;
}
```

Це основна сутність, до якої прив’язуються витрати, доходи та бюджети. Має унікальне поле email.

Клас Expense:

```
@Entity
@Table(name = "expenses")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class Expense {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
  private Long id;

  @ManyToOne(optional = false)
  private User user;

  @Column(nullable = false, precision = 12, scale = 2)
  private BigDecimal amount;

  private LocalDate date;

  @ManyToOne
  private Category category;

  private String description;
}
```

Витрата пов’язана з користувачем і категорією. Типовий приклад поля amount реалізовано з високою точністю (BigDecimal).

Клас Income:

```
@Entity
@Table(name = "incomes")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class Income {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne(optional = false)
  private User user;

  @Column(nullable = false, precision = 12, scale = 2)
  private BigDecimal amount;

  private LocalDate date;

  private String source;
}
```

Дохід містить джерело (наприклад, зарплата). Зв’язок – багато доходів → один користувач.

Клас Category:

```
@Entity
@Table(name = "categories")
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class Category {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  private String type;
}
```

Служить як для витрат, так і для доходів. Містить поле type, що позначає призначення (EXPENSE або INCOME).

Клас Budget:

```
@Entity
@Table(name = "budgets",
    uniqueConstraints = @UniqueConstraint(columnNames = {"user_id",
"month"}))
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @Builder
public class Budget {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne(optional = false)
  private User user;

  private String month;

  @Column(name = "budget_limit", nullable = false, precision = 12,
scale = 2)
  private BigDecimal limit;
```

```
}
```

Бюджет зберігає обмеження витрат на місяць у форматі YYYY-MM. Комбінація користувача і місяця є унікальною.

Клас AppUser (для авторизації)

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "auth_users")
public class AppUser {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(unique = true, nullable = false)
  private String username;

  @Column(nullable = false)
  private String password;

  private String role;
}
```

Цей клас не пов'язаний напряму з основною системою обліку, але відповідає за зберігання логінів, паролів і ролей користувачів для JWT- та OAuth2-автентифікації.

## 3.2 Репозиторії

У проекті використовуються репозиторії, створені на основі Spring Data JPA, які забезпечують стандартні CRUD-операції без необхідності реалізовувати SQL-запити вручну. Інтерфейси розміщені у пакеті ua.opnu.practice1_template.repository, а також security.repository – для роботи з авторизованими користувачами.

Усі інтерфейси наслідують JpaRepository<T, ID>, де T – тип сутності, а ID – тип ідентифікатора (у нашому випадку Long). Spring автоматично реалізує всі основні методи, такі як save, findById, findAll, deleteById.

```
UserRepository
public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User> findByEmail(String email);
}
```

Забезпечує доступ до таблиці users та дозволяє знайти користувача за email (унікальне поле).

ExpenseRepository:

```
public interface ExpenseRepository extends JpaRepository<Expense,
Long> {
List<Expense> findByUserId(Long userId);
List<Expense> findByUserIdAndDateBetween(Long userId, LocalDate
from, LocalDate to);
}
```

Методи для отримання витрат певного користувача та підтримка фільтрації за датою (місячні звіти, витрати за день тощо).

IncomeRepository:

```
public interface IncomeRepository extends JpaRepository<Income,
Long> {
  List<Income> findByUserId(Long userId);
  List<Income> findByUserIdAndDateBetween(Long userId, LocalDate
from, LocalDate to);
}
```

Аналогічна структура як у ExpenseRepository та дозволяє будувати звіти за доходами.

CategoryRepository

```
public interface CategoryRepository extends JpaRepository<Category,
Long> {
}
```

Базовий репозиторій для категорій та забезпечує операції створення, редагування, видалення та перегляду всіх категорій.

BudgetRepository

```
public interface BudgetRepository extends JpaRepository<Budget,
Long> {
  Optional<Budget> findByUserIdAndMonth(Long userId, String month);
}
```

Містить метод для перевірки, чи існує вже бюджет користувача на певний місяць та забезпечує унікальність по комбінації user_id + month.

