VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
INFORMATION TECHNOLOGIES STUDY PROGRAM

Software Architecture

Personal finances storing and analyzing website

Done by:
Oleksii Morozov

Supervisor:
Teach. Asst. Andrius Vytautas Misiukas Misiūnas

Vilnius
2025

Contents

Introduction

The essence of the project is to effectively control personal financial flows by entering data into the system. Each user will enter data on received and spent money to view the overall picture. The analysis will be available with total expenses, expenses for a certain period (day, week, month, year), saved money, deferred money, borrowed money, creating goals for the month, and budget distribution.

1 Planned functionality

- **I iteration:**
  1. User registration and authentication;
  2. Adding financial transactions.
- **II iteration:**
  1. Transaction Categorization;
  2. Filtering transactions;
  3. CRUD of categories.
- **III iteration:**
  1. Financial summary dashboard;
  2. Setting budget limits, and targets.
- **IV iteration:**
  1. Editing and deleting process for records;
  2. Expense Predictions.

2 Technologies

The Personal Finances Storing and Analyzing Website was developed using Python with the Django framework for both the backend and frontend, utilizing simple CSS development. Django handled user authentication, transaction management, and business logic, while also rendering dynamic HTML templates using Django's built-in Template Engine for the frontend. MySQL served as the database management system, ensuring efficient storage and retrieval of financial data. The application was hosted on an Apache web server, which managed incoming requests and served the application securely. For deployment, Virtual Machines (VMs) were used, providing scalability, system isolation, and efficient resource allocation. This technology stack ensured a reliable, secure, and high-performance platform for managing personal finances.

3 Non-functional requirements

The Personal Finances Storing and Analyzing Website must meet the following non-functional requirements to ensure performance, security, and usability:

- Performance & Scalability – Support 1,000+ concurrent users, response time 2s, optimized queries and caching.
- Security – Encrypt data, use hashed passwords, protect against SQL injection, XSS, CSRF, and support MFA.
- Availability & Reliability – Ensure 99.9% uptime, daily automated backups, and automatic recovery.
- Usability & Accessibility – Provide a responsive UI, follow WCAG 2.1, and offer an intuitive dashboard.
- Maintainability & Extensibility – Use Django best practices, modular design, and maintain clear documentation.

4 UML diagrams

4.1 Class diagram

The class diagram represents a financial tracking system where users can manage their income and expenses.

Each User has an account and can track multiple financial Records, which represent transactions such as income or spending. Users can sign up, log in, and access their financial records to monitor their budget and spending habits. Each record belongs to a specific user and includes details like the amount, category (e.g., food, rent), and transaction type (income or expense). The system allows users to add, update, and delete records, as well as view summaries of their total expenses and income over a given period. This structure ensures that users can effectively track and analyze their financial activities in a simple and organized way.

4.2 Use-Case diagram

The use case diagram illustrates how users interact with the Finances Storing and Analyzing Website to manage their financial transactions.

Users can sign up to create an account and sign in to access their financial records. They can add, edit, categorize, and delete records to track income and expenses, with the option to set transactions as recurring. The system allows users to filter records and view financial analysis, helping them monitor spending habits and set budget limits. Additionally, users can access expense predictions based on past data, providing insights for better financial planning. The diagram ensures a smooth and intuitive experience for managing personal finances effectively.

