MIDDLE EAST TECHNICAL UNIVERSITY
NORTHERN CYPRUS CAMPUS

Computer Engineering Program

CNG 495

CLOUD COMPUTING

FALL 2025
Capstone Progress Report
SmartRent

Team Members:

Mahlet Bekele- 2643146

Zeeshan Imran - 2640779

Miguel Tunga Mbabazi- 2600195

TABLE OF CONTENT

Cloud-based Smart Property Management & Rental Platform

SmartRent is a web-based property and rental management system that unifies landlords and tenants under one integrated system. The main purpose of the system is to allow tenants to securely log in, pay rent/bills online, view property details, and submit maintenance requests. Landlords can manage multiple properties, track rent payments, view maintenance history, and update request status in real-time.

We plan to achieve the following mentioned functionalities by the end of this project. These important features include user authentication, role-based access (landlord and tenant), rental and utility payments, maintenance requests submission, and a bill payment reminder.

For scalability and privacy, the system will use a multi-tenant SaaS approach, with each landlord's data segregated. The application will include rent reminders, payment reminders, and maintenance records, and provide a transparent, streamlined, and efficient process for all users.

IMPLEMENTATION:

We plan on implementing a full stack project with proper integration of the frontend with the backend and the database. In our system, the user interacts with the buttons for registering, paying rent or other functionalities, these requests are sent in the form of APIs with the backend. It should have a seamless integration and accurately update the house details(occupied/not_occupied) in real-time and the tenant details for that occupied house.

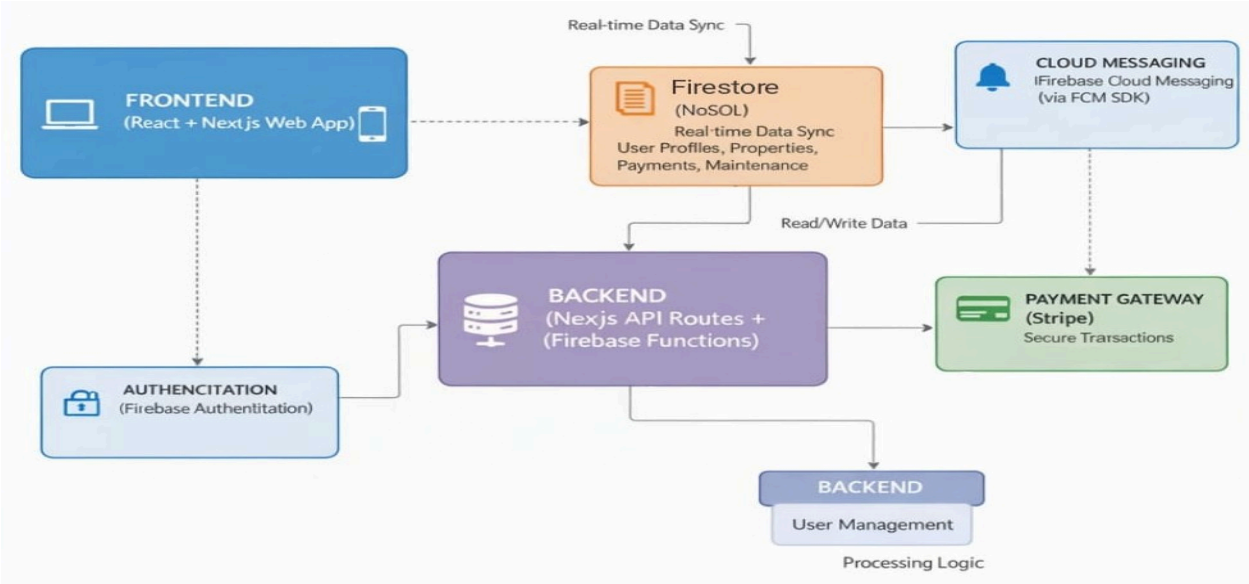


Figure 1: Application Environment Integration flow

Frontend (React + Next.js)

We will build a responsive web frontend for landlords as well as tenants. There would be a separate registration page and login page. The pages we create will be a rent payment dashboard, maintenance request form, reminders and notification page. We'll build the project in React in a Next js framework, which will give us faster loading time, better performance and future scalability for growth.

Backend (Next.js API Routes + Firebase Functions)

Backend would be built through Next js API routes and Firebase Functions to host the server-side code. Next js API gives a straightforward interaction of frontend with the backend

server for achieving a Secure authentication and role-based authorization, Payment gateway integration (Paddle/Stripe), request routing for maintenance, and rent reminder system.
Firebase is a cloud-based app development platform that provides developers with pre-built backend services. It contains numerous services from authentication, databases, cloud functions, and cloud storage. It is ideal as it enables rapid development and scalability.

Database (Firestore NoSQL)

For data storage and retrieval, we will use Firestore, Firebase's cloud NoSQL database. It's data as collections and documents and is ideal as it offers real-time data synchronization across the clients and the backend and fast development. It will hold user profiles, property details, payments, maintenance requests, and payment status. It handles large volumes of data well.

CLOUD DELIVERY MODELS:

1. **SaaS**

Stripe / Paddle
Provides integrated secured payment processing functionality via cloud APIs.

2. **PaaS**

Firebase (Firestore, Cloud Messaging, Hosting, Authentication):
Provides backend and platform features like real-time database, push, secure authentication, and hosting for web. We can focus on app development while Firebase takes care of infrastructure, scalability, and servers.
Vercel / Render
Hosts React/ Next.js frontend and API routes with automatic building, deployment, and future scaling. Offers a managed platform environment in which we are not required to configure it or manage servers.

DIAGRAMS:

