**EZBASE - A Backend-as-a-Service Solution**

**SPROJ Report**



**Ahmed Mozammil Iqbal 24100085  
Moiz Raza Amir 24100296  
Abdul Wahab 24100305**

**Faraz Mansur Ahmad 24100121**

**Muhammad Saad 24100012**

**Advisor:   
School of Science and Engineering  
Lahore University of Management Sciences  
Submission Date: 28/04/2024**

**Acknowledgement and Dedication**

**We are grateful to our supervisor, Dr Waqar Ahmad, for his continual support and guidance throughout the project.**

**We also thank Dr Sohaib Ayub for co-advising us alongside Dr Waqar Ahmad.**

**Certificate**

I certify that the senior project titled “**EZBASE: A Backend-as-a-Service Solution**” was completed under my supervision by the following students:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

and the project deliverables meet the requirements of the program.

------------------------------------- Date:

**Advisor (Signature)**

------------------------------------- Date:

**Co-advisor (if any)**

# Table of Contents

**Table of Contents 5**

**1. Introduction 7**

a. Introduction 7

b. Objective and Scope 10

c. Development Methodology 10

d. Contributions 10

**2. System Requirements 12**

a. System Actors 12

b. Functional Requirements 12

c. Non-functional Requirements 13

**3. System Architecture 15**

[a. Architecture Diagram](#_gjdgxs) 15

[b. Architecture Description](#_30j0zll) 15

[c. Justification of the Architecture](#_1fob9te) 18

d. Tools and Technologies 20

**4. Requirements Specifications 21**

a. Use Cases 21

[22. Class Diagram](#_2et92p0) 41

[23. Sequence Diagrams](#_tyjcwt) 42

**1. Create a Record through the API and UI 42**

**2. Delete a Record through the API and UI 43**

**3. Create a Collection through API and UI 43**

[**Software Development Methodology and Plan**](#_9abqf0vkoelb) **51**

[a. Software Process Selection](#_3dy6vkm) 51

[b. Gantt Chart](#_1t3h5sf) 54

**5. Database Design and Web Services 54**

a. Database Design 55

b. API Specification 55

**6. System User Interface 56**

**7. Project Security 66**

a. Project Threats 66

b. Potential Losses 66

c. Security Controls 66

d. Static and Dynamic Security Scanning Tools 67

**8. Risk Management 68**

[Potential Risks and Mitigation Strategies](#_4d34og8) 68

**9. Testing and Evaluation 71**

**10. Deployment Guidelines 74**

**11. Conclusion 75**

a. Summary 75

b. Challenges 75

c. Future 76

[**12. Review checklist**](#_2s8eyo1) **77**

**13. References 78**

# Introduction

## Introduction

Backend-as-a-Service (BaaS) is a cloud computing service model that allows developers to outsource the backend aspects of web or mobile application development. This model enables developers to focus primarily on the frontend while relying on BaaS providers for essential server-side functions.

By using APIs (Application Programming Interfaces) and SDKs (Software Development Kits) provided by the BaaS provider, developers can seamlessly integrate these backend services into their applications. This eliminates the need for them to develop and maintain the backend infrastructure themselves. Consequently, BaaS facilitates faster development and deployment of both mobile and web applications, including single-page applications, by significantly reducing backend programming requirements. 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 and Supabase 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.

We will be writing our backend in Typescript: a fast, strongly typed programming language that builds on JavaScript, providing a scalable tooling for projects. In the context of a BaaS solution, Typescript offers the following advantages:

* TypeScript's static typing system helps catch errors early in the development process by enforcing type safety. This is particularly useful in a BaaS environment where data types and structures need to be predictable and consistent for API communication and data storage.
* The type system not only catches errors but also enhances code readability and maintainability. It makes the codebase easier to understand and navigate, which is crucial when building complex backend systems that multiple developers might work on.
* TypeScript's features make it well-suited for developing large-scale applications. As BaaS solutions might scale to handle many different clients and large datasets, TypeScript’s organizational benefits (like namespaces and modules) help manage large codebases efficiently.
* It is a superset of JavaScript and has access to the vast npm ecosystem, allowing developers to use countless libraries and tools that are essential for building BaaS solutions.

The targeted users will mainly be the following:

* App developers (web or mobile).
* 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.
* As an open-source project, the EzBase community will be able to add their contributions to the solution as well in the form of feedback and additional features.

The EzBase BaaS solution will have the following components:

* A self-hostable server, the instance of which can be run directly on the target user’s i.e., developer machine.
* User Interface (UI) Client for directly managing the configurations, resources, and services.
* A SDK published on npm. The developers will be able to install the packages and utilize the API functionalities within.

## Objective and Scope

The primary objective of the EzBase is to provide developers with a flexible, scalable, and easy-to-use backend platform that can be self-hosted allowing quick development and deployment of websites This project aims to address the limitations associated with closed source solutions like Firebase by offering an open-source alternative that gives developers greater control over their backend environments. This includes full ownership of the data, the ability to customize and extend the backend services as needed, and independence from proprietary technologies that restrict data portability and integration. The scope of this project includes the development of a lightweight backend server bundled with a user-friendly UI client for managing server configurations, along with an SDK published on npm, enabling developers to easily integrate and utilize the API functionalities provided by EzBase.

## Development Methodology

For the development of the EzBase, we adopted the Agile software development methodology. This approach was chosen for its flexibility, iterative nature, and focus on collaboration, which is crucial for responding to the evolving problems and features as we progressed in the development. After each sprint, the team held a meeting with the advisor to reflect on the sprint process and identify improvements for future sprints.

## Contributions

Our solution prioritizes self-hosting, meaning the user (developer) can host our system depending on their budget. Since our system is open-source the developer can clone the project repository and additional features as required for their use case making our solution dynamic. Since the server is written in typescript, a popular and well-known language, making such changes would be easier. Furthermore, we support linking additional fields like files and arrays directly through the admin UI, which many of the other solutions don’t support. Server statistics like API hit rate, memory usage and time taken for each request is also shown which allows the user to manage the cost of their deployed server accordingly. We provide a robust Role Based Authorization system based on user metadata. These rules can be easily created and chained in the Admin UI.

# System Requirements

## System Actors

|  |  |
| --- | --- |
| **Actor Name** | **Description** |
| Developers | Developers are the primary users of our BaaS solution. They will interact with the system extensively to create the frontend of their applications and use the BaaS service's methods to handle the backend such as utilizing:  ● REST API CRUD operations  ● Authentication  ● File Storage  ● OAuth Adapters  ● Realtime Databases (optional) |
| Admins | Admins will be responsible for configuring and managing the BaaS system. Hence, their responsibilities, interactions, and system privileges with the system will include, but not limited to:  ● User Management  ● System Configuration & Customization  ● Security Management  ● Data Management  ● Scaling & Resource Management  ● Backup & Recovery  ● Troubleshooting & Support |

## Functional Requirements

List down system requirements. You may group requirements according to actors or modules

|  |  |
| --- | --- |
| **Sr#** | **Requirement** |
| 1 | As an admin, I want a clean and usable GUI to interact with the system. |
| 2 | As a developer, I need control over all of my data. |
| 3 | As a developer, I need the system to be easily self-hostable. |
| 4 | As a developer, I need a modular and expansive SDK to use the system. |
| 5 | As a developer, I need excellent documentation to consult when necessary. |
| 6 | As a developer, I need good logging mechanisms to allow analysis and optimization |
| 7 | As a developer, I need cross-platform support to be able to host my solution anywhere, preferably freely. |
| 8 | As a developer, I need a safe and easy to use authentication system. |
| 9 | As a developer, I need several OAuth mechanisms out of the box. |
| 10 | As a developer, I need to be able to create, read, update and delete records in the database. |
| 11 | As a developer, I need to be able to upload files to the system. |
| 12 | As a developer, I need to be able to access and modify all of my uploaded files. |

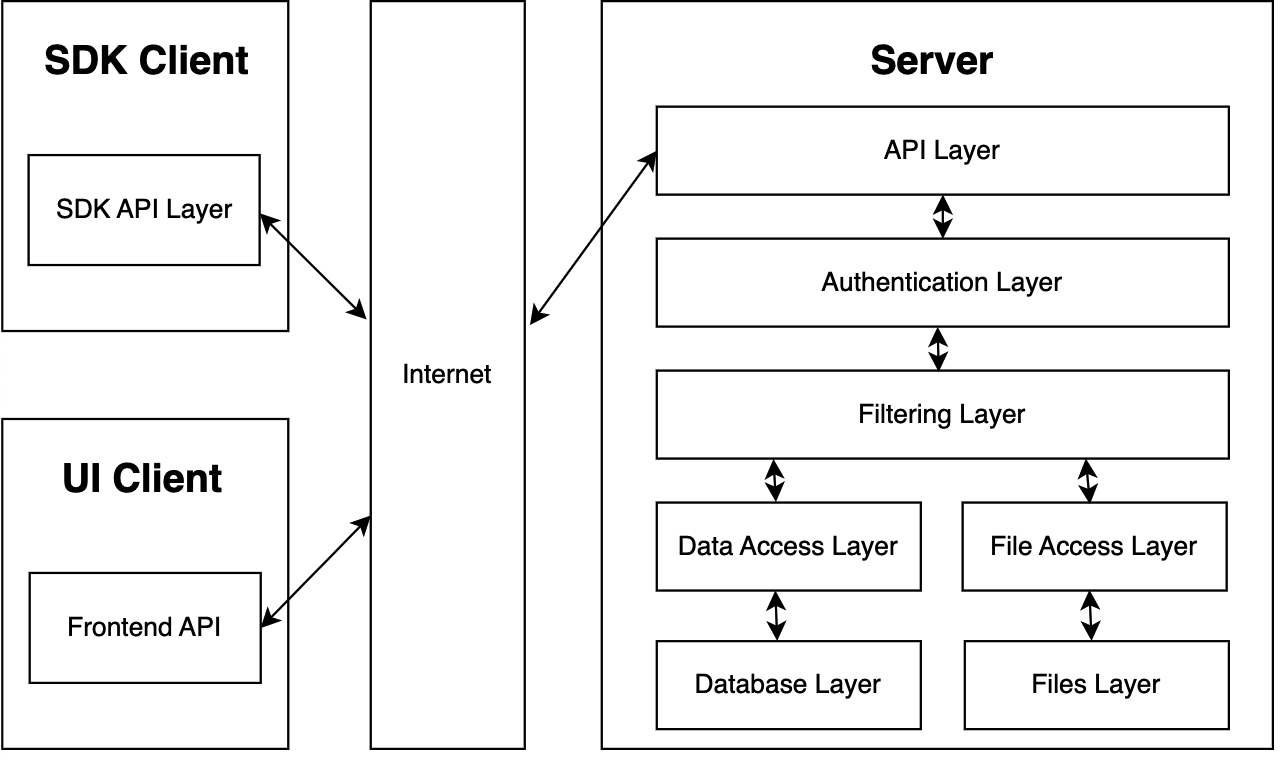
## Non-functional Requirements

List down non-functional requirements.

|  |  |
| --- | --- |
| **Sr#** | **Requirements** |
| 1 | The system should not utilize more than 1 GB of memory at any time during its execution. |
| 2 | The system should be available 99% of the time. |
| 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. |
| 4 | The system should be able to handle concurrent requests from 1000 users. |
| 5 | The system should be able to handle 32000 database writes in 1 second. |
| 6 | The system should be cross-platform, supporting Windows, macOS, and Linux. |

# System Architecture

## Architecture Diagram



## 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 Azure 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:
   * The system should be available 99% of the time.
   * The system should not utilize more than 1 GB of memory at any time during its execution.
   * 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:
   * The system should be able to handle concurrent requests from 1000 users.
   * 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:
   * 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.

## Tools and Technologies

* Frontend UI:
  + React
  + JSX
  + Vite
  + Tailwind
  + ESLint
* Backend:
  + Hono
  + NeDB
  + Node
  + Typescript
* SDK:
  + Typescript
  + npm
* Documentation Website:
  + Docusaurus
* Other:
  + Azure Blob Storage
  + Github
  + Bun

# 

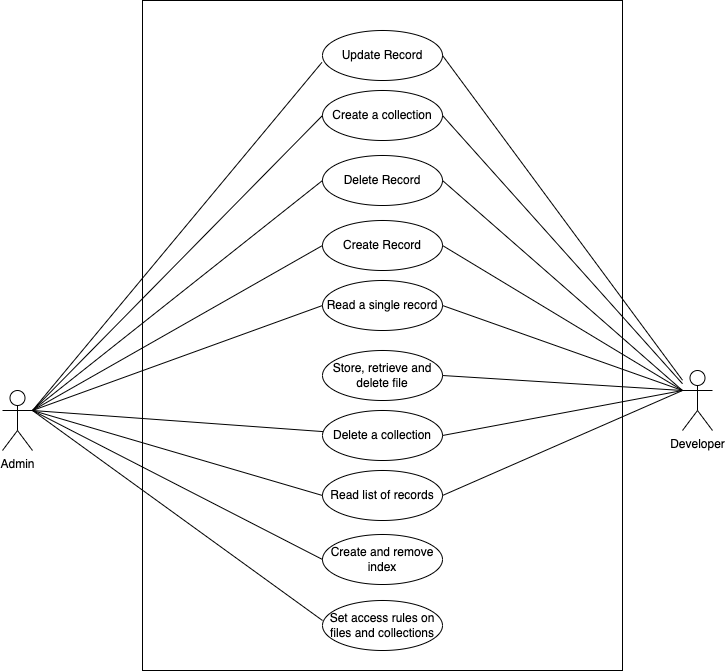
# Requirements Specifications

## Use Cases

**Use Cases:**

1. Create a record through the API and UI
2. Update a record through the API and UI
3. Delete a record through the API and UI
4. Read a record through the API and UI
5. Read a list of records through the API and UI
6. Create a collection through the API and UI
7. Delete a collection through the API and UI
8. Create and remove an index on a collection through the UI
9. User Logs.
10. Store, retrieve and delete a file through the API
11. Set access rules on files and tables through the UI
12. OAuth adapters for Google and Facebook
13. Create a user with username and password through API and UI
14. Log in/out user
15. Allow user to reset their passwords through the API
16. Create an Admin, Delete an Admin through the UI
17. Information website with Documentation (https://docusaurus.io/)
18. Start a Server to serve all the requests
19. Upload and alter Azure credentials to change file upload destination
20. Upload and alter email credentials to change email server destination
21. Create a record through the API and UI
    1. **Use Case Diagram:**

Part 1:

Part 2:  


1. Create a record through the API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-001 |
| **Purpose** | | The purpose of this use case is to allow developers and system admins to create a new record in the backend database using both the API and the user interface (UI). |
| **Pre-conditions** | | 1. The user has appropriate permissions and authentication credentials. 2. The backend service is up and running. |
| **Post-conditions** | | 1. A new record is successfully and accurately created in the database. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user accesses the API endpoint or navigates to the "Create New Record" section in the UI. | |
|  | The user provides the necessary data for the new record. | |
|  | The user submits the request for the new record to be created. | |
|  | The backend processes the request, validates the data, and creates a new record in the database. | |
|  | The backend sends a success response to the user. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | The user can either access the API endpoint or the UI to create a new record. | |
| **Step #** | **Exception Paths** | |
|  | In step 4, if the user-provided data is invalid or there are issues with database connectivity, the backend sends an error response to the user. | |
|  | In step 5, if the backend encounters an error while creating the record, it sends an error response to the user. | |

