Knockk

Grace Radlund

CST-451 Capstone Project Final Architecture & Design

Grand Canyon University

Instructor: Professor Mark Reha

Revision: 1.1

Date: 4/8/2025

**ABSTRACT**

Although social media is prominent, there is no social network to connect residents of an apartment building. This project aims to fill that void with Knockk. Knockk is a social network composed of two applications: a mobile application and a web application. The mobile application is designed for residents, with the primary goal of connecting the resident with neighboring units. The web application is for building administrators to verify and manage residents of a building. Although this project can be defined as a social network, it is not like most other social applications – social media postings are omitted. This project solely focuses on connecting residents together and is not focused on being a comparison game of likes, comments, and followers.

To make this project successful, the project is laid out in terms of planning, analyzing, designing, and developing, with a completion date of May 3, 2025. This document is the proposal for this project and is in the planning phase. A vague description in terms of what the project consists of has been developed, with more technical terminology in the high-level solution. Very technical terminology has been omitted. Basic functionality of the applications will be completed before any other features are pulled in to ensure the project can be completed with the success criteria met. Since this project is a large undertaking, there is an in-depth risk management plan to minimize issues occurring in this project. Proposed technologies will be researched, and proof of concepts will be developed before a final decision on the technology stack is made.

|  |
| --- |
| History and Signoff Sheet |

**Change Record**

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Revision Notes** |
| 11/17/2024 | Grace Radlund | Initial draft for review/discussion |
| 4/8/2025 | Grace Radlund | Preparing document for showcase |
|  |  |  |

|  |
| --- |
| **Overall Instructor Feedback/Comments** |

|  |
| --- |
| **Overall Instructor Feedback/Comments** |

**Integrated Instructor Feedback into Project Documentation**

Yes  No

**TABLE OF CONTENTS**

Design Introduction 5

Detailed High-Level Solution Design 6

Logical Solution Design 7

Physical Solution Design 8

Detailed Technical Design 9

General Technical Approach 9

Key Technical Design Decisions 9

Database ER Diagram 14

Database DDL Scripts 15

Sitemap Diagram 15

User Interface Diagrams 16

Component Design 16

Service API Design 18

NFR’s 19

Operational Support Design 19

Other Documentation 21

Appendix A – Technical Issue and Risk Log 23

Appendix B – References 25

Appendix C – External Resources 26

Design Introduction

This document provides in-depth design specifications of the Knockk project. The full stack project is composed of two frontend applications, one backend, and a database. Detailed High-Level Solution Design of both the logical and physical solutions will be provided. Following the solutions, is a Detailed Technical Design, which includes sections like: Key Technical Design Decisions for rational behind the technologies/frameworks that were chosen and the proof of concepts that validate these decisions; a Database ER Diagram that describes the database schema; Database DDL Scripts for creating the schema and showing database constrains; a Sitemap Diagram for showing how to navigate screens in the mobile and web applications; User Interface Diagrams which include both low and high fidelity wireframes of the mobile and web applications; Component Designs of the mobile and web applications for information on screens and components in the React Native applications; a Service API Design which includes documentation and UMLs of the REST API backend; NFRs for how the non-functional requirements will be implemented; an Operational Support Design for how the frontend, backend, and database will support logging and monitoring; Other Documentation for other information not listed in the above sections. Following the detailed design, are the Appendices – Technical Issues and Risks, References, and External Resources.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Deliverable Acceptance Log | | | | | |
| ID | Deliverable Description | Comments | Evaluator | Status | Date of Decision |
| 1 | Data Dictionary | Information pertaining to the structure of the database. Non-nullable columns will not have a default value – if a default value is not given, there will be an error; the row will not be added/updated. | Grace | Complete | 11/17/2024 |
| 2 | API Design | Fully documented API. Contains all HTTP requests with descriptions, requests, and responses. | Grace | Complete | 11/17/2024 |
| 3 | Project Proposal | Initial project proposal for Knockk that identifies the purpose of the project and outlines the work needed to be done. | Grace | Complete | 9/22/2024 |
| 4 | Project Requirements | Identify requirements for the project including user stories, technical requirements, logical models, and reports. | Grace | Complete | 10/13/2024 |

Below includes the deliverable acceptance log. These are documents external to the specification.

