CSC 431

QuantEstate

System Architecture Specification (SAS)

Group 9

Eddy Boris	Product Manager
Anish	Lead Developer
Diego	Scrum Master

Version History

Version	Date	Author(s)	Change Comments
1.0	3/24/25	Eddy, Anish, Diego	Create initial draft of SAS up to and including functional design
1.1	4/3/25	Eddy, Anish, Diego	Complete the rest of the first draft
2.0	4/14/25	Eddy	Final touches to ensure that SAS document is properly formatted and complete.
2.1	4/20/25	Eddy	Small changes to Sequence Diagrams and Structural Design to align with Professor Nguyen's feedback

Table of Contents

1. System Analysis

1.1 System Overview

QuantEstate is a web-based real estate recommendation application that provides investors or homebuyers with return-on-investment (ROI) insights. The system aggregates property listings, interest rates, and mortgage data, runs machine learning or statistical models to compute ROI forecasts, and then recommends properties to users based on their preferences.

Key capabilities from the SRS include:

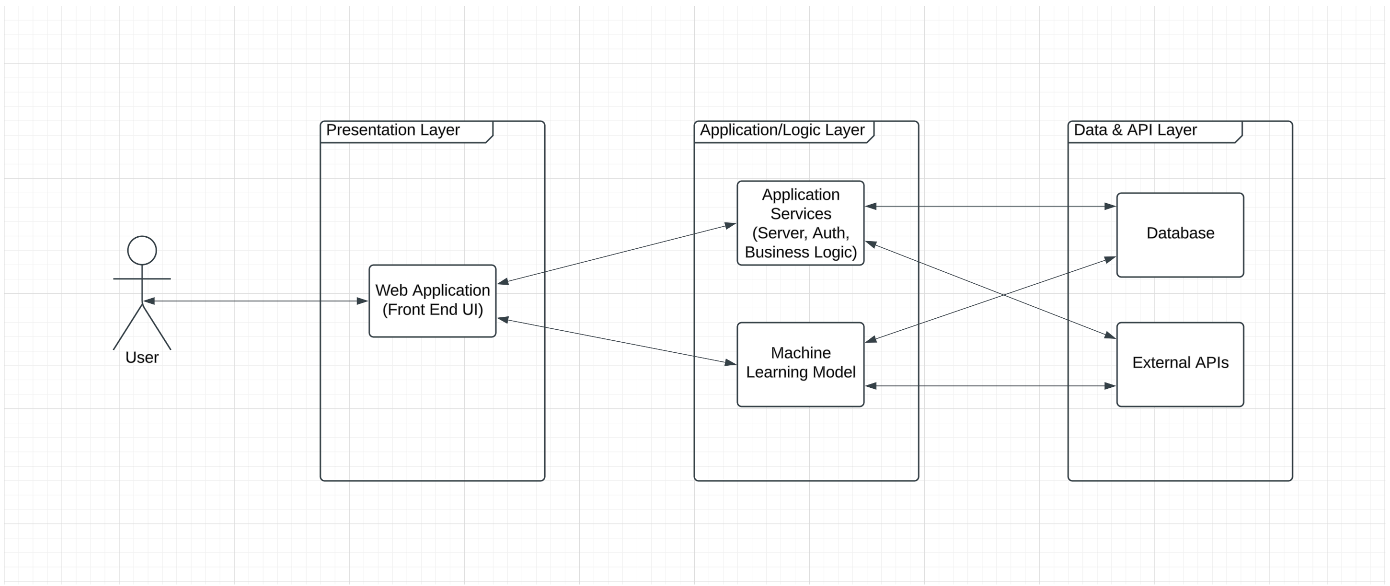
- User account management (register, login, profile updates)
- Data collection from external APIs (Zillow, mortgage providers)
- Data cleaning & transformation (via Python scripts/services)
- ROI computation (using machine learning/statistical models)
- Property recommendation based on user profiles & computed ROI

To accomplish this, QuantEstate follows a three-tier architecture ensuring separation of concerns:

1. Presentation Tier (front-end)
2. Application Logic Tier (business logic in Node.js or Python services)
3. Data / API Tier (databases and third-party APIs)

This architecture promotes maintainability, scalability, and ease of feature extension—key attributes for a startup-like environment with a small team of three engineers.