2. Update a record through the API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-002 |
| **Purpose** | | The purpose of this use case is to allow developers and system admins to update an existing record in the backend database using both the API and the user interface (UI). |
| **Pre-conditions** | | 1. The user has appropriate permissions and authentication credentials. 2. The backend service is up and running. 3. The record to be updated exists in the backend database. |
| **Post-conditions** | | 1. The selected record is updated successfully and as required. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user accesses the API endpoint or navigates to the "Update Record" section in the UI. | |
|  | The user selects the record to be updated. | |
|  | The user modifies the necessary data. | |
|  | The user submits the request. | |
|  | The backend processes the request, validates the data, and updates the selected record in the database. | |
|  | The backend sends a success response to the user. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | The user can either access the API endpoint or the UI to update a new record. | |
| **Step #** | **Exception Paths** | |
|  | In step 5, if the provided data is invalid or there are issues with database connectivity, the backend sends an error response to the user. | |
|  | In step 6, if the backend encounters an error while updating the record, it sends an error response to the user. | |

3. Create a Collection through the API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-006 |
| **Purpose** | | The user can create a new collection with any name they like through the API and UI. |
| **Pre-conditions** | | The system is online (the server is running), UC- 014 is operational which means the User is logged in, and the User is on the Collections view. |
| **Post-conditions** | | An empty collection with the name specified by the user is created in which the User can handle CRUD of records. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| 1. | The user clicks the button that says ‘Create Collection’ in the UI or sends an API call to the system passing the name of collection in the method. | |
| 2. | In the case of UI, it displays ‘what would you like your collection name to be?’ | |
| 3. | The User inputs the name of Collection in the UI. | |
| 4. | A request is dispatched to the system to create the Collection and the system checks whether any other collection has the same name. | |
| 5. | If no other collection shares the same name, a collection is created with a unique ID and the system sends a successful response back to the user in both cases of the UI and API. In the case of API, the name and ID of collection is returned. | |
| 6. | A collection is created by the system which the user can access through API calls. The user can also view the created empty collection in the UI. | |
| 7. | The use case ends. | |
|  |  | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. | In the case of creating collection with API, The steps go from Step 1 directly to Step 4 | |
| **Step #** | **Exception Paths** | |
| 1. | In Step 4,. if a collection with the same name already exists, an error message saying ‘collection already exists’ is returned. Or in the case of API, an error object is returned to the user. Which means the user is unable to create the given collection. | |
| 2. | If the Rules permit the User to create a collection in Step 1 and Step 2, the User will be able to create a collection. Else an error response is returned. | |

4. Delete a Collection through the API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-007 |
| **Purpose** | | The user can delete a collection through the API and UI. |
| **Pre-conditions** | | The system is online (the server is running), UC- 014 is operational which means the User is logged in, and the User is on the Collections view. |
| **Post-conditions** | | The collection is removed from the list of collections, and all data in the collection is deleted. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| 1. | The user selects the collection to delete in the UI or sends an API call to the system passing the name of collection in the method. | |
| 2. | The User clicks the button that says ‘Delete Collection’ in the UI | |
| 3. | A request is dispatched to the system to delete the Collection. | |
| 4. | A successful response is returned if the system successfully deletes the collection | |
| 5. | All of the Data in the collection is deleted and the Collection can not be accessed anymore unless it is created again | |
| 6. | The use case ends. | |
|  |  | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. | In the case of deleting collections with API, the steps go from Step 1 to Step 3 directly. | |
| **Step #** | **Exception Paths** | |
| 1. | In step 3, if a collection is unable to be deleted due to certain rules being set on the collection, then an error response is returned mentioning the collection can not be deleted. | |
| 2. | If collection does not exist then an error is returned stating that the given collection does not exist. | |

5. Store, retrieve and delete a file through the API

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-010 |
| **Purpose** | | To allow users to manage their files seamlessly, ensuring they can upload, access, and remove files as needed. |
| **Pre-conditions** | | The user must be authenticated and authorized to perform file operations. The API server along with the Azure Blob Storage bucket is up and running along with sufficient storage space. |
| **Post-conditions** | | For storing: The file is successfully uploaded and stored in the Azure Blob Storage bucket and the database is updated with a new file url . For retrieving: The user receives the requested file. For deleting: The file is permanently removed from the Azure Blob Storage bucket and the url is removed from the database. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1.** | For storing the user initiates a file upload via the API. For deleting and retrieving, the API receives an id of the file to delete or retrieve. | |
| **2.** | For storing, the API receives the file and uploads it to the server.  For deleting, API deletes the file from the server.  For retrieving, the API responds with the url to the file from where it can be accessed/downloaded. | |
| **3.** | For storing, API returns a unique url for the file which is stored in the database. For deleting, API deletes the file from the server and the url from the database. | |
|  |  | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| **1** | For each operation user can choose to perform these actions for multiple files in one request to the API. | |
| **Step #** | **Exception Paths** | |
| **1** | The user is not authenticated and not authorized to execute file operations: the API returns an unauthorized error message. | |
| **2** | The server is inaccessible or out of storage: The API returns an error message. | |
| **3** | The requested file for retrieval or deletion does not exist: The API informs the user of the missing file. | |
| **4** | If the system or the authentication module becomes unavailable or too slow to respond, the API and browser times out and returns an error. | |

6. Set access rules on files and collections through the UI.

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-011 |
| **Purpose** | | The Admin can set rules on files and collections and specify which user can access which collections and files. This is also so user data remains secure. |
| **Pre-conditions** | | The system is online (the server is running), UC- 014 is operational which means the Admin is logged in, and the Admin is on the Collections view or Files view. |
| **Post-conditions** | | Once the rules are set on files and collections, only specified users can access and modify the files and collections. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| 1. | The Admin opens the required collection/file. | |
| 2. | The Admin clicks the ‘Set Rules’ button. | |
| 3. | A form is opened with the options:   1. Who can read the collection/file? 2. Who can update the collection/file? 3. Who can delete the collection/file? 4. Who can create collections/files? | |
| 4. | In each field, the Admin can add the other users that are allowed to do each of the actions in Step 3 or alternatively make the actions admin only. | |
| 5. | The User clicks ‘Save Changes’ | |
| 6. | The rules are updated for the collection and only authorized users can do each of the given actions in Step 3. | |
| 7. | The use case ends. | |
|  |  | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. | There are no alternative course of actions | |
| **Step #** | **Exception Paths** | |
| 1. | If there is an error processing the users that are given permissions, an error is returned to the user. | |

7. OAuth adapters for Google and Facebook

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-012 |
| **Purpose** | | To provide users with a secure authentication experience by allowing them to log in using their Google or Facebook accounts. |
| **Pre-conditions** | | The user has an active Google or Facebook account. The server has registered as an OAuth client with both Google and Facebook. The user has registered to the server with their chosen OAuth client. The API server is up and running. |
| **Post-conditions** | | The user is authenticated and has an active session while making requests to the API. The API server has access to specific user information from Google, based on the granted permissions. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1.** | The user logs in and chooses either Google as the authentication method. | |
| **2.** | The user to the chosen OAuth provider's login page. | |
| **3.** | The user logs in to the OAuth provider and grants permission to the API server to access specific information. | |
| **4.** | The OAuth provider redirects back with an authorization code. | |
| **5.** | The server exchanges the authorization code for an access token. | |
| **6.** | The access token is used to retrieve user information from the OAuth provider. | |
| **7.** | The user is logged in, and a session is established. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| **1** | In step 3, the user may deny the permission to access the information for which the server should display the error and allow the user to login with another method. | |
| **Step #** | **Exception Paths** | |
| **1** | The OAuth provider returns an error during the authentication process, the server should send an error message. | |
| **2** | The server fails to retrieve user information after obtaining the access token from the Oauth provider, the server should send an error message. | |

8. Create a user with username and password through API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-013 |
| **Purpose** | | The user creates a new account through the API or UI |
| **Pre-conditions** | | The system is online, the database is available and the authentication module is operational. For the UI sub case the user is in the user collection view. |
| **Post-conditions** | | A new user account is created, the user receives confirmation through an API response or a success message on the UI. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user makes the create user API call or presses the create user button on the UI. | |
|  | The user enters the username and password while making the API call or is prompted for the username and password on the UI. | |
|  | The request is dispatched containing the user’s credentials. | |
|  | The system validates input data and the uniqueness of the username. | |
|  | A new user record is made in the database where the credentials are stored securely. | |
|  | A unique user ID is generated and made part of the new user record . | |
|  | A successful response containing the newly created user record is sent back to the user using the API or in the case of the UI a success message is displayed. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | In step 4 the system can reply with a failure response of the form “user already exists” in the case of non uniqueness of username | |
| **Step #** | **Exception Paths** | |
|  | If the system, database or the authentication module becomes unavailable or too slow to respond, the client side for the API and the browser for the UI times out and returns an error. | |

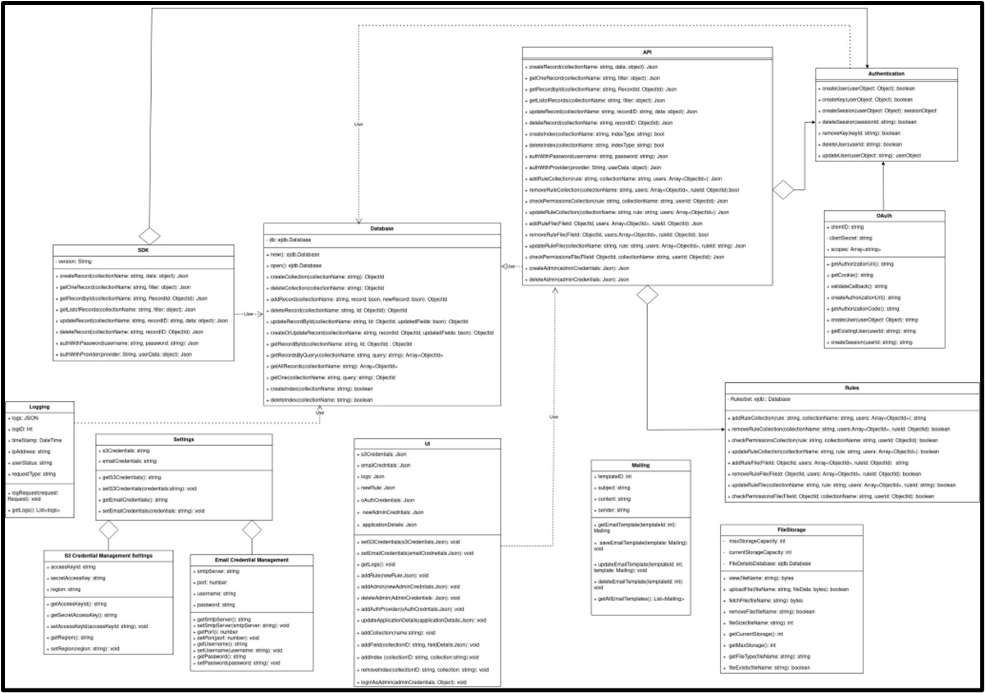
9. Start a Server to serve all the requests

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-18 |
| **Purpose** | | To initialize the server for serving requests. |
| **Pre-conditions** | | Server setup is completed successfully. |
| **Post-conditions** | | The server is running and ready to handle requests. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The Admin downloads the zipped project from the info site. | |
|  | The Admin unzips the project. | |
|  | The Admin uploads the unzipped project to a linux server. | |
|  | The Admin runs the executable with the relevant options on the server. | |
|  | The server is now ready to handle requests. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. 1 | N/A | |
| **Step #** | **Exception Paths** | |
| 1. | An error occurs in the configuration in step and the server does not start. | |

10. Upload and alter Azure Blob Storage credentials to change file upload destination

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-19 |
| **Purpose** | | To upload and modify Azure Blob Storage credentials for changing the file upload destination. |
| **Pre-conditions** | | Admin has appropriate access privileges. |
| **Post-conditions** | | Azure Blob Storage credentials are updated, affecting file upload destinations. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | Admin accesses the Azure Blob Storage credentials credential management settings. | |
|  | Admin uploads new Azure Blob Storage credentials. | |
|  | The Azure Blob Storage credentials are altered. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. 1 | N/A | |
| **Step #** | **Exception Paths** | |
| 1. | The Azure Blob Storage credentials are invalid and an error is displayed | |

## Class Diagram [Make it more visible]



**Description:**1. Database: Handles all interfacing with the underlying NeDB database. Controls all CRUD operations on collections and records, along with creation/removal of indexes.

2. File Storage: Controls all things pertaining to file uploads, including file destination, backups, removal, compression and resizing.

3. API: Authentication: Provides the authentication features for both users and admins, including password reset and confirmation.

4. OAuth: Inherits from the Authentication class, and allows usage of third party social login providers.

5. Rules: Used for access control, allows the user to define access rules on database collections that control access.

6. Mailing: Used for password reset/email confirmation and other emails.

7. Logging: Used for logging user requests, and generating intelligent insights

8. UI: Primary interface for the admin, allows DDL operations, changing of settings, monitoring and other admin actions.

9. Settings: Used to tweak project settings, such as setting log retention time, admin credentials, ports, SSL certificates and so on.

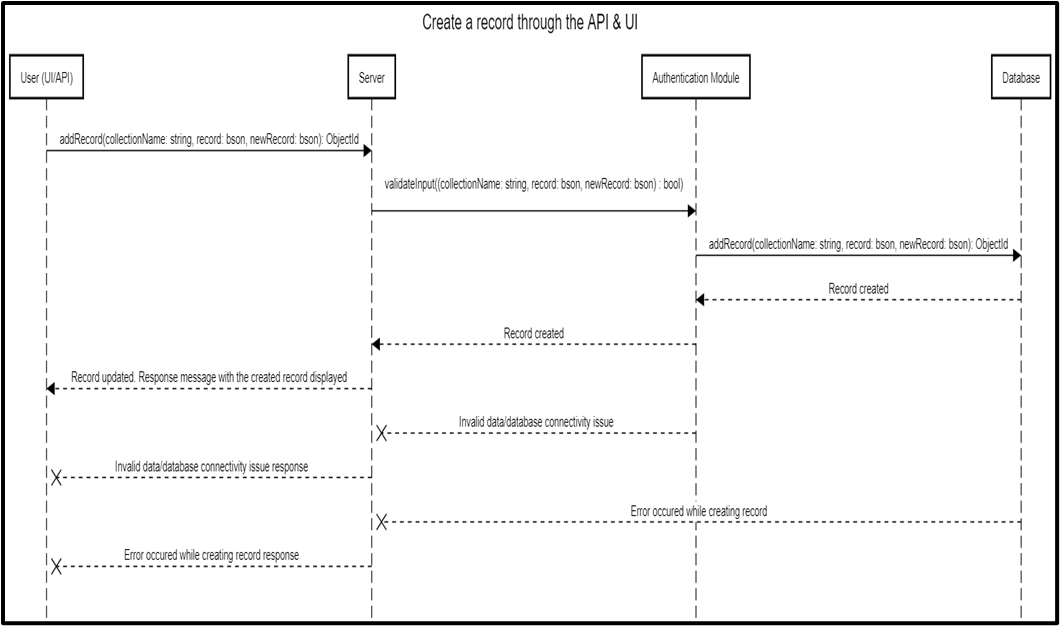
10. Azure Blob Storage Credential Management Settings: Used to change the default file upload destination to Azure Blob Storage.

