**Requirements Specifications**

**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** |

|  |  |  |
| --- | --- | --- |
| **Content** | **Totals** | **Obtained** |
| Introduction & system actors | 5 | 5 |
| Use case diagram | 10 | 7 |
| Use case descriptions | 20 | 17 |
| Class diagram | 20 | 16 |
| Sequence diagram | 20 | 14 |
| State diagram | 5 | 5 |
| Non-functional requirements | 5 | 5 |
| Who did what | 5 | 5 |
| Review checklist | 5 | 5 |
| Overall formatting/template | 5 | 5 |
| Late submission penalty | -20 |  |
| Github folder structure | -5 |  |
| **Total** | **100** | **84** |
| Review | 20 |  |
| **Grand Total** |  |  |

**Table of Contents**

[1. Introduction 2](#_gjdgxs)

[2. System Actors 4](#_30j0zll)

[3. Use Cases 5](#_1fob9te)

[4. Class Diagram 28](#_1t3h5sf)

[5. Sequence Diagrams 30](#_17dp8vu)

[6. State Diagrams 49](#_lnxbz9)

[7. Non-functional Requirements / Quality Attributes 50](#_44sinio)

[8. Who Did What? 50](#_2jxsxqh)

[9. Review checklist 51](#_z337ya)

# 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 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 |

# Use Cases

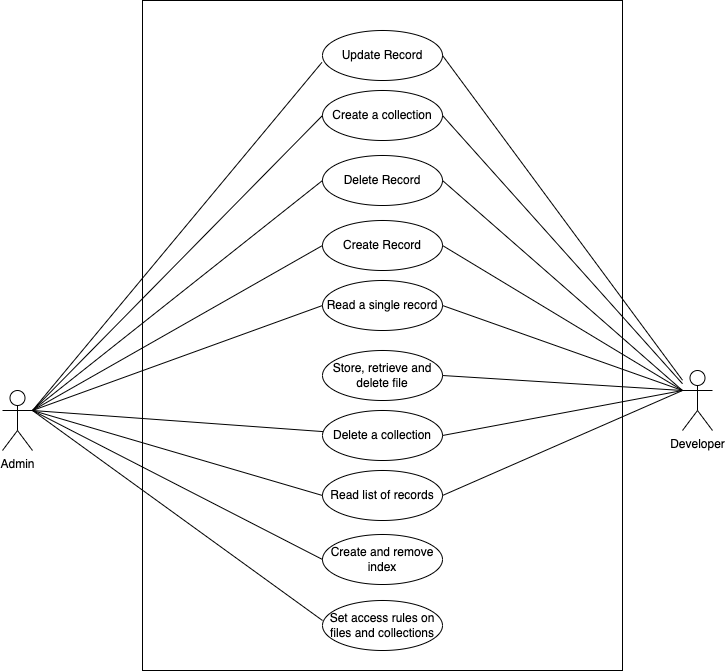
## Use Case Diagrams

Part 1 -

[Administrative tasks are bundled with API tasks. Add one more actor which is responsible for configurations like super user and admin is responsible for accounts management. Clarify responsibilities of each again which will improve the overall use case diagram.]



Part 2-



## Description of Use Cases

[Add more details in key use cases, otherwise it appears as if your system functionality is quite small. If use cases are actually quite simple, add some more features in your system. ]

**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

### Create and remove an index on a collection through the UI

### User Logs.

1. Store, retrieve and delete a file through the API
2. Set access rules on files and tables through the UI
3. OAuth adapters for Google and Facebook
4. Create a user with username and password through API and UI
5. Log in/out user
6. Allow user to reset their passwords through the API
7. Create an Admin, Delete an Admin through the UI
8. Information website with Documentation (https://docusaurus.io/)
9. Start a Server to serve all the requests
10. Upload and alter S3 credentials to change file upload destination
11. Upload and alter email credentials to change email server destination
12. 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 [How this course of action would execute?]. | |
| **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

[Step numbers missing for referring to fall-back conditions]

| **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. | |

1. Delete a record through the API and UI

| **Identifier** | | UC-003 |
| --- | --- | --- |
| **Purpose** | | The purpose of this use case is to allow developers and system admins to delete an existing record from 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 deleted exists in the database. |
| **Post-conditions** | | The selected record is deleted successfully. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user accesses the API endpoint or navigates to the "Delete" section in the UI. | |
|  | The user selects the record to be deleted. | |
|  | The user confirms the deletion request. | |
|  | The backend processes the request and deletes the selected record from 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 delete a record | |
| **Step #** | **Exception Paths** | |
|  | In step 4, if there are issues with database connectivity or the record cannot be deleted, the backend sends an error response to the user. | |
|  | In step 5, if the backend encounters an error while deleting the record, it sends an error response to the user. | |

4. Read a record through the API and UI

| **Identifier** | | UC-004 |
| --- | --- | --- |
| **Purpose** | | The purpose of this use case is to allow developers and system admins to retrieve and view an existing record from 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 retrieved exists in the database. |
| **Post-conditions** | | The selected record is successfully retrieved and displayed to the user. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user accesses the API endpoint or navigates to the "Read" section in the UI. | |
|  | The user specifies the criteria (e.g., record ID or search parameters) to retrieve the record. | |
|  | The user submits the request. | |
|  | The backend processes the request, retrieves the requested record from the database. | |
|  | The backend sends the retrieved record as a response to the user. | |
|  | The user views the record. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | The user can either access the API endpoint or the UI to read a record | |
| **Step #** | **Exception Paths** | |
|  | In step 4, if there are issues with database connectivity or the specified record cannot be found, the backend sends an error response to the user. | |
|  | In step 5, if the backend encounters an error while retrieving the record, it sends an error response to the user. | |

5. Read a list of records through the API and UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-005 |
| **Purpose** | | The user wishes to retrieve or view the list of records in a collection stored in the database through API or UI. |
| **Pre-conditions** | | The user must be authenticated and authorized to access the records.  The database contains records that the user is permitted to view.  The API server is up and running.  The UI is accessible and responsive. |
| **Post-conditions** | | The user is presented with a list of records from the database.  The records are displayed in a readable and organized manner in the UI.  No changes are made to the actual records in the database. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1.** | The user is authenticated into the system. | |
| **2.** | If using the admin UI, admin is navigated to the collection section of the UI where records are displayed. If using API, API receives a request to get a list of records from a collection. | |
| **3.** | The API retrieves the records from the database after verifying permissions from rules module. | |
| **4.** | The API returns the list of records to the admin UI or sends a JSON object list of records as a response to the request. | |
|  |  | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| **1** | In step 2, the request to retrieve a list of records can be made with specific parameters like filters on fields, sorting the records based on specific fields or expanding a foreign field. | |
| **2** | The API queries the database according to the given parameters and returns a response. | |
| **Step #** | **Exception Paths** | |
| **1** | The user is not authenticated and not authorized to view certain records: The API returns an unauthorized error message. | |
| **2** | There are no records found, the API returns an empty list. | |
| **3** | 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. 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. | |

### 

7. 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. | |

### 

### 8. Create and remove an index on a collection through the UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-008 |
| **Purpose** | | The User can index a collection to make reading the collection faster. Or the User can delete an Index if in case the Index is taking extra space. |
| **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 Indexes view. The index must exist for the delete index to work. |
| **Post-conditions** | | An index is created if a ‘create’ request is sent, making reads faster. Or the index is deleted if the ‘delete’ request is sent. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1** | The User clicks on the button that says ‘Create Index’. | |
| **2** | The user proceeds to click the ‘+’ button. | |
| **3** | The user selects all the collections from the list which he wants as indexed. | |
| **4** | The user clicks ‘Save Changes’. | |
| **5** | Indexes of all the needed collections are created and displayed in the view. | |
| **6** | In order to delete a collection’s index, all the collections indexes to be deleted are selected. | |
| **7** | The user clicks the ‘Delete’ button to delete the indexes | |
| **8** | The indexes are removed from the view of all indexes. | |
| **9** | The use case is complete. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. | There are no other alternative courses of action. | |
| **Step #** | **Exception Paths** | |
| 1. | In step 7, if a collection index is unable to be deleted due to certain rules being set on the collection, then an error response is returned mentioning the index can not be deleted. | |

### 

9. User Logs

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-009 |
| **Purpose** | | To maintain a traceable record of all user interactions with the system, aiding in debugging, and analyzing user activities. |
| **Pre-conditions** | | The user is interacting with the system through the API [has access?]. |
| **Post-conditions** | | A log entry is generated and stored, detailing the user's request and any relevant metadata. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1.** | The user sends a request to the API. | |
| **2.** | The server captures the request details, including: timestamp, IP address, status of user (authenticated or guest) and request type. | |
| **3.** | While the request is processed normally, a log entry is added with the captured details. | |
| **4.** | The log entry is made available on the admin UI panel. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| **1** | N/A | |
| **Step #** | **Exception Paths** | |
| **1** | The logging system or service is down or unreachable, the system should respond to this with an error message on admin panel. | |
| **2** | In step 3, the request can be erroneous however, a log entry is made regardless. | |

### 

10. 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 S3 bucket is up and running along with sufficient storage space. |
| **Post-conditions** | | For storing: The file is successfully uploaded and stored in the S3 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 S3 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 S3 bucket.  For deleting, API deletes the file from the S3 bucket.  For retrieving, the API responds with the url to the file from where it can be accessed/downloaded. | |
| **3.** | For storing, S3 API returns a unique url for the file which is stored in the database. For deleting, API deletes the file from the S3 bucket 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 S3 bucket 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. | |

### 

11. 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. | |

### 

12. 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 or Facebook, based on the granted permissions. |
|  | | |
| **Step #** | **Typical Course of Action** | |
| **1.** | The user logs in and chooses either Google or Facebook 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. | |

### 

13. 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. | |

14. Log in/out a user

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-014 |
| **Purpose** | | The user logs in or out. |
| **Pre-conditions** | | The system is online the database is available and the authentication module is operational |
| **Post-conditions** | | In the case of logging in the user receives an authenticated user model that contains data that can be used for future authentication purposes. In the case of logging out, this model and associated data is cleared. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user makes the login or logout API call. | |
|  | In the case of logging in the user has to provide the username and password for the account they want to login to. In the case of logging out the authenticated user model is cleared and the logout sub use case ends. | |
|  | In the case of logging in, the supplied credentials are validated by comparing them with the existing user record in the database. | |
|  | After successful validation an authenticated user model is returned along with a successful response, this user model contains all the data required for future authentication purposes. | |
|  | Upon receiving the successful response, the authenticated user model is stored and the user is considered to be logged in. | |
|  | The login sub use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | In step 3 no existing user entry is found and the system responds with a “user does not exist” error. | |
| **Step #** | **Exception Paths** | |
|  | If the system, database or the authentication module becomes unavailable or too slow to respond, the browser times out and returns an error. | |