Detailed High-Level Solution Design

The high-level solution design includes proof of concepts, hardware and software technologies, and the physical and logical solution designs for Knockk. Proof of concepts were created to validate the technical decisions made and teach the developers more about the given technology. The results from these concepts led to the technical decisions made for this project – more information can be found in the Key Technical Design Decisions section of the Detailed Technical Design.

The logical solution design provides a high-level logical design of the project, with n-layer architecture patterns, and overview of the folder structure and how they interact with one another. The physical system design provides a high-level physical design of the project with hardware requirements, ports, and protocols. The physical design also specifies how the frontend apps will be built using Expo.

|  |  |  |
| --- | --- | --- |
| Proof of Concepts | |  |
| **Description** | **Rationale** | **Results** |
| React | Library for building web applications. | React Native was chosen because it decreases the amount of learning required to complete this project; instead of learning both React and React Native, React Native only has to be learned. |
| React Native | Library for cross-platform development. Used to develop both mobile and web applications. | Chosen library to build mobile and web applications. Uses a bridge that renders native components. |
| Flutter | Framework for cross-platform development. Used to develop both mobile and web applications. | React Native was chosen because of the use of the JavaScript programming language and the larger community. Companies, such as what the develop is hired at, uses React Native more than Flutter. |
| Expo | Framework built on top of React Native that simplifies development. | Expo was chosen because of the increased simplicity for developing and scaling React Native applications. The React Native documentation recommends using this framework. |
| Express | Framework for building REST APIs. | Spring Boot was chosen because of the structure it provides compared to JavaScript. |
| Supabase | Cloud provider for a Postgres database. | Chosen cloud provider for Postgres database. Not utilizing the API Supabase provides within the Spring Boot API because it would lock in the Spring Boot API; would not be interoperable. If the database is shifted to AWS, the whole Spring Boot API would have to be rewritten. |

|  |
| --- |
| Hardware and Software Technologies |
| 1 - React Native v0.74 |
| 2 - Expo v51.0.0 |
| 3 - Typescript v5.6.3 |
| 4 - Spring Boot v3.1.4 |
| 5 – PostgreSQL v153.6 |
| 6 – Supabase v1.204.4 |
| 7 – Draw.io v24.7.17 |
| 8 – Figma v16.13.3 |
| 9 – GitHub v3.4.6 |
| 10 – Jira v9.17.0 |
| 11 – MacBook Pro M1 chip Venture 13.5.2 |
| 12 – Visual Studio Code v1.91.4 |
| 13 - Word v16.80 |
| 14 – Sentry v6.1.0 |

Logical Solution Design

Refer to the “Logical Solution Design” attachment for the logical solution design.

This design provides a high-level overview of the project utilizing Expo, Spring Boot, and Supabase.

The two client applications, Knockk-Admin and Knockk-Resident will be built with Expo. React has a twist to N-layer architecture and combines the presentation and business layer together. This presentation and business layer contains the app and components folders. Utilizing the expo-router library, file-based routing will be used to manage screens. Subfolders inside of the app folder will have files that define the screens of the application. To abstract components from the screens, there is a separate component folder to where these components will reside. The presentation layer will contain the assets folder which will include stylesheets (utilizing NativeWinds), fonts, images, etc. The business layer will make use of state, and util. State will utilize Zustand, which is a lightweight state management library, and will eliminate excess prop drilling. The util folder will have services that make requests to the database - axios will be used for making network calls. Securing the API is an out-of-scope feature (high on the list) which means that the connection – for now – will not be secure. BCrypt will be used to hash passwords, making it harder for a hacker to steal credentials because they aren’t in plain text. The response between the Expo application and Spring Boot API will be transmitted with JSON, using HTTP.