11. Email Credential Management: Used to change the default SMTP credentials.

12. SDK class: Main interface for the developer, allows the developer to interact programmatically with the backend.

## Sequence Diagrams

# Create a Record through the API and UI



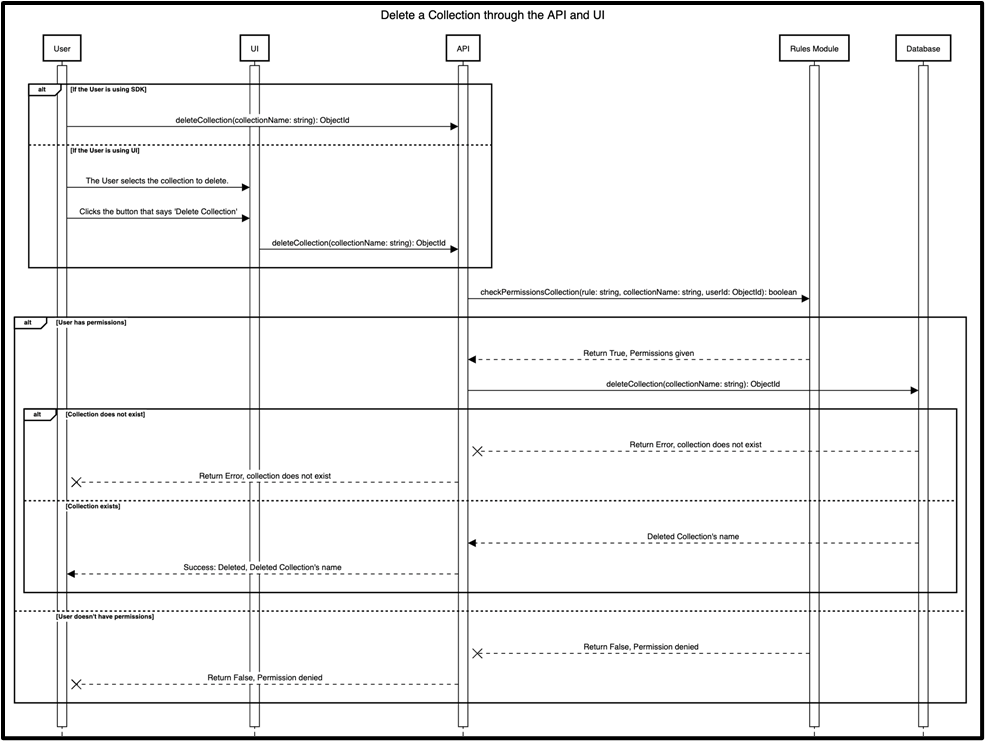
# Delete a Record through the API and UI

# 

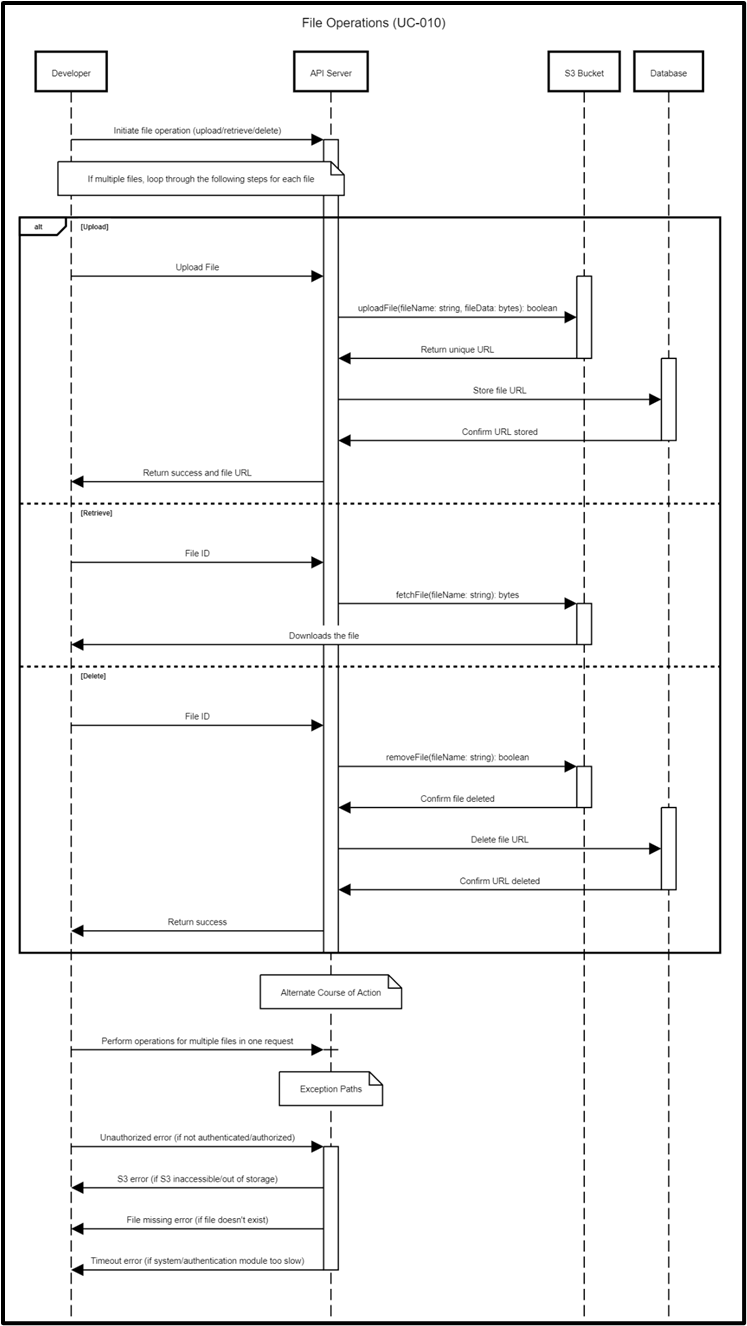
# Create a Collection through API and UI

# 

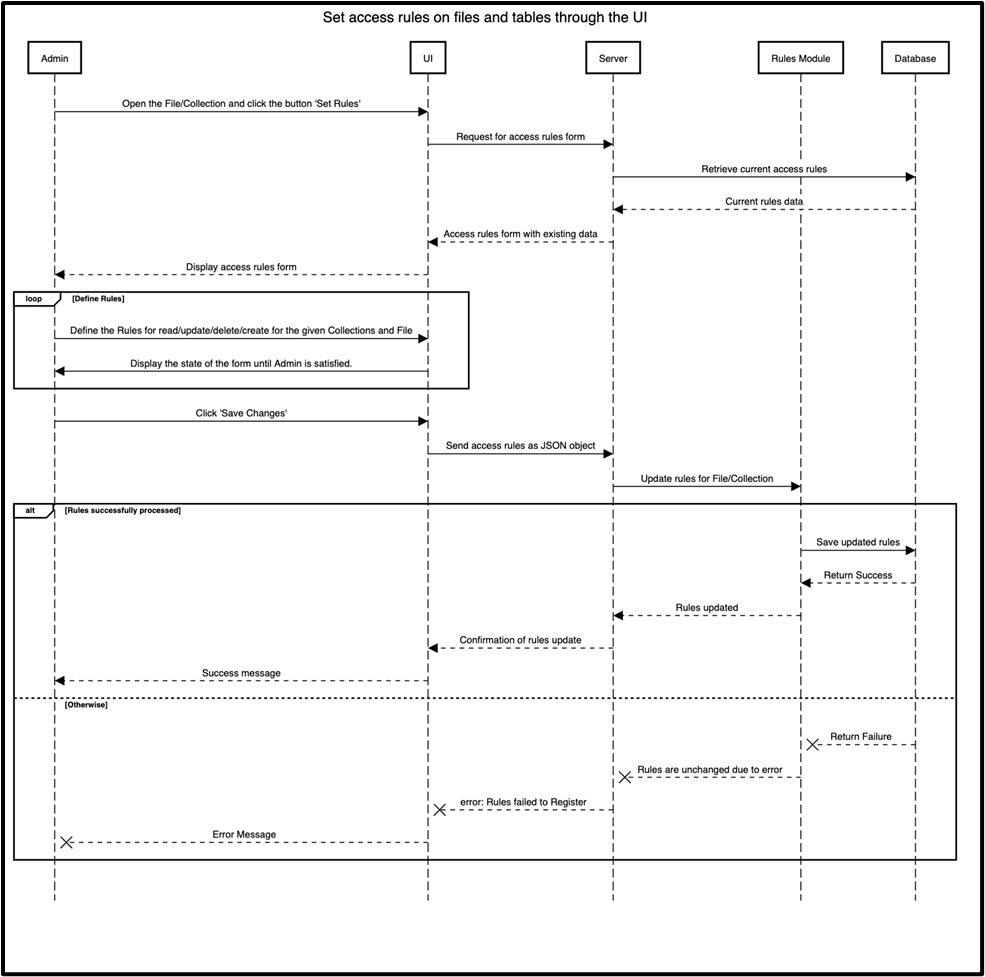
4. Delete a collection through the API and UI



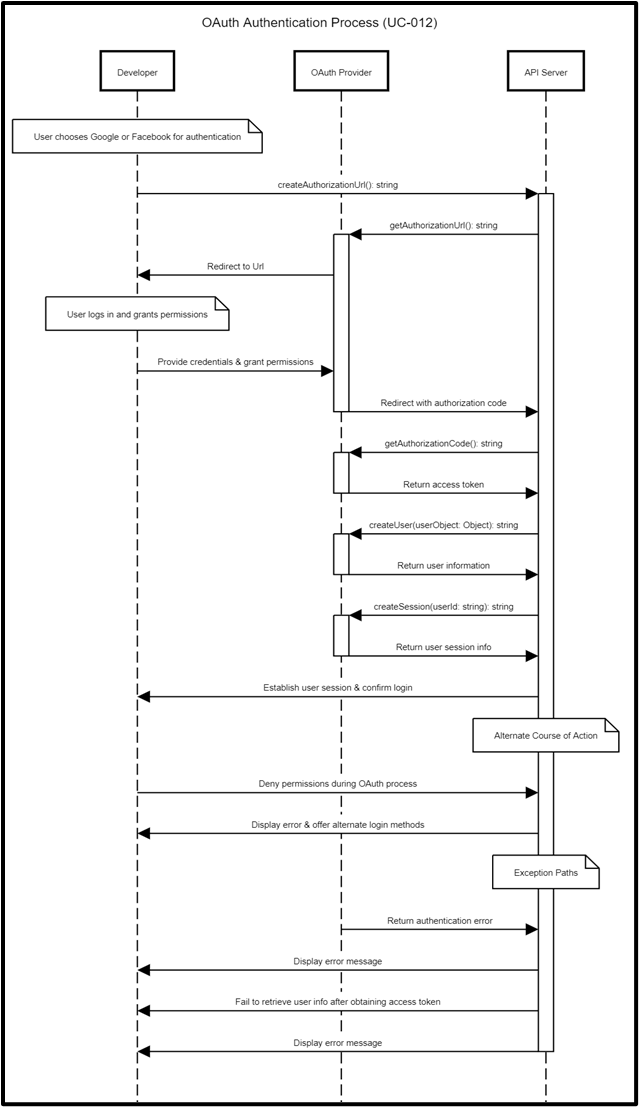
5. Store, retrieve and delete a file through the API



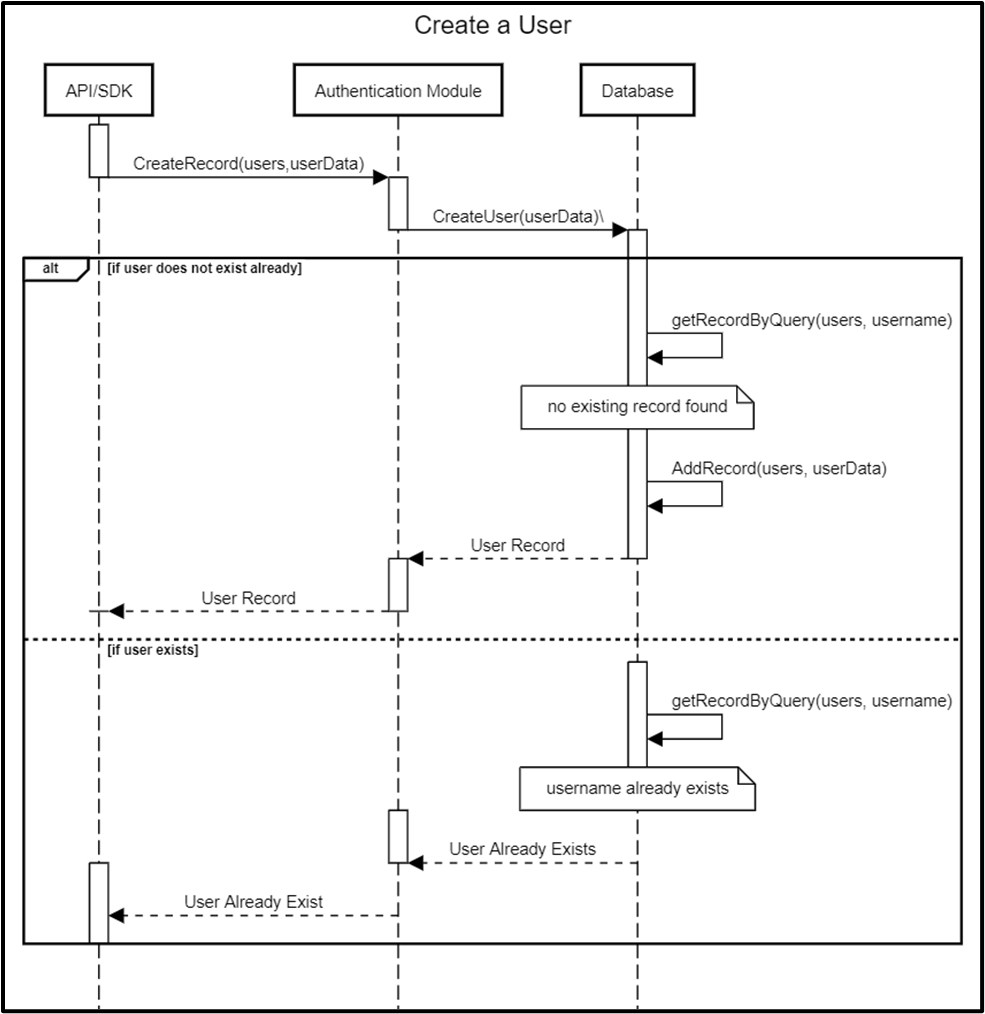
6. Set access rules on files and tables through the UI