1.2 System Diagram



1.3 Actor Identification

There are three main actors who may interact with the QuantEstate system: Non-registered Users, Registered Users, and System Administrators.

1. Non-Registered Users:
    a. This actor visits the web application without having created an account. Their primary interactions include viewing limited, public-facing information and initiating the registration process. They do not have access to personalized recommendations or ROI metrics.
2. Registered Users:
    a. This is the typical user who logs into the system with valid credentials. They can access the complete functionality, which includes browsing available properties, applying filters (like budget or geography), and viewing computed ROI metrics to help make informed investment decisions. Registered users can also save or favorite properties, manage their account settings, and receive relevant alerts about market changes.
3. System Administrator:
    a. The System Administrator is responsible for overseeing and configuring the application environment. Administrators ensure the machine learning models stay updated, control database migrations, and handle troubleshooting ensuring the system continues to function as intended.

1.4 Design Rationale

1.4.1 Architectural Style

QuantEstate adopts a three-tier architectural style because it clearly separates the user interface (front end) from the business logic (application layer) and the data storage (database and external APIs). This approach promotes:

1. Modularity: Each tier focuses on a distinct set of responsibilities, display logic, application logic, or data management, making it easier to maintain and scale specific components independently.
2. Flexibility: The front-end could evolve separately without disrupting back-end logic, while the back end can swap out or upgrade underlying data structures with minimal impact on the user interface.
3. Scalability: If usage grows, additional Node.js servers or containerized ML modules could be spun up without requiring a full rewrite. Similarly, the database can be scaled horizontally or vertically depending on performance requirements.

This layering ensures consistent user experiences, isolates business logic for maintainability, and allows for scalable future enhancements.

1.4.2 Design Pattern(s)

In order to streamline the development process and ensure the greatest degree of compatibility between the various platforms and the application logic, some common software design patterns will be deployed. Some examples of these patterns include:

- Model-View-Controller (MVC)
    - At the back-end level, the Node.js application loosely follows the MVC pattern. We have models to represent data entities such as Users or Properties, Controllers to handle incoming HTTP requests, translating them into service calls and returning appropriate responses and views that are not strictly defined in the back-end since the presentation is handled by React/Next.js on the front end. However, the concept remains valuable for structuring controllers and model logic in separate modules.

- Factory Pattern
    - As data acquisitions can involve creating new property objects or user objects systematically, a Factory Pattern may be used to simplify the logic for constructing these objects based on external data. This pattern provides a consistent interface for generating domain-specific objects that represent the information from Zillow or other APIs, simplifying the code needed to handle diverse incoming data formats.

- Singleton Pattern
    - Database connections or application-level configuration can often follow the Singleton Pattern, ensuring that only a single instance of a connection pool or configuration manager is in use throughout the system. This helps manage resources efficiently and avoid synchronization issues that might arise if multiple connections attempt to modify the same shared data concurrently.