The Server, built with Spring Boot, adheres to the standard N-layer architecture. The presentation class includes the REST controllers, which utilize business services (apart of the business layer). The business services will utilize data services, which will utilize repositories to communicate to the Postgres database. Models will be passed between controllers and business services, and entities will be passed between services and between data services and repositories.

Physical Solution Design

Refer to the “Physical Solution Design” attachment for the logical solution design.

Expo Deployment Builds will be used for building the front-end applications. The Knockk-Admin application will be opened in the web on port 8081. For development, Google Chrome will be used, however Expo also supports other browsers like Edge and Safari. The Knockk-Resident application can be run on Android or iOS using an Android Emulator (using Android Studio) or iOS Simulator (using XCode). Expo SDK 51 supports iOS 13.4+ and Android 6+, so the devices and their operating systems versions could be changed by the developer if necessary. Both the admin and resident applications will make requests to the API using HTTP, port 3000. The REST API will connect to Supabase using port 6543, with HTTPS. Port 6543 is Supabase’s session port which supports prepared statements, unlike Supabase’s transaction port.

Detailed Technical Design

General Technical Approach

The Knockk development team will implement agile and the scrum methodology. Although software is not expected to be delivered every two weeks, the team will follow industry standards of two-week sprints. It is important to remember that this project is a marathon, not a sprint – figuratively speaking. Sprints will be used to break apart the project into manageable tasks; this project is not meant to be completed in a week. It’s a marathon, lasting until 5/3/2025 (and potentially beyond this is turned into a production level project). Productivity tools such as Jira will be used to organize and facilitate tasks.

The software the Knockk team will be delivering includes two frontend applications, a backend, and a database. The front-end applications will utilize Expo, a React Native framework. The frontend will communicate to the REST API backend, which utilizes Spring Boot. The API then communicates with the Postgres database, hosted on Supabase. Mentorship from Professors who have worked in the industry will be key to making sure that the software is properly architected. The team has and will continue to work hard to ensure that best practices are being implemented.

Technical details can be found in the sections below: Key Design Decisions, Database ER Diagram, Database DDL Scripts, User Interface Diagrams, Component Design, Service API Design, NFR’s, Optional Support Design, and Other Documentation.

Key Technical Design Decisions

Before the final tech stack was decided, proof of concepts were developed. The final decisions were:

**Frontend**

React Native was chosen for developing the cross-platform front-end applications. Initially, React was going to be used for developing the admin application, but it was decided that learning two different frameworks for two frontend applications would be a lot of work. Before React Native was decided upon, Flutter was also pursued. Flutter’s common struggle with “nesting h\*ll” and the use of Dart, which is Flutter specific, ultimately led to choosing React Native. Expo is the framework of choice for building the React Native application - this framework is the recommended choice of React Native developers and speeds up the development process of React Native applications.

Builds:

Expo development builds were chosen over Expo Go. Expo Go (an app from the App Store) is a sandbox environment for running Expo projects - once the app is built in the terminal, you scan the QR code, and the app will run on your device. The problem with this is that Expo Go uses pre-installed libraries, so if you use an external library that is packaged, the build will fail. Development builds create the native app bundle, giving full control at runtime. Note that the development mode of Expo development builds supports hot reloading, while production mode does not. For the current scope of this project, only development mode will be used. These builds can be stored in the cloud using Expo Application Services, but current build practices do not do so.

**Backend**

Spring Boot was chosen for developing the REST API. REST was chosen over GraphQL, simply because the team is more familiar with it and there are already other technologies that need to be learned. Building the API with Express was pursued as an option, but the lack of object-oriented principles because of the use of JavaScript led to Spring Boot being selected. Spring Data JDBC will be used for querying the database.

**Database**

Supabase was chosen as the cloud provider for the Postgres database. Not only was it chosen because it has a cool name, but because it hosts the database in the cloud, making it accessible. Postgres was decided because of the relational database advantages – creating relationships between tables.

