**High level Architecture**

**P06: Open Source Backend In Rust**

| **Student ID** | **Name** |
| --- | --- |
| **24100305** | **Abdul Wahab** |
| **24100121** | **Faraz Mansur Ahmad** |
| **24100012** | **Muhammad Saad** |
| **24100296** | **Moiz Raza Amir** |
| **24100085** | **Ahmed Mozammil Iqbal** |

**Table of Contents**

[1. Introduction 2](#_gjdgxs)

[2. System Architecture 4](#_30j0zll)

[2.1 Architecture Diagram 5](#_1fob9te)

[2.2 Architecture Description 5](#_3znysh7)

[2.2 Justification of the Architecture 9](#_2et92p0)

[3. Risk Management 12](#_tyjcwt)

[3.1 Potential Risks and Mitigation Strategies 12](#_3dy6vkm)

[4. Tools and Technologies 13](#_1t3h5sf)

[5. Hardware Requirements 14](#_4d34og8)

[6. Who Did What? 14](#_2s8eyo1)

[7. Review checklist 15](#_17dp8vu)

# Introduction

To provide context for our project, Backend as a Service (BaaS) solutions essentially abstract away the complexities of REST API such that the developer only needs to create the frontend and use the ready-made BaaS service’s methods to handle the backend. This increases a developer’s productivity as there is no need to write complex backend code as a result. Many known BaaS services come bundled with several available functionalities such as:

● Built-in REST API CRUD operations

● Out-of-the-box authentication

● File Storage

● OAuth Adapters

● Realtime Databases (useful for chat applications)

This makes BaaS solutions attractive for developers. There exist several BaaS services, such as ‘Firebase’ by Google. However, Firebase is closed source and any hosting of the database and other media is done by Google itself which some developers find problematic. Firebase also uses a proprietary data store called “Firestore” which makes data migration a hassle.

As such, there is a growing trend in self-hosting for reasons such as freedom and independence in hosting one’s own services, as well as having the ability to customize applications. Due to the increasing need of customizable services and providing transparency to users, Open Source projects are becoming popular. However, self-hosting open-source BaaS solutions can be tricky as there are several services that need to be configured for them to work securely and efficiently. Most of the existing BaaS solutions provide first-class support for usage as a service. However, they are hosted by the provider, and support for self-hosting in this domain is limited.

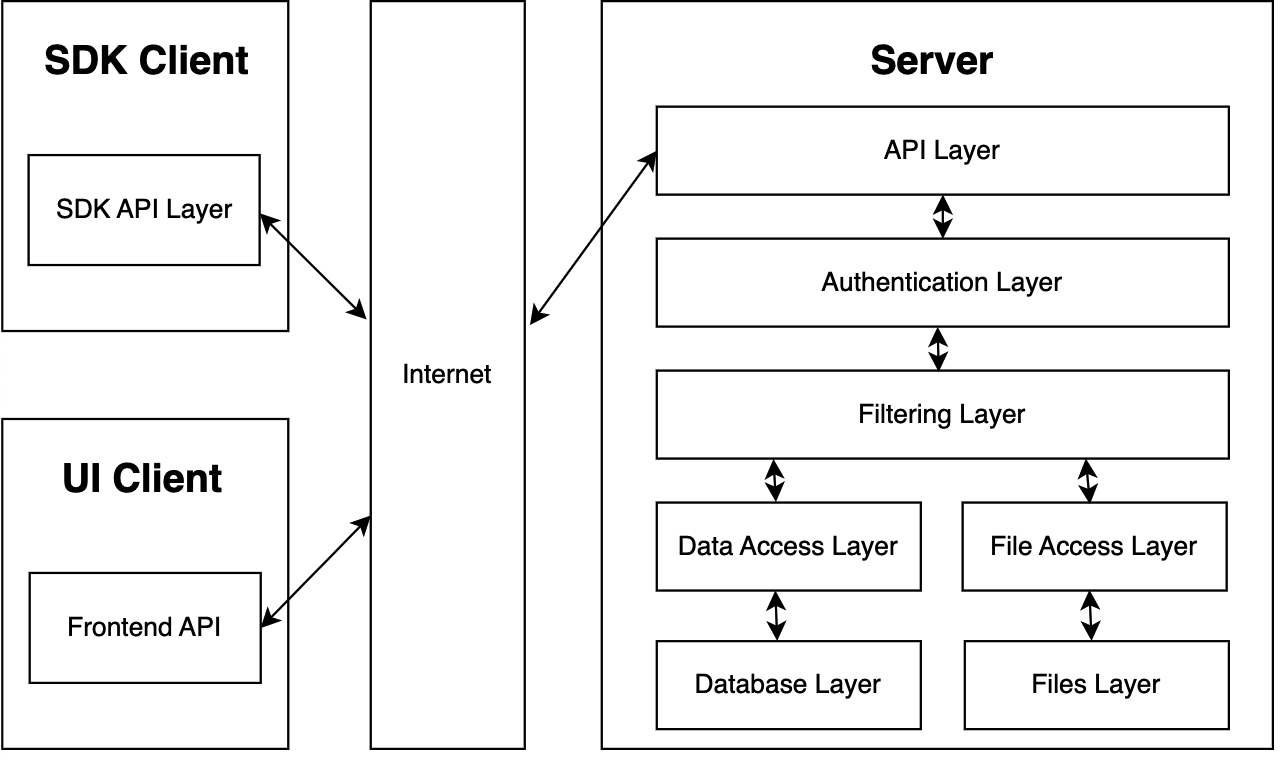
Hence we were motivated to create a lightweight backend similar to Firebase that is open source and can be self-hosted. [Pocketbase](https://pocketbase.io/) and [Supabase](https://supabase.com/) are close relatives of the idea, and are the references that will be used throughout the development of our project. The goal is to create a lightweight and fast backend while providing users well made documentation and a clean UI to easily navigate our service. There is a high demand for efficient and less storage intensive backend solutions and we are choosing to address this need.