15

15. Allow users to reset their passwords through the API.

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-015 |
| **Purpose** | | The user resets their account password through the API or UI. |
| **Pre-conditions** | | The system is online the database is available and the authentication module is operational |
| **Post-conditions** | | The previous password entry is securely replaced by the newly provided password in the user’s record in the database. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | The user makes the API call containing their authenticated user object and new password. | |
|  | The authenticated user object is passed to the authentication module along with the new password. | |
|  | The authenticated user object is successfully validated by comparing to the existing user record in the database. | |
|  | The old password is securely replaced with the provided new password. | |
|  | A successful response is returned with the updated authenticated user model. This model contains all the data needed for future authentication purposes. | |
|  | Use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | In step 3 an invalid authenticated user object will result in a failure scenario with a failure response being received from the system. | |
| **Step #** | **Exception Paths** | |
|  | If the system, database or the authentication module becomes unavailable or too slow to respond, the API and browser times out and returns an error. | |

16. Create or delete an Admin through the UI

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-016 |
| **Purpose** | | Create or delete an admin through the UI. |
| **Pre-conditions** | | The system is online and the authentication module is operational.For creating an admin the user is on the create account page. For deleting an admin the user is on the delete account page. |
| **Post-conditions** | | A new admin is created or deleted. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | In the case of creating an admin the user is prompted to enter a username and a password. In the case of deleting an admin the user presses the delete account button. | |
|  | In the case of creating an admin the user presses the sign up button. In the case of deleting an admin the user’s credentials are sent to the authentication module. | |
|  | In the case of creating an admin the user input is sent to the authentication module. In the case of deleting an admin the user’s credentials are validated with an existing admin record in the database. | |
|  | In the case of creating an admin the username is validated for uniqueness. In the case of deleting an admin, after successful validation the admin record is deleted from the database. | |
|  | In the case of creating an admin after successful uniqueness validation, the entered password is securely stored along with a system generated ID in a newly created admin database record. In the case of deleting an admin after deletion and successful response any authentication data in the browser is cleared. | |
|  | In the case of creating an admin a successful response is sent along with the authentication data that is stored in the browser for future authentication. In the case of deleting an admin a successful deletion message is shown on the UI. | |
|  | In the case of creating an admin the authentication data is stored in the browser. In the case of deleting an admin the user is directed to the login screen | |
|  | In the case of creating an admin a successful creation message is shown on the UI. The delete admin sub case ends. | |
|  | In the case of creating an admin the user is directed to the admin dashboard. | |
|  | Admin creation sub case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | In step 3 for deleting an admin, invalid credentials will result in a deletion failure response. | |
|  | In step 4 for creating an admin, a non unique username will result in an admin already existing response. | |
| **Step #** | **Exception Paths** | |
|  | If the system or the authentication module becomes unavailable or too slow to respond, the browser times out and returns an error. | |

17. Information website with Documentation

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-17 |
| **Purpose** | | To provide users with information and documentation through a website. [Actor?] |
| **Pre-conditions** | | N/A |
| **Post-conditions** | | Users have access to the information and documentation. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | Developer or Admin navigates to the website. | |
|  | Developer or Admin browses the available information and documentation. | |
|  | Developer or Admin searches for specific content if needed. | |
|  | Developer or Admin can download or access documentation as required. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
|  | N/A | |
| **Step #** | **Exception Paths** | |
|  | If the Developer or Admin does not find the required information on the website, they file an issue on Github or start a discussion. | |

18. 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. | |

19. Upload and alter S3 credentials to change file upload destination

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-19 |
| **Purpose** | | To upload and modify S3 credentials for changing the file upload destination. |
| **Pre-conditions** | | Admin has appropriate access privileges. |
| **Post-conditions** | | S3 credentials are updated, affecting file upload destinations. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | Admin accesses the S3 credential management settings. | |
|  | Admin uploads new S3 credentials. | |
|  | The S3 credentials are altered. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. 1 | N/A | |
| **Step #** | **Exception Paths** | |
| 1. | The S3 credentials are invalid and an error is displayed | |

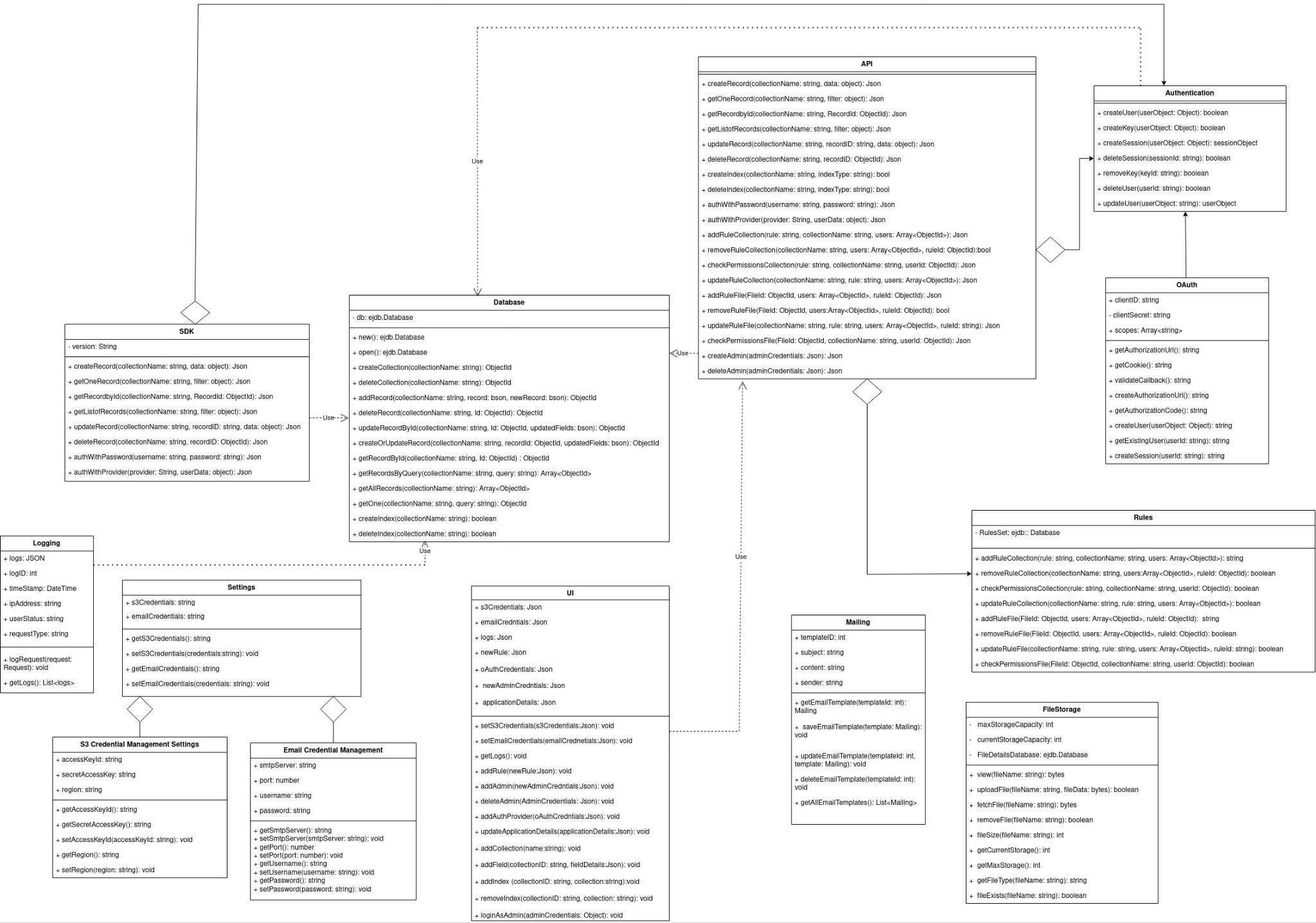
20. Upload and alter SMTP credentials to change email server destination

|  |  |  |
| --- | --- | --- |
| **Identifier** | | UC-20 |
| **Purpose** | | To upload and modify SMTP credentials for changing the email server destination. |
| **Pre-conditions** | | Admin has appropriate access privileges. |
| **Post-conditions** | | Email credentials are updated, affecting the email server destination. |
|  | | |
| **Step #** | **Typical Course of Action** | |
|  | Admin accesses the email credential management interface from the settings | |
|  | Admin uploads new email credentials.  Admin confirms the update. | |
|  | The email credentials are altered. | |
|  | The use case ends. | |
|  | | |
| **Step #** | **Alternate Courses of Action** | |
| 1. 1 | N/A | |
| **Step #** | **Exception Paths** | |
| 1. | The email credentials are invalid and an error is displayed | |

# Class Diagram

[Most of your classes does not contain any attributes, how the objects make interaction with other functions?]

## Diagram



The Diagram can be viewed more clearly on: <https://app.diagrams.net/#G1MqcLCWahPCHaNNq4k0rbNtsHlooHGqCP>