Use Case diagram

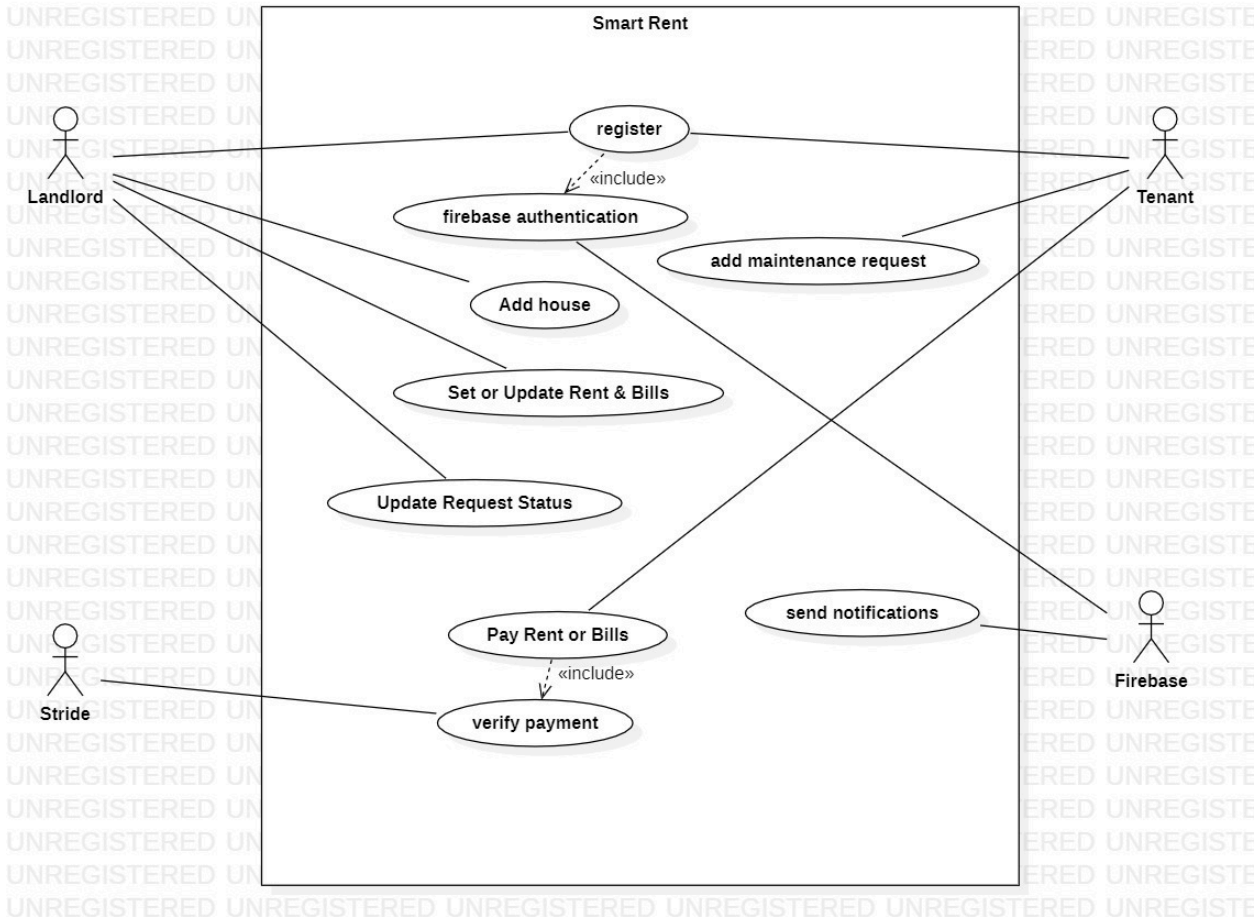


Figure 2: UseCase Diagram

Data flow diagrams

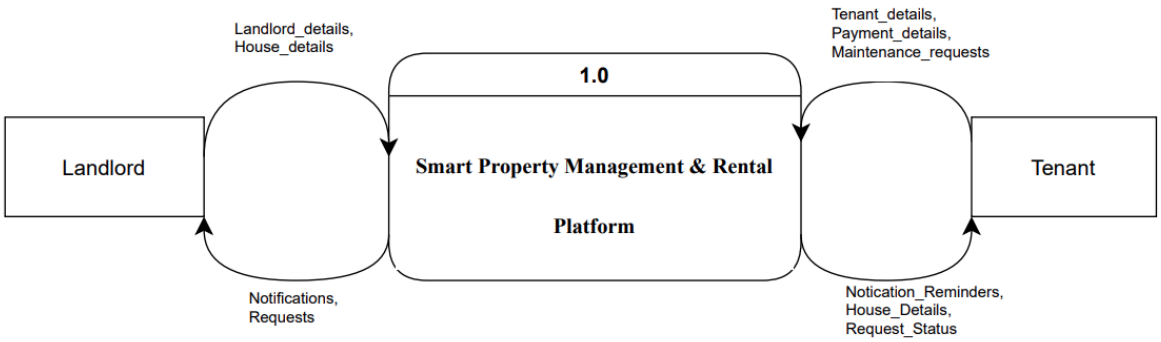


Figure 3: Level-0 Dataflow Diagram

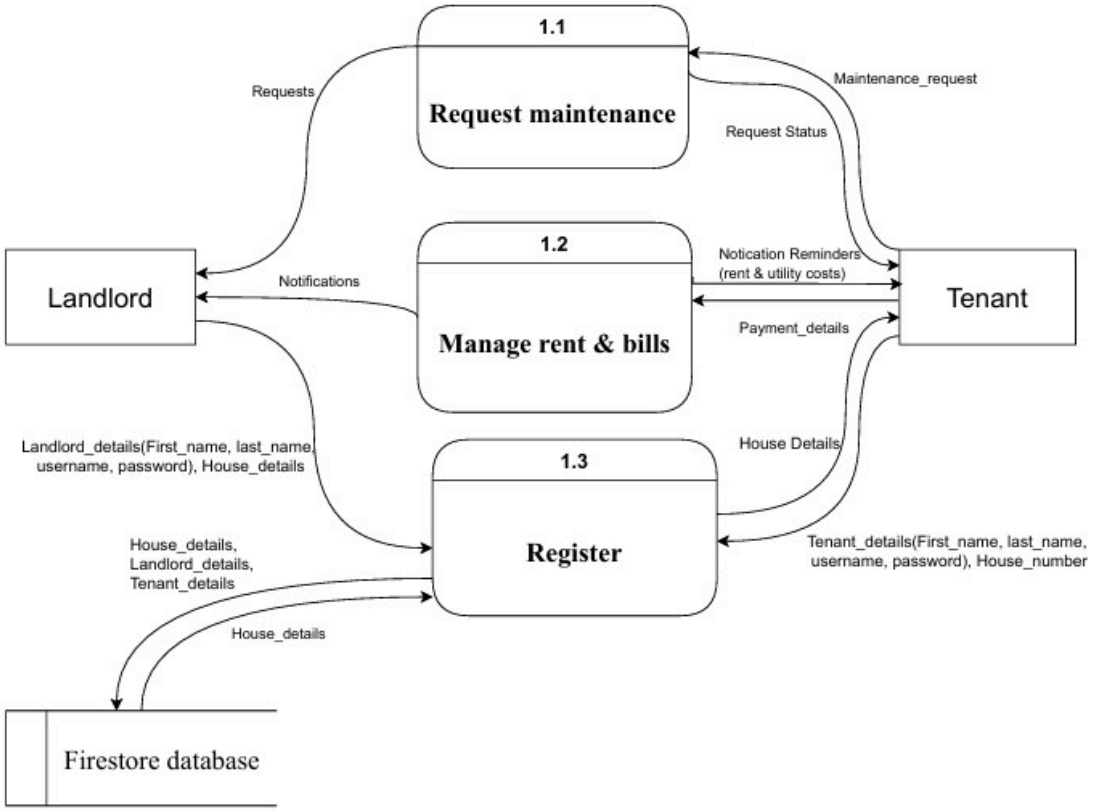


Figure 4: Level-1 Dataflow Diagram

Sequential diagrams

Landlord Interface

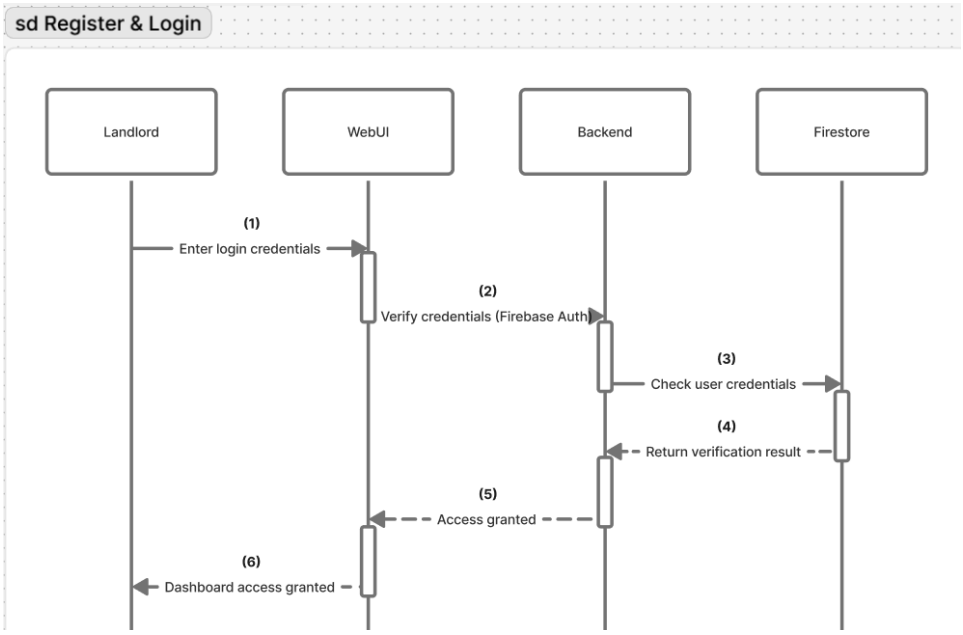


Figure 5: Register & Login

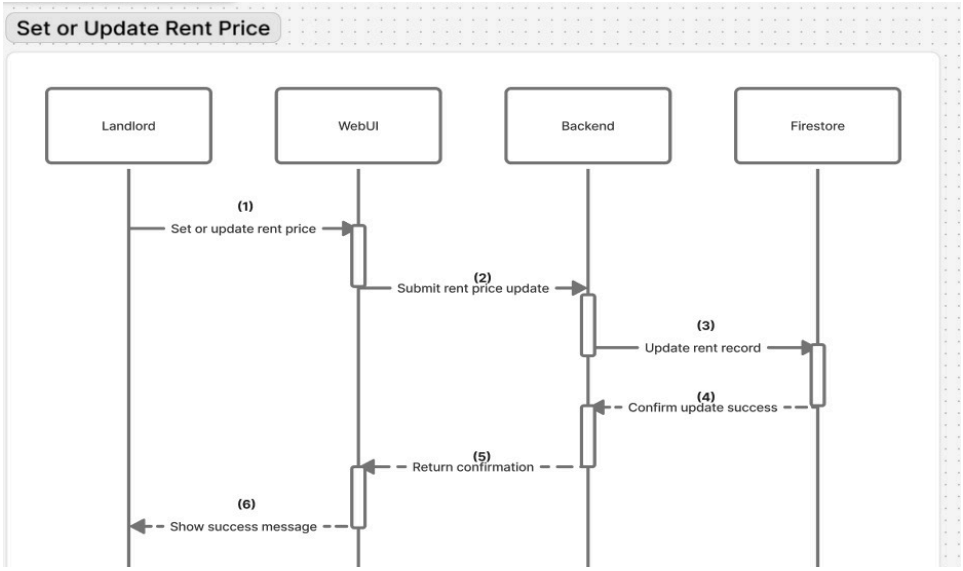


Figure 6: Set or Update Rent Price