1.4.3 Framework

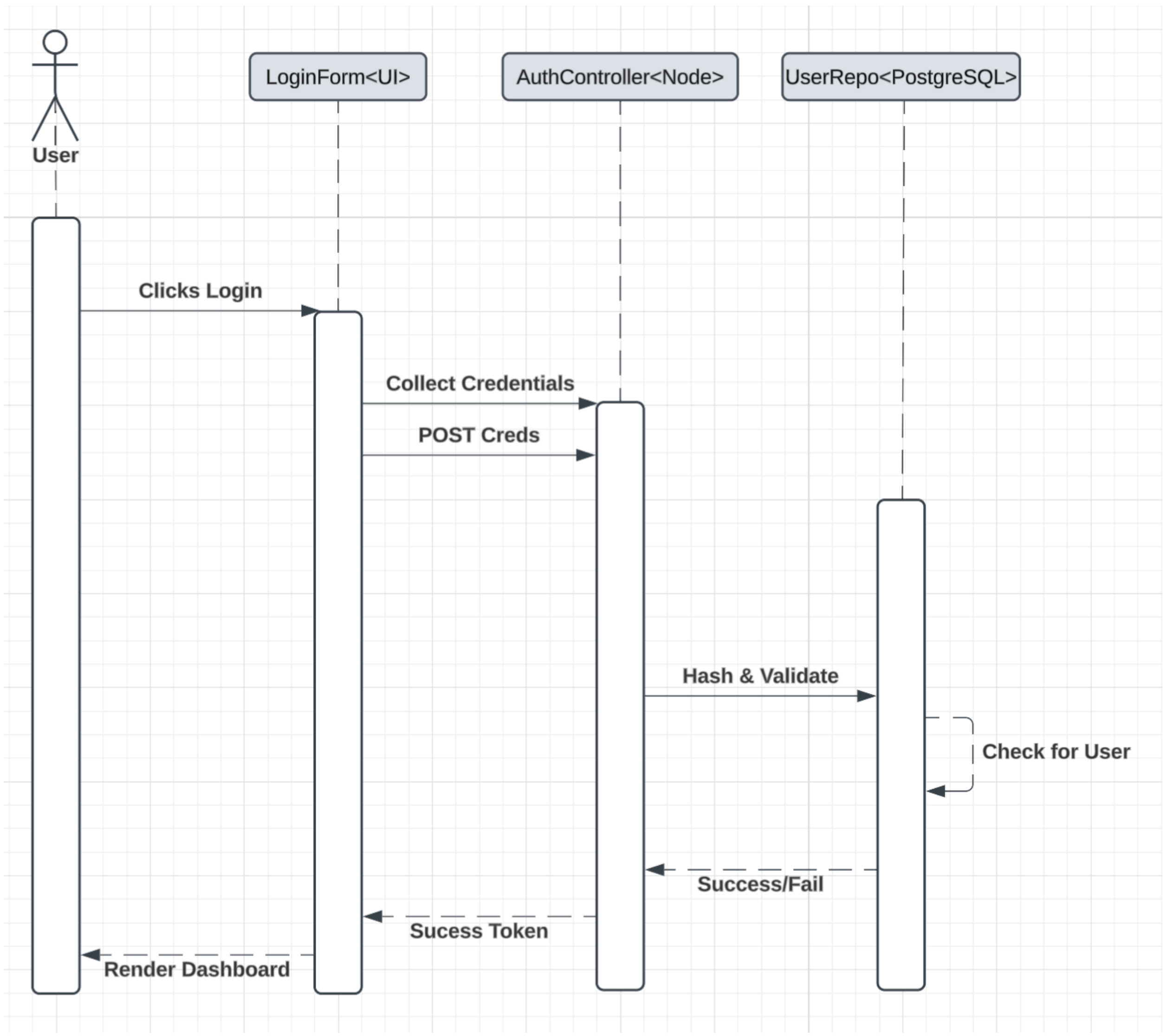
QuantEstate leverages several frameworks and technologies:

1. React/Next.js
    a. This framework is used in the front end to create a dynamic, responsive web interface. Next.js enables server-side rendering to improve performance and SEO while React delivers a modular component model that facilitates maintainable UI development.
2. Node.js/Express
    a. This environment provides a lightweight, flexible setup for developing RESTful APIs or GraphQL endpoints. Express routing enables clear definitions of endpoints for user management, property listing retrieval, and ROI computations.
3. Python (with pandas, scikit-learn, or TensorFlow)
    a. Chosen for data ingestion, cleaning, and machine learning tasks. Python's data science foundation ensures fast development for ROI computation and advanced analytics, which can be exposed to Node.js through a microservice architecture or simple script calls.
4. PostgreSQL
    a. An open-source relational database, PostgreSQL is both robust and flexible, making it suitable for storing user information, property records, and results of ROI calculations. SQL's maturity and reliability make it an ideal candidate in a real-world setting.