Unlike Pocketbase which uses Go and Supabase which uses Typescript, we will be writing our backend in Rust: a fast, systems programming language that performs orders of magnitudes better than both Go and Typescript in benchmarks. Rust is also known for its robust error handling mechanisms and type-safety that make software safe from expensive errors such as “null pointer exceptions” and allows for a great degree of compiler level optimization. For these reasons we felt that it was appropriate to program our project in Rust.

The potential users will mainly be developers. However when developers deploy our service as a backend for their softwares, System Admins will be able to use our provided User Interface to make any edits.

# System Architecture

## Architecture Diagram



## 2.2 Architecture Description

The system is divided into 3 main components that follow the ‘Client-Server Model’ externally.

* The Server
* The SDK client
* The UI client

The Server itself has a ‘layered architecture’ with the following components:

* **API Layer**: is used to perform HTTP requests to receive or send data across the Internet. As such it receives the JSON objects from SDK client and UI client before passing it on for further processing in the lower server layers. The API Layer further sends the processed data from the server to the SDK client and UI client for the Admin and Developer to interact with. The use cases handled by this layer are as follows:
  + Create a record
  + Update a record
  + Delete a record
  + Read a record
  + Read a list of records
  + Create a collection
  + Delete a collection
  + Store, retrieve and delete a file
  + Allow user to reset their passwords
* **Authentication Layer:** When the data is passed from the API layer down to the Authentication layer, it is checked whether the user is logged in to decide whether the user is authenticated to interact with the databases and files. Other features of the Authentication layer are to handle logins, signups and the handling of cookies and sessions as well as the handling of Authentication adapters. The authentication layer handles the following use cases:
  + User Authentication i.e., User login/logout.
  + User Profile Management i.e., resetting password, usernames etc.
  + Authentication of user input.
  + Security against common security threats i.e., SQL injection, cross-site scripting, and Insecure Direct Object References.
* **Filtering Layer:** This layer is used to check permissions and rules whether the user is allowed to interact and perform certain databases or is allowed to access certain files. If the user has permissions, then the request is forwarded to the Data Access Layer and File Access Layer. It handles the following use cases:
  + Access Control and Permissions Management to check whether a user has the appropriate permissions to read, write, update, or delete data in specific databases
  + File Permissions: It checks whether a user is allowed to access or manipulate files stored within the application.
  + Permission Denied Handling such that when a user lacks the necessary permissions, the Filtering Layer can return appropriate error responses or redirection.
* **Data Access Layer:** This is a protected data structure that is used to perform CRUD operations on the database for the allowed users in the Filter Layer. It interacts with the Database Layer and is used to handle records and collections. The specific use cases are as follows:
  + Create a record through the API and UI
  + Update a record through the API and UI
  + Delete a record through the API and UI
  + Read a record through the API and UI
  + Read a list of records through the API and UI
  + Store, retrieve and delete a file through the API
  + Create a user with username and password through API and UI
* **Database Layer:** The database layer comprises files with the ‘.db’ extension that hold all the database records. The Data Access Layer is used to interact with these Database files.
* **File Access Layer:** This layer is also a data structure that is used to interact with the files stored by the user. The file access layer handles the deletion, creation and storing of Files. The use cases handled by this layer are as follows:
  + Store, retrieve and delete a file through the API
  + Set access rules on files and collections through the UI.
  + Upload and alter S3 credentials to change file upload destination
* **Files Layer:** In this layer is the directory storing all the user files with which the File Access Layer interacts. Hence, it contributes to the same use cases the File Access Layer does.