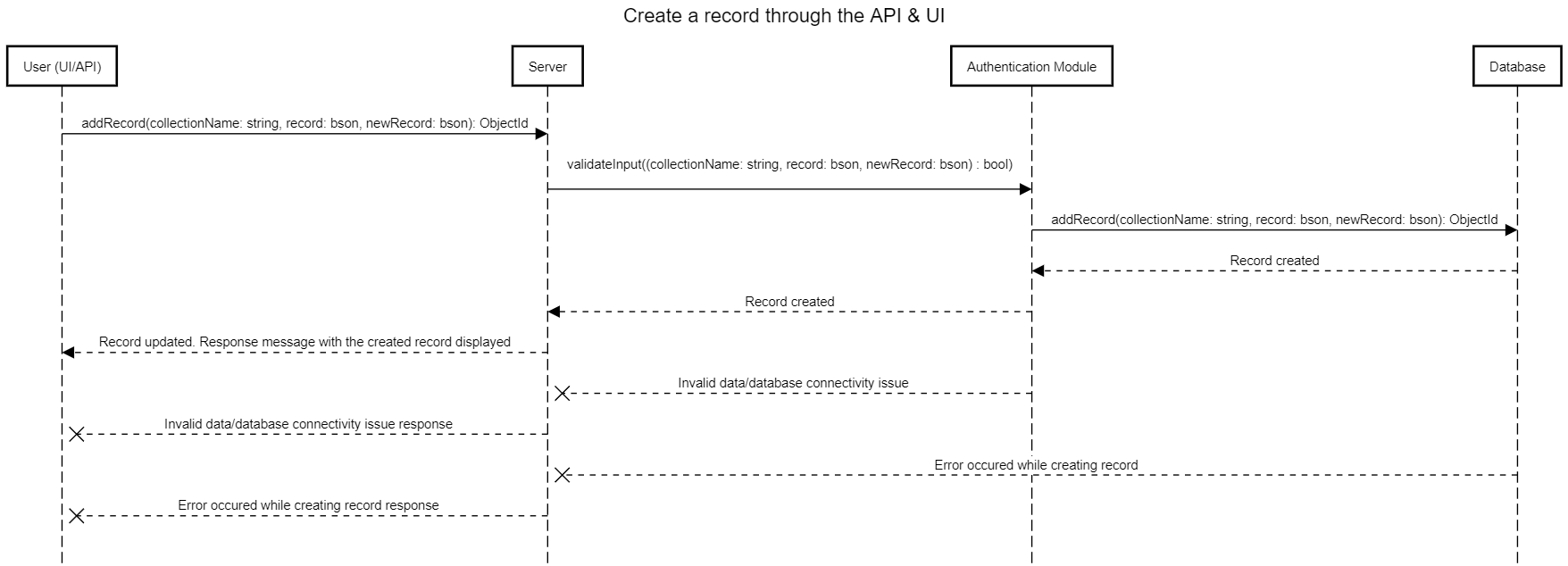
## Description

1. Database: Handles all interfacing with the underlying EJDB 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: The primary interface for all actions performed by the SDK. Core class with which almost every other class communicates.
4. Authentication: Provides the authentication features for both users and admins, including password reset and confirmation.
5. OAuth: Inherits from the Authentication class, and allows usage of third party social login providers.
6. Rules: Used for access control, allows the user to define access rules on database collections that control access.
7. Mailing: Used for password reset/email confirmation and other emails.
8. Logging: Used for logging user requests, and generating intelligent insights
9. UI: Primary interface for the admin, allows DDL operations, changing of settings, monitoring and other admin actions.
10. Settings: Used to tweak project settings, such as setting log retention time, admin credentials, ports, SSL certificates and so on.
11. S3 Credential Management Settings: Used to change the default file upload destination to S3.
12. Email Credential Management: Used to change the default SMTP credentials.
13. SDK class: Main interface for the developer, allows the developer to interact programmatically with the backend.

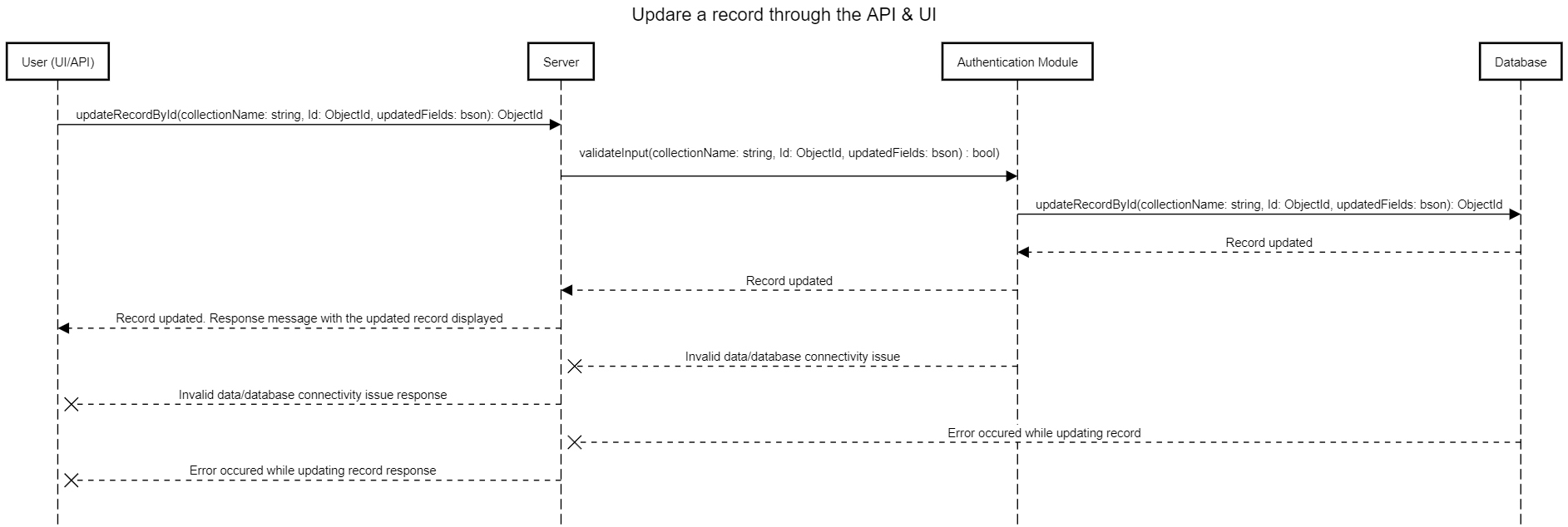
# Sequence Diagrams

[Sequence diagrams are inconsistent throughout the document. In some diagrams arrows are not as per the template. Request and responses are not aligned. Actors are not shown in most of the diagrams].

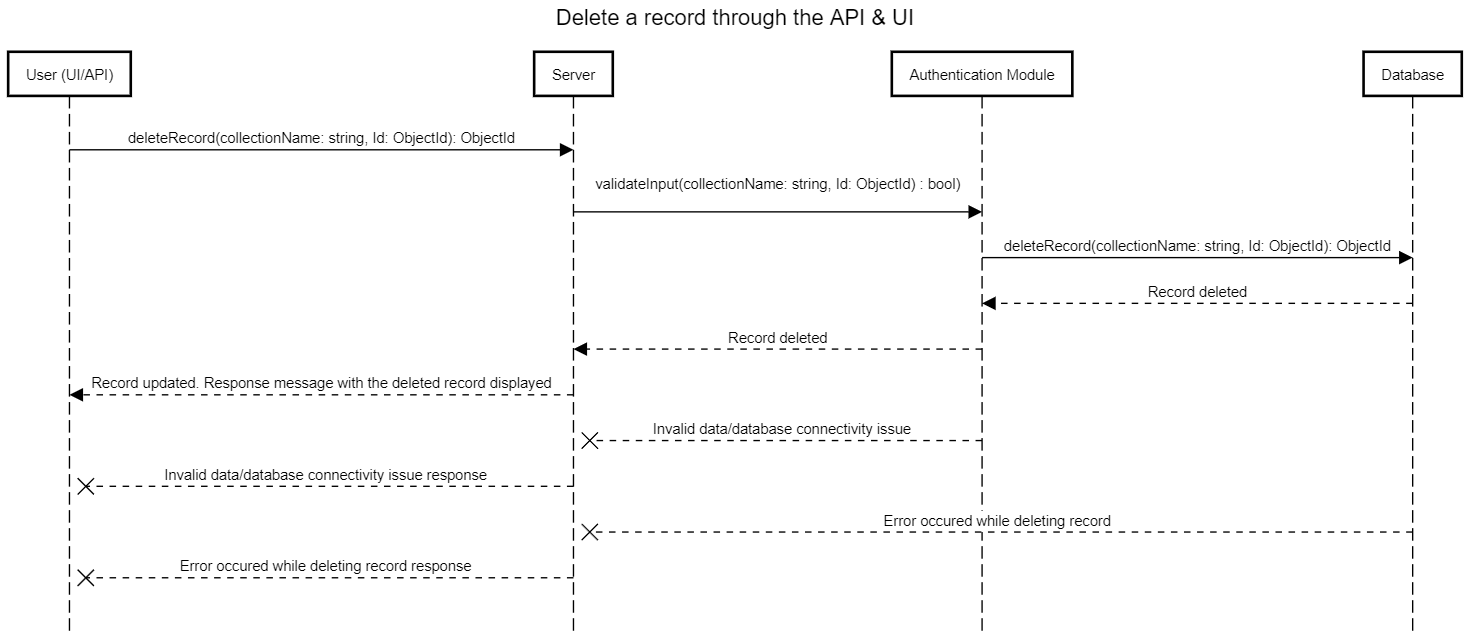
1. **Create a Record through the API and UI**



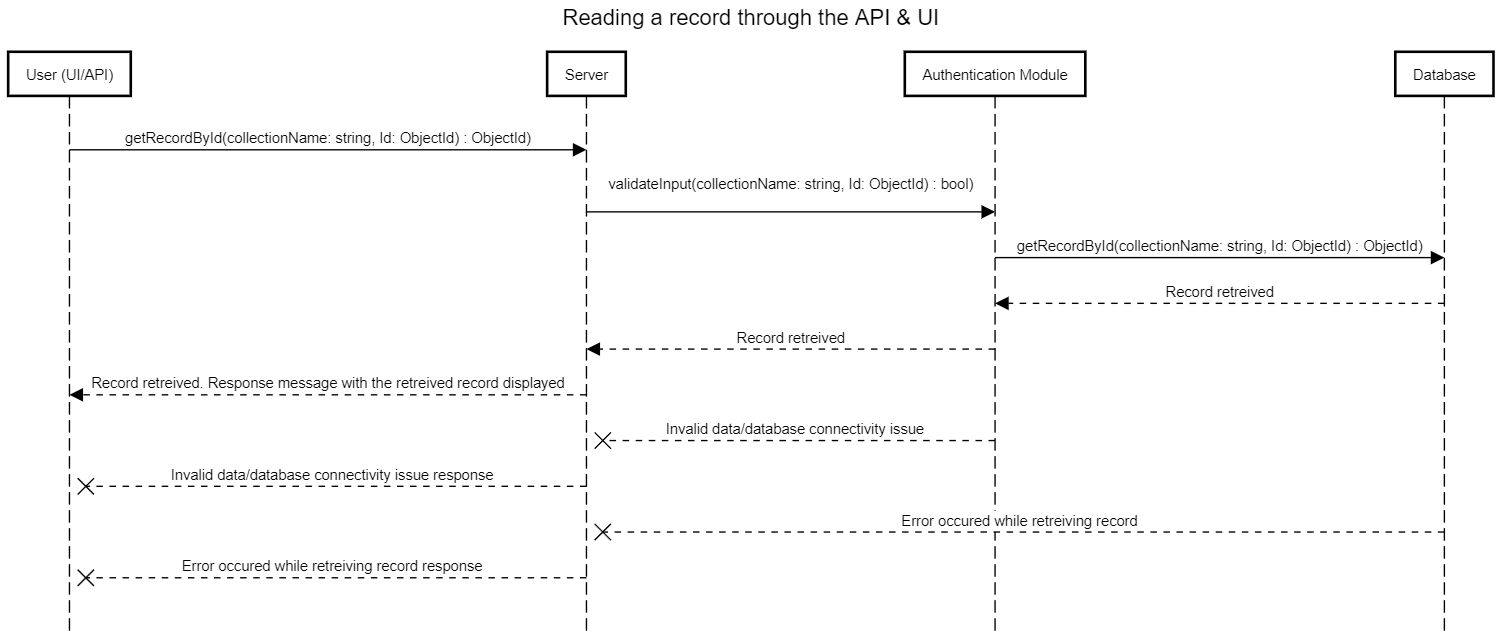
1. **Update a Record through the API and UI**