2. Functional Design

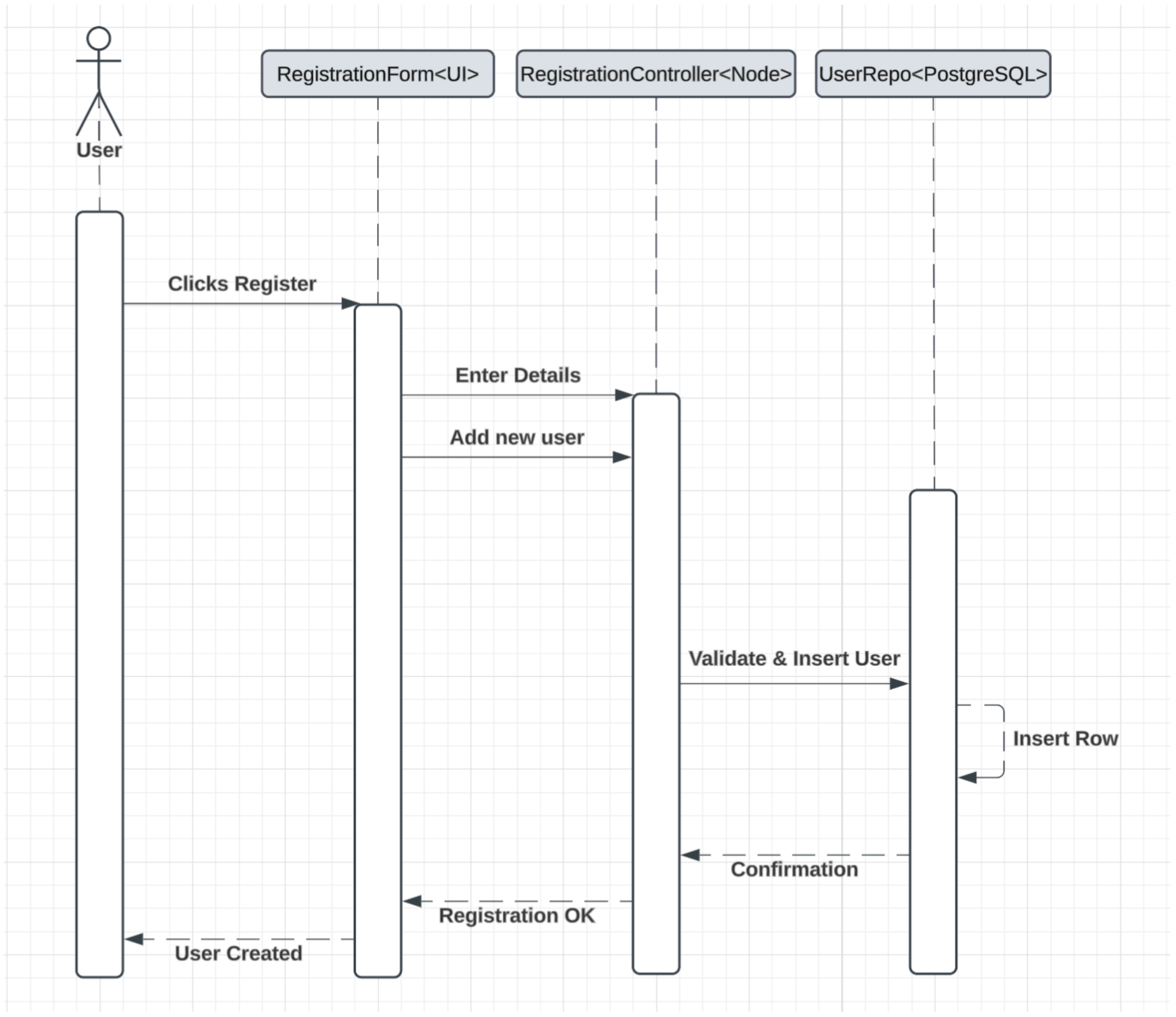
<Identify all significant workflows as sequence diagrams using the following format>