Everything has pros and cons, and software is no exception - but as the saying goes, you have to pick your poison. The following section describes the “poisons”/technologies that have been chosen and the purpose and reasoning behind each below:

React Native v0.74

Purpose: Framework to build mobile and web applications.

Chosen: Popular framework with large community support. Selected over Flutter because JavaScript is a popular programming compared to Dart; selected over React because React is not cross-platform.

Expo v51.0.0

Purpose: Accelerates development.

Chosen: Recommended framework in the React Native documentation. Companies like Pizza Hut, codeacademy, and Insider all chose to integrate Expo within their apps.

TypeScript v5.6.3

Purpose: JavaScript superset used in React Native, strongly typed.

Chosen: Offers additional safety to JavaScript which allows for more readable and maintainable code.

Node.js v20.18.0

Purpose: Runtime environment that will execute the Expo applications.

Chosen: Needed for Expo Development Builds.

Spring Boot v3.1.4

Purpose: To develop the backend REST API.

Chosen: Commonly used in the industry. Utilizes object-oriented principles, which is unlike other frameworks like Express (which uses a non-OOP language, JavaScript).

Supabase v1.204.4

Purpose: Cloud service provider that hosts PostgreSQL databases. Alternative to Firebase.

Chosen: Relational database provider in the cloud. Provides monitoring and reporting of the Postgres instance.

PostgreSQL v15.6

Purpose: Database to store data.

Chosen: Database instance that Supabase provides.

Draw.io v27.7.17

Purpose: Tool for technical diagraming.

Chosen: Lots of shapes for different diagramming uses.

Figma v16.13.3

Purpose: Tool for wireframing.

Chosen: Commonly used in the industry. Sleek user interface that makes wireframing enjoyable and easy.

GitHub v3.4.6

Purpose: Version control in the cloud.

Chosen: Commonly used in the industry. Allows developers to store code in repositories and track changes in the cloud.

Jira v9.17.0

Purpose: Tool for project management.

Chosen: Commonly used in the industry. Will help the team deliver code faster by managing tasks and tracking progress.

MacBook Pro M1 Chip Ventura 13.5.2

Purpose: Development environment.

Chosen: Developers’ current computer.

Visual Studio Code v1.9134

Purpose: Tool for code development.

Chosen: Commonly used in the industry. Lightweight IDE with a simple user interface and rich extensions.

Word v16.80

Purpose: Tool for documentation.

Chosen: Team is familiar with Word. Easy to create documents.

Sentry v6.1.0

Purpose: Capture reporting and performance issues for Expo. Crash reporting, performance monitoring, error tracking, gathering user feedback related to errors.

Chosen: Can easily integrate into Expo project and provides great interface for reporting, monitoring, and error tracking.

Database ER Diagram

A screenshot of a computer

Description automatically generated

Knockk utilizes a Postgres instance in Supabase for storing information. There will be seven tables in the Knockk schema: User, Resident, Friendship, Lease, Unit, Building, Admin. The tables are associated to other tables together through the use of foreign keys. The User table stores login credentials of a Resident such as email and password. This table abstracts the login information from the Resident. The Resident table includes all information about a resident including first name, last name, gender, age, hometown, biography, profile photo, background photo, Instagram, Snapchat, X, Facebook, and if they are verified. Some of these fields are optional, because information like a biography is not required by the user of the resident application to input. Residents can be friends with other residents, which is shown in the Friendship table. This table uses two foreign keys to tie the Resident’s id, along with accepted to see if the invitee has accepted the request. The Lease table stores information about a lease including the start date, end state. The Lease table is what ties a Resident to a Unit, and a Unit to a Building, so that the admin can verify the resident in the admin application. The Unit table stores information about the unit including floor, room, and number of rooms. The Building table stores information about a building including the name, number of rooms, top floor, bottom floor, an array of rooms with no neighbors to the right, and an array of rooms with no neighbors to the left. The Admin table includes information about an admin like their username and password.