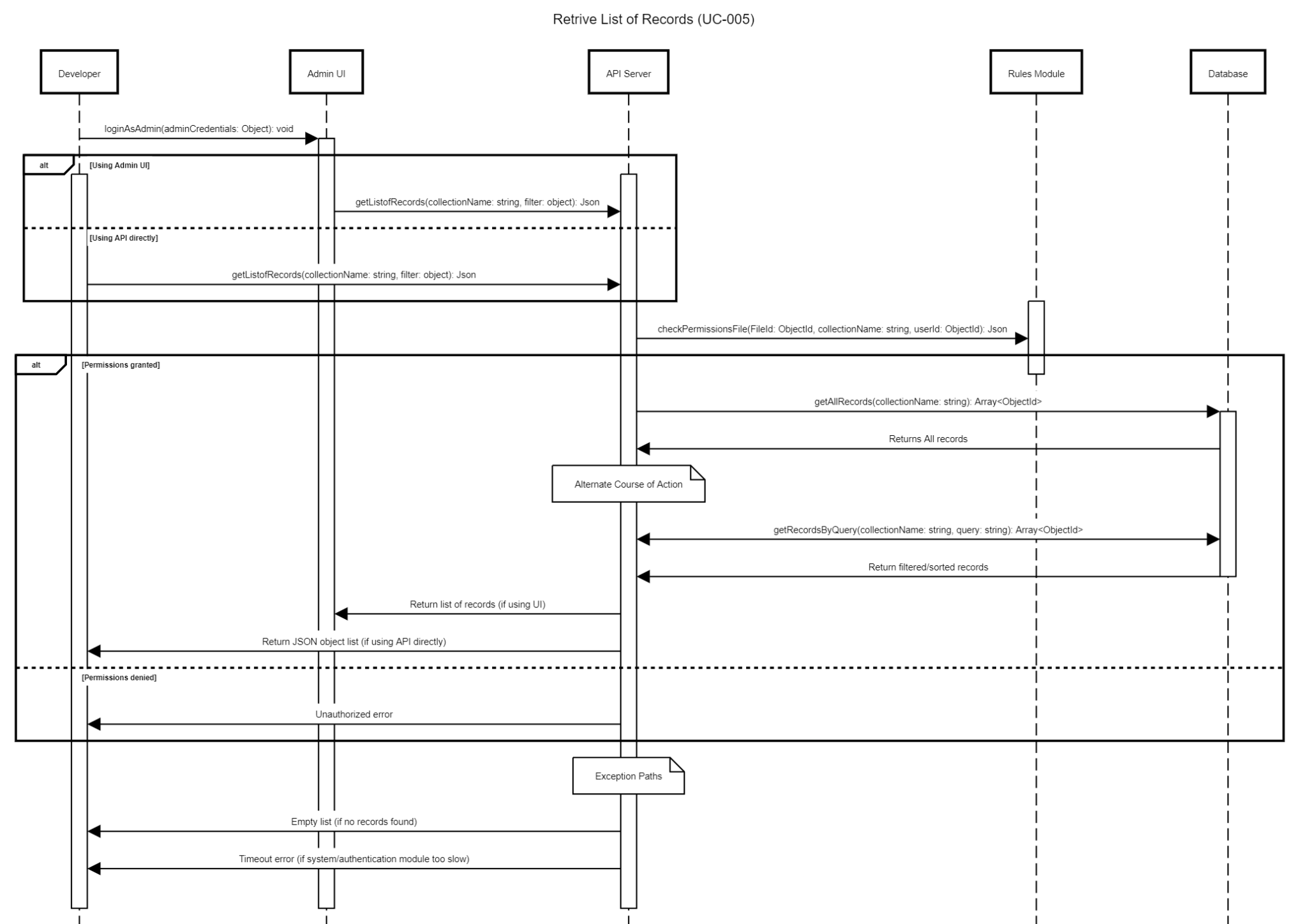
1. **Delete a Record through the API and UI**



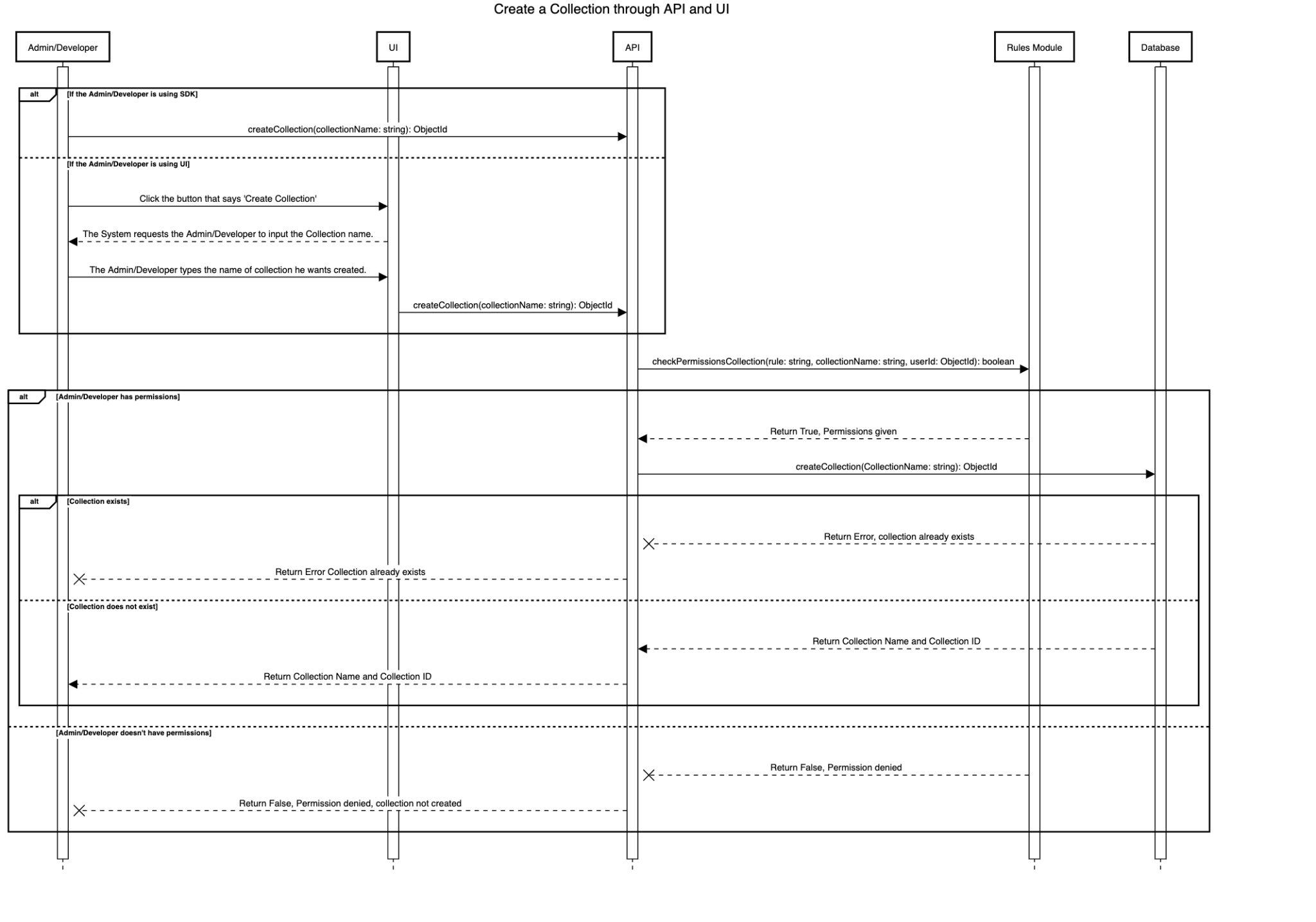
1. **Read a Record through the API and UI**



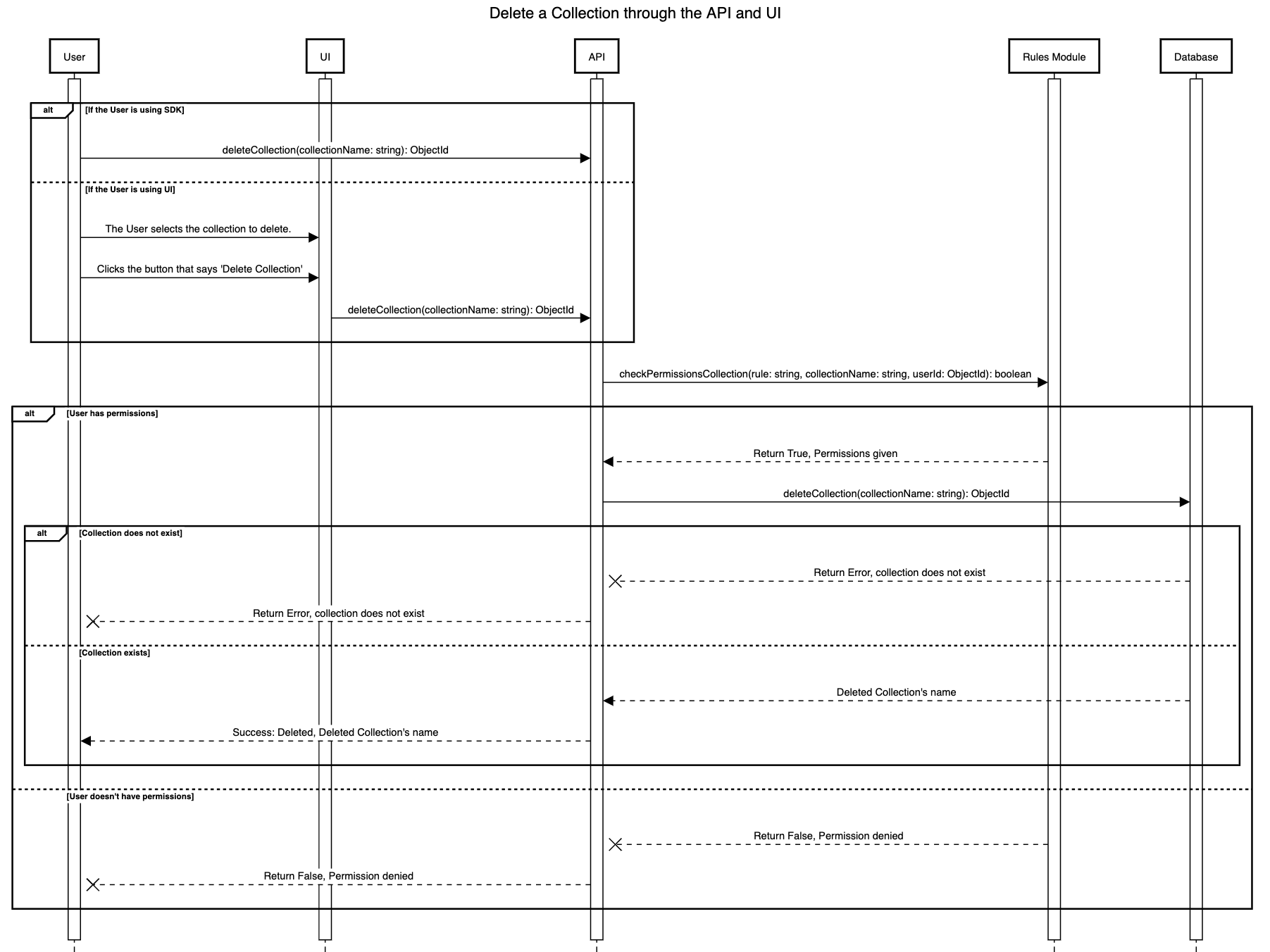
1. **Read a list of records through the API and UI**