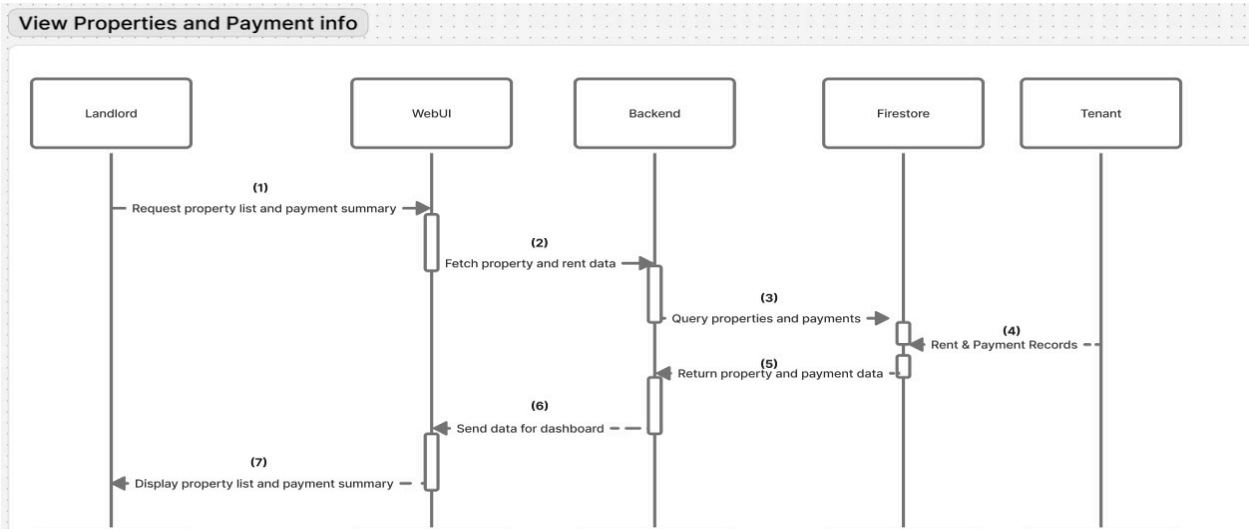


Figure 7: View Properties and Payment info

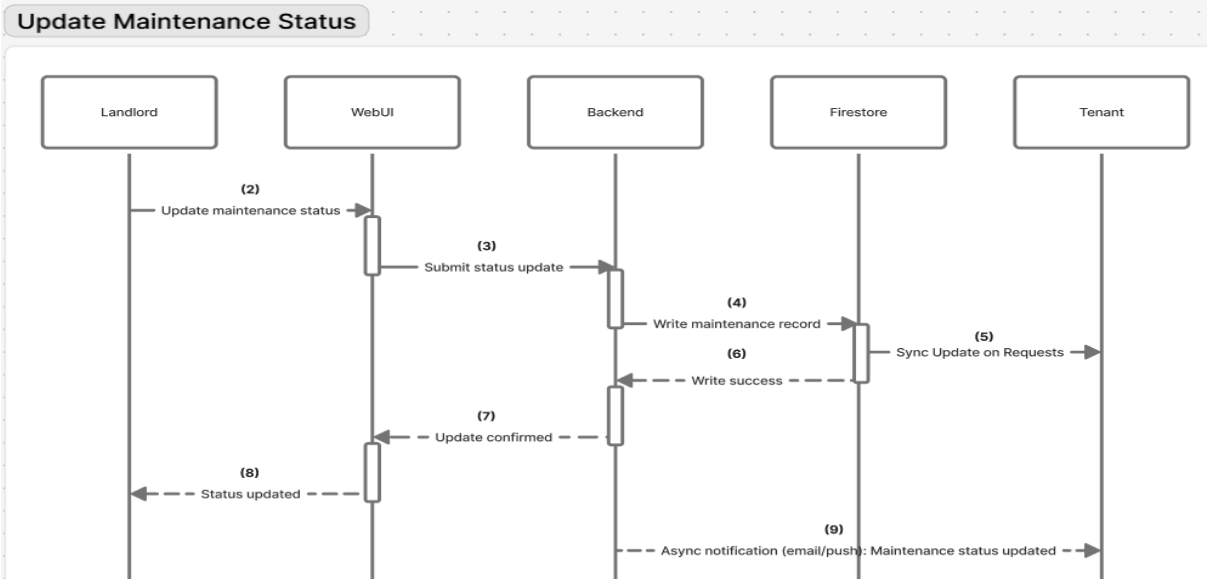


Figure 8: Update Maintenance Status

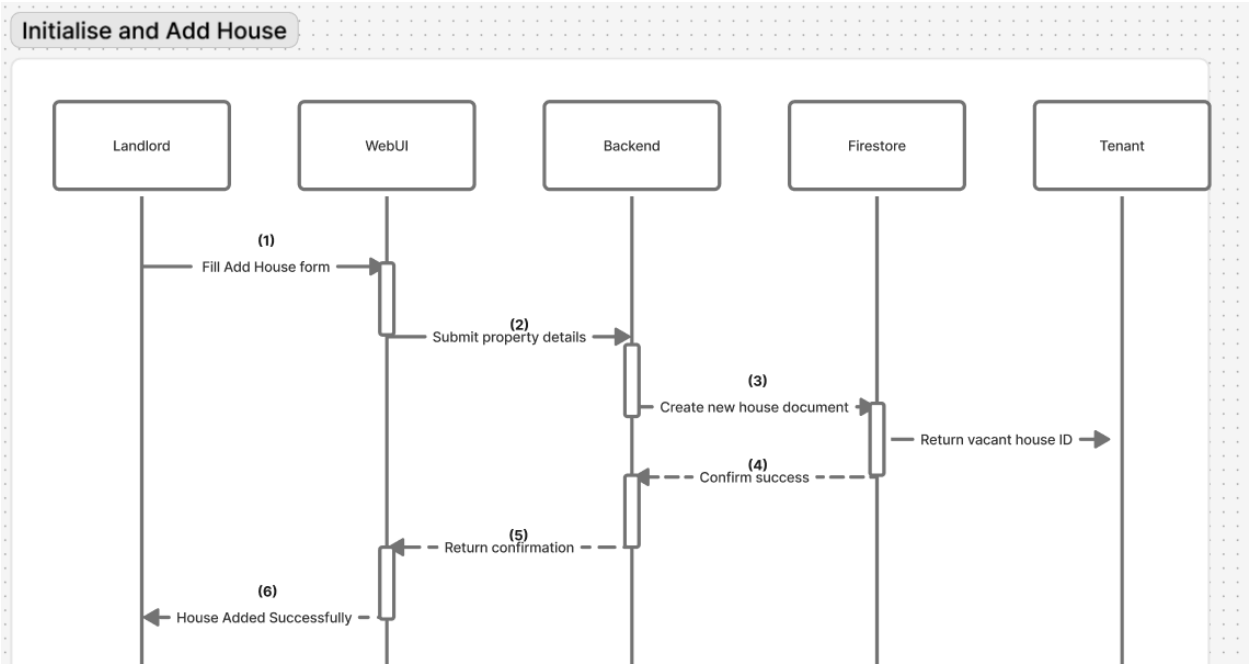


Figure 9: Initialising and Add House functionality

Tenant Interface

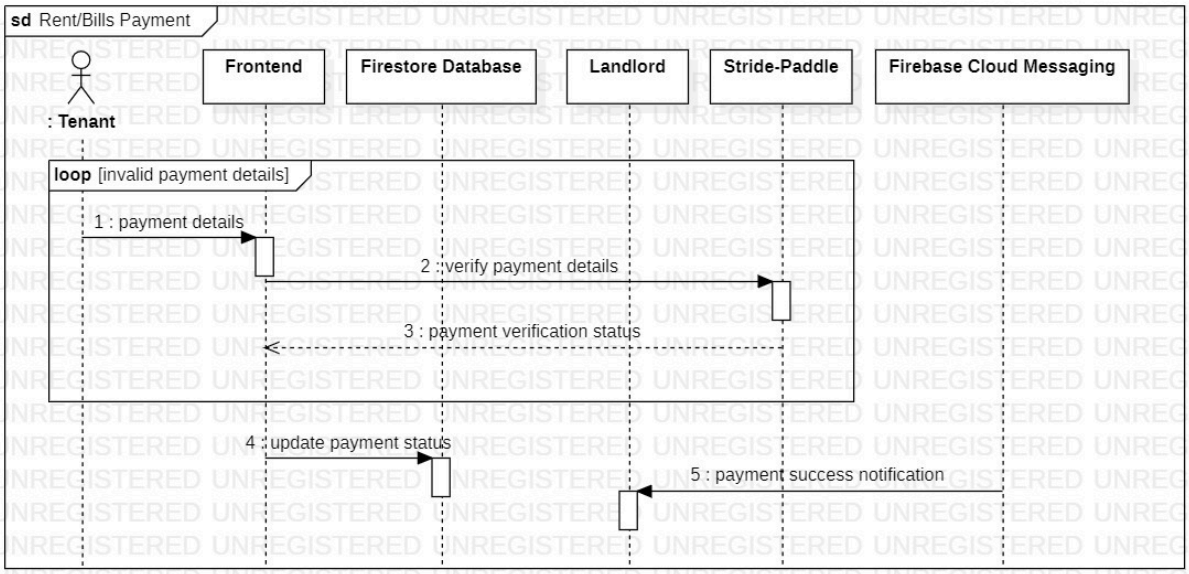


Figure 10: Pay Rent/Bills

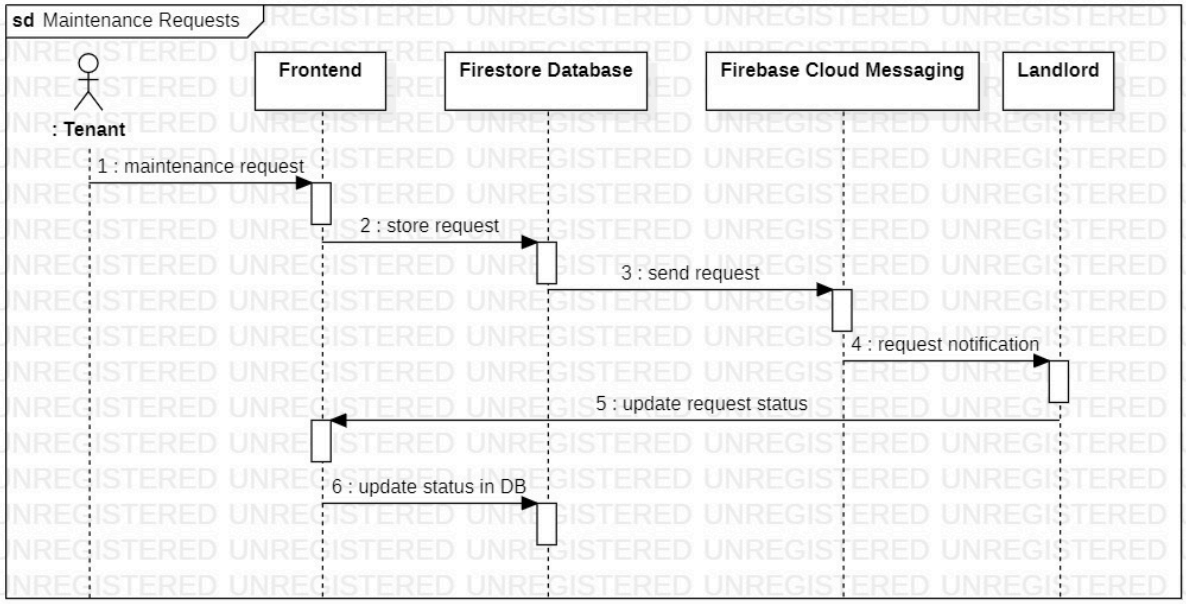


Figure 11: Add Maintenance Requests

(Updated Diagram according to feedback)