NOTE: A resident’s credentials were abstracted using the User table with a foreign key to the Resident table, but admin was not. Since the Admin table does not have any other information about an admin other than the login credentials, it was felt as though a separate table was not necessary.

NOTE: The resident needs to be verified before their account is active. For admin to be able to verify residents, the resident needs to be tied to a lease. The lease needs to be created before a resident can register.

NOTE: Future implementation should store an array of admin ids in the Building table so that multiple admins can manage a building.

NOTE: Throughout the project, the terms “unit” and “room” will be used – these are not interchangeable. A unit refers to the number with the floor number and the room number. Room refers to the room number only. Example: a room on floor 4, room 51, is unit 451, and room 51.

Database DDL Scripts

Refer to the “KnockkDDL” zip for the DDL script of this schema. To create the database in Supabase, create a new Project, import the script into the SQL Editor, and click run.

NOTE: Supbase’s direct connection to the Postgres instance does not support prepared statements, so the session mode for the JDBC connection will be used.

Sitemap Diagram

Refer to the “Sitemap” attachment for the sitemap diagram. The diagram displays the screens/pages for both the resident and admin applications. A key is provided for more information on meanings of different symbols, colors, etc. The applications will utilize Expo router for navigation between screens.

The resident application will start at the login screen. Future implementation will include a splash screen for the app. The user can either login, or, if they don’t have an account, register. Registration spans through many screens – once registration is complete, the user will be navigated back to login. At any point through the registration, the user can go back to the previous screen. Access to screens other than login and those associated with registration will not be permitted until the user is logged in. Once logged in, the user will be directed to the home screen. From there, the user can select a room (if it exists) – above, to the right, below, or to the left – or view their profile. Upon selecting a room, the neighbor can view a specific neighbor who lives in the room. If the user did not choose a room, and instead chose their profile, they will be able to see their profile, and from their either edit their profile or log off. Once the user logs off, they will have restricted access to only the login or registration screens until the login again.

The admin application will start at the login screen. From there, the user can only login – all other pages are restricted. Currently, there is no functionality for admin to register. Once logged in, the user will be able to view the buildings they manage in the side navigation bar. Selecting a building will let the user manage the residents of that building. The user can choose a resident to verify, which will take them to the verification screen.

User Interface Diagrams

Low fidelity wireframes provide basic sketches of page structure and layout. Refer to the “Knockk Low Fidelity Wireframes” folder for both the admin (web) and resident (mobile) applications high fidelity wireframes.

High fidelity wireframes provide a detailed mockup of the application. Refer to the “Knockk High Fidelity Wireframes” folder for both the admin (web) and resident (mobile) applications high fidelity wireframes.

Component Design

Refer to the “Component Design” folder for both admin (web) and resident (mobile) applications’ component designs.

Each page in the design folder includes one screen of the application which has:

* Screen name
* Screen description
* Composed of (the components that make up this screen)

Screens can - and will - have multiple components per screen. The components within the screen are listed and include:

* Component name
* Component description
* Props
* State
* Methods

Components are commonly used for code reusability. While some components, like Resident and Navigation, are reused across multiple screens, others are not. However, breaking sections into components allows for abstraction. Instead of having a screen with lots of code, a component can be created and import into the file.

NOTE: Reused components and styling are listed on a separate page to eliminate redundancy. The screens’ name can be tied back to the wireframe (UnknownResident and KnownResident share the same screen in the wireframe, view a neighbor).

NOTE: The current scope of the project does not include publishing the mobile app to the app store, and lifting and shifting the web app. In the future, however, this will be revisited. The mobile application will be supported on both iOS and Android devices, running iOS 13.4+ and Android 6+. The web application will be compatible with web browsers such as Google Chrome, Safari, and Microsoft Edge.