2.1 QuantEstate Login



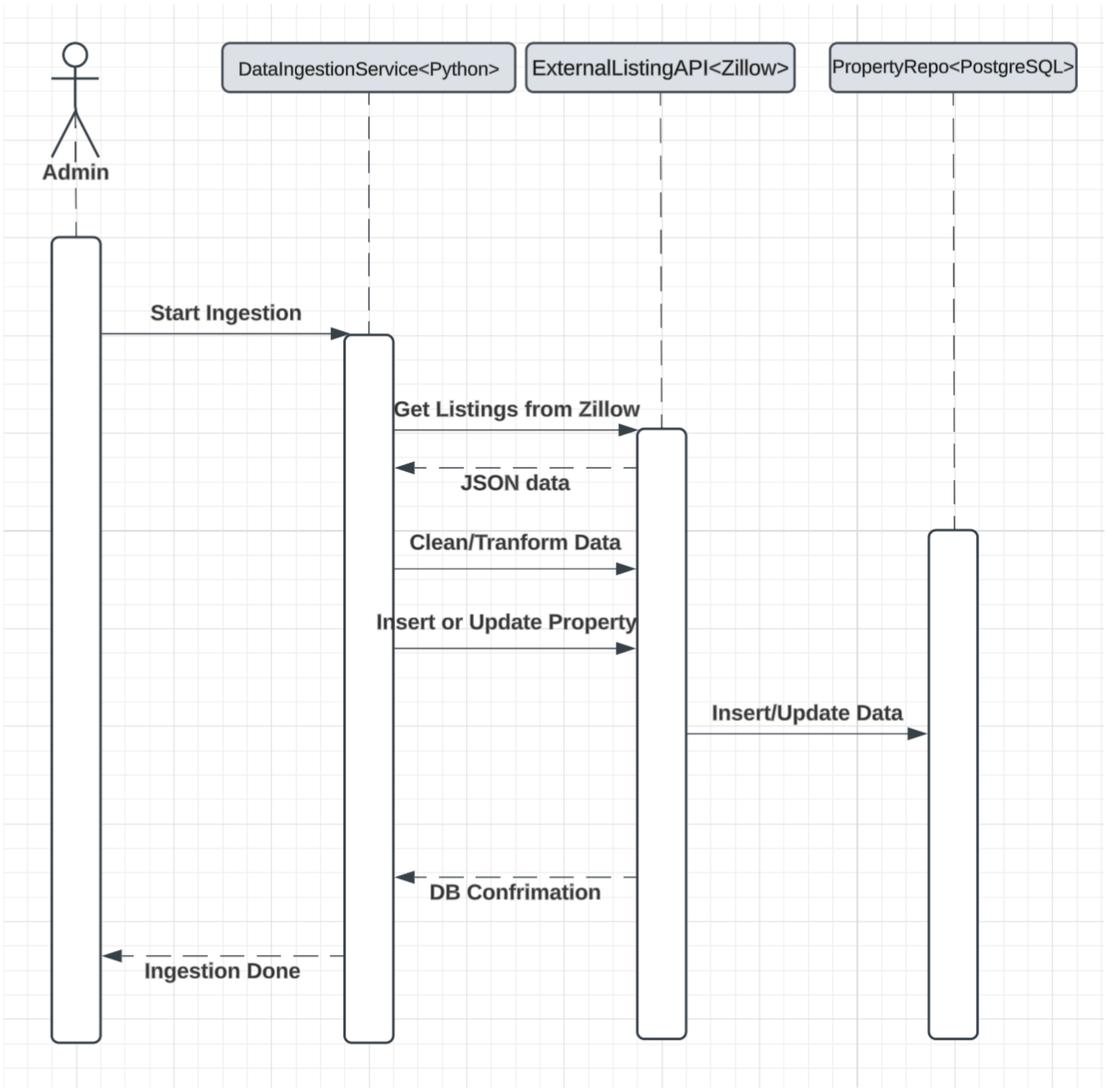
When a user presses Login on the site, the LoginForm React component gathers the email and password and posts them over HTTPS to the AuthController class in the Node.js/Express layer. AuthController hashes the password and asks UserRepository to look up a matching record in PostgreSQL. If a match is found, the controller returns a signed token; otherwise, it responds with an error message. The token is stored client-side so subsequent page loads can display the user's personalized dashboard.