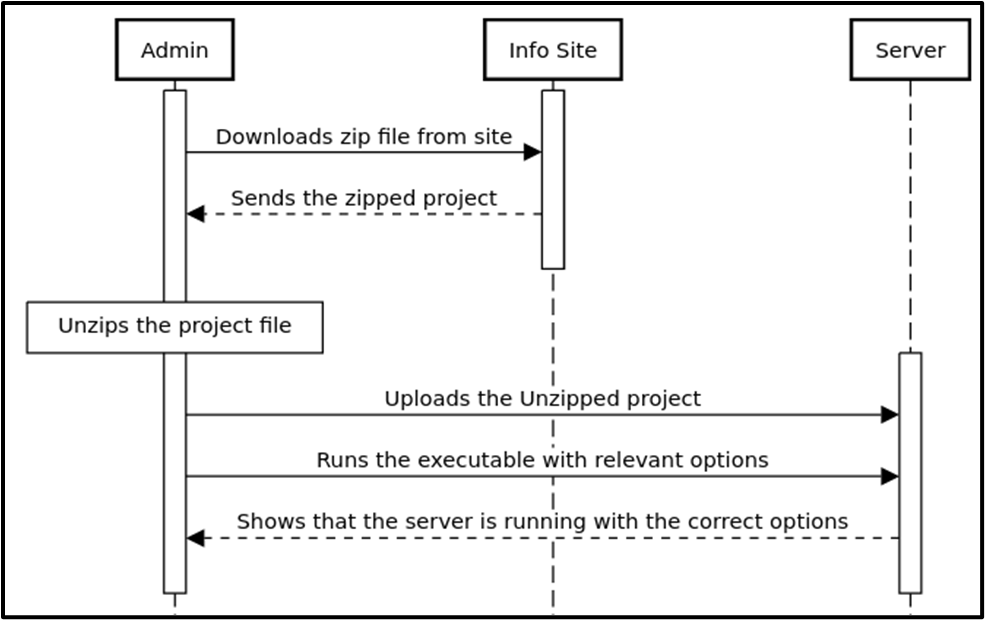
7. OAuth adapters for Google and Facebook



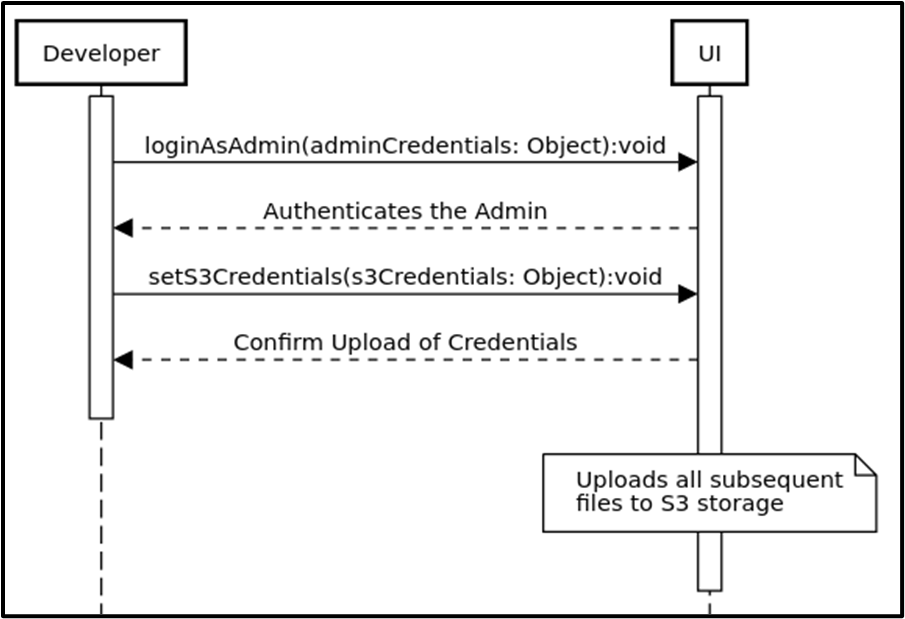
8. Create a user with username and password through API or UI



9. Start a Server to serve all the requests



10. Upload and alter Azure FIle Storage/Azure Blob Storage credentials to change file upload destination



# Software Development Methodology and Plan

## Software Process Selection

**Agile:**

**Pros:**

1) Ability to respond rapidly to changing requirements since any change of direction can be discussed in the daily scrum meetings.

2) Teams are not too tightly bound by any stage or phase requirements and can creatively solve problems.

3) Often and more direct contact with the product owner through scrum meetings allowing for concepts, ideas, requirements and feedback to be communicated between the developers and the customer more effectively.

4) A potentially smaller time taken to get a minimum viable product to market owing to the principle of working software being the primary measure of progress.

5) Improved project management since development is divided into smaller goals rather than being monolithic.

6) Agile promotes cross-functional teams that include developers, designers, and stakeholders. Regular meetings and open communication foster teamwork.

**Cons:**

1) For very large projects that may be mission critical and have rigid requirements, agile may not be the best methodology since cross team coordination and the scaling up of agile practices can be challenging.

2) Additional features and requirements may be continually added to the project, potentially impacting timelines and budgets.

3) Working software is one of the principles of agile, however, for some organizations, it can be a drawback in situations where extensive documentation is necessary for compliance or regulatory reasons.

4) A highly involved customer may end up slowing down the development process if they are not very sure of their own needs or expectations.

5) The fast-paced, iterative nature of Agile can lead to burnout if not managed properly.

**Waterfall:**

**Pros:**

1) A heavy focus in waterfall is on comprehensive documentation, this can lead to a solid understanding of the project's scope and objectives, which allows for minimal scope creep and a predictable project timeline.

2) The Waterfall model has well defined milestones and phases, making it easier to assess progress at different stages of the project.

3) Software releases are less frequent and thus each release can be more carefully vetted and of a higher quality compared to the fast paced releases of agile.

4) A good amount of time being spent in the earlier stages like the requirements stage allows for early detection of potential issues.

5) Mission critical systems greatly benefit from a waterfall development methodology owing to comprehensive documentation and the stability of development.

**Cons:**

1) Owing to its rigidity, accommodating changes once a project has started may prove to be a time consuming and difficult process.

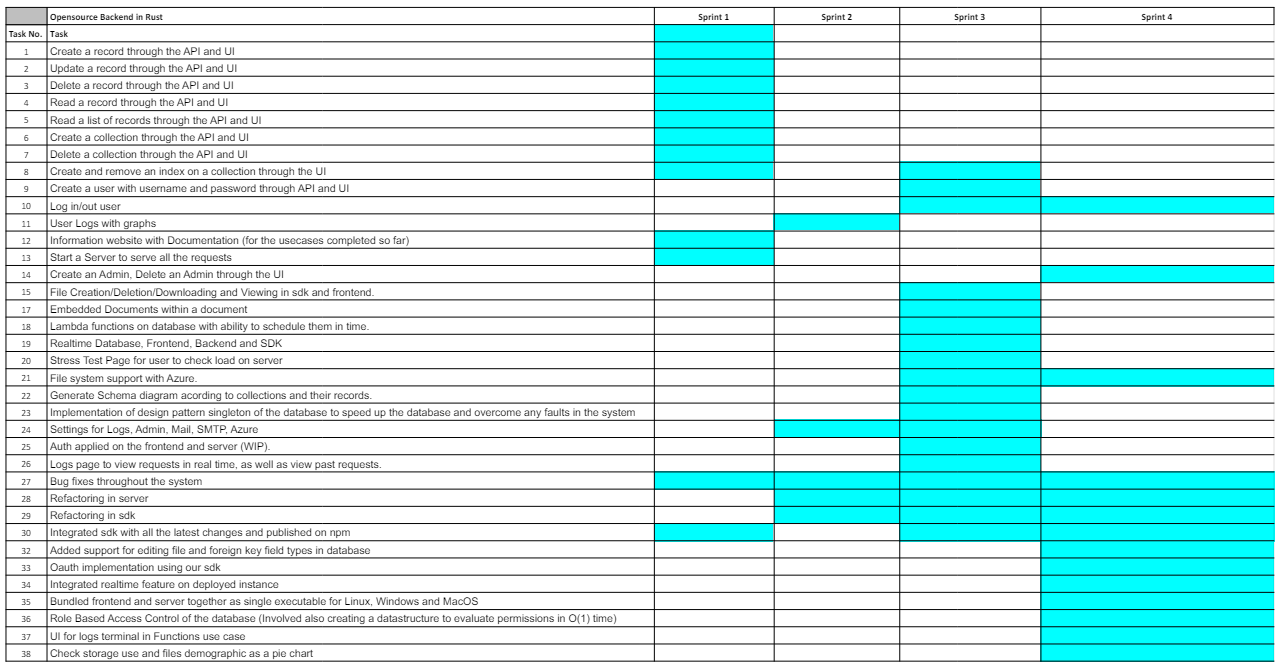
2) The entire project must be completed before any deliverables are ready for testing or client review resulting in a potentially slower time to market.

3) Producing extensive documentation can be very time consuming and costly, especially for small projects where documentation may outweigh the development effort.

4) The earlier stages of a waterfall project are immensely crucial, any major mistakes or oversights during these stages can lead to critical issues down the line.

We adopted an Agile development methodology in our project. From a project context analysis point of view, our potential loss due to defects/bugs is low to medium since our system is not mission critical and bugs may not present too big a loss to users. The developer’s skill levels range from medium to high, the rate of requirement of change is high, the team size is low, the team is highly adaptive to change, there is high pressure to develop prototypes and early releases, business staff’s commitment to work extensively with development team is high, developer’s experience with similar systems is medium and the availability of reusable components is high. All these aforementioned points lead us to follow Agile principles since they fulfill our and the project's requirements most completely.

## Gantt Chart



**Milestones**

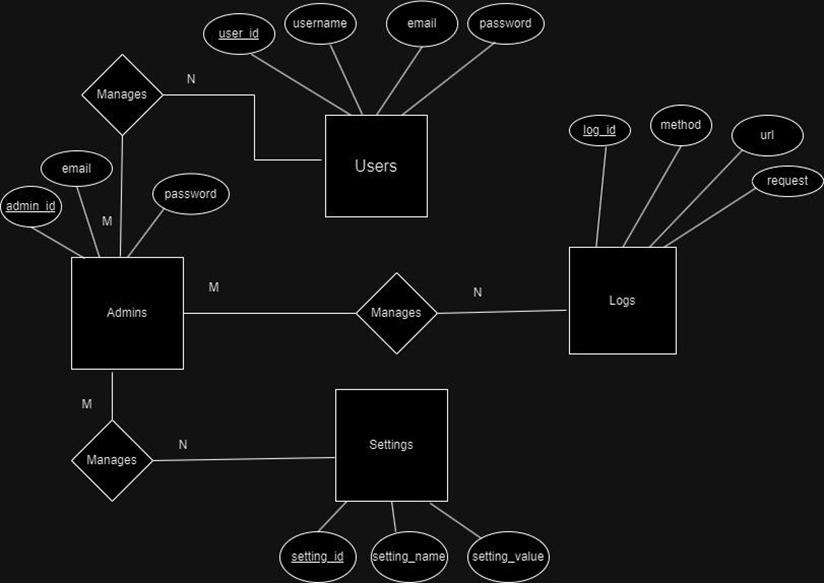
* Completion of screens (Week 1)
* Completion of CRUD (Week 4)
* Completion of Authentication (Week 7)
* Completion of Rules (Week 10)
* Completion of File Storage (Week 9)
* Completion of UI for all required use cases (Week 10)
* Completion of SDK for all required use cases (Week 12)
* Completion of Documentation and Info Site (Week 11)
* Working Executable produced of the typescript server (Week 12)

# Database Design and Web Services

Brief introduction of this chapter in a paragraph

## Database Design [Picture should be in white background]

The data model has four entities Admins,Users, Logs and Settings. Additional tables are made by the user and it depends how each developer makes the data model. Admin manages all the models in the server, performs crud operations and changes permissions. Settings model is used to keep track of the general settings set by admin for example SMTP email address and Azure Blob Storage configuration. Logs keep track of all API requests made to the server. Users table is used to authenticate every user interacting or making requests to the server.



## API Specification

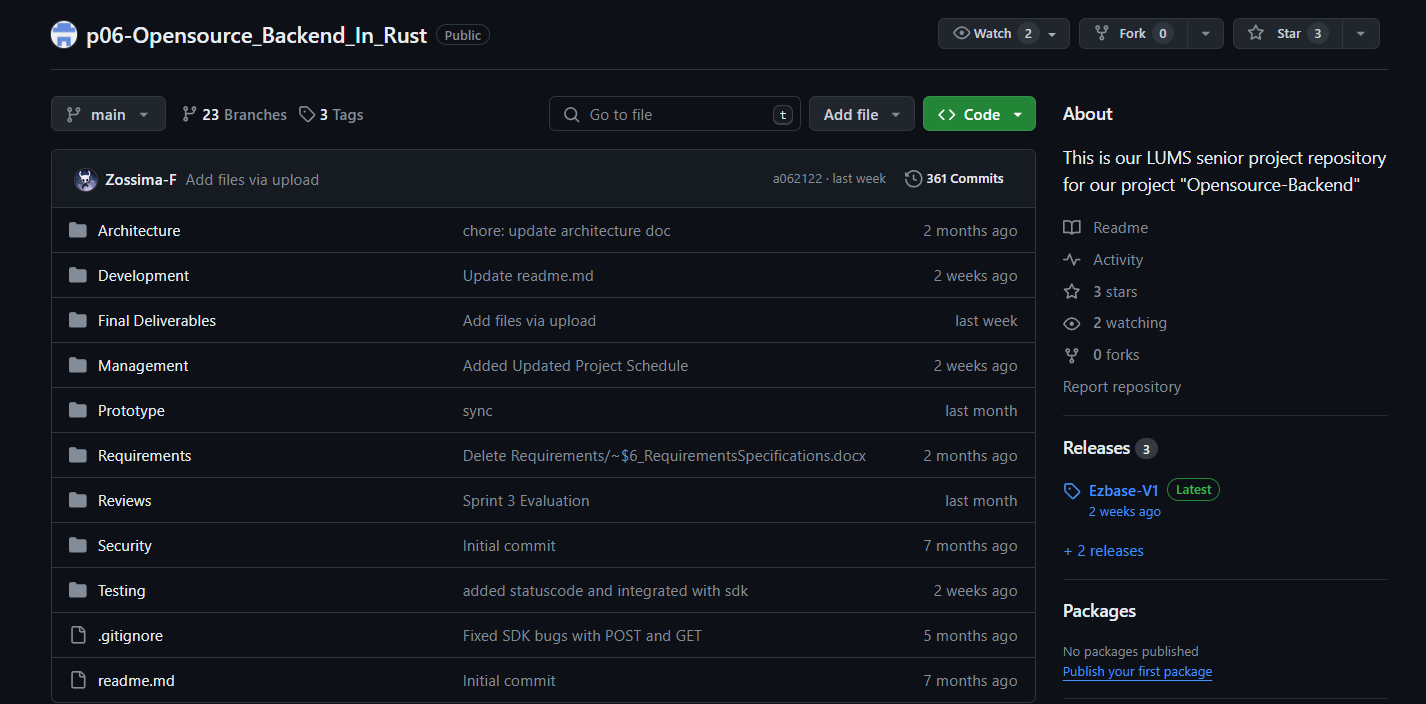
This external API used in this project is Google’s Oauth provider API.

# System User Interface

This section explains the System User Interface and the functionality of our application to our end users.

1. **Project Executable:**

The executables for the project are available in the ‘Releases’ section of the Github repository. The details for deployment are available in the ‘Deployment and Guidelines’ section.