**Expo Diagrams**

Refer to the “Expo Diagrams” folder for both the admin (web) and resident (mobile) applications’ (sort of) UML diagrams. This is not a typical class diagram and has been modified. UMLs are for OOP languages, which JavaScript is not, but it was important to show parts of the Expo application.

The diagrams include utility, screen, component, constant, and state. Assets and stylesheets were not included. Please refer to the “Key” page in the “Component Design” folder for assets that will be used and styling that will be created.

NOTE: This is not a typical UML diagram and was modified for React Native. Where class properties are listed in a typical UML, state was listed. Methods were listed as normal. Access modifiers were not listed because React Native does not have these. Props being passed, and variables declared, are not listed in the diagrams. For example, the APIService will declare a string and initialize it with a string of the API’s URL; this string was not listed in the APIService diagram.

NOTE: Zustand is a lightweight state management library the frontend applications will utilize. State is the data that is being managed, and actions (listed in the methods part of the diagram) allow for modifying the state. The store is used as a hook within screens and components.

Service API Design

API documentation can be found either in the External Resources section of the Appendix or in “API Documentation” attachment. This includes information about the API like how to access it, examples of applications that consume it, information regarding requests/responses, etc.

Refer to the “API UML” attachment for the UML diagram of the Spring Boot backend application.

The UML depicts all classes of the Spring Boot application. Having two controllers – admin and resident will separate the requests coming from the two frontend applications. The controllers will have a request mapping of /admin and /residents, respectively. Get and delete requests, will make use of request parameters - the use of request bodies is not industry standard for get and delete requests. Posts and puts will use request bodies - these bodies will be mapped to specific models. The controllers will interact with their respective business services class’ with the use of models. The business services will communicate with the data services class through entities. The use of Spring Data JDBC will use repositories to make queries to the database.

All requests will return a ResponseEntity. This entity will include a status and body. The body will be a ResponseModel. For an unsuccessful response, the response will be a ResidentModel with a message. For successful responses, the ResponseModel will include data of a generic type. For example, a request that gets the resident will return the ResidentModel as the data property of the ResponseModel.

NOTE: There are models that are only specific for requests or responses. Responses that only are returning one property will just pass that property, but two or more properties will have a model. For example, getting a friendship sends the residentId and neighborId (also known as friend) in the request as a parameter. The response will use a FriendshipModel (with residentId, neighborId, and accepted properties) as the data property of the ResponseModel. Models, like the UserModel, will only be used in a request because the resident’s credentials should not be returned (we want to only return the id of the resident - if the request is successful, of course).

NOTE: Common properties of a Resident were abstracted into a separate OptionalResidentModel class that both the Resident and Register models extend. This OptionalResidentModel class is also used for updating the Resident, as these are the only properties a resident can update after the registration.

NOTE: A key has been provided. The outlined shapes (most of which are rectangles) depict the high-level package the class(es) are located in. Controller classes will be in the controller package; models classes in the model package; business service class in the business package; data service classes in the data.services package; logging classes in the util package. There are also packages with the data class that were not shown in the diagram: repositories classes reside in the data.repositories package, entities in the data.entities package, mapper classes in the data.mappers package.

NFR’s

NFR: As a system, I wouldn’t not like to store passwords in clear text so that passwords are secure.

Implementation: Using react-native-bcrypt all passwords will be hashed before being sent in the request body. Pseudo code:

Login(email, password){  
 hash password

initialize and declare login model with email and hashed password

make http post request to login with login model

get response back and do further logic

}

Testing: To ensure that the passwords are hashed, the NFR will be tested by:

* Scanning the database to ensure passwords are not in plain text.
* Using Wireshark for packet sniffing to see if the password was hashed.