2.2 QuantEstate Register



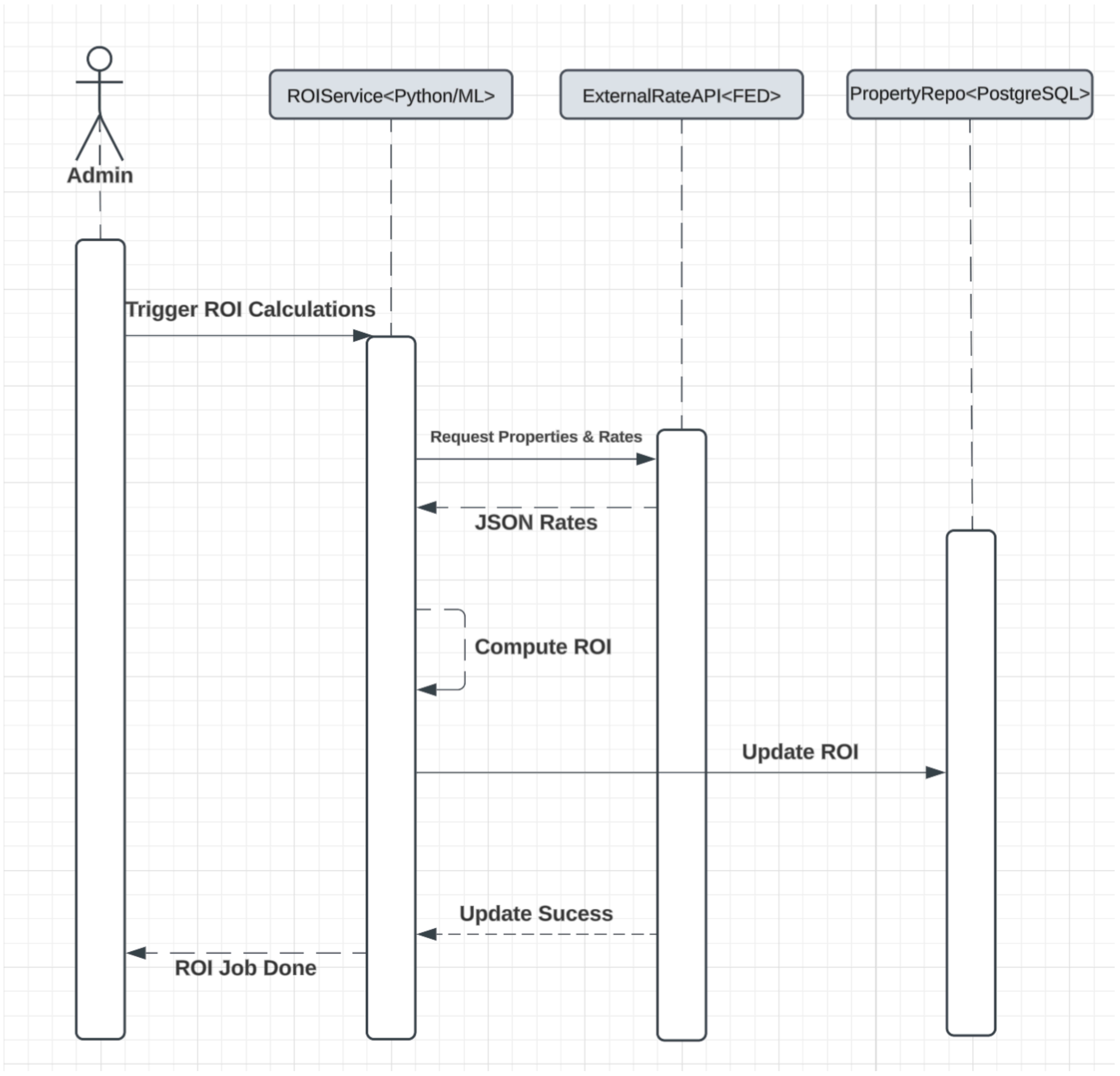
During registration, the RegistrationForm component submits the prospective user's details to the RegistrationController. That controller first calls UserRepository to be sure the email is not already taken, then hashes the password and inserts a new user row. On success, the controller sends a confirmation that the front end turns into either an automatic login or a prompt to log in. This workflow guarantees every account remains unique and meets password-strength rules before it is written to the database.