1. **Create a Collection through API and UI**



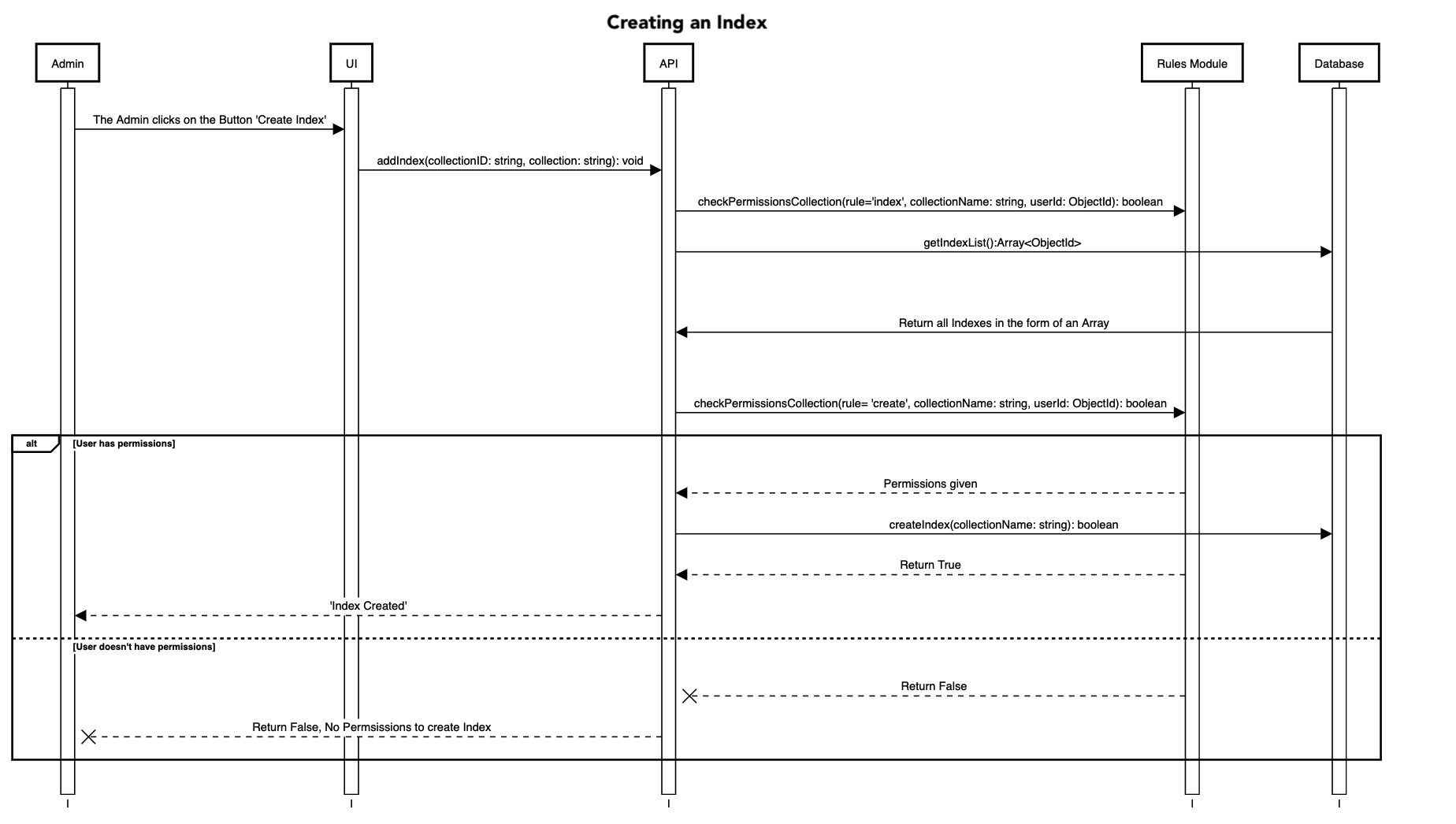
1. **Delete a collection through the API and UI**

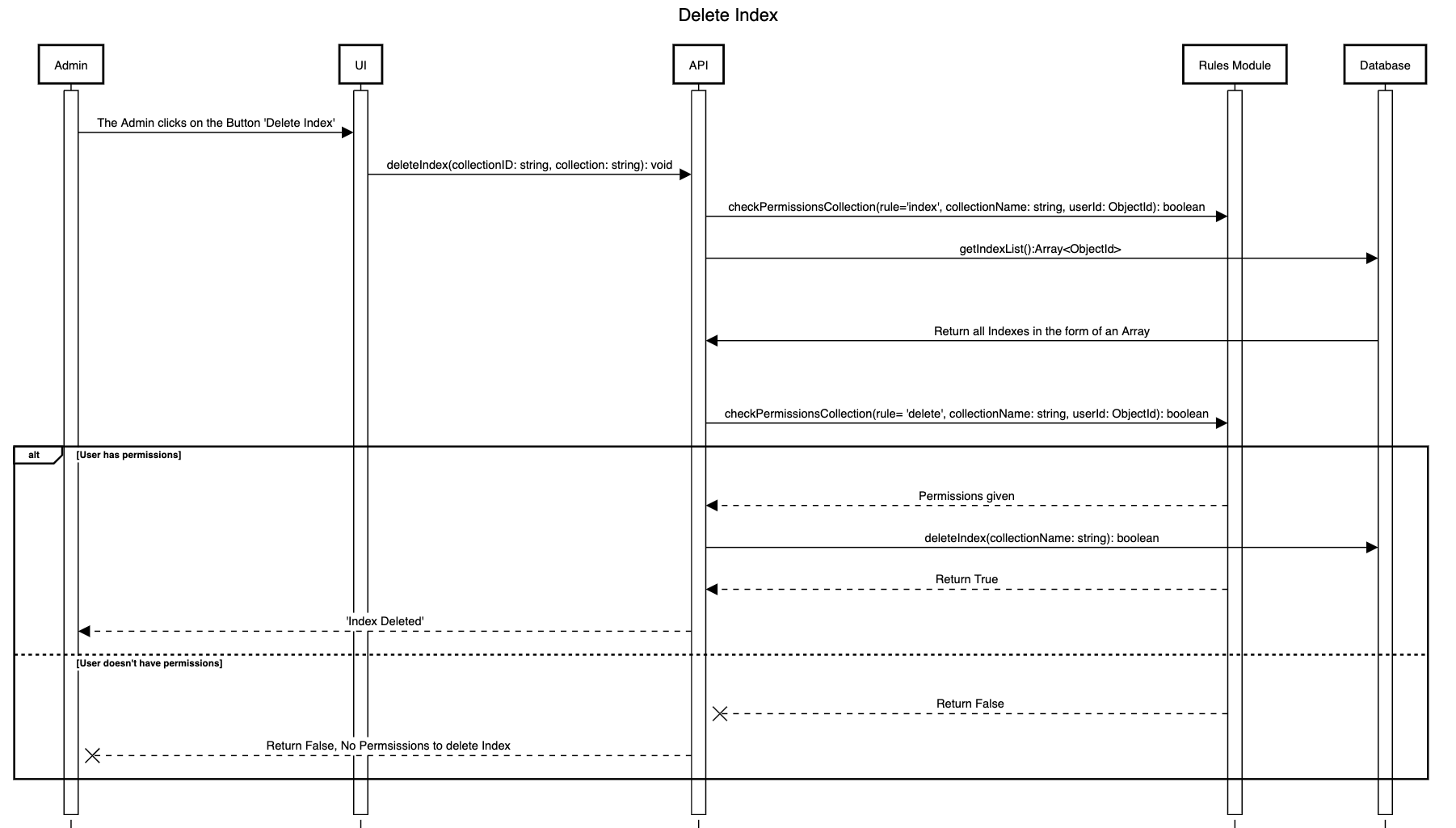


### **Create and remove an index on a collection through the UI**

[here getindexlist would be invoked in any case if the user has permissions or not]

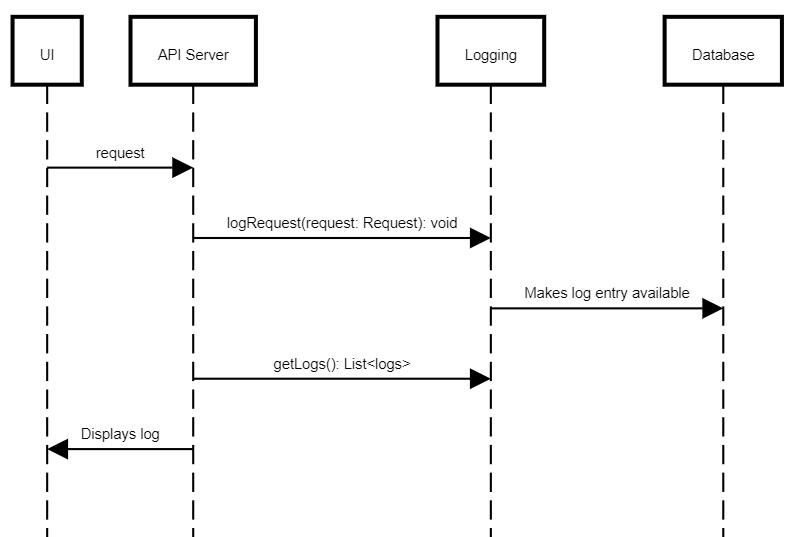
Do note, both the following sequence diagrams are about the same. One is performing create and other delete of Index.