Operational Support Design

**React Native (Knockk-Resident and Knockk-Admin)**

Logging:

react-native-logs will be used for logging the frontend. These logs will have:

levels: debug, info, warn, error.

format: [YYYY-MM-DD HH:MM:SS] {log level} : {function}-{method} method : {message}

example: [2024-11-14 12:00:00] error : KnownResident – getResident method : HTTP GET returned invalid data.

axios-logger will be used for logging Axios.

axios methods: request, response

format: [Axios {axios method}] [YYYY-MM-DD HH:MM:SS] {HTTP Method} {URL}

example: [Axios Request] [2024-11-14 12:00:00] GET http//localhost:3000/residents/

Current implementation is for development, so these logs will be outputted to the console.

NOTE: error logs can, and will, not only be console logged, but also sent to Sentry. Sentry’s primary purpose, however, is performance monitoring. In the future, for production, this should be revisited. A new strategy will need to be implemented for improved data retention and storage.

Monitoring:

Sentry will be used to monitor the app. The implementation of Sentry into the Expo project is simple and offers a lot of features like error monitoring, tracing, session replay, cron monitoring, and user feedback. The logs and reports have a retention policy of 90 days. For production, this will need to be revisited to implement long retention policies – and possibly exporting the data somewhere (like to a storage bucket). Other pricing plans will be sought out in production, which include upgraded features, like an increase in the limit of error monitoring logs.

**Spring Boot REST API**

Logging:

Spring AOP dependency will be used for logging. Logs will be sent to the console for debugging. Spring AOP allows for method tracing in the application. Future production level implementation will include sending logs to Loggly once the application is lifted and shifted to the cloud. These logs will have:

levels: debug, info, warning, error

format: YYYY-MM-DD HH:MM:SS {level} : {class} - {method}

example: 2024-11-15 12:00:00 INFO : com.gcu.controller.ResidentController – Entering ResidentController.getResident()

Monitoring:

Current implementation of this project is running locally. Future implementation will lift and shift to the cloud so that it will be available always (in relation to the cloud provider’s SLA). Since it will only be running when the developer is running it locally, there so no need to monitor the application.

NOTE: In the future, Uptime Robot will be used to monitor the applications availability.

**Supabase Database**

Logging:

Supabase offers log tracing for the database. The current pricing plan (free) provides log retention for one day. Selecting the log allows displays metadata about the log. They, Supabase, have its own log format:

DD Month HH:MM:SS {level} {event message}

Monitoring:

Supabase offers a user-friendly dashboard with metrics related to the Postgres instance. These auto-generated reports include monitoring charts for the database, memory, CPU. The retention policy for these metrics is one month. For production, a different pricing plan will be pursued for features critical for production apps, like increased log retention, log draining, uptime SLAs, etc.

Other Documentation

Reporting:

**Frontend**

The admin application has two main reports that are generated within the application:

*Managing residents*

The admin user will be able to view residents. This report will include:

* User id
* First name
* Last name
* Email
* Floor number
* Room number
* Bedroom
* Lease start date
* Lease end date

The report will populate 10, 25, or 50 residents based on what the admin user chooses; default will be 25 residents. The user will have 5 visible pages of pagination. The admin user will be able to sort based on floor number increasing, floor number decreasing, last name a to z, or last name z to a; default will be floor number increasing. There will be no filtering on the data as this is an out-of-scope feature.

*Verifying residents*

The admin user must first verify the data that the user entered during registration. This report will only display one user and will include:

* User id
* First name
* Last name
* Email
* Floor number
* Room number
* Bedroom number

From this information, the admin user will either reject the resident or activate their account (takes a few extra steps). Verification will only verify one resident at a time, so there will be no sorting or filtering of information.

The user application has no reports.

**Backend**

Supabase provides reporting of the database. These autogenerated reports include one month’s view of data, including: API requests, auth and storage.