2.3 Data Collection Workflow



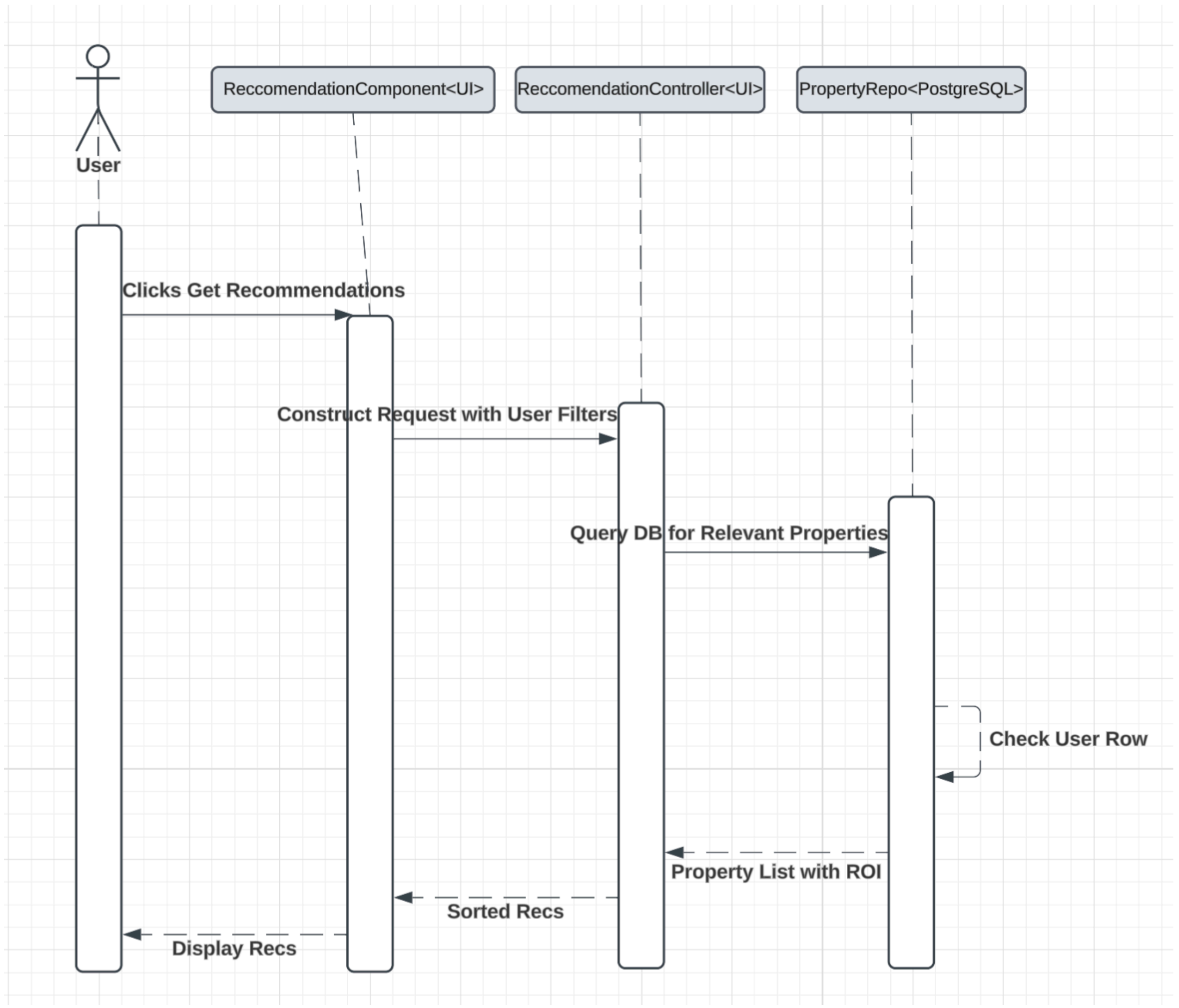
An admin triggers the DataIngestionService script, which fetches the latest listings from the ExternalListingAPI (e.g., Zillow). The service cleans and normalizes each JSON record, then calls PropertyRepository to insert those records into PostgreSQL. Any anomalies or duplicates are logged, and the service publishes a completion message for administrators. This automated pipeline keeps property data and interest-rate snapshots fresh.