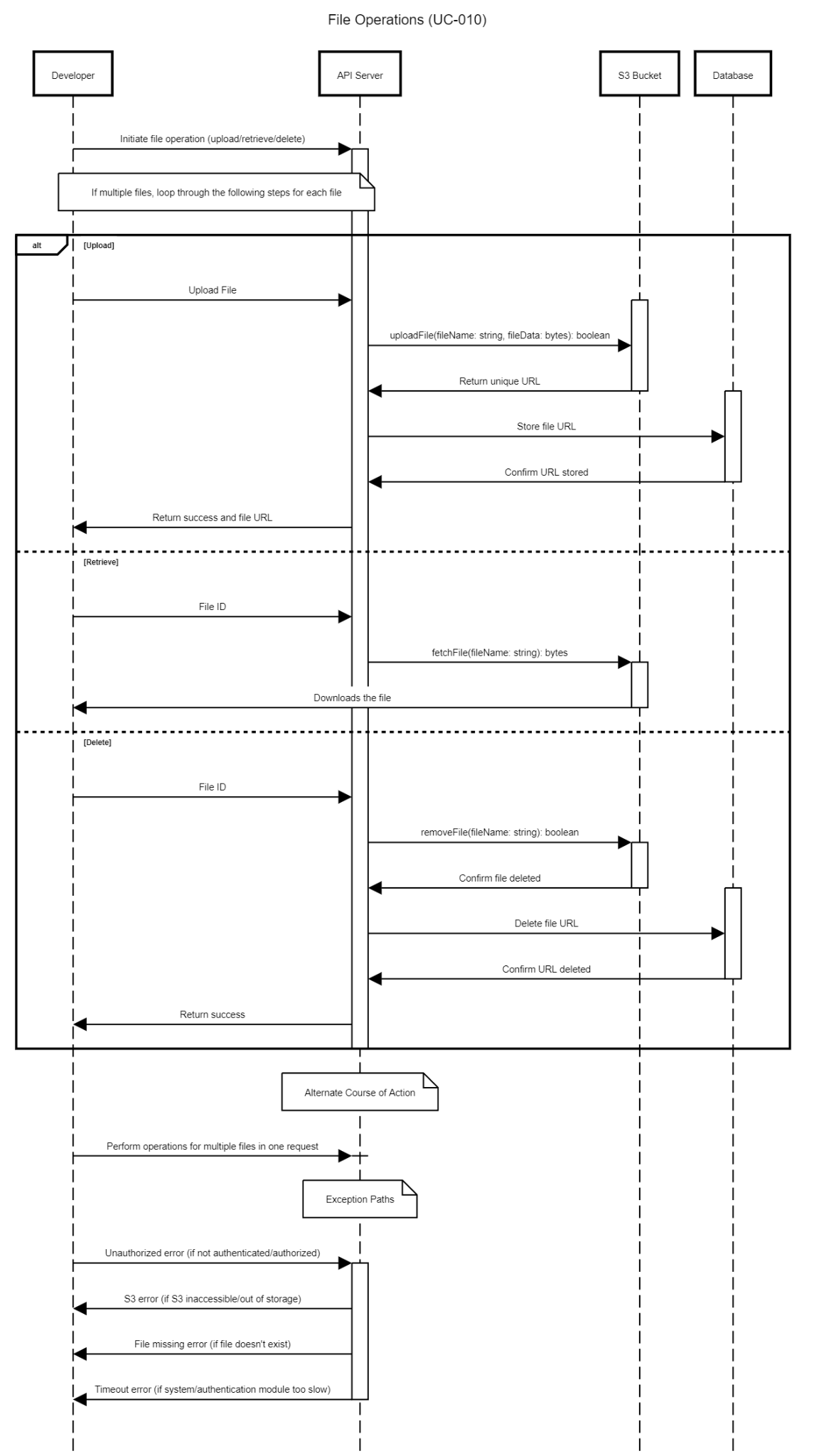


1. **User Logs**

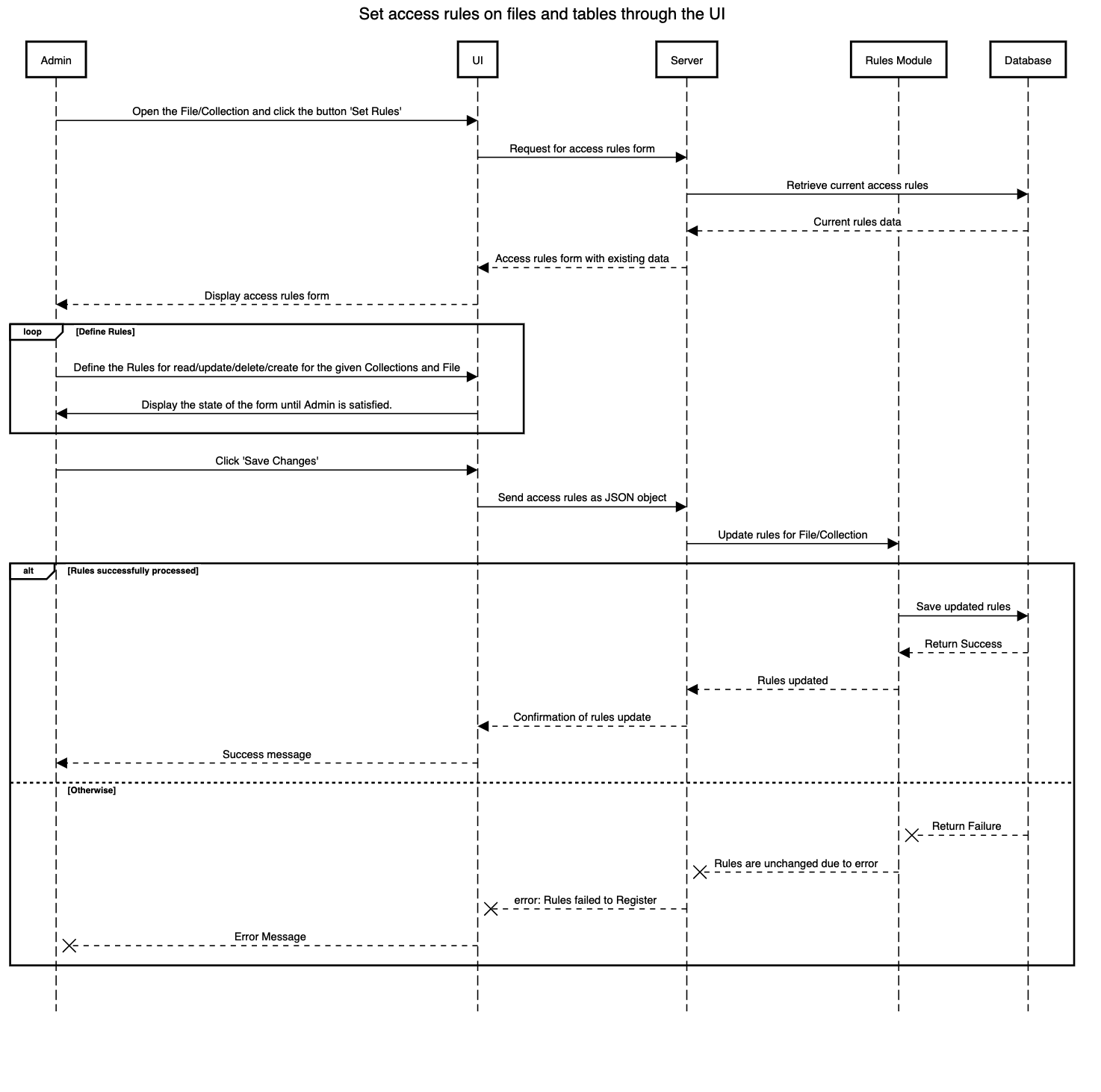
**[Inconsistent to other diagrams, the request response arrows are different, see the template and lecture slides.]**



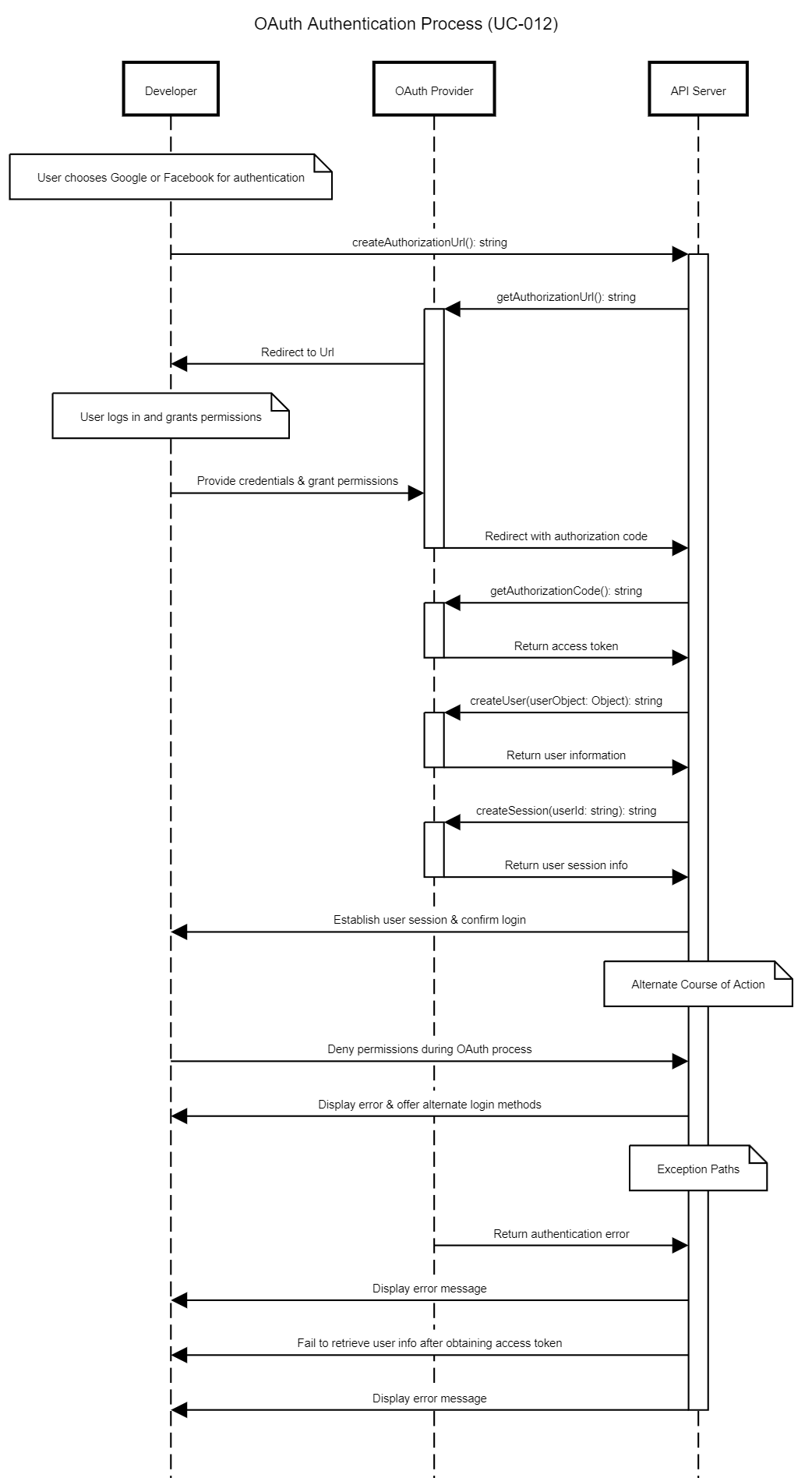
1. **Store, retrieve and delete a file through the API**