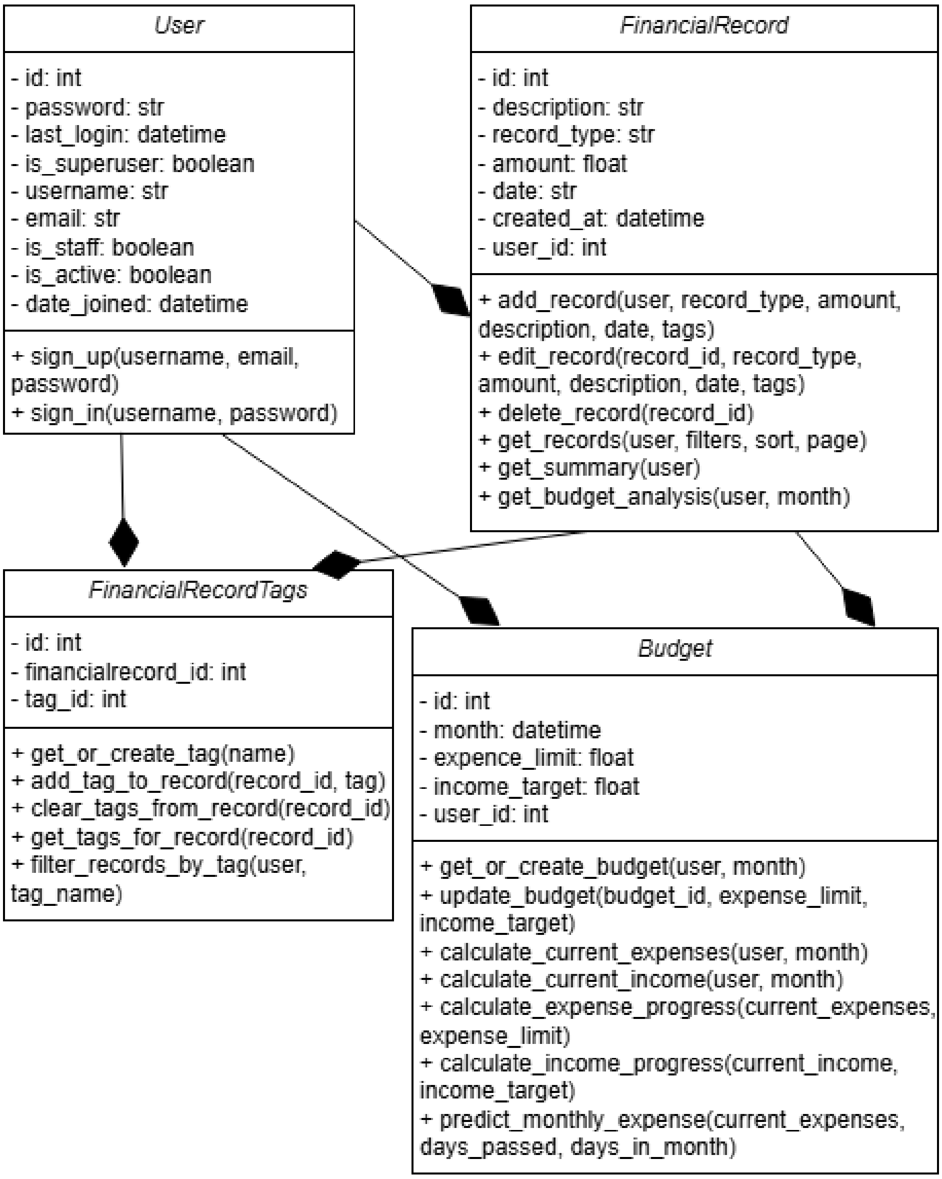
User
- id: int
- password: str
- last_login: datetime
- is_superuser: boolean
- username: str
- email: str
- is_staff: boolean
- is_active: boolean
- date_joined: datetime
+ sign_up(username, email, password)
+ sign_in(username, password)
FinancialRecord
- id: int
- description: str
- record_type: str
- amount: float
- date: str
- created_at: datetime
- user_id: int
+ add_record(user, record_type, amount, description, date, tags)
+ edit_record(record_id, record_type, amount, description, date, tags)
+ delete_record(record_id)
+ get_records(user, filters, sort, page)
+ get_summary(user)
+ get_budget_analysis(user, month)
FinancialRecordTags
- id: int
- financialrecord_id: int
- tag_id: int
+ get_or_create_tag(name)
+ add_tag_to_record(record_id, tag)
+ clear_tags_from_record(record_id)
+ get_tags_for_record(record_id)
+ filter_records_by_tag(user, tag_name)
Budget
- id: int
- month: datetime
- expence_limit: float
- income_target: float
- user_id: int
+ get_or_create_budget(user, month)
+ update_budget(budget_id, expense_limit, income_target)
+ calculate_current_expenses(user, month)
+ calculate_current_income(user, month)
+ calculate_expense_progress(current_expenses, expense_limit)
+ calculate_income_progress(current_income, income_target)
+ predict_monthly_expense(current_expenses, days_passed, days_in_month)

Figure 1. Class diagram

5 Database

The Entity-Relationship (ER) Diagram represents the database structure for a financial management system, illustrating the relationships between entities. The User entity contains attributes such as ID, username, password, and email, which uniquely identify and authenticate each user in the system. A User has a one-to-many (1:M) relationship with the *Financial_Record* entity, indicating that a single user can manage multiple financial records. The *Financial_Record* entity, uniquely identified by RID (Record ID), stores essential transactional attributes, including *R_type* (record type: income or expense), description, amount, category, and date. The Controls relationship ensures that users can manage their financial data effectively. This ER diagram provides a structured representation of how financial transactions are stored, categorized, and linked to users within the database, ensuring data integrity and efficient retrieval.