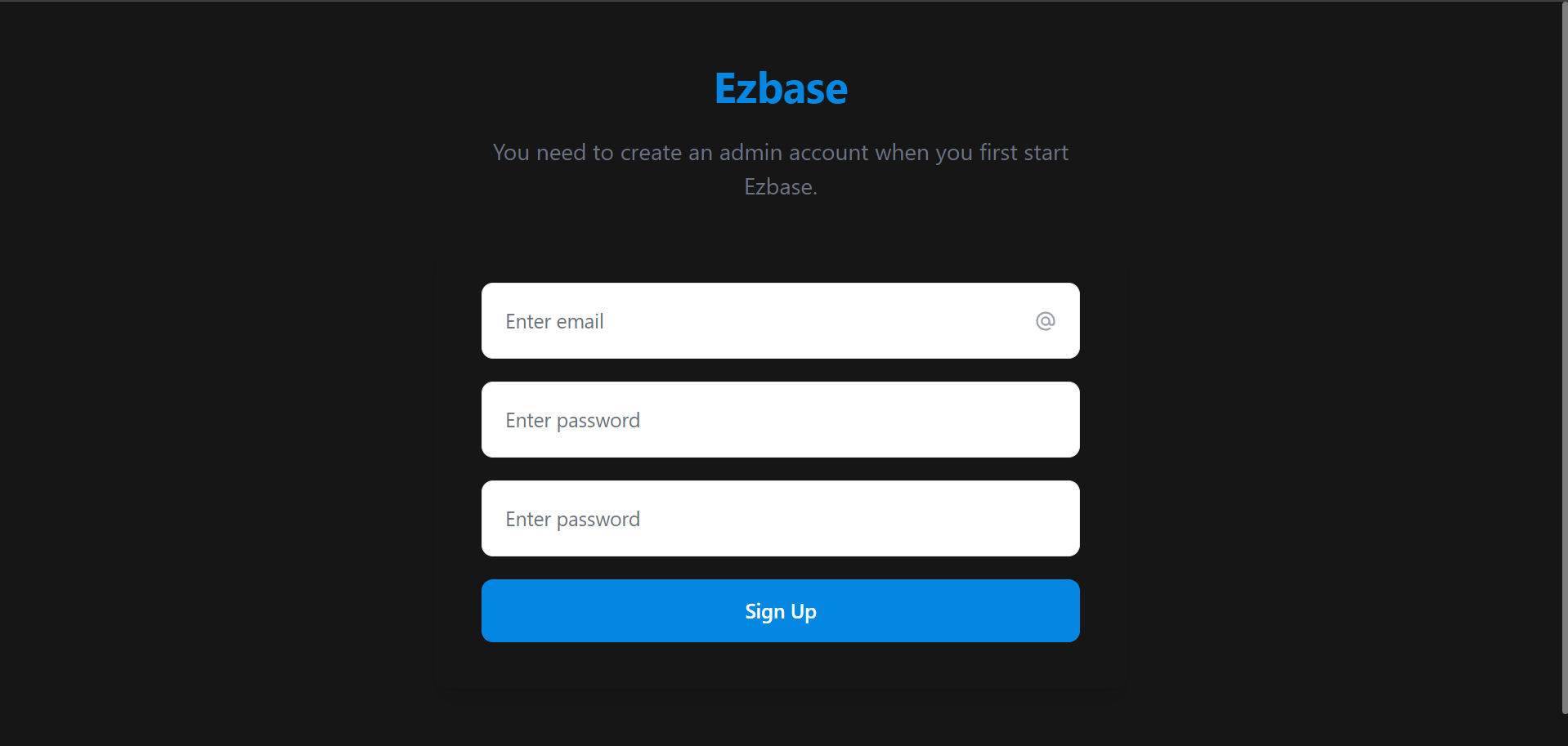
1. **Server Instance:**

After the EZBASE executable is run, the user will be able to see the server instance running on their machine along with the link for the Admin UI.

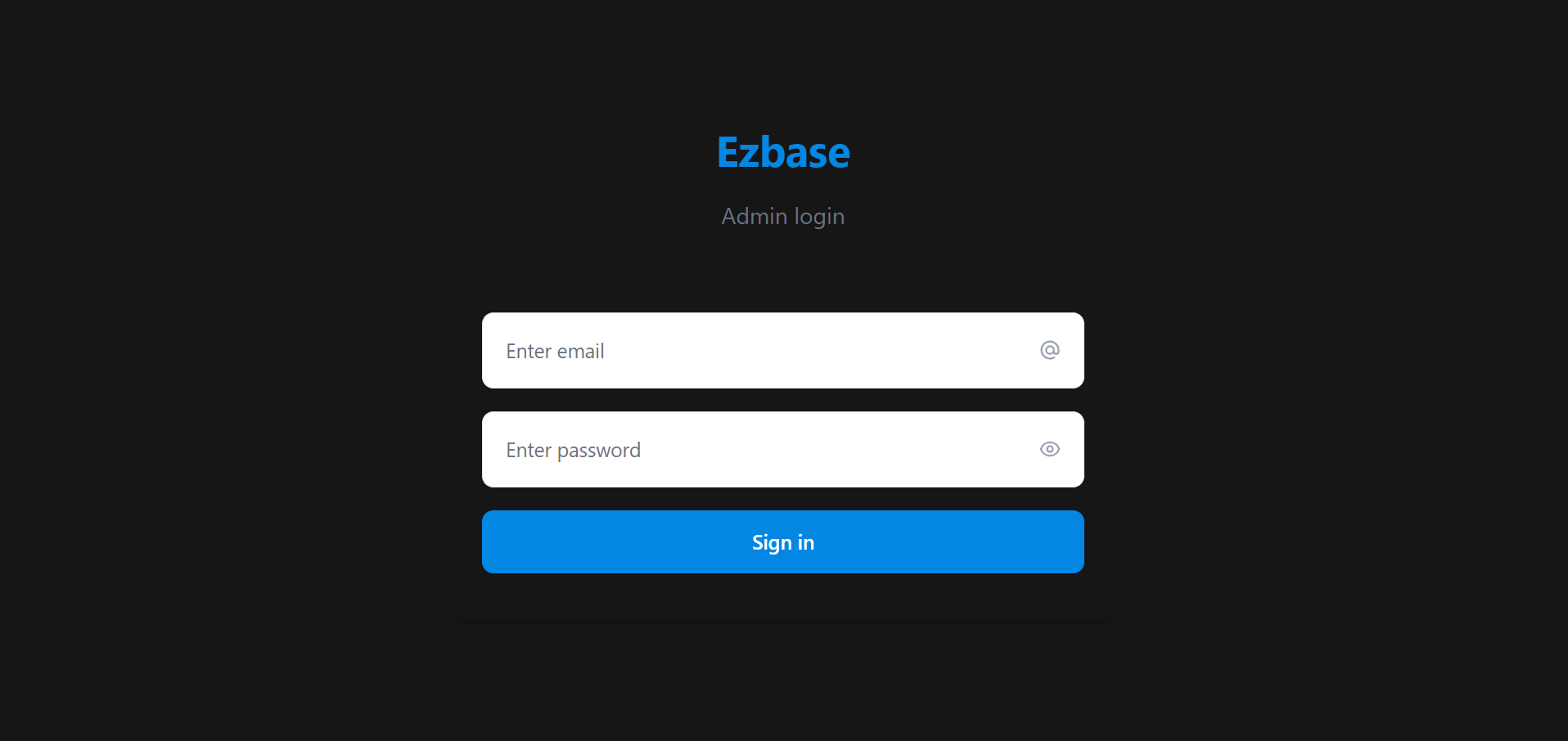


1. **Signup/Login for Admin UI:**

On running the Admin UI, the user will be taken to the signup page of EZBASE:

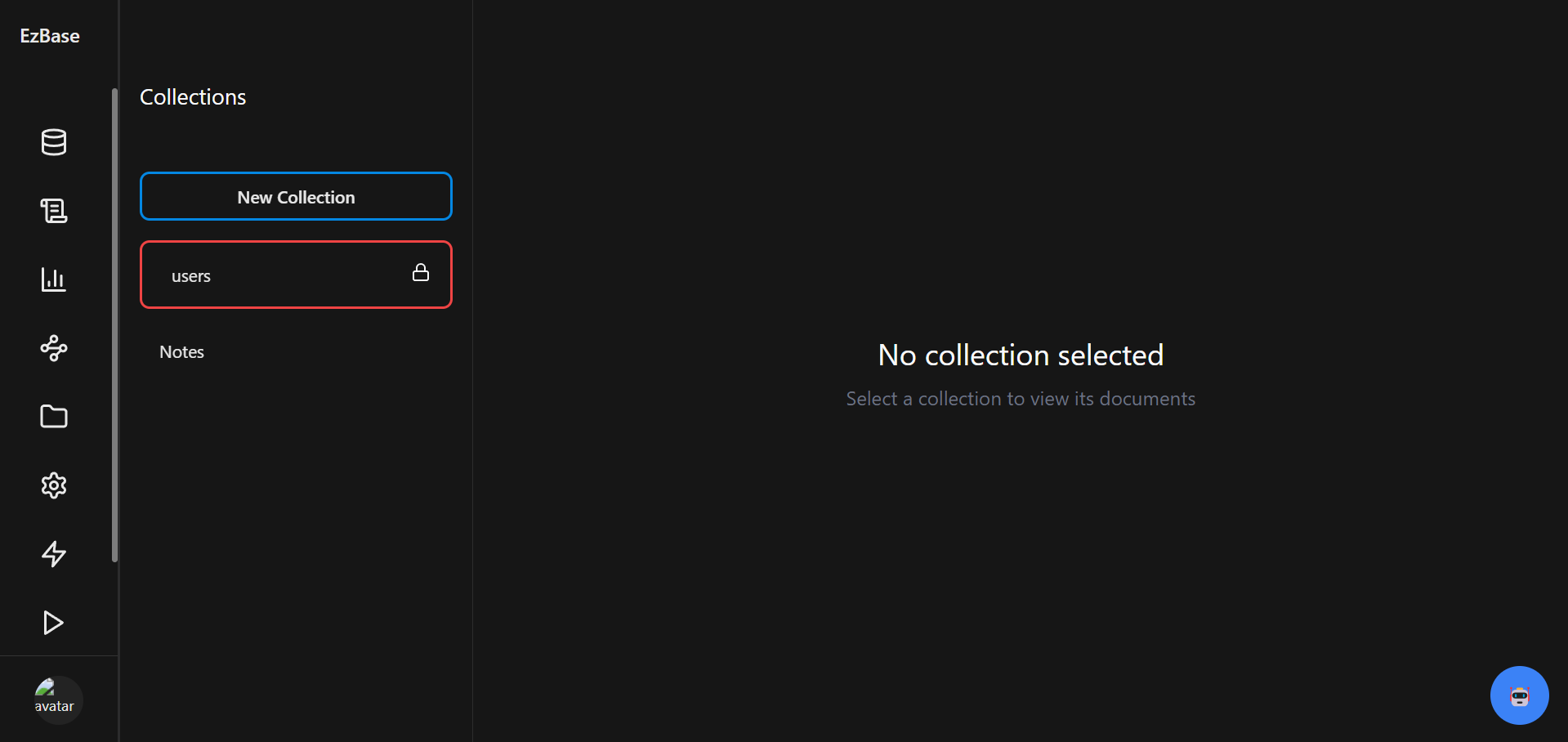


After signing up, the user will sign in to their admin account:



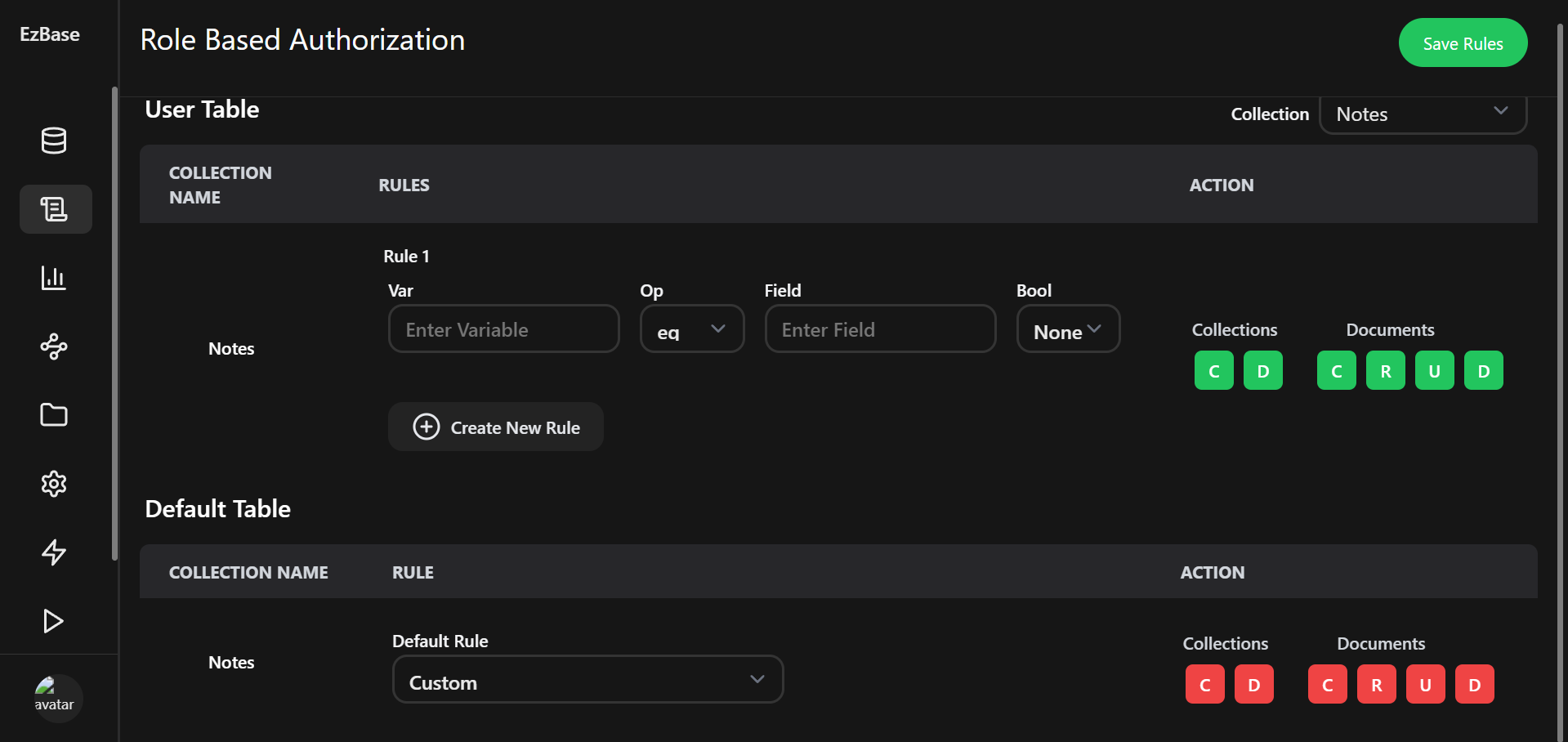
1. **Admin UI Dashboard and Collections Page:**

After the successful login, the user will be taken to the Admin UI Dashboard. The first screen is the collections page where the user perform CRUD operations for collections and their respective records directly. The user will be make the desired changes to their collections except for the user collection which is predefined.