2.4 ROI Computation Workflow



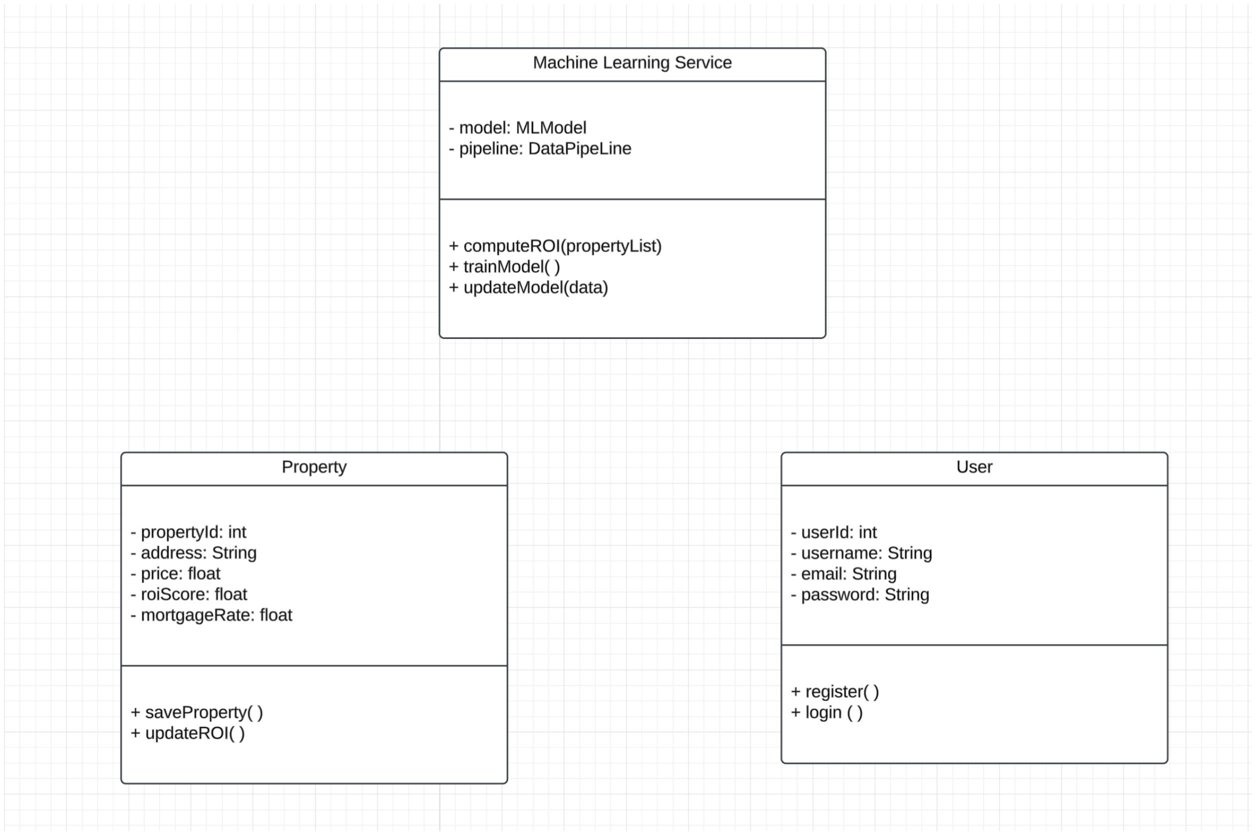
In this workflow, the ROIService loads property rows from PropertyRepository and current mortgage rates from an ExternalRateAPI. Using its embedded machine-learning model, the service computes a projected ROI for every property and writes the new scores back to the database. This workflow ensures that every recommendation is backed by the latest financial calculations.

2.5 Property Recommendation Workflow


User
ReccomendationComponent<UI>
ReccomendationController<UI>
PropertyRepo<PostgreSQL>
Clicks Get Recommendations
Construct Request with User Filters
Query DB for Relevant Properties
Check User Row
Property List with ROI
Sorted Recs
Display Recs

When a logged-in user clicks “Get Recommendations,” the RecommendationComponent posts the user’s filters to the RecommendationController. The controller queries PropertyRepository for properties that match those filters, then sorts the results by the ROI scores produced in the previous workflow. The top listings are returned to the front end, which renders them as interactive cards with price, location, and ROI highlights. In this way, the system combines real-time preferences with data-science outputs to present high-value investment opportunities.

3. Structural Design


Machine Learning Service
- model: MLModel
- pipeline: DataPipeLine
+ computeROI(propertyList)
+ trainModel( )
+ updateModel(data)
Property
- propertyId: int
- address: String
- price: float
- roiScore: float
- mortgageRate: float
+ saveProperty( )
+ updateROI( )
User
- userId: int
- username: String
- email: String
- password: String
+ register( )
+ login ( )