The SDK Client comprises of the following layers:

* **SDK API:** The SDK API layer is used by the developer to interact with the server. The API serves as the primary interface for all actions performed by the SDK and hence is a core class with which almost every other class communicates. Currently we are making our SDK compatible with Javascript based frameworks and it will be downloadable through the npm package registry.

The UI Client comprises of the following layers:

* **Frontend API:** This layer will be made in react and is the UI interaction that users will have with the system for all our use cases.

Collectively, the frontend and the SDK API handle the following use cases:

* Create a record
* Update a record
* Delete a record
* Read a record
* Read a list of records
* Create a collection
* Delete a collection
* Store, retrieve and delete a file
* Create a user with username and password
* Allow user to reset their passwords
* Create an Admin, Delete an Admin through the UI

## Justification of the Architecture

Pros:

1. A layered server architecture promotes a modular design, making it easier to develop, test, and maintain different components independently. This allows us to work on our server stack on a component by component basis, meaning that components can be swapped out, in or even upgraded without extended server downtime. The layered architecture can also help deal with increasing user load through the introduction of layers with the sole purpose of load balancing so as to not overwhelm the server.
2. A layered server architecture allows for the addition and/or modification of layers without affecting the server as a whole. This makes it scalable, allowing for easy adjustments to handle increased load or new features.
3. The overall client server architecture of the system allows for a singular centralized server instance that is the sole source of truth and enforcer of security. Data syncing, backup and security is much easier to implement on a singular node.
4. The overall client server architecture of the system allows for cross platform compatibility since the client instance can be run on a variety of platforms.

Cons:

1. Data transformation to make it compatible for different layers in the layered server architecture adds computational load and latency.
2. Inter layer data security is easy to overlook in a layered server architecture. Data must be secured when going from layer to layer to minimize attack vectors available to a squatting adversary.
3. The system’s server client architecture is difficult to maintain since both server and client instances need to be kept in sync in terms of changes to API, Data, etc.

Keeping both the pros and cons in mind, a layered server architecture and an overall client server system architecture is justified for our project requirements since it helps implement our non-functional requirements like high availability, scalability, cross platform use, concurrent user access. Following are some of the justifications for the architecture and how they relate to the non-functional requirements:

1. **Modularity and Maintainability:** The layered server architecture promotes a modular design, which is highly beneficial for this BaaS system. This modularity aligns with the goal of creating a lightweight and customizable backend. Developers can also work on different components independently, allowing for easier development, testing, and maintenance. This modularity is particularly important for an open-source project, as it enables the community to contribute to specific components without disrupting the entire system. This accomplishes the following non-functional requirements (NFRs) as were described in the Requirements Specification document:
   1. The system should be available 99% of the time.
   2. The system should not utilize more than 1 GB of memory at any time during its execution.
   3. The system's user interface (UI) should have at most 15 pages with direct buttons to perform tasks like CRUD for example. And the UI should require less than one week training for end-users to use proficiently.
2. **Data Flow and Integrity:** While data transformation and handling can add computational load and latency, the described architecture acknowledges this challenge. This awareness of potential latency issues indicates a focus on optimizing data flow within the layers by making a safe compromise with data integrity. The overall client-server architecture, in which the server is the sole source of truth and enforcer of security, is well-aligned with the requirement for a secure BaaS system. It simplifies data syncing, backup, and security implementation. The NFRs this justification covers are:
   1. The system should be able to handle concurrent requests from 1000 users.
   2. The system should be able to handle 32000 database writes in 1 second.
3. **Cross-Platform Compatibility:** The ability to run the client instance on various platforms aligns with the goal of providing a solution that can cater to a wide range of developers and applications. Cross-platform compatibility is important, as it broadens the potential user base and thus achieves the following NFRs:
   1. The system should be cross-platform, supporting Windows, macOS, and Linux.