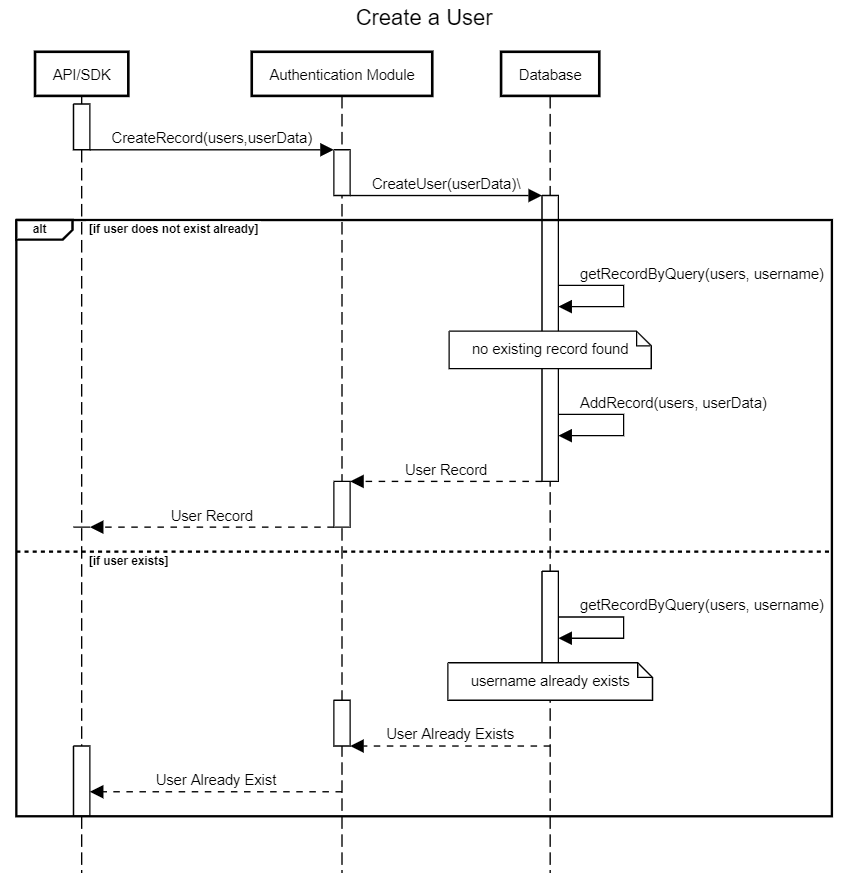
1. **Set access rules on files and tables through the UI**



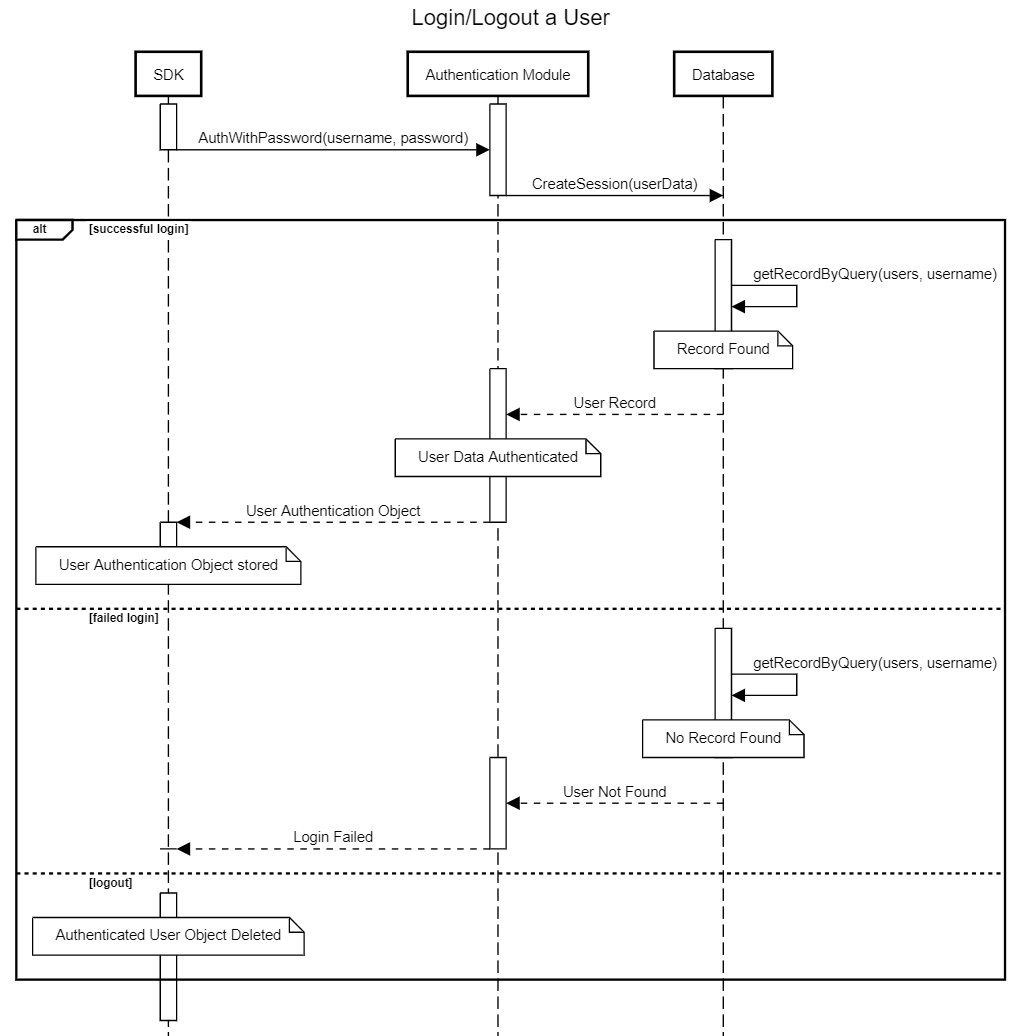
1. **OAuth adapters for Google and Facebook**



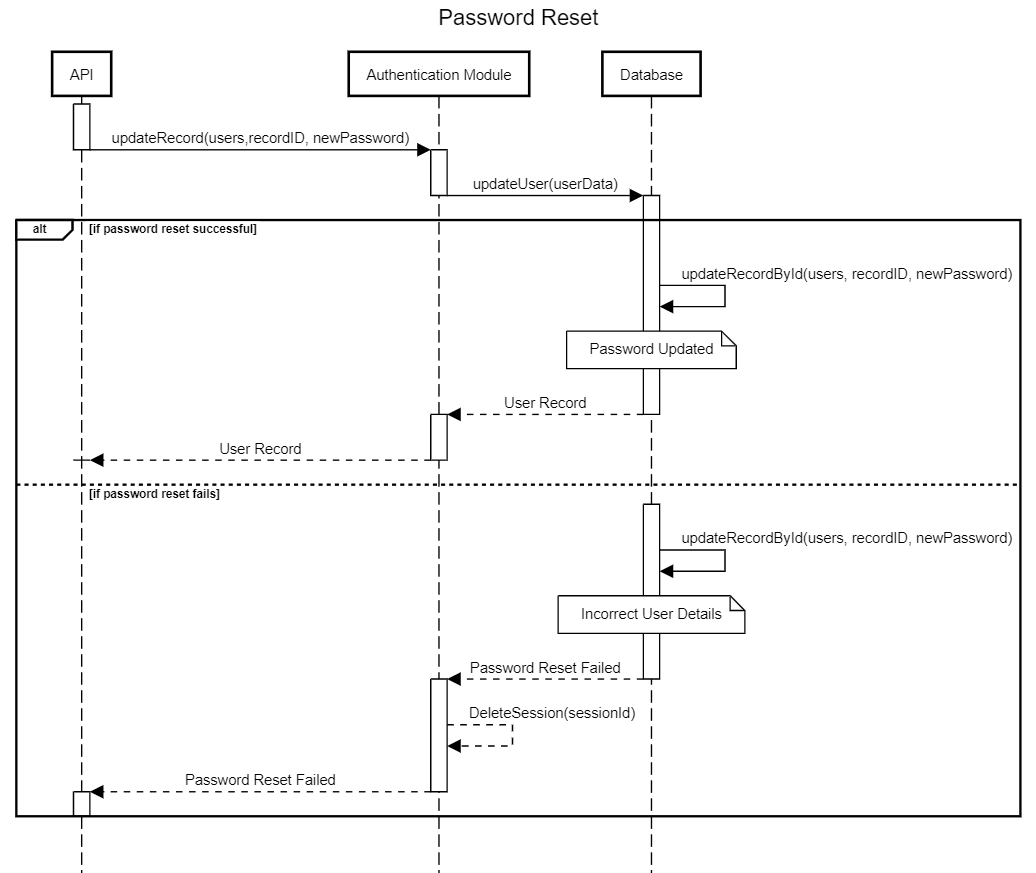
1. **Create a user with username and password through API or UI**