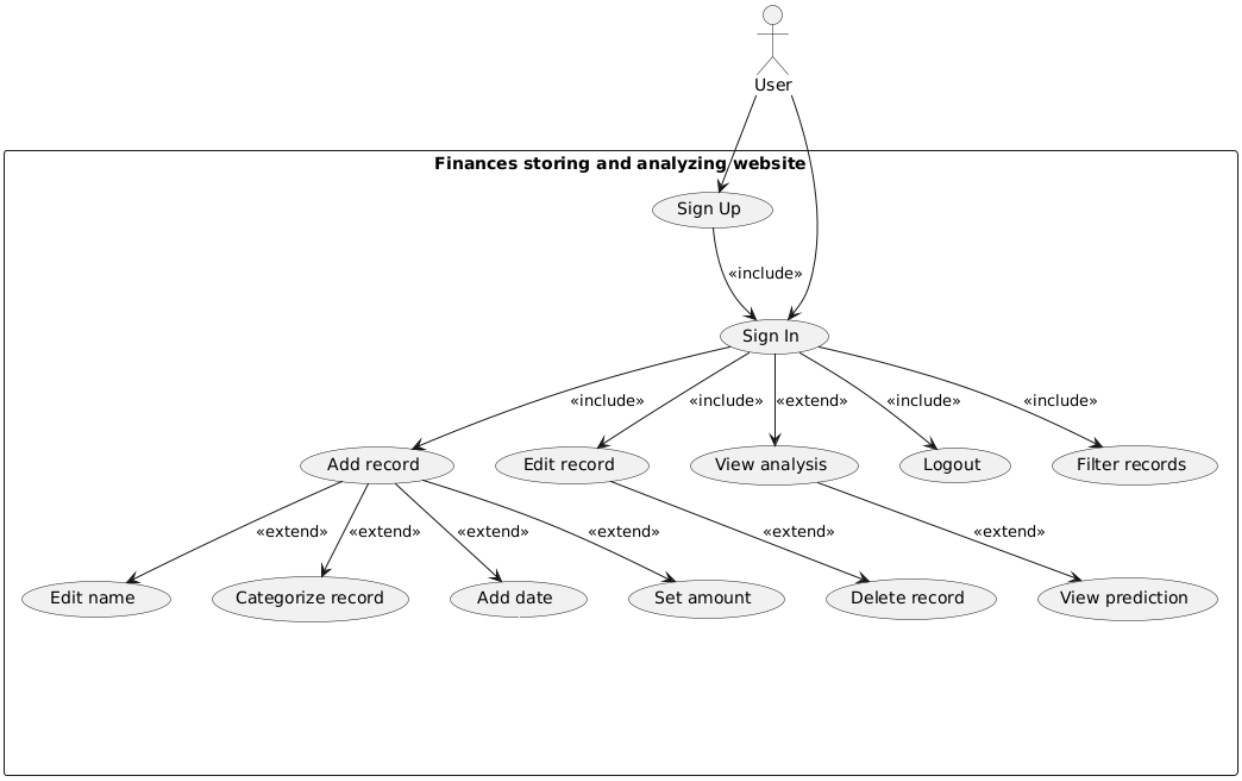

Figure 2. Use case diagram

6 Changes from preliminary plan

In the updated version of the system diagrams, several significant improvements have been made compared to the original version, which only included two entities: User and Record. The revised ER diagram introduces two new entities — Budget and FinancialRecordTags — enabling the implementation of personalized budgeting per user per month and many-to-many relationships between financial records and tags. The class diagram has been expanded accordingly: Budget includes fields such as income_target and expense_limit, with methods for calculating progress and predictions, while FinancialRecordTags manages tag creation, association, and filtering. The FinancialRecord class itself now supports additional operations like get_summary and get_budget_analysis, reflecting its integration with budgeting and analysis features. The use case diagram has also become more detailed, now including extended user interactions such as editing record details, categorizing, adding dates, marking records as recurring, and viewing predictions, all of which provide a clearer and more functional representation of the system's capabilities.