Together, these architectural decisions reinforce and extend the following foundational layers of a BaaS\*:

* **Foundation Layer:** At the foundation layer, the core components of the application are established. This layer forms the basis upon which the entire system is constructed. The architectural decisions for this layer include the choice of data storage mechanisms, and data backup strategies. The justification for this layer's existence lies in the need to establish a robust and reliable infrastructure to support the application's operations.
* **Applications Layer:** This is primarily handling user requests such as logins and various application-specific functions. The architectural decisions in this layer include defining the APIs and endpoints to handle user interactions.
* **Connection Layer**: This layer enables the application servers to access the internet. It involves network communication, data exchange with external services, and managing connections to ensure data flows seamlessly between the application and external resources. The architectural decisions here revolve around network protocols, security mechanisms, and optimizing data transfer.

*\*(Reference:* [*https://radixweb.com/blog/backend-as-a-service-baas*](https://radixweb.com/blog/backend-as-a-service-baas)*)*

# Risk Management

## Potential Risks and Mitigation Strategies

| **Sr.** | **Risk Description** | **Mitigation Strategy** |
| --- | --- | --- |
|  | Unauthenticated user might try to access the api endpoints. | Usee industry standard Auth protocols like OAuth to ensure a robust authentication mechanism. |
|  | Access to sensitive user data during data leaks. | Secure storage through hashing of highly sensitive data to mitigate damage from leaks. |
|  | Injection attacks which can cause data loss, data tampering leading to financial and business loss. | All input data should be properly validated and sanitized, validation and sanitization should not be limited to the client side but should be extended to the server side in order to ensure maximum coverage. Parameterized and prepared queries should be used as much as possible. |
| 4. | Authenticated users might try to access the data for which they are unauthorized | Use robust filtering layer allows role based authorisation giving the user fine grained access control on who has access to some data. |
| 5. | Storing user data might bring about legal and regulatory challenges, especially if data crosses borders. | Users are responsible for their own data as the server is self-hosted. |
| 6. | Third-party libraries or dependencies used might contain vulnerabilities. | Use well-audited libraries which regularly receive security updates. |
| 7. | As user demand grows, the backend might struggle to cope with scalability issues. | The architecture to be horizontally scalable from the outset. All the bottlenecks are identified and addressed during stress tests. |
| 8. | Loss of user data due to server failures or other issues. | Allow the user to easily create a backup for all the data stored in the database. |
| 9. | Users might misconfigure the backend leading to vulnerabilities. | Provide clear documentation, to guide users through the configuration process. |
| 10. | Without proper logging, malicious activities might go unnoticed. | Implement comprehensive logging and monitoring so users can clearly see all the requests being made to their server. |

# Tools and Technologies

Some of the tools that we will be using to develop our project are:

**Rust 1.73:** A memory-safe systems programming language.

**ReactJS 18:** A Javascript DOM manipulation library.

**EJDB 2.0:** An embedded NoSQL database.

**Docusaurus 2.0:** A ReactJS Static Site Generator for documentation.

**Javascript ES11:** An interpreted programming language for the web.

**Axum 0.6:** A lightweight Rust library for implementing REST APIs.

# Hardware Requirements

There are no strict requirements for hardware in both the development and software machines, but the following specs should allow the system to run comfortably. The performance of the system would be better if the available CPU is better, the RAM is a newer version and the disk storage is modern. But the system should have the following specifications to run comfortably.

**Deployment Servers:**

* At least 1GB of RAM
* At least 4GBs of disk space.
* A modern CPU (>2003)

**Development Machines:**

* At least 8GBs of RAM
* At least 128GBs of storage
* A modern CPU (>2003)

# Who Did What?

| **Name of the Team Member** | **Tasks done** |
| --- | --- |
| M.Saad | Introduction, Section 2.3 |
| Abdul Wahab | Section 4 and 5 |
| Ahmed Mozammil Iqbal | Section 2.1 and Section 2.2 |
| Faraz Mansur Ahmad | Section 2.2, 2.3 |
| Moiz Raza Amir | Section 3 |

# Review checklist

Before submission of this deliverable, the team must perform an internal review. Each team member will review one or more sections of the deliverable.

| **Section** **Title** | **Reviewer Name(s)** |
| --- | --- |
| 1, 2.1, 2.2, 2.3 | Faraz Mansur Ahmad |
| 1, 2.2 | Abdul Wahab |
| Section 4 | Ahmed Mozammil Iqbal |
| Section 5 | Muhammad Saad |
| Section 2.1,2.2 | Moiz Raza Amir |