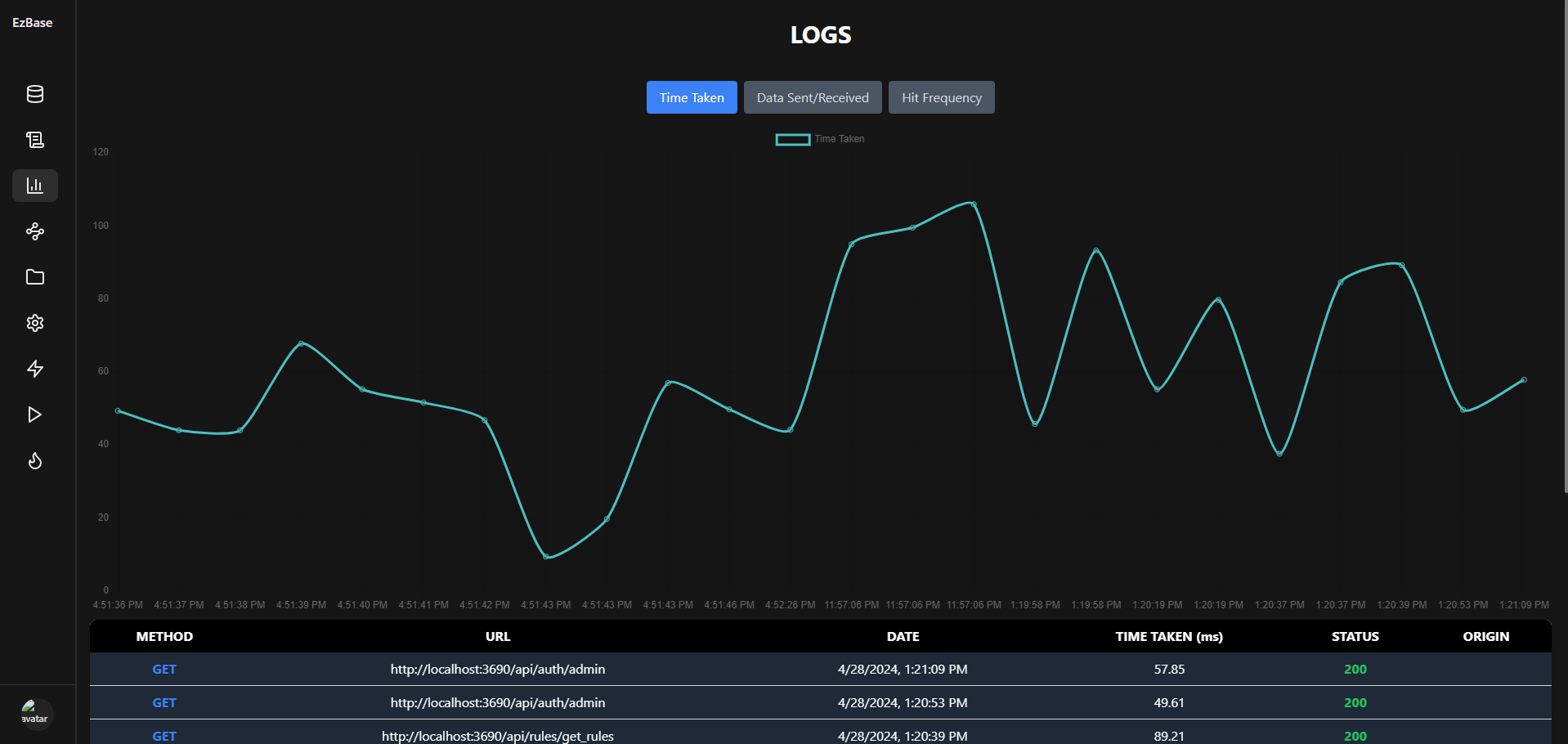
1. **Role Based Authorization:**

The users are able to add rules to User Tables and Default Table for creating and deleting collections and CRUD operations on Documents.



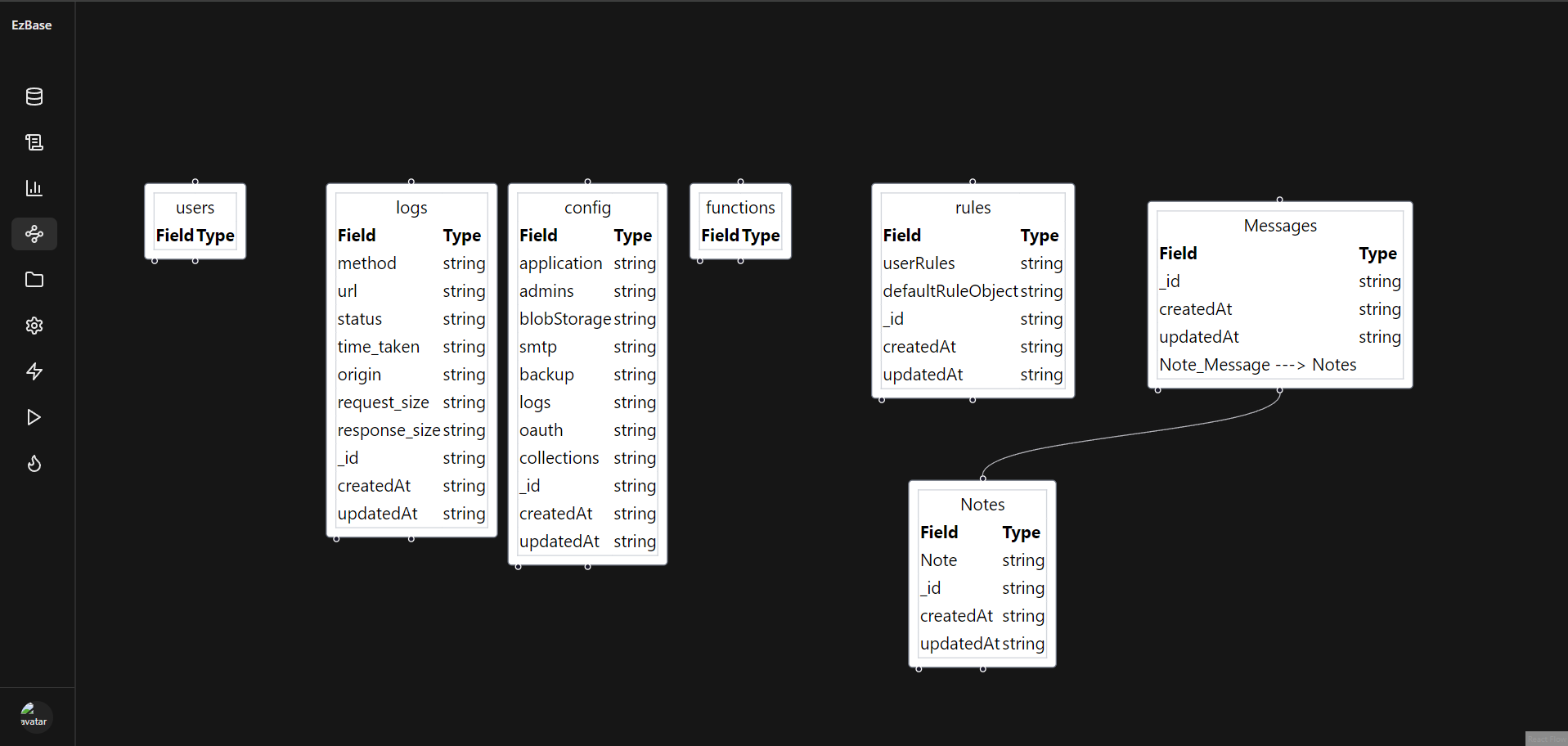
1. **Log Metrics:**

The user can view all the request that they have sent to the server and the respective metrics i.e., Time taken for the requests, Data sent and received, and Hit frequency for the requests.



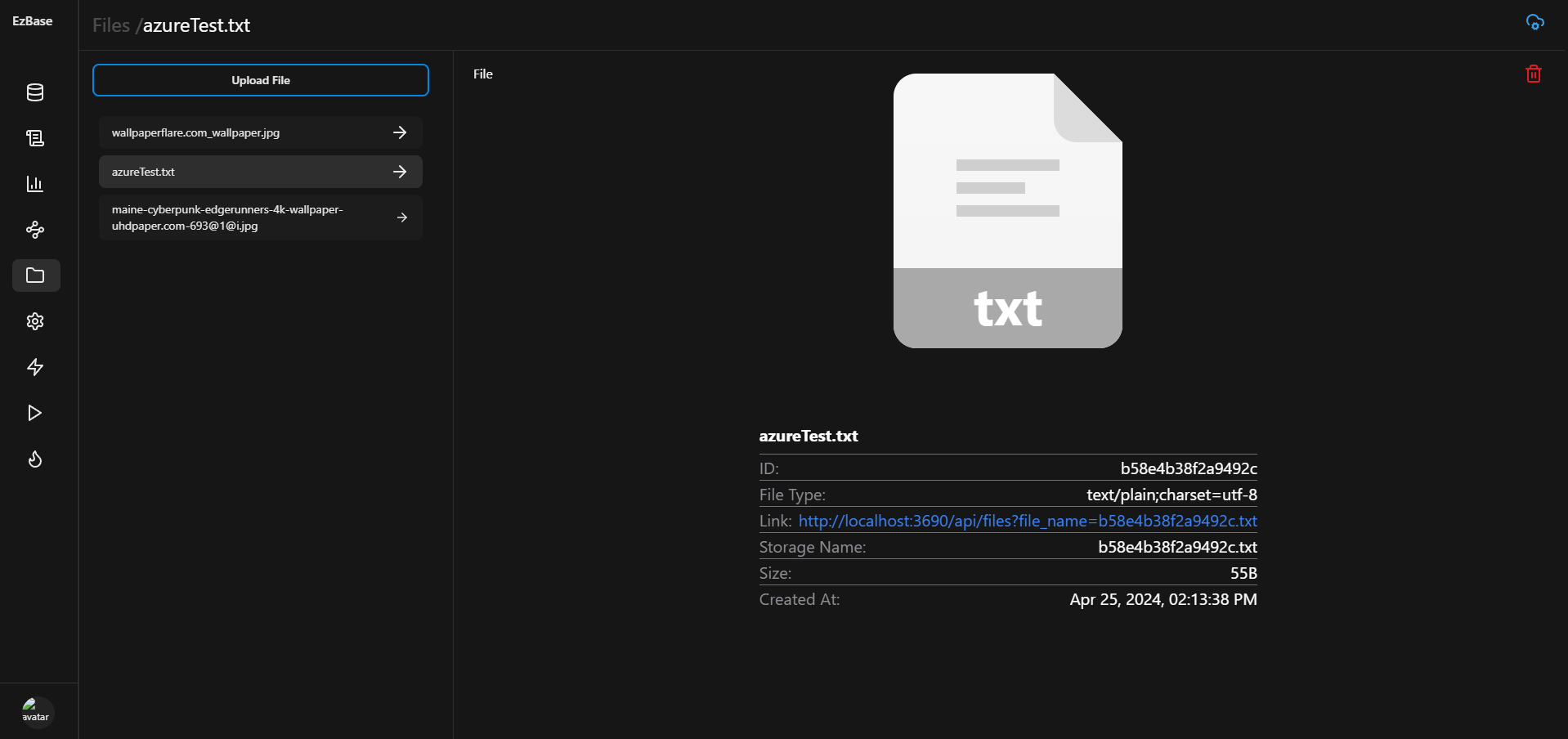
1. **Collections Schema:**

Although we are following a non-relational database for the server, we have added the feature for the user to be able to view the current schema of all their collections and any connections between collections based on foreign keys.



1. **File Storage:**

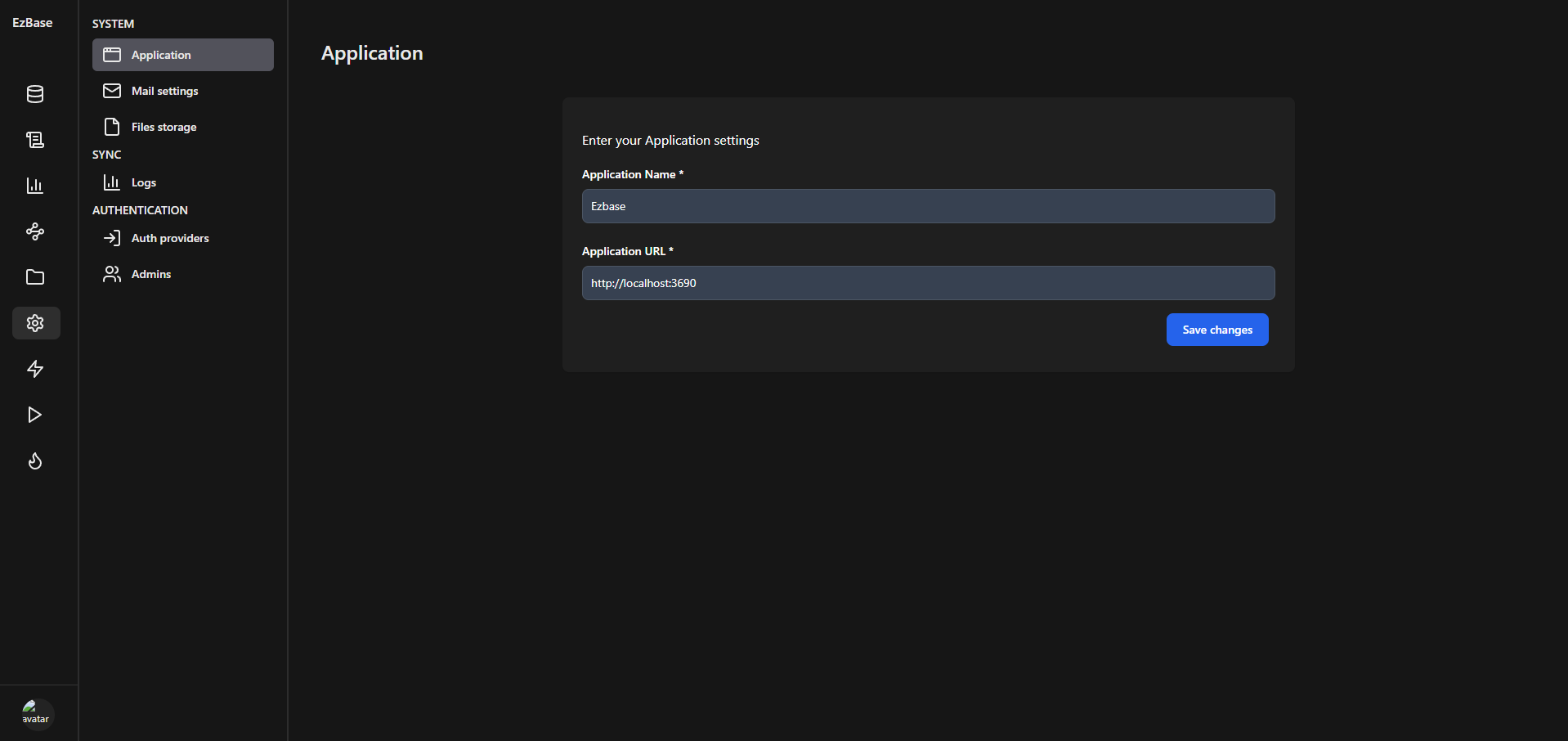
The user is able to add files to either local storage or Azure Blob Storage after uploading the appropriate settings. After a file has been uploaded, the user can view the relevant metadata for it and can delete it. Furthermore, they can view the space occupied and distribution by file type by the icon in the top right corner.