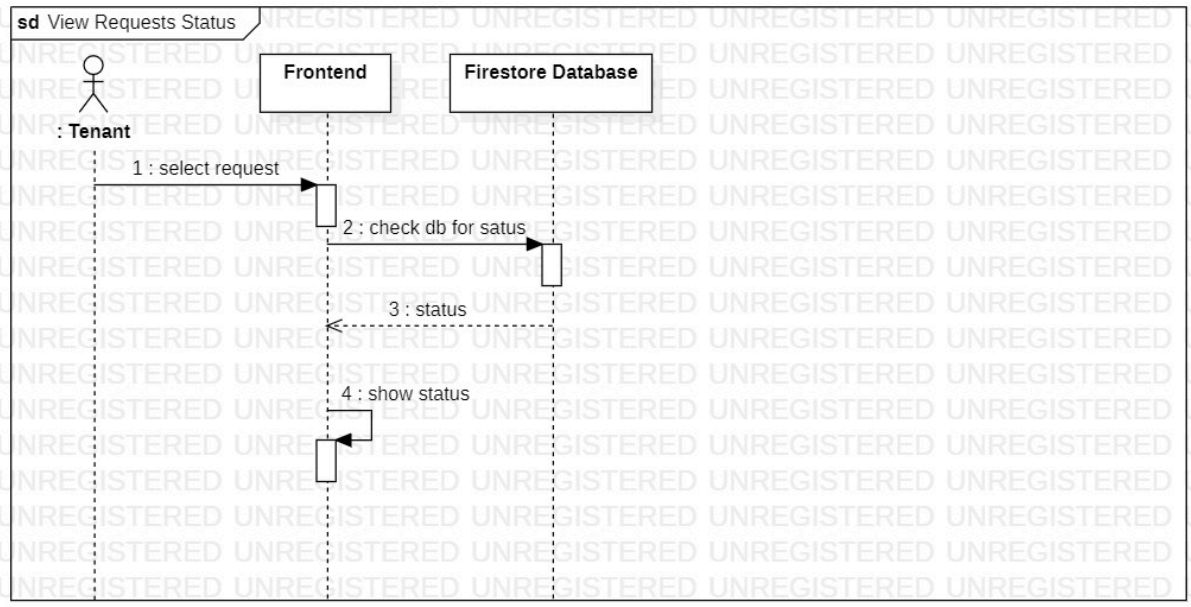


Figure 12: View Maintenance Requests Status

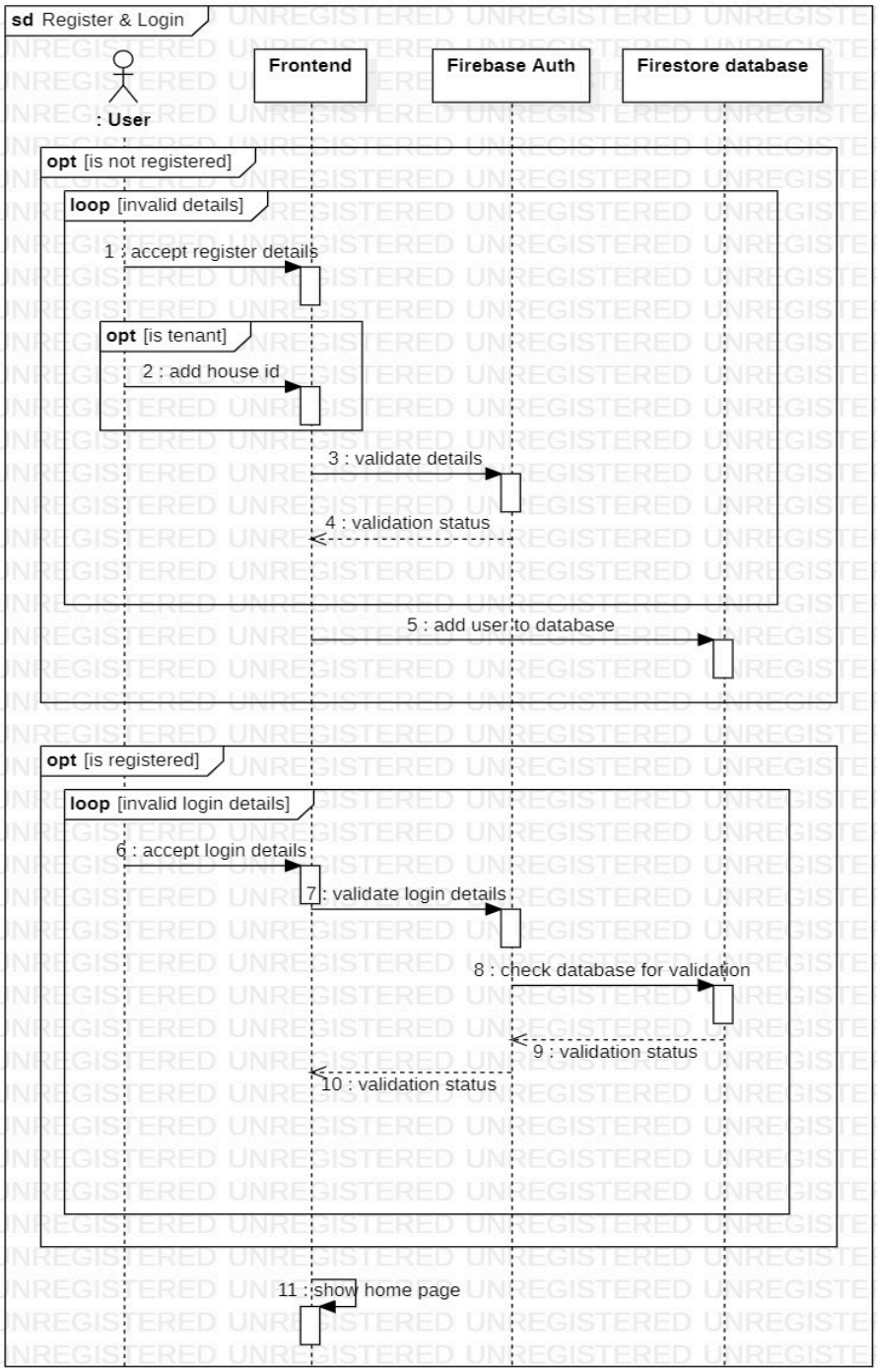


Figure 13: Register & Login

DATA TYPES

- Text (VARCHAR(n)))
- Numbers (INT, FLOAT)
- Boolean
- Date-time/Timestamps
- Metadata

COMPUTATIONS

Authentication:

- Validate credentials or OAuth token.
- Check user type (tenant or landlord).
- Issue a secure session (JWT or Firebase token).

Calculation:

- Rent base + utilities cost calculations

Notification:

- Calculate due dates and trigger reminders
- Decide which users should receive a notification and when based on request forms submission, maintenance status updation and payment verifications

EXPECTED CONTRIBUTION

Frontend designing and development - **Mahlet Bekele**

API development, hosting and integration with frontend - **Zeeshan Imran**

Database setup and hosting - **Miguel Tunga**

System Functionalities and design - **All Members**

Cloud services Integration and hosting - **All Members**

Testing - **All Members**

MILESTONE ACHIEVED

Oct 27 - Nov 2: (Database Schema design and API implementation).
We created a Firestore database schema model that supports both the landlord and tenant side , their individual characteristics and relations with each other, plus properties, requests and bills schemas to fetch all the necessary data wherever required. We imagined how a full muti tenant system should behave, focusing on data isolation and query filtering. The Firestore data model supports multi-tenancy with users, properties, leases, maintenance, and payments collections; tenant documents now include a landlordId field linking them to a landlord and properties include an owner/landlord identifier used for filtering. Migration scripts for initial schema changes and updates are present, and the recent work added and populated the landlord linkage so queries and endpoints can enforce tenant data isolation. The database is ready for QA of the multi-tenant flows, with a small operational task remaining to backfill any preexisting tenant records created before the landlordId fix.

Nov 03 - 09: (Database integration and authentication flow implementation)
The authentication was implemented using Firebase's user authentication system. Whenever a user registers, their data is stored in Firebase, and their password is encrypted before being saved. User roles are stored in the user account as either landlord or tenant. We made sure that tenants register under a specific landlord, so they're required to provide a landlord ID during registration. The backend registration endpoint was updated to validate the provided landlord ID (checking that it exists and that the user has the role landlord) and to store that landlordId on new tenant user documents. The users service was also fixed so that createUser now saves the landlordId correctly. The authentication middleware verifies Firebase ID tokens and the property routes now enforce multi-tenant so that tenants only receive their own landlord's properties.

Nov 10-16: (User interface designing and development)
We implemented the sign-in user interface, which is the same for both landlords and tenants once they register. For the sign-up page, we designed it so the user can choose to register as either a landlord or a tenant. Based on their choice, they're required to fill in the necessary credentials. After that, we implemented separate dashboards for tenants and landlords.

Nov 17 - 23: (API routes and UI implementation)
We implemented the major backend routes for interacting between landlords, tenants and especially the interaction with properties such as creation of properties, lease creation, maintenance request tracking to ensure real time updates between the frontend and the backend and the Firestore. These included functions like toggling house status and updating rents, creating new properties and adding tenants to it and creating message requests and also setting up bills and stuff. Apart from the api routes we also created major UI pages for each of the tasks mentioned above to integrate those routes with the frontend and test the flow of the entire system

Nov 24 - 30: (Testing and Integration)
We were mainly testing the full flow, writing integration tests to make sure everything worked end-to-end. During this week, we made sure the major components, property creation APIs, maintenance request APIs and pages were functional. So mainly, we were testing the work done, and writing integration tests to make sure everything will work end to end.

TASKS DONE BY EACH MEMBER

Zeeshan Imran:

1. **Designed the firestore database schema and implemented migrations:** Designed a complete Firestore data schema (multi-tenant). It covers users, leases, all properties, maintenance as well as the billing. The model shows us the relationship between the landlords and the tenants clearly which includes on the tenants documents the landlordID field and on the properties the ownership fields. In order to add these new fields and update records that already exist migration scripts were executed. To be able to support filtered and complex role based queries Firestone indexing was improved.
2. **Implemented the Authentication and Registration Flow:** Implemented the authentication part of the system. It was implemented using Firebase Authentication which allowed secure user creation (with encrypted passwords) and a role assignment which is either a tenant or a landlord. The tenants are supposed to register themselves under an already existing landlord by providing a valid landlord ID, which the backend will take and verify before creating the tenant record. The backend registration endpoint was also updated to store landlordId correctly, and middleware was added for the validation of Firebase ID tokens and enforcement of multi-tenant scoping. This resulted in a consistent verified and a role-aware registration and login flow.
3. **Tested and Implemented the API Routes:** Created and tested core backend API routes which included functionalities such as property creation, updating the occupancy status, managing the leases and also handling maintenance requests, and generating or updating bills. A connection was made to the firestore through these routes and it was integrated with the frontend to make sure there was real-time synchronization of state across the system. After the implementation was done, each route was tested, first individually and then through end-to-end integration tests with available UIs, to confirm that all of the flows are being operated correctly and that the frontend, backend, and database worked together seamlessly.