7 How to run the project

To run the project, start by creating virtual machines to host the application. Then install Apache web server and Django framework along with all necessary dependencies such as Python, pip, and required Python packages. After that, import the project code while preserving the existing folder structure. Once the code is in place, apply database migrations using Django's migration system to set up the required tables. Finally, configure and start the Apache server to serve the Django application.

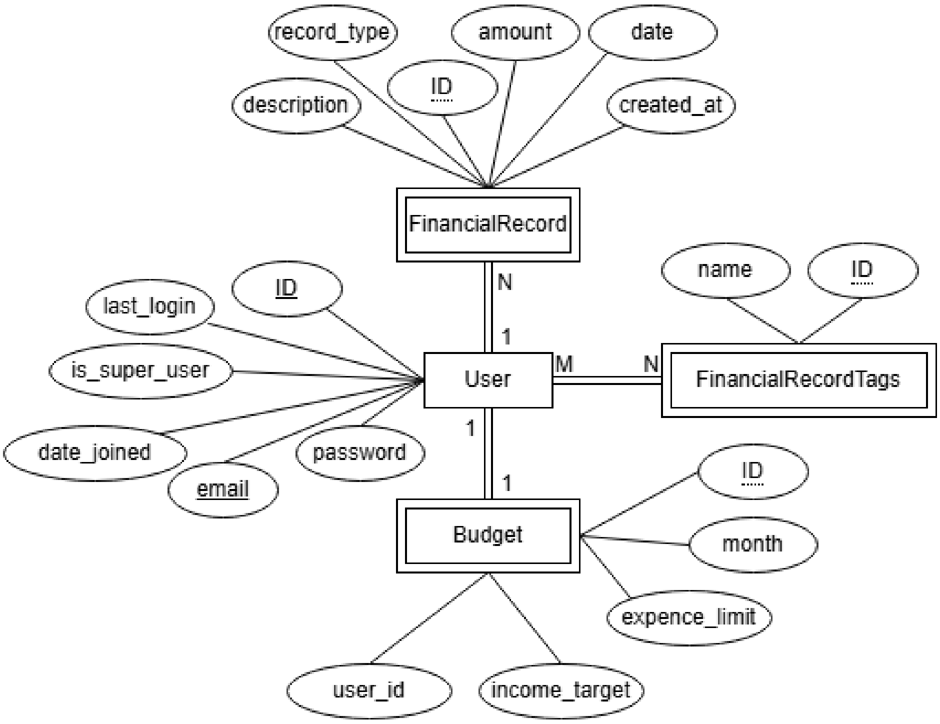

Figure 3. ER diagram

8 What's used from the framework

In the views.py file, all business logic, including handling form data, validating user input, processing financial records, managing tags, calculating summaries, and rendering templates, was written manually by the developer. At the same time, the code makes extensive use of Django's built-in features and libraries such as render, redirect, messages, authenticate, login, login_required, and model-related utilities like get_object_or_404, Paginator, Sum, and TruncMonth to simplify common web development tasks such as authentication, messaging, pagination, and database querying. This combination of custom logic and Django framework tools provides a structured, efficient approach to building and managing the application's user interactions and financial data operations.

The models.py file combines custom model definitions with Django's built-in ORM functionality. The developer manually defined the structure and logic of three models: Tag, FinancialRecord, and Budget, specifying fields, relationships, and behaviors tailored to the application's financial tracking needs. For instance, the FinancialRecord model includes user linkage, typed choices for records, timestamps, and a many-to-many relationship with tags. Similarly, the Budget model includes unique constraints per user per month and custom ordering. These models inherit from django.db.models.Model, leveraging Django's built-in field types (CharField, DecimalField, DateField, ForeignKey, ManyToManyField), model behaviors like auto_now_add, and meta-options such as unique_together and ordering. Thus, while the developer defined the data structure and domain logic, Django's framework provides the underlying implementation for data persistence, validation, and relationship handling.