AppUserRepository (для авторизації):

```
public interface AppUserRepository extends JpaRepository<AppUser,
Long> {
  Optional<AppUser> findByUsername(String username);
}
```

Працює з таблицею auth_users та використовується при вході через JWT або Google OAuth2. Додано метод findByUsername – базовий для Spring Security.

Усі репозиторії можуть бути розширені в майбутньому для складніших запитів (JPQL, Criteria API або SQL). Вони легко інтегруються в сервісний рівень через ін’єкцію залежностей.

## 3.3 Сервіси (бізнес-логіка)

Сервіси є проміжною ланкою між контролерами та репозиторіями. Вони реалізують бізнес-логіку застосунку, виконують перевірки, обчислення, модифікації даних, логіку агрегації та інші операції.

Кожен сервіс створений як окремий клас з анотацією @Service та отримує залежності через @RequiredArgsConstructor (Lombok).

UserService:

```
@Service
@RequiredArgsConstructor
public class UserService {
  private final UserRepository userRepository;
```

```
  public User create(User user) { return userRepository.save(user);
}

  public Optional<User> getById(Long id) { return
userRepository.findById(id); }

  public List<User> getAll() { return userRepository.findAll(); }

  public User update(Long id, User updatedUser) {
    User existing = userRepository.findById(id).orElseThrow();
    existing.setName(updatedUser.getName());
    existing.setEmail(updatedUser.getEmail());
    return userRepository.save(existing);
  }

  public void delete(Long id) { userRepository.deleteById(id); }
}
```

Стандартні CRUD-операції над користувачами та забезпечені оновлення лише дозволених полів (name, email).

ExpenseService:

- метод create перевіряє наявність User та Category перед збереженням витрати;
- getByUserId() – повертає всі витрати користувача;
- getByDate() та getByMonth() – дозволяють фільтрацію витрат по даті або місяцю;
- update() – редагує існуючу витрату (сума, дата, опис, категорія);
- delete() – видаляє витрату за ID.

IncomeService

```
@Service
@RequiredArgsConstructor
public class IncomeService {
  private final IncomeRepository incomeRepository;
```

```
  public Income create(Income income) {
    return incomeRepository.save(income);
  }

  public List<Income> getByUserId(Long userId) {
    return incomeRepository.findByUserId(userId);
  }

  public void delete(Long id) {
    incomeRepository.deleteById(id);
  }
}
```

Проста реалізація CRUD для доходів, додано додаткову фільтрацію можна легко додати за аналогією до ExpenseService.

CategoryService:

```
@Service
@RequiredArgsConstructor
public class CategoryService {
  private final CategoryRepository categoryRepository;

  public Category create(Category category) {
    return categoryRepository.save(category);
  }

  public List<Category> getAll() {
    return categoryRepository.findAll();
  }

  public Category update(Long id, Category updated) {
    Category existing =
categoryRepository.findById(id).orElseThrow();
    existing.setName(updated.getName());
    existing.setType(updated.getType());
    return categoryRepository.save(existing);
  }
```

```
  public void delete(Long id) {
    categoryRepository.deleteById(id);
  }
}
```

Підтримка створення, редагування та видалення категорій. Додано тип категорії (EXPENSE / INCOME) також змінюється під час оновлення.

BudgetService:

```
@Service
@RequiredArgsConstructor
public class BudgetService {
  private final BudgetRepository budgetRepository;

  public Budget create(Budget budget) {
    return budgetRepository.save(budget);
  }

  public List<Budget> getByUserId(Long userId) {
    return budgetRepository.findAll()
      .stream().filter(b ->
b.getUser().getId().equals(userId)).toList();
  }

  public Budget update(Long id, Budget updated) {
    Budget existing = budgetRepository.findById(id).orElseThrow();
    existing.setLimit(updated.getLimit());
    existing.setMonth(updated.getMonth());
    return budgetRepository.save(existing);
  }

  public Budget getByUserAndMonth(Long userId, String month) {
    return budgetRepository.findByUserIdAndMonth(userId,
month).orElse(null);
  }
}
```

Дозволяє створювати та оновлювати бюджети. Додано метод getByUserAndMonth() необхідний для подальших обчислень залишку та перевищення бюджету.