Mahlet Bekele:

1. **Frontend UI Implementation:** Designed and implemented the React-based UIs for both tenants and landlords, including maintenance request forms, lease forms, and property management pages. These interfaces support role-specific workflows, such as tenants submitting maintenance requests under a specific landlord and landlords receiving and managing those requests (updating status, assigning a contractor), creating and managing properties, and sending lease invitations to tenants.
2. **Frontend-Backend Integration:** Verified that leases, maintenance requests, user accounts, and property data were correctly created, stored, and retrieved according to the multi-tenant data model in Firestore, and ensured that the backend APIs were functioning correctly. Then improved some functionalities in some edge cases such as when a tenant tries to create multiple maintenance requests, the system would give them a message saying they have already created this request. Ensured that during lease creation, landlords could select from their available properties, assign tenants, specify lease

periods and financial details, and confirm that this information was correctly stored in Firestore, with tenants receiving corresponding lease invitations.

3. **Testing and Validation:** Conducted extensive testing of the frontend and backend integration to ensure that the full workflow operated correctly and consistently across tenant and landlord roles.

Miguel Mibabazi:

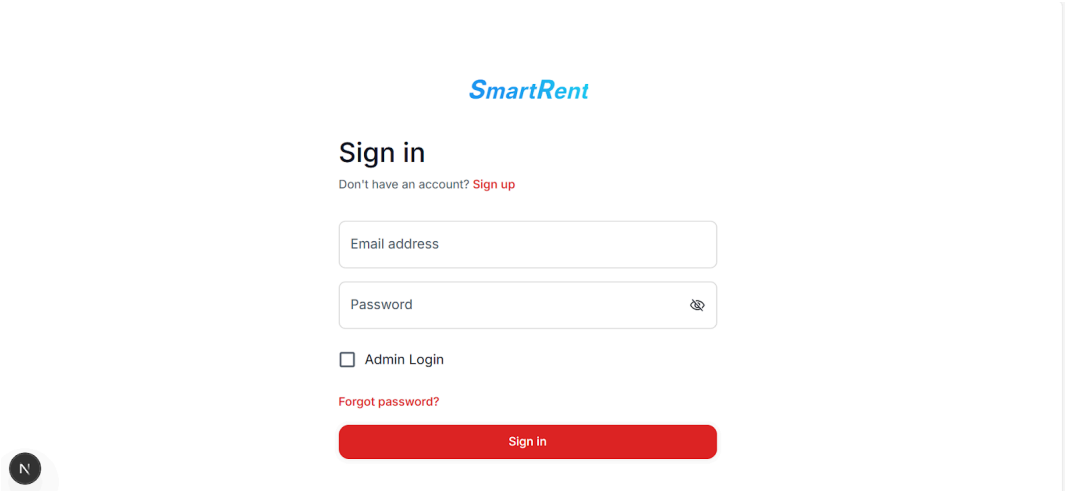
1. **Backend API Development:** Built RESTful backend APIs and connected them to Firestore, enabling operations on users, properties, leases, maintenance requests, and other entities in the system. These APIs handle role-based access control.

2. **Client Services Implementation:** Developed client services to perform CRUD operations and synchronize data with Firestore, ensuring consistent and reliable communication between the frontend and backend.

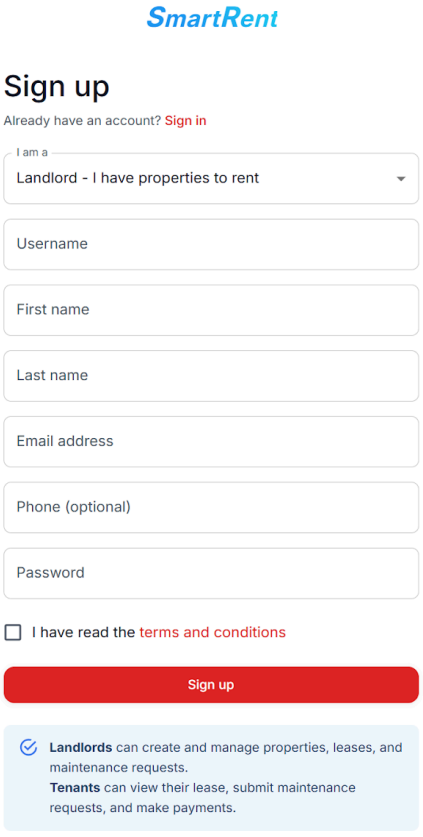
3. **Integration Testing and Documentation:** Added integration tests to verify end-to-end functionality of the backend APIs and client services, covering scenarios such as user registration, property management, and maintenance requests.

USER INTERFACES:

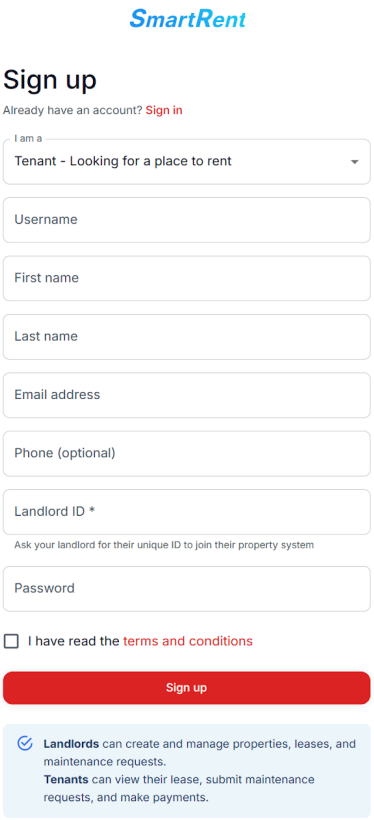
1. Sign in page:



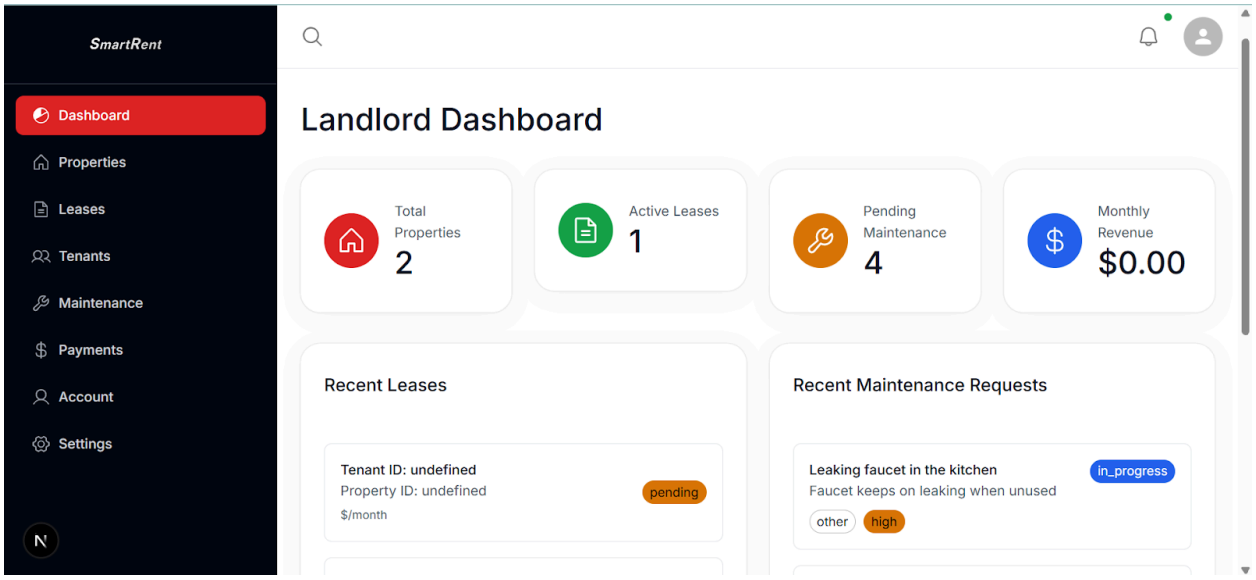
2. Landlord Registration page:



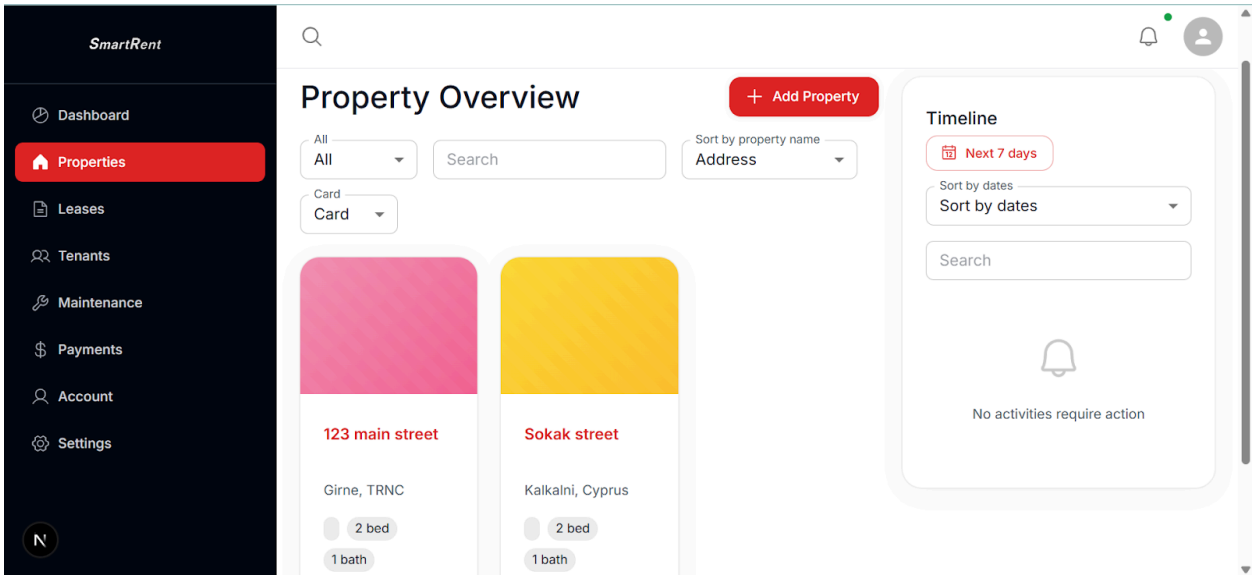
3. Tenant Registration page:



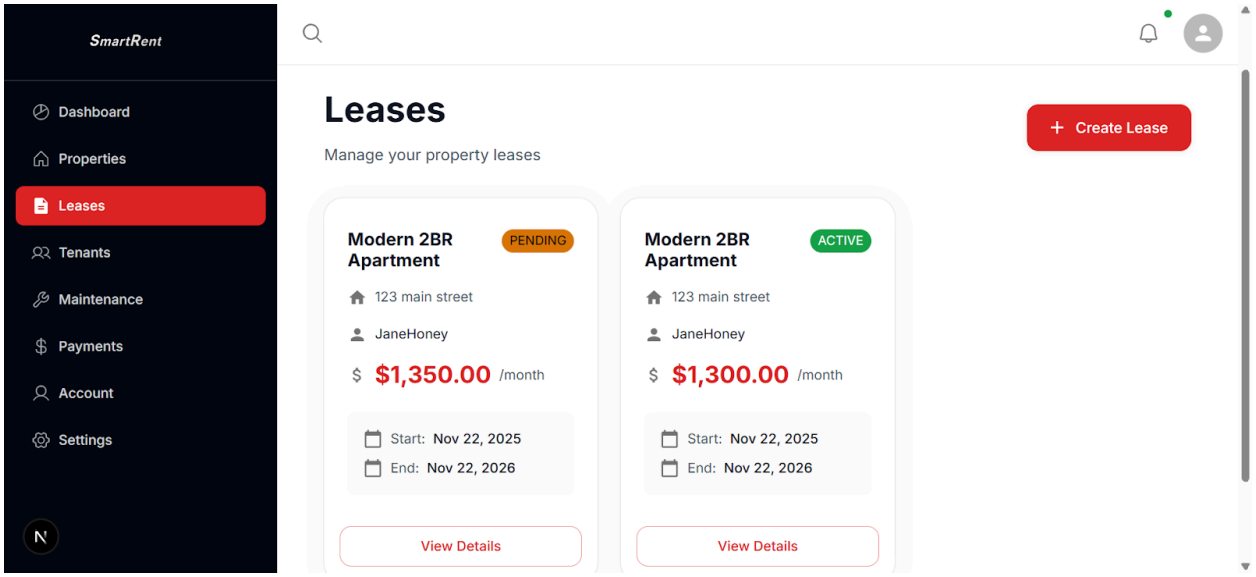
4. Landlord Dashboard



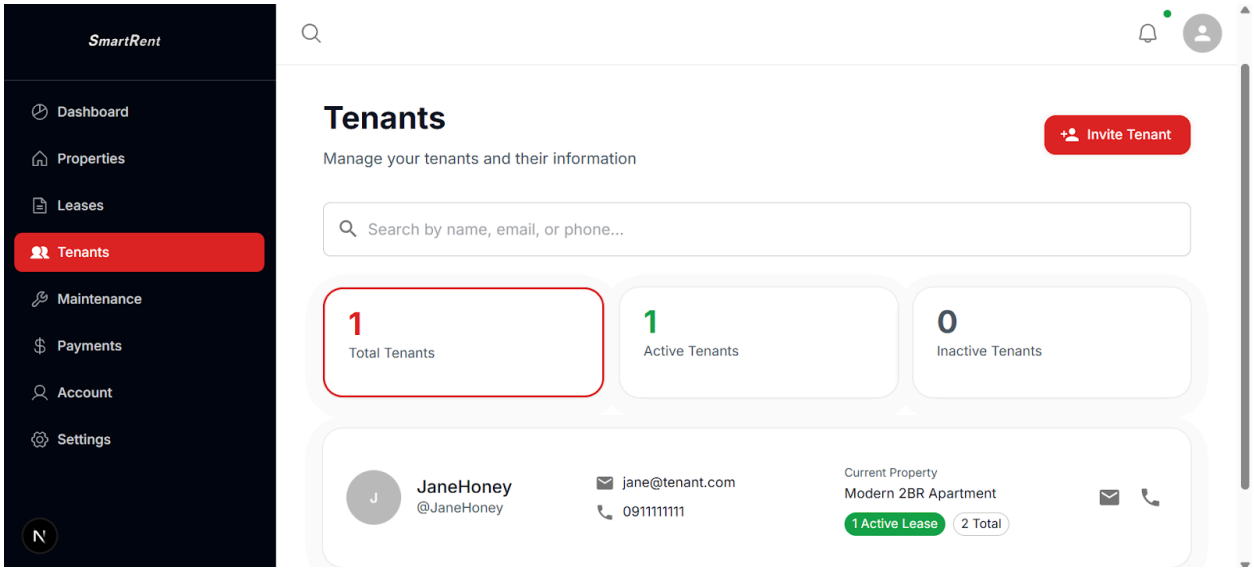
5. Landlord Properties page:



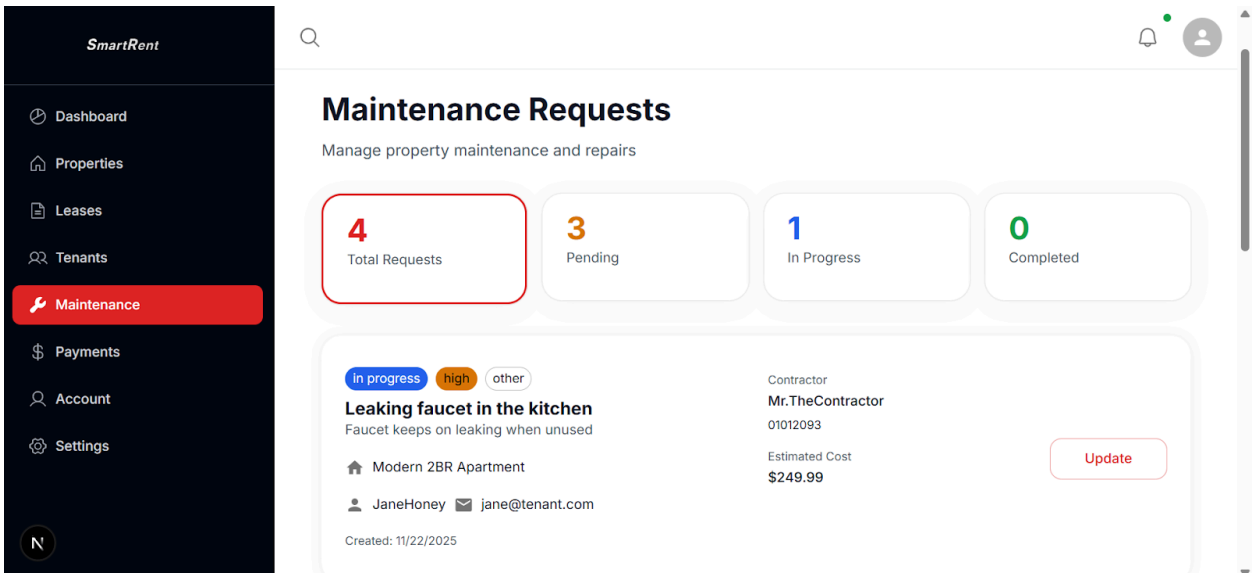
6. Landlord Leases page:



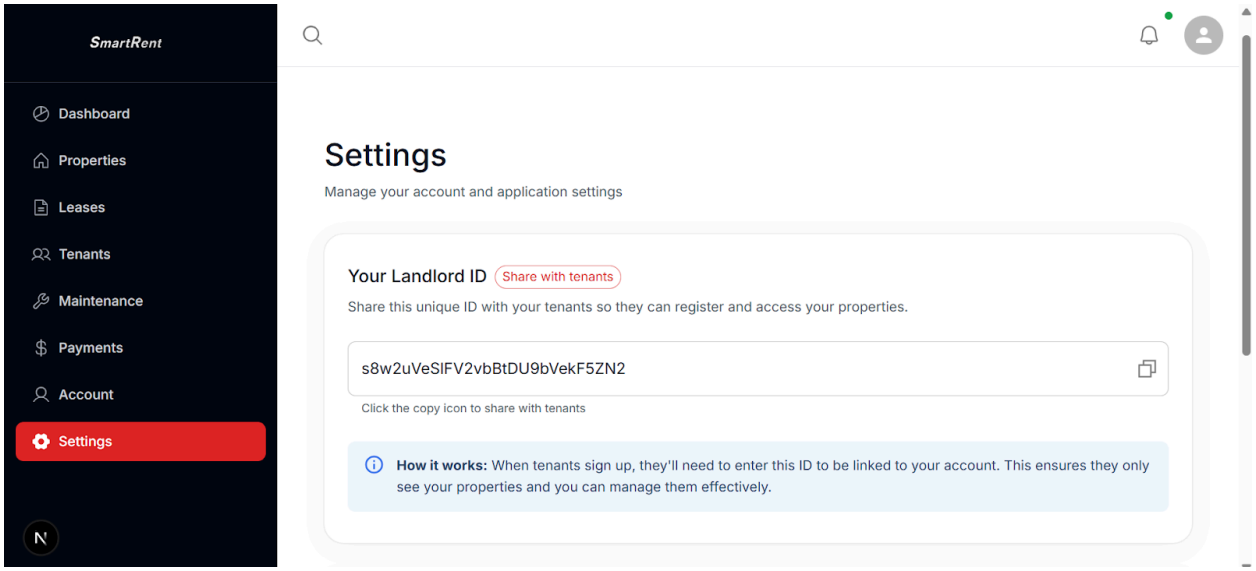
7. Landlord Tenants page:

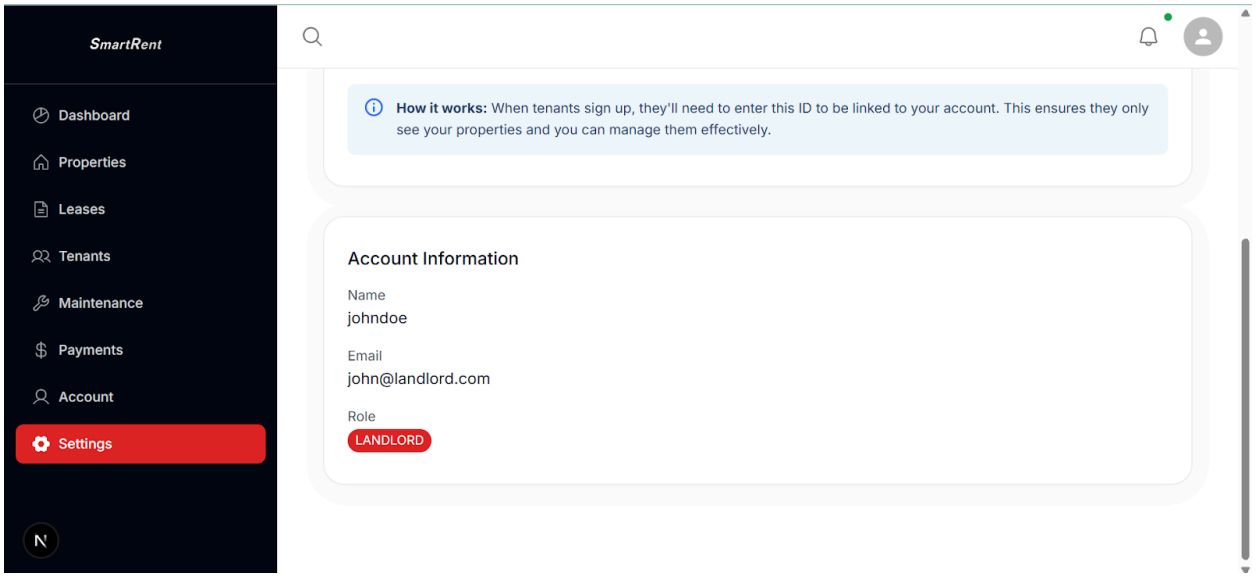


8. Landlord Maintenance Requests page:

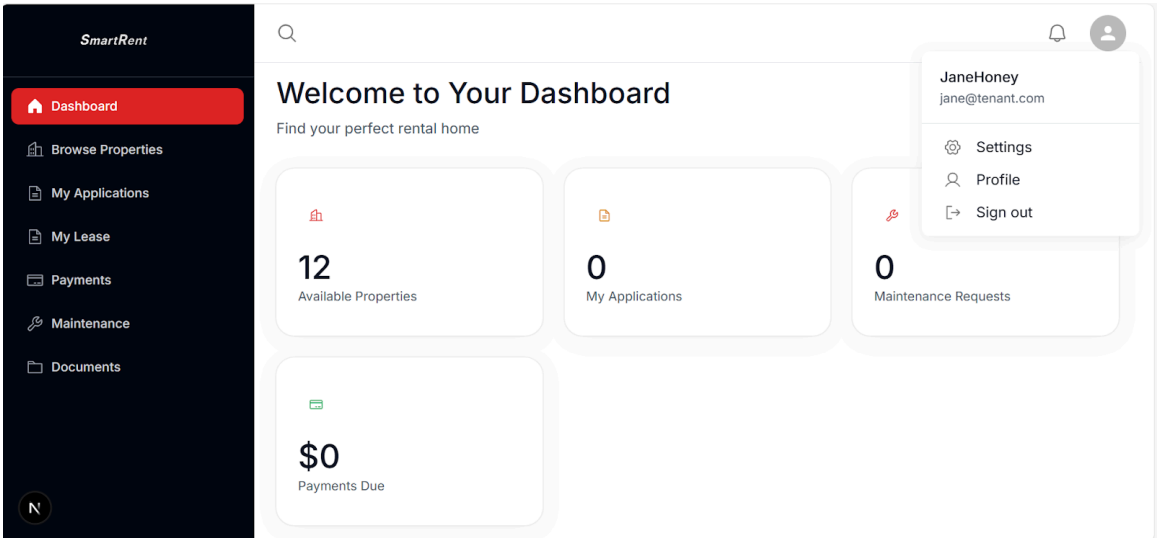


9. Landlord Settings page:

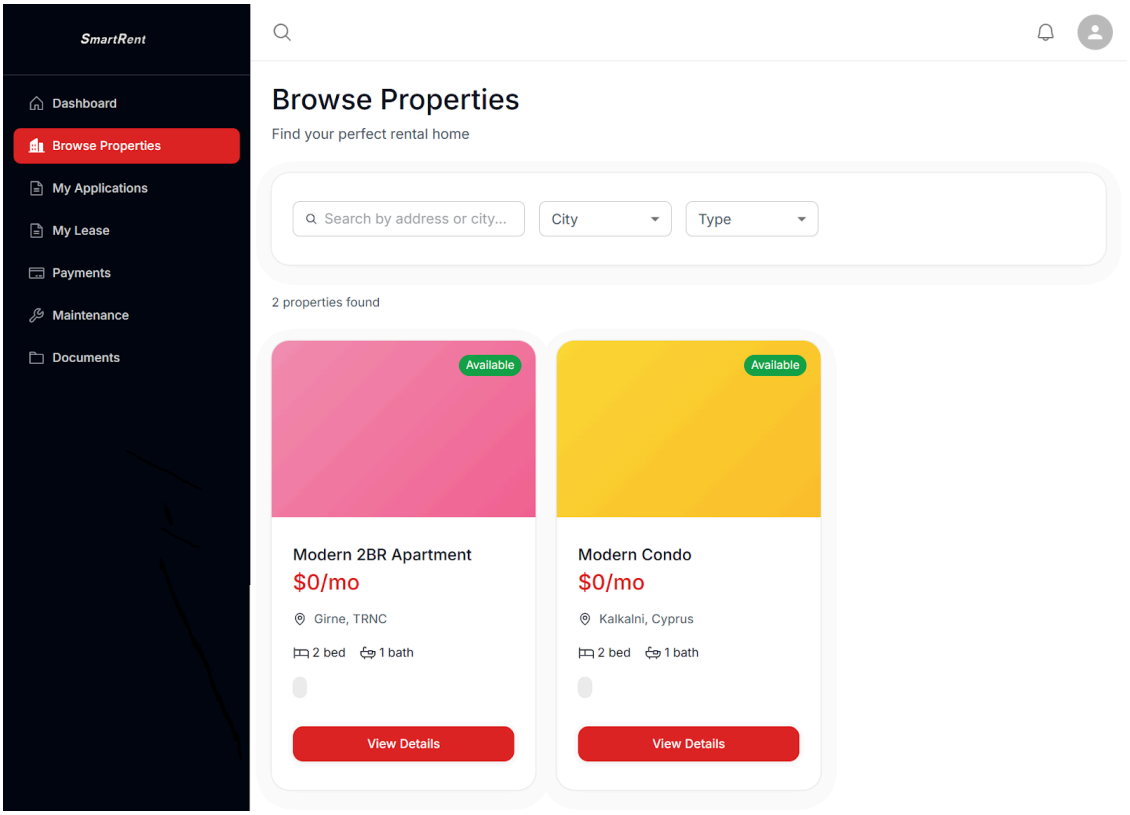




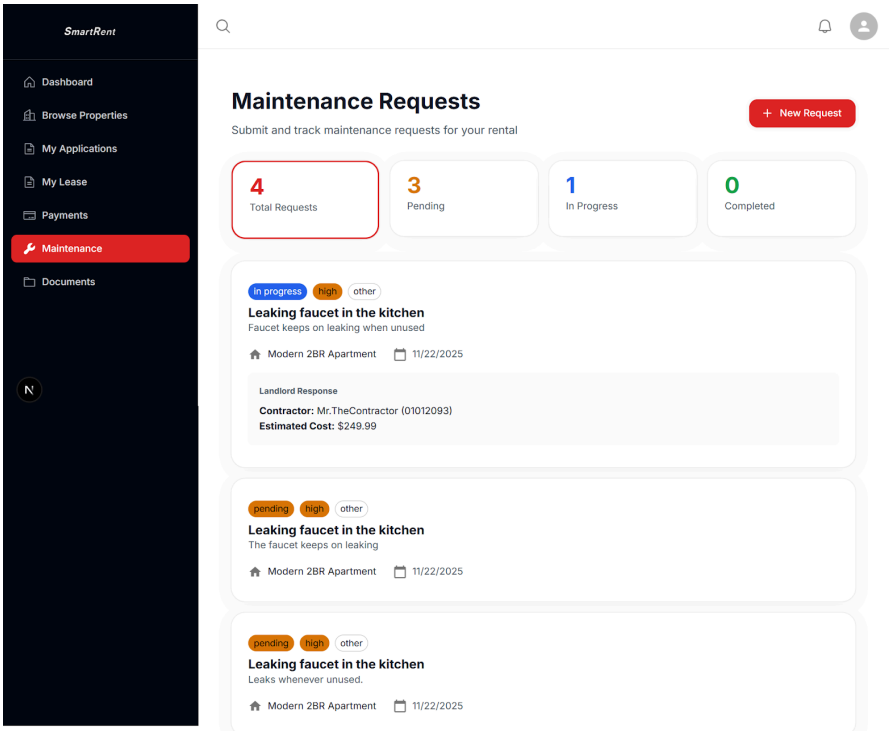
10. Tenants Dashboard page:



11. Tenants Browse Properties page:

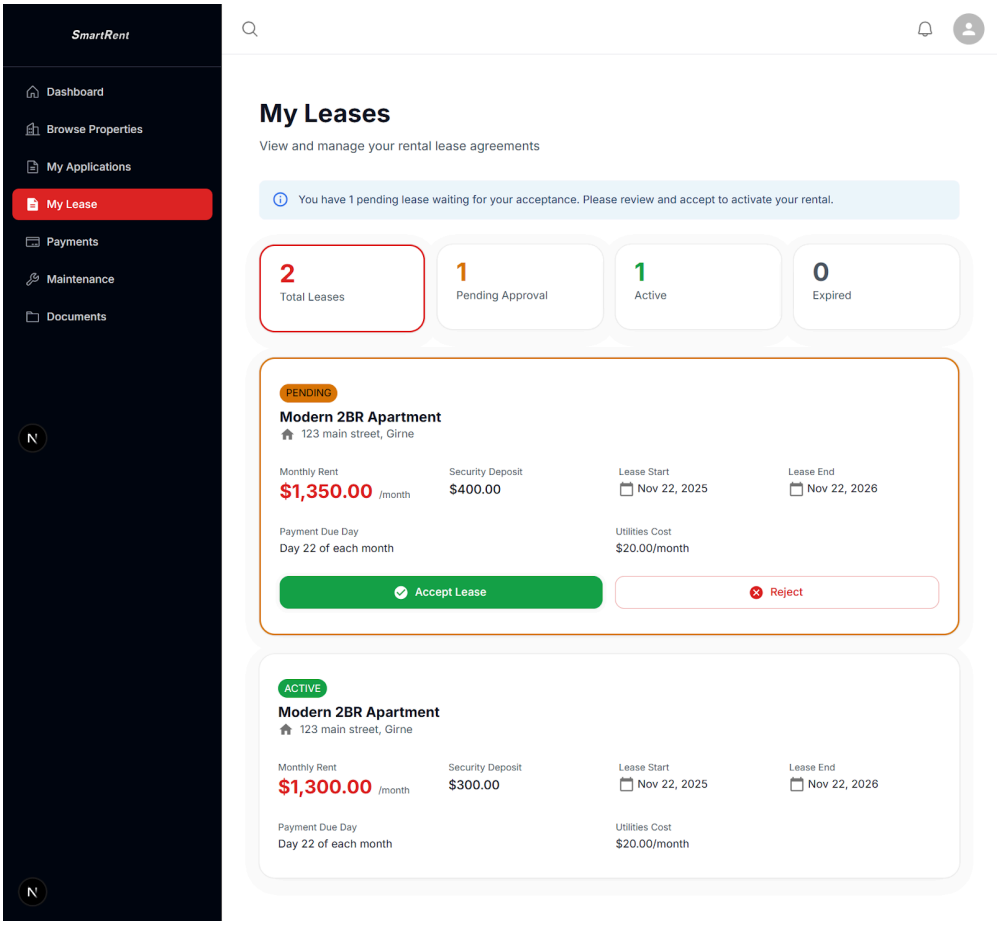


12. Tenants Maintenance Requests page:



Note: While developing the maintenance requests feature, it didn't seem logical for a tenant to keep creating and sending the same request over and over. So we updated the request creation page so that the tenant can't submit a duplicate request. Even though the screenshot above shows the same pending request, we've tested that the tenant won't be able to create it again. If they try, they'll get a message telling them they've already submitted this request before.

13. Tenants Leases page:



TUTORIAL FOR CLOUD BASED TECHNOLOGIES:

1. **Firebase Firestore:** The Firebase Firestore was used to create a NoSql database for our project. Its choice was made considering the ease of development and update of working with a NoSql database. It is hosted on Firebase Milan servers to make sure that our system has to experience less latency, keeping in view the current location of the system being TRNC. We can simply login to Firebase, create an account and open the console where we can start creating our firestore database. The creation of tables and their attributes is simple and we can add, update or delete attributes from a table of a particular instance without having to change the rest of the schemas. The connection URL and the project id, project key is used to connect with the database hosted for real time db connection and data storage, which are provided to us once we create a successful database on the firestore for the project. Since it is simple to build and upgrade, we decided to utilize Firestore, a cloud NoSQL database, for our project. Firestore allows us to create collections and documents rather than traditional database tables. To activate Firestore and begin work, we go to the Firebase console, create a project, and then activate Firestore. Firestore allows us to add, modify, and remove fields to documents without having to change a rigid schema, which is useful for our changing landlord–tenant model. Firebase gives you keys including ‘apiKey’, ‘authDomain’, and ‘projectId’, along with a few others. We paste those keys into our frontend, for instance in a ‘firebaseConfig’ object, and we utilize those keys to initialize Firebase and Firestore in our software, which provides us with a constant access to the database hosted in Milan for quick data storage and retrieval.

2. **Firebase Authentication:** Firebase Authentication is a cloud service that allows for account creation and secure logins with email and passwords. Landlords and tenants within our system have their accounts created with Firebase, where tenants are required to provide a valid landlord ID. During account creation, Firebase sends a unique ID to us that corresponds to the created account. We then save this ID to our backend to manage user roles and associate tenants with their respective landlords. When an account is logged into, Firebase checks the email and passwords, and stores logged in status in the browser's ID token. These tokens are continually updated so that the user does not have to log in again after reloading the page. We have backend middleware that checks these tokens to make sure only logged in users are able to access specific backend features. This process allows for a secure and streamlined experience when logging in for landlords and tenants.
   Our implementation in firebase:
   Created Firebase project → Enabled Firestore database → Set up collections (users, properties, leases, maintenanceRequests, payments) → Enabled authentication(enable email & password authentication) → Connected frontend & backend → Populated demo data(via sign up page) → Enforced role-based access.

GITHUB LINKS:

Github Organization link: https://github.com/SmartRent-495

Backend Repo link: https://github.com/SmartRent-495/Backend

Frontend Repo link: https://github.com/SmartRent-495/Frontend

MILESTONES REMAINING

Milestone Remaining	Tasks per milestone	Responsible Member	Timeline
1. Maintenance Request Workflow (E2E)	1.1 Tenant UI for submitting requests with photo uploads.	Mahlet Bekele	Dec 1 - 7
	1.2 Landlord UI for viewing and updating the request status.	Mahlet Bekele	
	1.3 Testing of the routes with the UI integration.	Zeeshan Imran	
	1.4 Automated notifications for all new and updated requests.	Miguel & Zeeshan	
	1.5 Implement Firebase messaging system.	Zeeshan & Miguel	
2. Fix data fetching issues	2.1 Implement a way to fetch the data in appropriate time to display on the UI without any unnecessary latency.	Mahlet Bekele	Dec 1 - 7
3. End-to-End Payments Integration	3.1 Wiring the frontend payment interface to Stripe backend endpoints.	Mahlet & Zeeshan	Dec 8 - 14
	3.2 Recording all payment transactions in Firestore	Miguel Mbabazi	
	3.3 Handling necessary webhooks.	Miguel Mbabazi	
	3.4 Displaying a comprehensive payment history to users	Zeeshan Imran	
4. Property	4.1 Finalize the Create, Read, Update, and	Miguel &	Dec 8 - 14

Management (CRUD) with Media	Delete (CRUD) functionality for properties, including image upload capability (migrate image storage to Firebase Storage or S3 for production environments	Zeeshan	
5. Data Cleanup and Backfill	5.1 **:** Execute a script or administrative task to resolve data inconsistencies by populating the missing landlordId values for existing tenant records.	Miguel Mbabazi	Dec 8 - 14
6. Admin Portal/Dashboard	6.1 Create Admin UI	Mahlet Bekele	Dec 8 - 18
	6.2 Set API routes/end points for admin access and functions	Zeeshan Imran	
	6.3 Implement Services and Test APIs	Miguel Mbabazi	
7. Test the Overall System:	7.1 Test the systems performance from start to end from both the Landlord and Tenants side to ensure all the deliverables are achieved and accurate functionality is implemented with proper session management and data updation and notification function all using solely cloud based technologies.	All Members	Dec 19 - 21

DELIVERABLES FOR COMPLETED PROJECT:

Frontend Code (Client):
Source code of React + Next.js for UIs of landlords and tenants- dashboards, properties, leases, maintenance requests, and payments. Includes all completed pages for both the tenant and landlord.
A separate Admin UI for controlling all the administrative tasks and extra privileges to access data and resources.

Backend Code (Server):
Next.js API routes and Firebase Cloud Functions handle authentication, role-based access, property/lease/maintenance/payments logic, and integrations like Stripe/Paddle and notifications.

Detailed Readme File: All information of our project, technologies used, cloud services used, directory structure, installation steps, how to run your project, and team member contact details.

Database Deliverables:

1. Firestore Schema Documentation Overview Collections: users, properties, leases, maintenanceRequests, payments, etc.
2. Demo Firestore export/backup with sample landlord and tenant data

Deployment & Configuration Files:
Vercel/Render or Firebase deployment configurations, Firebase project setup without secrets, and environment variable templates (.env.example)

Deployed Demo System:
Public demo URL of running SmartRent application (frontend and backend)

API Documentation:
Endpoint list or OpenAPI/Postman collection describing request/response formats for the main APIs: authentication, properties, leases, maintenance, and payments

User Documentation(Optional):
Short guides integrated in the UI for users that shows how to manage their properties, leases, and requests.

REFERENCES

1. **Firebase Firestore Documentation** – Official guide to using Cloud Firestore, Google's scalable NoSQL database. https://firebase.google.com/docs/firestore
2. **Firebase Cloud Messaging (Notifications)** – Documentation for integrating push notifications on web and mobile using FCM. https://firebase.google.com/docs/cloud-messaging
3. **Firebase Authentication** *(if you're using login/signup features)* – Securely manage users and roles with Firebase Auth. https://firebase.google.com/docs/auth
4. **Vercel Documentation** – Official documentation for deploying Next.js and other frontend apps with edge functions and CI/CD. https://vercel.com/docs
5. **Render Cloud Hosting Documentation** – Platform guide for deploying full-stack web applications, APIs, and background workers. https://render.com/docs
6. **Stripe API Reference** – Official Stripe documentation for payment integration, billing, and subscription management. stripe.com/docs/api