Appendix A – Technical Issue and Risk Log

Risks, such as lack of experience with Expo, have been mitigated through proof of concepts. Risks that are out of control of the Knockk development team, like Supabase going down, cannot be mitigated more than they have – which is making sure team knows the ins and outs of the software/technology. No issues have been reported at this time.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Issues and Risk Log | | | | | | | | |
| **Issue or Risk** | **Description** | **Project Impact** | **Action Plan/Resolution** | **Owner** | **Importance** | **Date Entered** | **Date to Review** | **Date Resolved** |
| 1 | Software tools like Figma, GitHub, Draw.io start incurring charges. | The team will start paying out of pocket. | Keep up to date with Figma, GitHub, and Draw.io policies and charges. If huge fees start incurring, pursue a different software. | Grace | Low | 9/22/2024 | Bi-monthly (to ensure charges aren’t being incurred). |  |
| 2 | The team has little experience with React Native and Expo. | Frontend applications will not be built. | The team created a proof of concept using Expo (framework for React Native). | Grace | Medium | 9/22/2024 | 9/22/2024 | 11/1/2024 |
| 3 | The team has little experience with Supabase. | The API will not be able to communicate with the Postgres database and all Supabase’s features will not be fully taken advantage of. | The team created a proof of concept using Supabase | Grace | Medium | 9/22/2024 | 9/22/2024 | 11/1/2024 |
| 4 | Lots of technologies to choose from; unknown technology stack. | Delays the progress moving forward. | The team created proof of concepts of proposed technologies and made decisions based on a wide set of reasons (see Technical Design Decisions for more information). | Grace | Medium | 9/22/2024 | 9/22/2024 | 11/1/2024 |
| 5 | Creating data in the database. | If there is no data, the API has nothing to return, and the frontend applications will have nothing to display. | Creating data of real information. | Grace | Low | 9/22/2024 | 1/1/2025 | 1/1/2025 |
| 6 | Industry best practices were not implemented. | This leads to poor design and the potential to have to redo in the future (double the work). | Seek mentorship from industry professionals to ensure industry best practices are being utilized. | Grace | Medium | 9/22/2021 | Weekly |  |
| 7 | The team has little experience with JWTs. | API will not be secured appropriately giving anyone access to the endpoints. | Initially out of scope. If all features in scope are completed before 5/3/2025, real-time databases will be researched, and proof of concept will be developed. | Grace | Medium | 9/22/2024 | 5/3/2025 |  |
| 8 | The team has little experience with real-time databases. | Residents will not be able to chat with one another. | Initially out of scope. If all features in scope are completed before 5/3/2025, real-time databases will be researched, and proof of concept will be developed. | Grace | Medium | 9/22/2024 | 5/3/2025 |  |
| 9 | The team has no knowledge of making the API public facing. | The API will only be used running locally if it is not public facing. | Initially out of scope. Seek a cloud provider that can host the API, like AWS, and provide SSL certificates. Self-signed certificates messy. | Grace | Medium | 9/22/2024 | 5/3/2025 |  |
| 10 | The team has no knowledge of building out a push notifications feature. | The app will not have a push notification feature. | Initially out of scope.  If all features in scope are completed before 5/3/2025, push notifications will be researched, and a proof of concept will be developed. | Grace | Medium | 9/22/2024 | 5/3/2025 |  |
| 11 | The team has no experience deploying to the App or Google Play stores. | The mobile application will not be deployed to the App or Google Play stores. | Initially out of scope. If all features in scope are completed before 5/3/2025, research will be done on how to deploy to the stores. | Grace | Medium | 9/22/2024 | 5/3/2025 |  |
| 12 | The team created too much work – bit off more than they can chew. | All features will not be completed. | The team will utilize Jira to break down the project into manageable tasks. It’s a marathon; not a sprint. | Grace | Medium | 9/22/2024 | Bi-weekly |  |

Appendix B – References

Not applicable – no external references were utilized.

Appendix C – External Resources

|  |  |
| --- | --- |
| **GIT URL:** | [*https://github.com/gradlund/knockk*](https://github.com/gradlund/knockk) |
| **Swagger:** | <https://app.swaggerhub.com/apis/gradlund_gcu/Knockk1/1.0.0> |
| **Hosting URL:** | Not applicable. |