Всі сервіси побудовані з урахуванням повторного використання коду, спрощеної обробки винятків (orElseThrow()) та з підтримкою перевірок наявності пов’язаних сутностей.

## 3.4 Контролери (REST API)

Контролери реалізують рівень API і є точкою входу для клієнтських запитів. Кожен контролер відповідає за певний тип сутностей (користувачі, витрати, доходи, бюджети, категорії). Вони розміщені у пакеті ua.opnu.practice1_template.controller.

Контролери взаємодіють із відповідними сервісами через ін’єкцію залежностей (@RequiredArgsConstructor) і обробляють запити згідно зі специфікацією REST (GET, POST, PUT, DELETE). Дані приймаються у форматі JSON, а відповіді також повертаються у вигляді JSON-об’єктів.

UserController:

```
@RestController
@RequestMapping("/users")
@RequiredArgsConstructor
public class UserController {
  private final UserService userService;

  @PostMapping
  public User create(@RequestBody User user) { return
userService.create(user); }

  @GetMapping("/{id}")
  public User get(@PathVariable Long id) { return
userService.getById(id).orElse(null); }
```

```
  @PutMapping("/{id}")
  public User update(@PathVariable Long id, @RequestBody User user)
{
    return userService.update(id, user);
  }

  @DeleteMapping("/{id}")
  public void delete(@PathVariable Long id) {
userService.delete(id); }
}
```

Забезпечує повний CRUD для користувачів.

ExpenseController:

```
@RestController
@RequestMapping("/expenses")
@RequiredArgsConstructor
public class ExpenseController {
  private final ExpenseService expenseService;

  @PostMapping
  public Expense create(@RequestBody Expense expense) { return
expenseService.create(expense); }

  @GetMapping("/user/{userId}")
  public List<Expense> getAll(@PathVariable Long userId) { return
expenseService.getByUserId(userId); }

  @PutMapping("/{id}")
  public Expense update(@PathVariable Long id, @RequestBody Expense
expense) {
    return expenseService.update(id, expense);
  }

  @DeleteMapping("/{id}")
  public void delete(@PathVariable Long id) {
expenseService.delete(id); }
```

```
  @GetMapping("/user/{userId}/month/{month}")
  public BigDecimal getTotalForMonth(@PathVariable Long userId,
@PathVariable String month) { ... }

  @GetMapping("/user/{userId}/month/{month}/average")
  public BigDecimal getAveragePerDay(@PathVariable Long userId,
@PathVariable String month) { ... }

  @GetMapping("/user/{userId}/month/{month}/max")
  public Expense getMaxExpense(@PathVariable Long userId,
@PathVariable String month) { ... }

  @GetMapping("/user/{userId}/date/{date}")
  public List<Expense> getByDate(@PathVariable Long userId,
@PathVariable String date) { ... }
}
```

Реалізує не лише CRUD, а й додаткові звітні запити.

IncomeController:

```
@RestController
@RequestMapping("/incomes")
@RequiredArgsConstructor
public class IncomeController {
  private final IncomeService incomeService;

  @PostMapping
  public Income create(@RequestBody Income income) { return
incomeService.create(income); }

  @GetMapping("/user/{userId}")
  public List<Income> getAll(@PathVariable Long userId) { return
incomeService.getByUserId(userId); }

  @DeleteMapping("/{id}")
```

```
  public void delete(@PathVariable Long id) {
incomeService.delete(id); }
}
```

Контролер для роботи з доходами (створення, перегляд, видалення).

CategoryController:

```
@RestController
@RequestMapping("/categories")
@RequiredArgsConstructor
public class CategoryController {
  private final CategoryService categoryService;

  @PostMapping
  public Category create(@RequestBody Category category) { return
categoryService.create(category); }

  @GetMapping
  public List<Category> getAll() { return categoryService.getAll();
}

  @PutMapping("/{id}")
  public Category update(@PathVariable Long id, @RequestBody
Category category) {
    return categoryService.update(id, category);
  }

  @DeleteMapping("/{id}")
  public void delete(@PathVariable Long id) {
categoryService.delete(id); }
}
```

Контролер для управління категоріями витрат/доходів.