1. **Application Settings:**

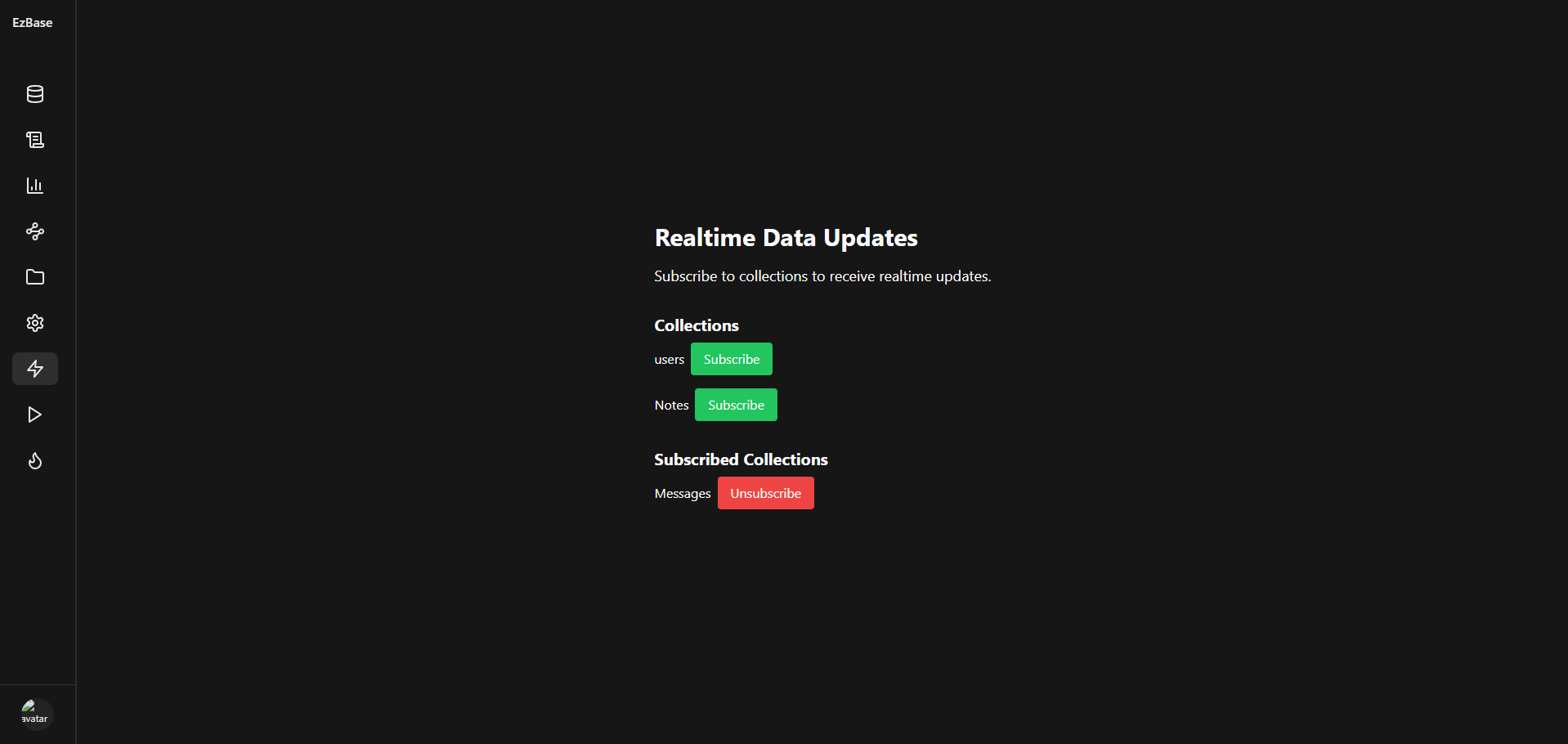
In this section, the user can add/change the settings for their:

* Application
* SMTP Mail Service
* Azure File Storage
* Logs
* Authentication Providers
* Admins



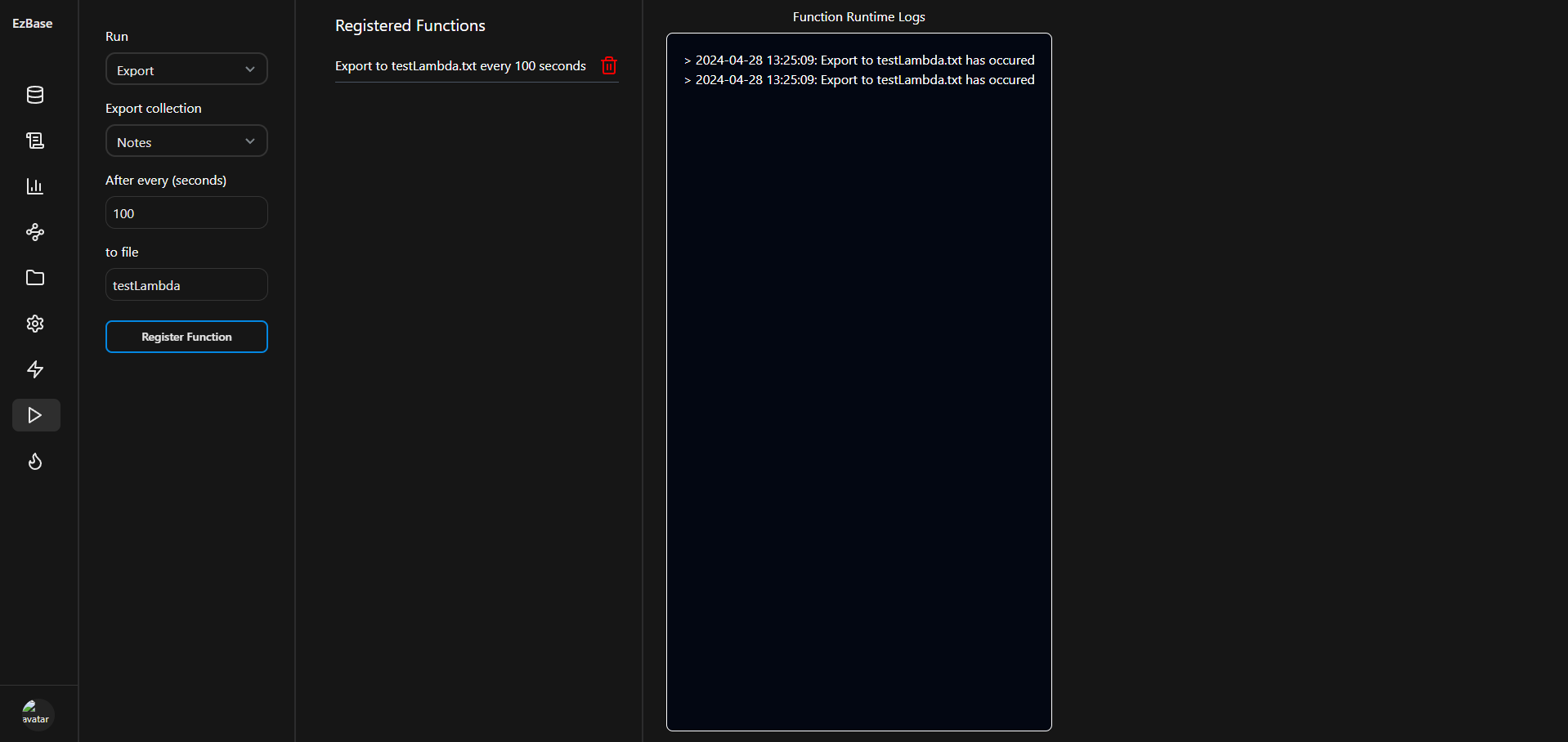
1. **Real Time Updates:**

The user can subscribe/unsubscribe from a collection for real time updates.



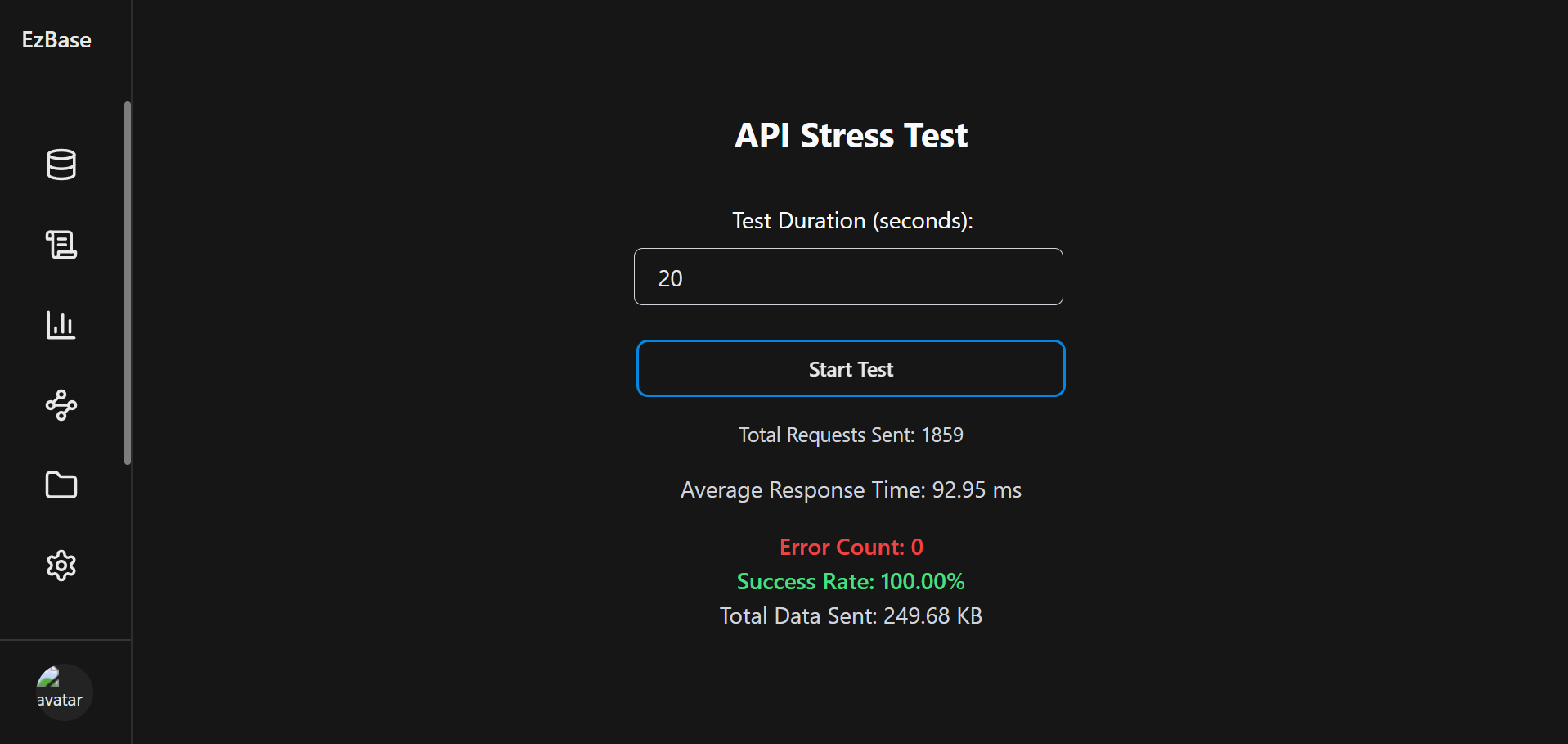
1. **Registered Functions:**

The user can specify a lambda function on a collection to periodically send data or server a request.



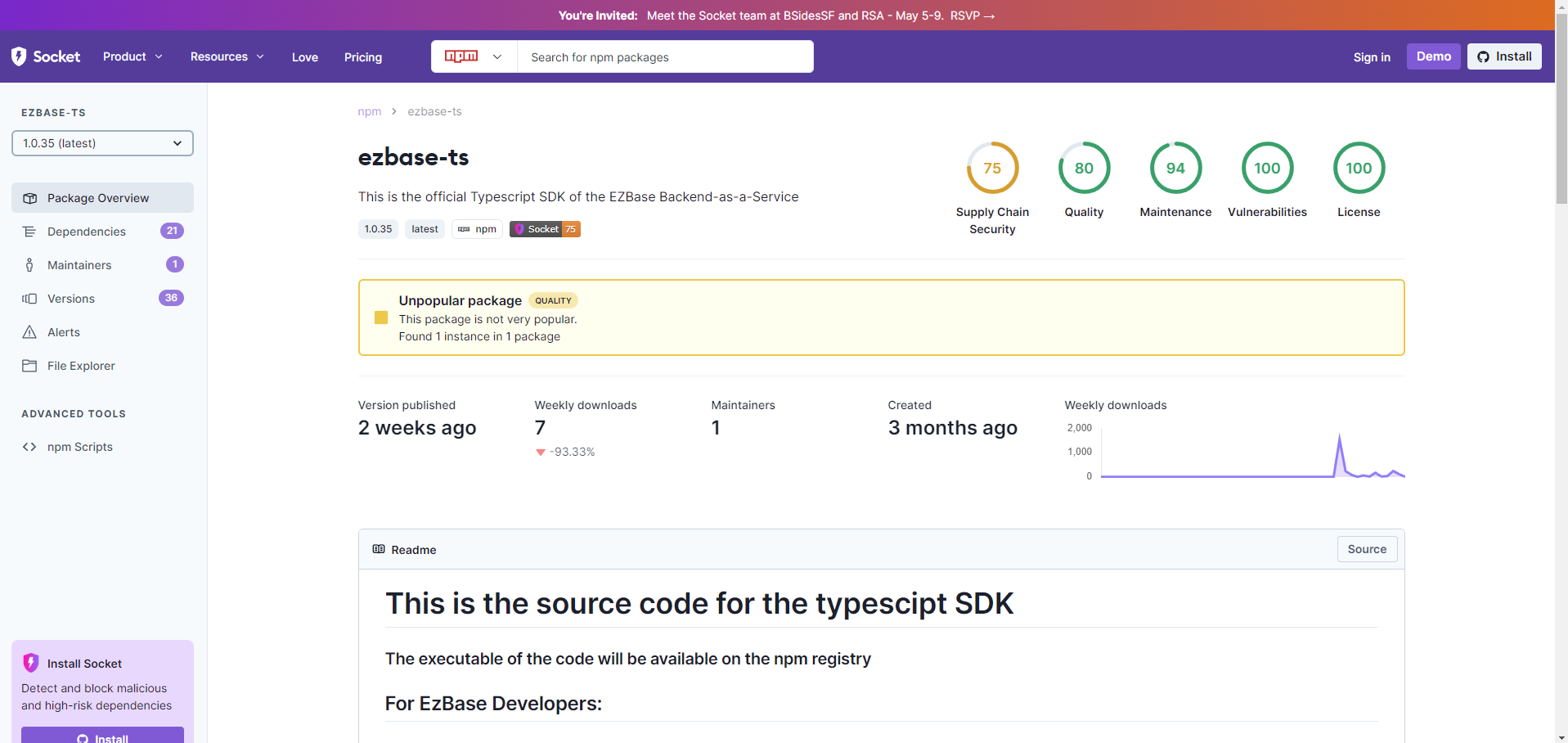
1. **API Stress Test:**

The user can check the availability and efficiency of the server in serving requests for a stipulated amount of time.