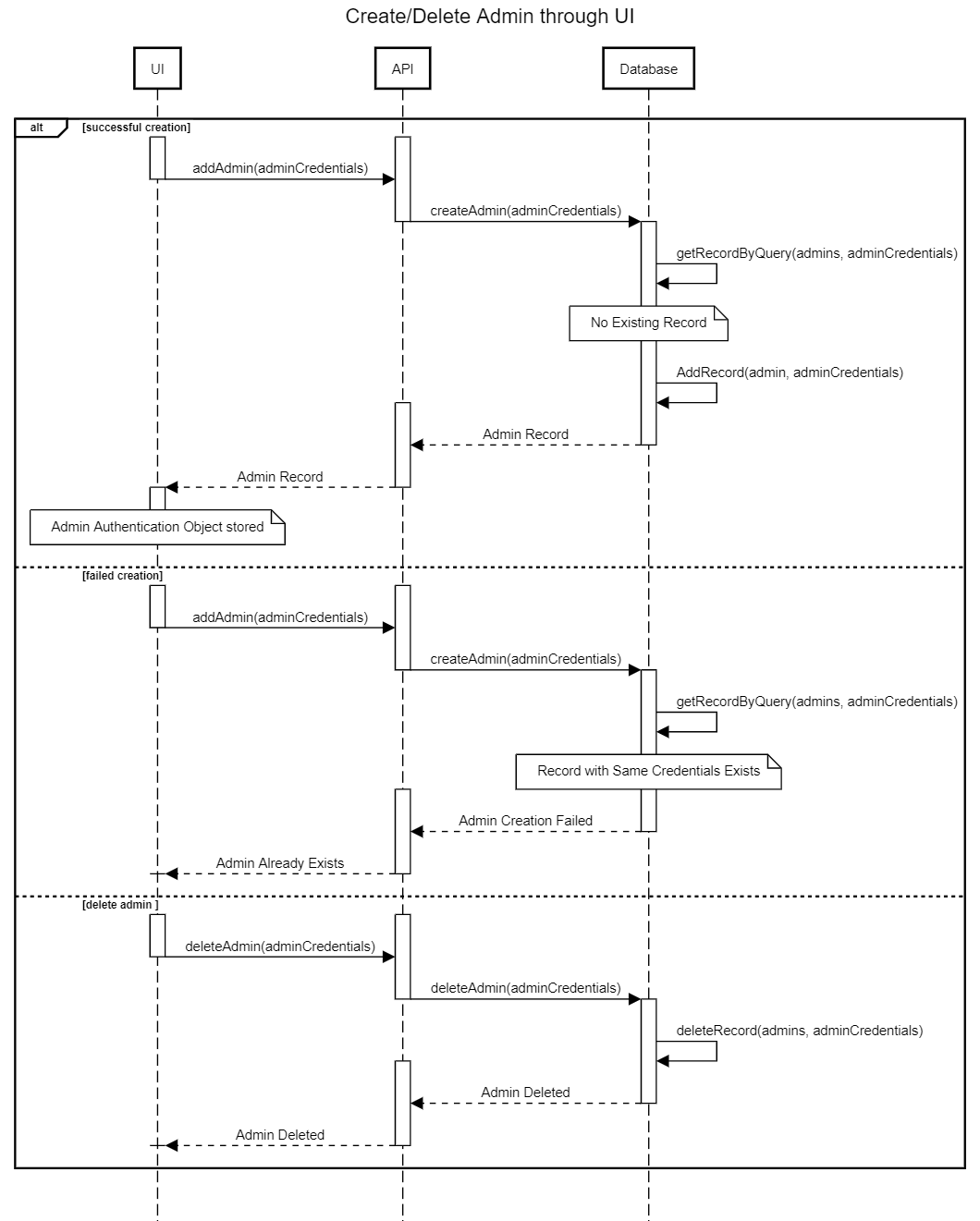
1. **Login/Logout a user**



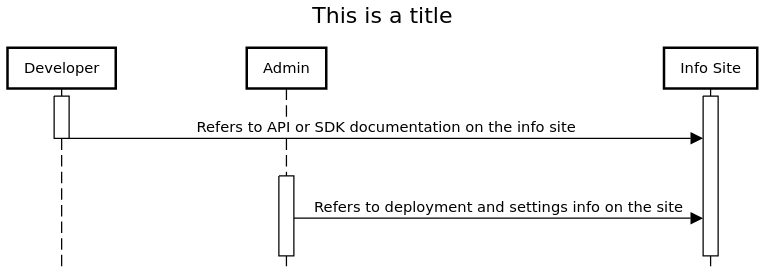
1. **Allow users to reset their password through the API**



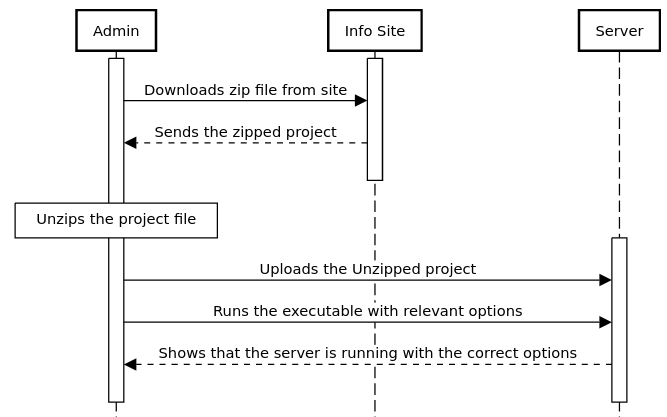
1. **Create or delete an admin through the UI**



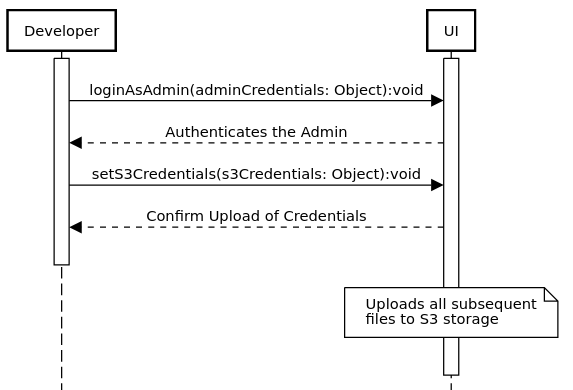
1. **Information website with Documentation (**[**https://docusaurus.io/**](https://docusaurus.io/)**)**



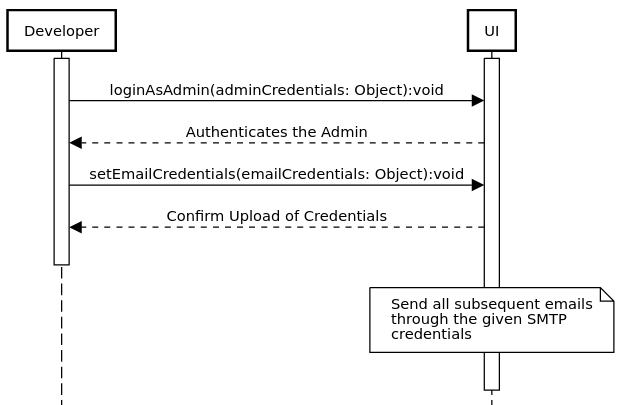
1. **Start a Server to serve all the requests**



1. **Upload and alter S3 credentials to change file upload destination**



1. **Upload and alter email credentials to change email server destination**

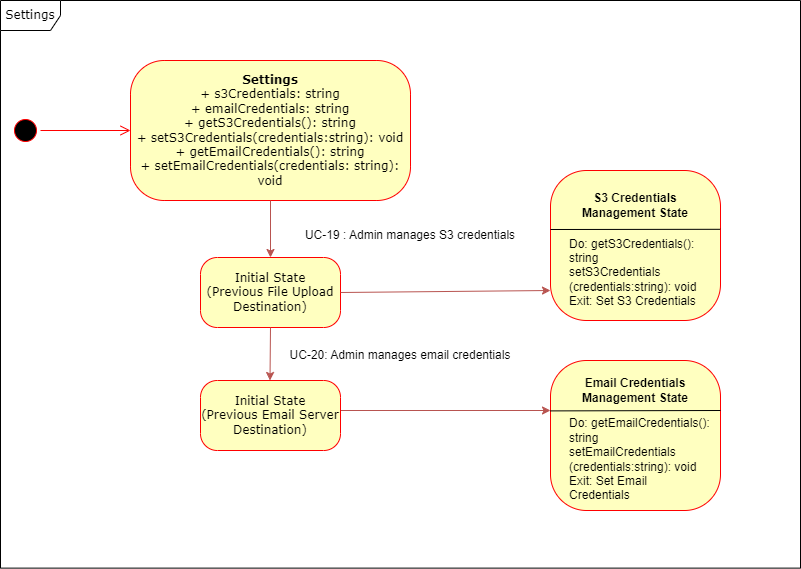


# State Diagrams

## Diagram details

The Settings class allows the admin to perform changes in the states of two of its subclasses, namely the S3 Credentials Management and the Email Credentials Management State. The initial state of these subclasses consists of empty or previously defined user fields. Through APi calls and the UI methods, the admin is able to successfully change and save the state of these fields on process completion.

## Diagram



# Non-functional Requirements / Quality Attributes

Below are the non-functional requirements for the system:

|  |  |
| --- | --- |
| **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. |

# Who Did What?

|  |  |
| --- | --- |
| **Name of the Team Member** | **Tasks done** |
| Muhammad Saad | 4 Use Case Descriptions + 4 Sequence Diagrams + Assigned Class Diagrams and Class Relations.  Completed UC Description of: UC013, UC014, UC0015, UC016. |
| Abdul Wahab | 4 Use Case Descriptions + 4 Sequence Diagrams + Assigned  Class Diagrams + Class descriptions.  Completed UC Description of: UC017, UC018, UC019, UC020. |
| Faraz Mansur Ahmad | 4 Use Case Descriptions + 4 Sequence Diagrams + Assigned  Class Diagrams + Class descriptions + State Diagram.  Completed UC Description of: UC001, UC002, UC003, UC004. |
| Ahmed Mozammil Iqbal | 4 Use Case Descriptions + 4 Sequence Diagrams + Assigned  Class Diagrams + Use Case Diagram + Pasting and Editing sections from last deliverable  Completed UC Description of: UC006, UC007, UC008, UC011. |
| Moiz Raza Amir | 4 Use Case Descriptions + 4 Sequence Diagrams + Assigned  Class Diagrams  Completed UC Description of: UC005, UC009, UC010, UC012. |

# 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)** |
| Use Case Diagram | Saad, Wahab |
| Sequence Diagrams | Ahmed |
| Class Diagrams | Faraz |
| State Diagrams | Moiz |
| UC001-UC004 | Saad |
| UC005-UC008 | Moiz |
| UC09-UC012 | Faraz |
| UC017-UC020 | Ahmed |
| UC013-UC016 | Wahab |