BudgetController:

```
@RestController
@RequestMapping("/budgets")
@RequiredArgsConstructor
public class BudgetController {
```

```
  private final BudgetService budgetService;
  private final ExpenseService expenseService;

  @PostMapping
  public Budget create(@RequestBody Budget budget) { return
budgetService.create(budget); }

  @GetMapping("/user/{userId}")
  public List<Budget> getAll(@PathVariable Long userId) { return
budgetService.getByUserId(userId); }

  @PutMapping("/{id}")
  public Budget update(@PathVariable Long id, @RequestBody Budget
budget) { ... }

  @GetMapping("/exceeded/{userId}/{month}")
  public BigDecimal getExceed(@PathVariable Long userId,
@PathVariable String month) { ... }

  @GetMapping("/report/{userId}/{month}")
  public List<String> reportByCategory(@PathVariable Long userId,
@PathVariable String month) { ... }

  @GetMapping("/left/{userId}/{month}")
  public BigDecimal getLeft(@PathVariable Long userId, @PathVariable
String month) { ... }
}
```

Контролер бюджету включає логіку звітності: перевищення бюджету, залишок, звіт по категоріях.

AuthController (JWT):

```
@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {
  private final AppUserRepository userRepository;
```

```
  private final BCryptPasswordEncoder passwordEncoder;
  private final JwtUtil jwtUtil;

  @PostMapping("/register")
  public String register(@RequestBody AppUser user) { ... }

  @PostMapping("/login")
  public ResponseEntity<?> login(@RequestBody AppUser loginData) {
... }
}
```

Забезпечує базову авторизацію через JWT: реєстрація та вхід.

Усі контролери відповідають принципам REST, мають зрозумілі шляхи, повертають JSON-результати та використовують HTTP-методи за призначенням.

## 3.5 Безпека та автентифікація (JWT, OAuth2)

Застосунок реалізує дві форми автентифікації користувача. Перша за допомогою логіну/паролю з використанням JWT-токенів, друга за допомогою облікового запису Google через протокол OAuth2.

Весь функціонал безпеки реалізований на основі Spring Security і структурований в окремий пакет security.

JWT-автентифікація

JWT (JSON Web Token) – це токен, який клієнт отримує після успішної автентифікації і використовує для авторизації подальших запитів.

AppUser – модель користувача для автентифікації:

```
@Entity
@Table(name = "auth_users")
@Data @NoArgsConstructor @AllArgsConstructor
public class AppUser {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String username;
  private String password;
```

```
  private String role;
}
```

AppUserRepository:

```
public interface AppUserRepository extends JpaRepository<AppUser,
Long> {
  Optional<AppUser> findByUsername(String username);
}
```

JwtUtil – генерує та перевіряє JWT:

```
@Component
public class JwtUtil {
  private final String SECRET = "...";
  public String generateToken(String username) { ... }
  public String extractUsername(String token) { ... }
  public boolean validateToken(String token) { ... }
}
```

JwtFilter – перехоплює запити, перевіряє токен, додає аутентифікацію у контекст:

```
@Component
public class JwtFilter extends OncePerRequestFilter {
  protected void doFilterInternal(...) { ... }
}
```

AuthController – контролер для логіну/реєстрації:

- @PostMapping("/register")  // /auth/register
- @PostMapping("/login")  // /auth/login
- OAuth2-автентифікація (через Google)

OAuth2 дозволяє користувачам входити у систему без реєстрації, використовуючи свій акаунт Google. Після успішного входу також видається JWT-токен. Розглянемо компоненти такої авторизації.