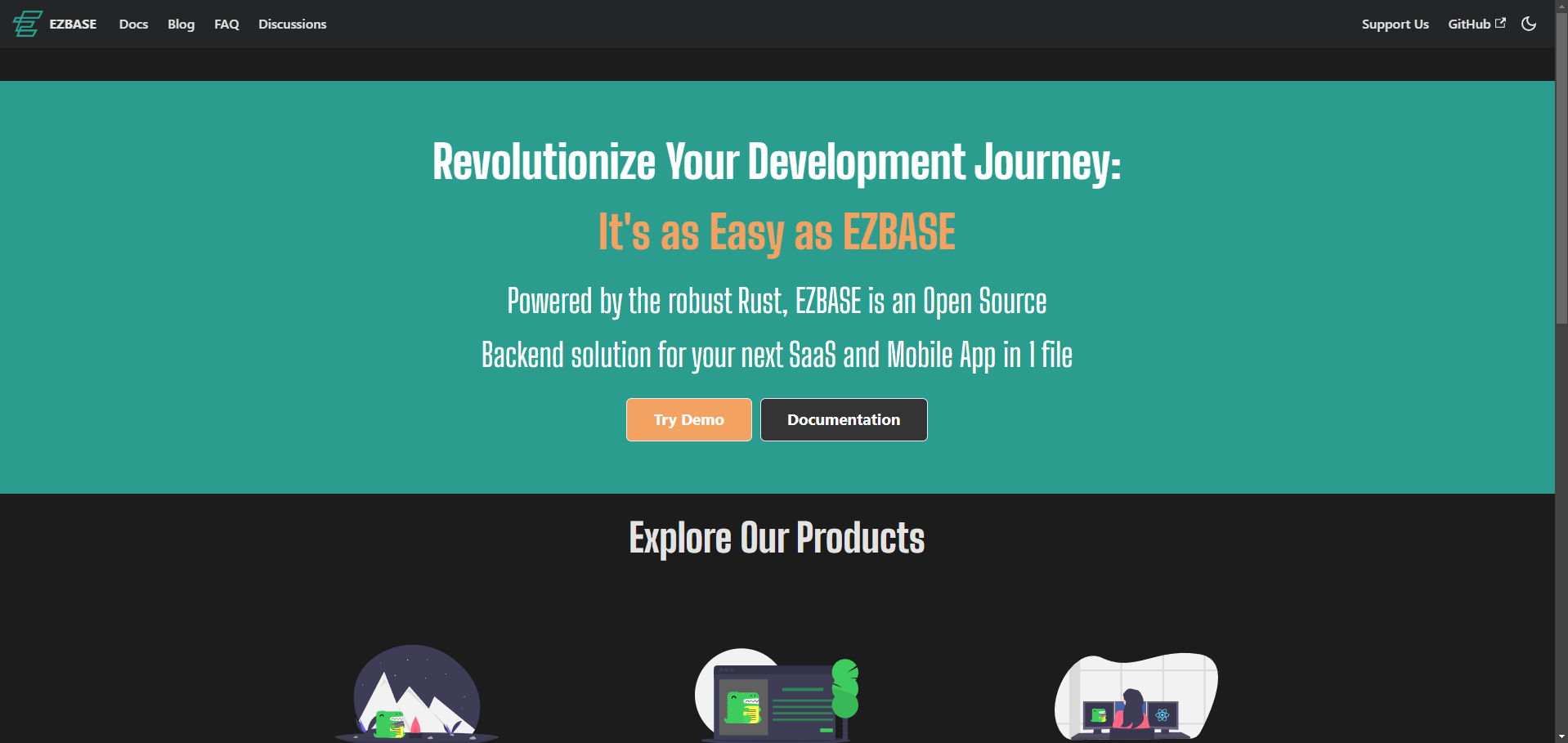
1. **NPM SDK Package;\**

The SDK for our project is published at NPM (details in Deployment and Guidelines section).



1. **Documentation Website:**

For details on the initialization, demo, and details on the usage of various functionalities, the user can access our documentation website.

****

# Project Security

Security is a critical part of any software. For projects like Ezbase, it is even more important as users rely on it to create their own applications.

## Project Threats

The biggest threat is an attacker gaining access to stored data in a deployed EZBASE server instance. Since this data is in the form of json it can be easily accessible, therefore we would recommend that users password lock the data files for additional security.

Other threats include but are not limited to: jwt hijacking (attackers could compromise a user’s session if they manage to steal their jwt, for applications built using EZBASE we would suggest password entry for critical actions just in case), byzantine functions (if an attacker gains access to the functions data file they could potentially run functions that are detrimental to the system), stress test DDOS (a hostile admin could run stress tests on the EZBASE server that effectively DDOS user applications) etc.

## Potential Losses

Data access potential losses: data leakage, user betrayal of trust, economic losses, financial losses, data held hostage, applications held hostage, data deletion.

Jwt hijacking potential losses: data leakage, user betrayal of trust, economic losses, financial losses, data held hostage, applications held hostage, data deletion.

Hostile functions: data leakage, user betrayal of trust, economic losses, financial losses, data held hostage, applications held hostage, data deletion.

DDOS attacks: economic losses, financial losses, data held hostage, user erosion of trust.

## Security Controls

Identify the **controls** (e.g., input validation, audit logs, multi-factor authentication, user roles etc.) that should be implemented in your system in order to address the above threats. Moreover, categorize the identified controls into one or more of the following: ***detective, protective, responsive, recovery***.

Protective:

* Data file password protection.
* Frequent jwt refresh to invalidate older potentially hijacked sessions.
* Backup/Export data frequently to minimize damage from data loss.

Detective:

* Checking logs frequently and looking for suspicious activity.
* Frequently validating admin's identity and credentials.
* Making sure functions cannot be created that could be potentially hostile through input validation in the admin ui as well as protection of the functions data file.

Responsive:

* Generating new user auth key to invalidate all sessions if some have been hijacked.
* Deleting hostile functions.
* Delete hostile admins.

Recovery:

* Restart server with stress test feature turned off to prevent that DDOS attack.
* Use backups and logs to recover any lost data.

## Static and Dynamic Security Scanning Tools

Static security scanning tools:

1. ESLint with TypeScript Plugin: ESLint can be configured with plugins to enforce TypeScript-specific rules and catch potential security issues through static analysis and can identify common security pitfalls like unsafe type coercions, insecure patterns, and potential vulnerabilities.
2. SonarQube: supports TypeScript and offers a good set of security rules to detect vulnerabilities, code smells, and bugs.

Dynamic security scanning tools:

1. OWASP ZAP: is suitable for dynamic security testing of TypeScript web apps. It can intercept requests, manipulate parameters, and identify vulnerabilities like XSS, as well as SQL injection.
2. Burp Suite: is a powerful web app security testing tool that includes features for scanning, intercepting, and modifying HTTP requests to uncover vulnerabilities. It supports Typescript apps.

Additionally tools like npm audit can be utilized to check for security vulnerabilities in the dependencies of EZBASE.

# Risk Management

## Potential Risks and Mitigation Strategies

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Risk Description** | **Mitigation Strategy** |
| 1. | Unauthenticated users might try to access the api endpoints. | Usee industry standard Auth protocols like OAuth to ensure a robust authentication mechanism. |
| 2. | Access to sensitive user data during data leaks. | Secure storage through hashing of highly sensitive data to mitigate damage from leaks. |
| 3. | 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 incorrectly configure 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. |

# Testing and Evaluation

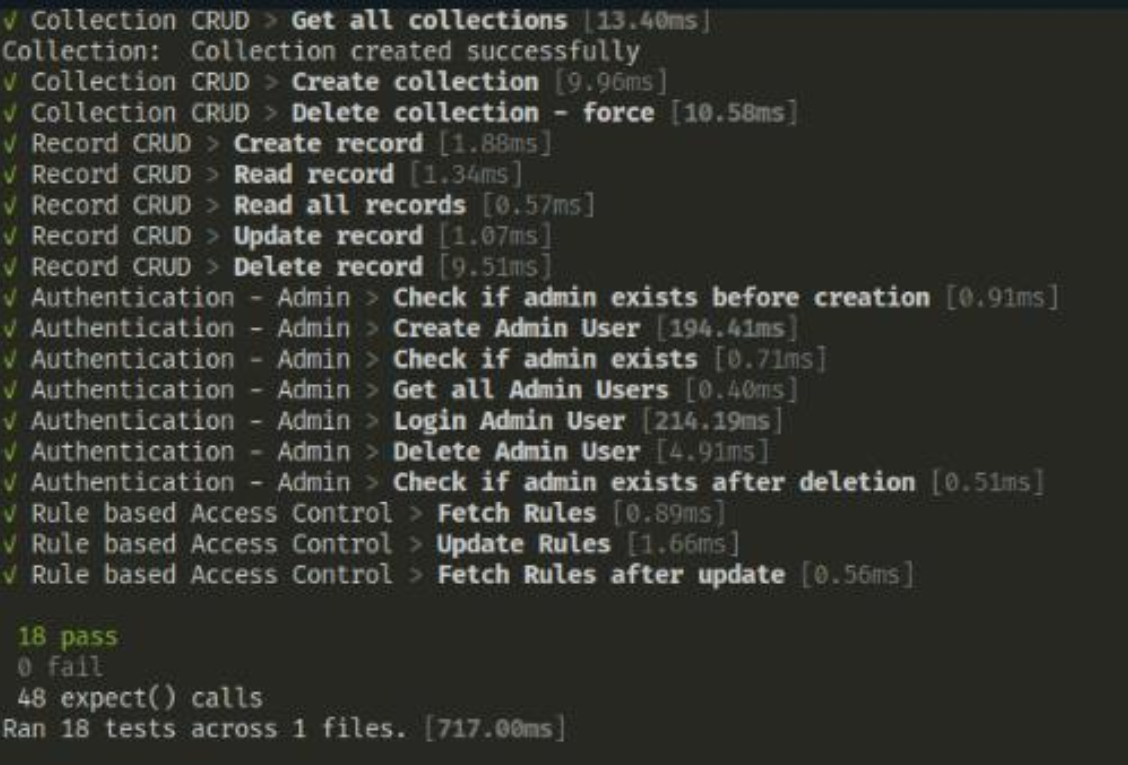
We tested our system the way it is meant to be used by building applications. The two applications we built were a Notes app and a Chat app. These applications used our client-side SDK, server, and Admin UI to manage the application.

* The Notes app tested our system’s capability of building any general CRUD application, along with Oauth integration and file handling. In the Notes app, user has two ways to sign up, using our built-in auth or using Google’s OAuth. Each user has their own avatar which they can update. This is possible through a way our system allows any file to be linked to a record and then can be fetched with the record.. All notes are fetched in descending order, newest first, through the SDK and server. This tests the different ways a developer can display the data using our system.
* The chat app tests the real-time database feature of our system. There are two collections: chats and messages. Each message is filtered based on which chat it belongs to and sent to those users which are part of that chat. The user can participate in different chats and receive real-time messages and notifications. This tests how our server and sdk fetch and filter data in realtime. Both applications require no additional third-party library.
* In addition, the server was also tested through unit tests. Some of these tests and the full test report is shown below. The automation tool used was Bun’s built-in tester, similar to Jest.









# Deployment Guidelines

As our project is open-source, the code is available at the following github [link](https://github.com/ahmediq-git/p06-Opensource_Backend_In_Rust).

For the deployment of our project in a production environment, follow the following steps:

* Go to the ‘Releases’ section of the aforementioned Github repository link or follow this [link](https://github.com/ahmediq-git/p06-Opensource_Backend_In_Rust/releases).
* In the section are the downloadable executables for our project Linux, Mac OS, and Windows, along with the source code files.
* Download the relevant executable according to your OS.
* Extract the zipped folder and run the ‘ezbase.exe’ file on your system. This would start an instance of your server at ‘<http://localhost:3690>’ by default.
* The Admin UI will be available at ‘<http://localhost:3690/api/index.html>’.
* The Documentation application is available at ‘<https://ezbase.vercel.app/>’.
* The npm SDK is available at ‘<https://socket.dev/npm/package/ezbase-ts>’. It can be downloaded in a project by running ‘npm i ezbase-ts’ in the project directory.
* The server must be deployed on an EC2 instance or similar, with socket port (3691) exposed as baseurl and the REST API port (3690) exposed with the url: “rest/”.

# Conclusion

## Summary

We approached this project with an open mindset. We learned that software decisions are not easy to make and often one has to step back, look at the bigger picture and reassess their position. Software development is also an ever changing process, the choices of yesterday may prove greatly useful tomorrow or they may come back to haunt you and force a complete overhaul of the system. Good software is also not easy to make, stuff like rules, auth, realtime etc. require actual planning for them to work out properly otherwise the system has to be held together with duct tape and dreams. With all this said, software development is still a fun activity and a source of great learning.

## Challenges

1. The biggest challenge faced was the technical difficulties in developing the server in Rust. Although the project started as developing a Rust server, however, we realized that some of our use cases i.e., role based authorization were difficult to implement in a language which we had limited knowledge of and that these use cases could have been standalone project/s of their own. Consequently, we decided to migrate our server and SDK to Typescript as the development team had much greater knowledge of it and this would result in the project being completed on time.
2. Additional challenges included stuff like cloning the project repository taking 30 minutes because there were multiple large videos and documents on it. We addressed this by waiting. When that did not work, we made a SSH key for a secure and stable download.
3. Serving the frontend from the backend was something that should not have been a challenge but proved to be one of the more annoying issues due to Hono’s poor documentation. We addressed this by overcoming technical skill issues.
4. Deploying a demo EZBASE instance on a hosting platform was also a challenge due to a lack of Bun support and issues with opening sockets on hosting platforms. We solved this by deploying the demo using docker and hard toil to fix socket issues.
5. RBAC was a feature that was most difficult to implement. It was addressed through raw engineering talent and hard work.
6. Some non-technical issues included the development team being spread across Lahore (except for one member who was living in the university’s hostel) and it being difficult to communicate properly. We addressed this through online meetings and meeting up at university.

## c. Future

Although Typescript provides great tooling and strong error-handling, it is not a relatively fast language. Future work can include developing the server in languages such as C, Erlang, Zig or Google Docs (as a database). Future work can also include adding unique features to the end product which sets it apart from the available products. These features could include:

* Advanced data analytics and monitoring tools.
* Integration with machine learning.
* Advanced security features like IP blocking, DDOS mitigation etc.
* Industry specific tooling and support.
* Database generation from provided schema.
* Horizontal database sharding.
* SDK’s for more languages.
* Deterministic simulation testing like that of TigerBeetleDB.
* Preset database schemas for popular use cases (banking, ecommerce etc)
* Separate dashboards for different projects.
* Database migration files.
* Plug-N-Play databases.

# Review checklist

|  |  |
| --- | --- |
| **Chapter/Section Name** | **Reviewer Name(s)** |
| Section 1, 2, 3 | Faraz Mansur Ahmad |
| Section 8, 10, 11 | Muhammad Saad |
| Section 6,7, 9 | Moiz Raza Amir |
| Section 4 | Ahmed Mozammil Iqbal |
| Section 5 | Abdul Wahab |