CustomOAuth2UserService – створює обліковий запис, якщо користувач вперше заходить:

```
@Service
public class CustomOAuth2UserService implements
OAuth2UserService<...> {
```

```
  public OAuth2User loadUser(...) {
    // перевірка наявності користувача у базі
    // створення нового AppUser, якщо не існує
  }
}
```

OAuth2LoginSuccessHandler – обробляє успішний логін, генерує JWT:

```
@Component
public class OAuth2LoginSuccessHandler extends
SimpleUrlAuthenticationSuccessHandler {
  public void onAuthenticationSuccess(...) {
    // генерує JWT та додає його у відповідь
  }
}
```

Конфігурація SecurityConfig – вмикає фільтри, задає правила безпеки:

```
@Bean
public SecurityFilterChain filterChain(...) {
  http
    .authorizeHttpRequests(...)
    .oauth2Login().userInfoEndpoint().userService(...)
    .successHandler(oAuth2LoginSuccessHandler)
    .and()
    .addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);
  return http.build();
}
```

Поведінка системи. При зверненні до захищених ресурсів користувач надсилає JWT у заголовку Authorization: Bearer <токен>. Без дійсного токена доступ до ресурсів буде заборонено. Користувач може автентифікуватися один раз, потім токен буде використовуватися для всіх запитів.

# 4 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ

Тестування через Postman. Було створено окремий набір запитів у Postman для кожного ресурсу: User, Expense, Income, Budget, Category, Auth. Запити супроводжувалися відповідними JSON-тілами, які змінювалися динамічно.

Приклади перевірених сценаріїв:

- реєстрація користувача (/auth/register) – відповідь 200 OK, створено користувача в БД;
- вхід користувача (/auth/login) – у відповідь приходить JWT-токен;
- додавання витрати (/expenses) – створено новий запис, перевірено збереження у базі;
- оновлення бюджету – зміна значення ліміту, перевірка впливу на розрахунок перевищення;
- звіт по категоріях – /budgets/report/{userId}/{month} – перевірено правильність агрегації;

Окрім позитивних, тестувалися також ситуації з помилками:

- спроба реєстрації з уже існуючим email – 400 Bad Request;
- спроба входу з неправильним паролем – 401 Unauthorized;
- створення витрати з неіснуючою категорією – 404 Not Found;
- запит без токена до захищеного ресурсу – 403 Forbidden.

Захист маршруту за токеном

Усі маршрути, крім /auth/** та /oauth2/**, були захищені авторизацією. При відсутності заголовка Authorization: Bearer ... сервер повертав помилку доступу.

Проєкт було завантажено на хостинг Render, після чого всі запити були повторно протестовані вже на продакшн-адресі. Основна увага приділялася:

- CORS-доступу;
- стабільності відповідей;
- обробці великих обсягів даних.

Тестування засвідчило стабільну роботу основного функціоналу, коректність автентифікації та валідну логіку розрахунків.

# ЗАГАЛЬНІ ВИСНОВКИ

У ході виконання курсової роботи було досягнуто цілісне розуміння процесу розробки сучасного веб-застосунку з використанням Java та Spring Boot. Робота над проєктом дозволила глибше зануритися у моделювання предметної області, побудову реляційних зв’язків між сутностями, організацію архітектури багаторівневого застосунку, а також впровадження механізмів автентифікації та безпеки.

Розроблена система на практиці довела ефективність підходу REST у побудові API, а також гнучкість фреймворку Spring у створенні масштабованих і підтримуваних рішень. Впровадження JWT та OAuth2 забезпечило сучасний рівень захисту доступу до даних. Інтерактивне тестування REST-запитів дозволило переконатися в надійності реалізованої логіки.

Загалом, реалізація цього проекту стала не лише практичним підтвердженням засвоєних теоретичних знань, але й прикладом створення прикладного програмного продукту, який може мати реальну цінність у повсякденному житті користувача.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Методичні вказівки до виконання курсової роботи з курсу «Об’єктно-орієнтоване програмування» для студентів всіх форм навчання спеціальності 122 «Комп’ютерні науки» / *Укл.: М.А. Годовиченко.* Одеса: Одеська політехніка, 2024. 59 с.

2. Spring Security Reference Guide – https://docs.spring.io/spring-security.
3. JWT – JSON Web Token Introduction – https://jwt.io/introduction.
4. OAuth 2.0 Overview – https://oauth.net/2/.