**CHAPTER 1**

**INTRODUCTION**

* 1. **Introduction**

Currently, Most of the communications between the client and server use REST architectural style through HTTP or simply RESTful API. The client requests resources through URL endpoints to access the specific resource provided by the server. The request and responses are usually formatted as JSON but not limited to it.

Not all the requests are authorized to access the services. The owner can access only his private information but not others, which requires the owner proving that they are the owner of that resources, which we call “Authentication”. While the process of granting authenticated party permission to do something is called “Authorization”.

JSON Web Tokens (JWT) have gained popularity as a secure and efficient method for handling authentication and authorization in client-server communications. When a user successfully authenticates themselves, the server issues a JWT, which is a compact and digitally signed token. The JWT contains encoded information about the user's identity and access rights.

One of the key advantages of using JWTs is their stateless nature.

Unlike traditional session-based authentication, JWTs eliminate the need for servers to store session information. Instead, the necessary information for authentication and authorization is encapsulated within the JWT itself. This design allows for scalability and ease of implementation across distributed systems due to the stateless nature of JWTs.

Upon receiving a JWT, the client includes it in subsequent requests as an authorization mechanism. The server can then validate the JWT's integrity and authenticity by verifying its digital signature. This process ensures that the token has not been tampered with or modified.

JWTs can carry additional claims, which can provide fine-grained access control, allowing servers to authorize specific actions based on the user's role or privileges. For example, a JWT may contain a "role" claim that specifies whether the user has administrative privileges or is restricted to certain operations.

Furthermore, JWTs can also be used to enable Single Sign-On (SSO) capabilities. With SSO, a user who has already authenticated with one application can seamlessly access other applications without needing to provide credentials again. JWTs serve as the mechanism for exchanging authentication information between different services, facilitating a unified and secure user experience. Additional standard mechanisms like JWE enables JWT’s otherwise easily decodable base64 to be encrypted.

Overall, JSON Web Tokens offer a versatile and standardized approach to authentication and authorization in client-server communications. By leveraging JWTs, developers can create secure and scalable systems that enable controlled access to resources while minimizing server-side state management.

* 1. **Problem Statement**

As the number of services within an organization increases, the complexity of managing access control and ensuring a secure authorization system becomes magnified. Each service may have dependencies on other services, leading to a web of interconnected access requirements. From a higher-level perspective, this can create a convoluted and challenging management process for access control.

To address this challenge, organizations can implement a centralized authorization system that provides a unified approach to managing access across services. By centralizing access control, organizations can establish a single point of control and enforce consistent authorization policies. This eliminates the need for individual services to manage their own access control mechanisms, reducing the potential for inconsistencies and making the overall system more manageable.

A centralized authorization system enables organizations to define access policies based on roles and other attributes, ensuring that the right users have the appropriate level of access. It simplifies the process of granting and revoking access privileges, as changes can be made in a centralized manner rather than having to modify access control configurations for each service separately. Additionally, it allows for better visibility and auditing, such as access logs.

By adopting a centralized authorization system, organizations can mitigate the challenges associated with managing access control across multiple services. It promotes consistency, simplifies administration, and enhances security by providing a centralized mechanism for defining and enforcing access policies.

Top of Form

**1.3 Existing system:**

In the existing system, developers of web services within an organization are responsible for implementing their own authorization mechanisms on their service APIs. This typically involves using established protocols like OAuth or relying on third-party providers such as Auth0 for authentication and authorization functionalities. However, this decentralized approach leads to inconsistencies and additional overhead in managing access control across multiple services.

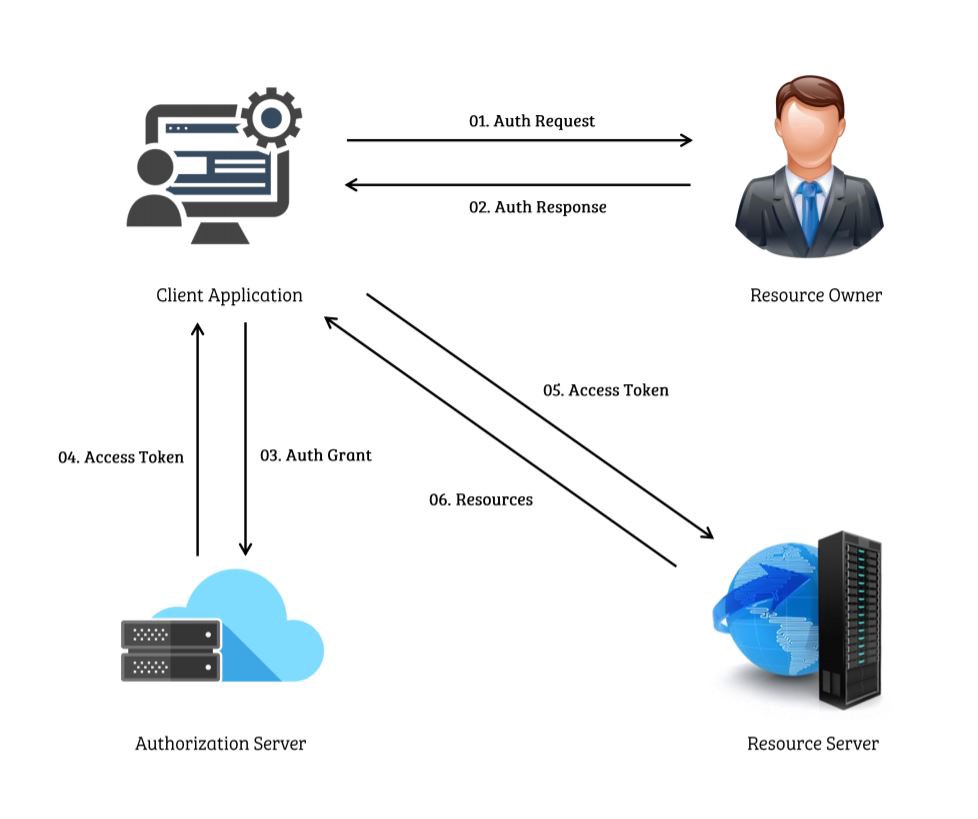
Similarly, each web service developed within the organization needs to implement its own monitoring system to track and manage access control by other services. This means that each service must independently handle logging and auditing of access attempts, maintaining access tokens, and enforcing access policies. This decentralized monitoring approach can result in a lack of unified visibility and makes it challenging to track and analyse access patterns and potential security issues across the organization's services.

To address these issues, a centralized authorization system can be implemented, which eliminates the need for individual services to handle authentication and authorization on their own. By centralizing the authorization process and implementing a unified monitoring system, the organization can achieve consistency, better control, and improved visibility over access control across all services. This streamlines the development process, enhances security, and simplifies the management of access rights and monitoring across the organization's services.

Top of Form

Drawbacks of existing system:

* Lack of centralized and standardized implementation across the organization.
* Increased development time, cost, and resource allocation for individual service implementations.
* Potential security risks and difficulties in addressing known security issues in a decentralized system.
* Limited visibility and centralized monitoring of access control activities.
* Challenges in enforcing consistent access policies and managing permissions across multiple services.



**Fig 1.1**: OAuth Working Example

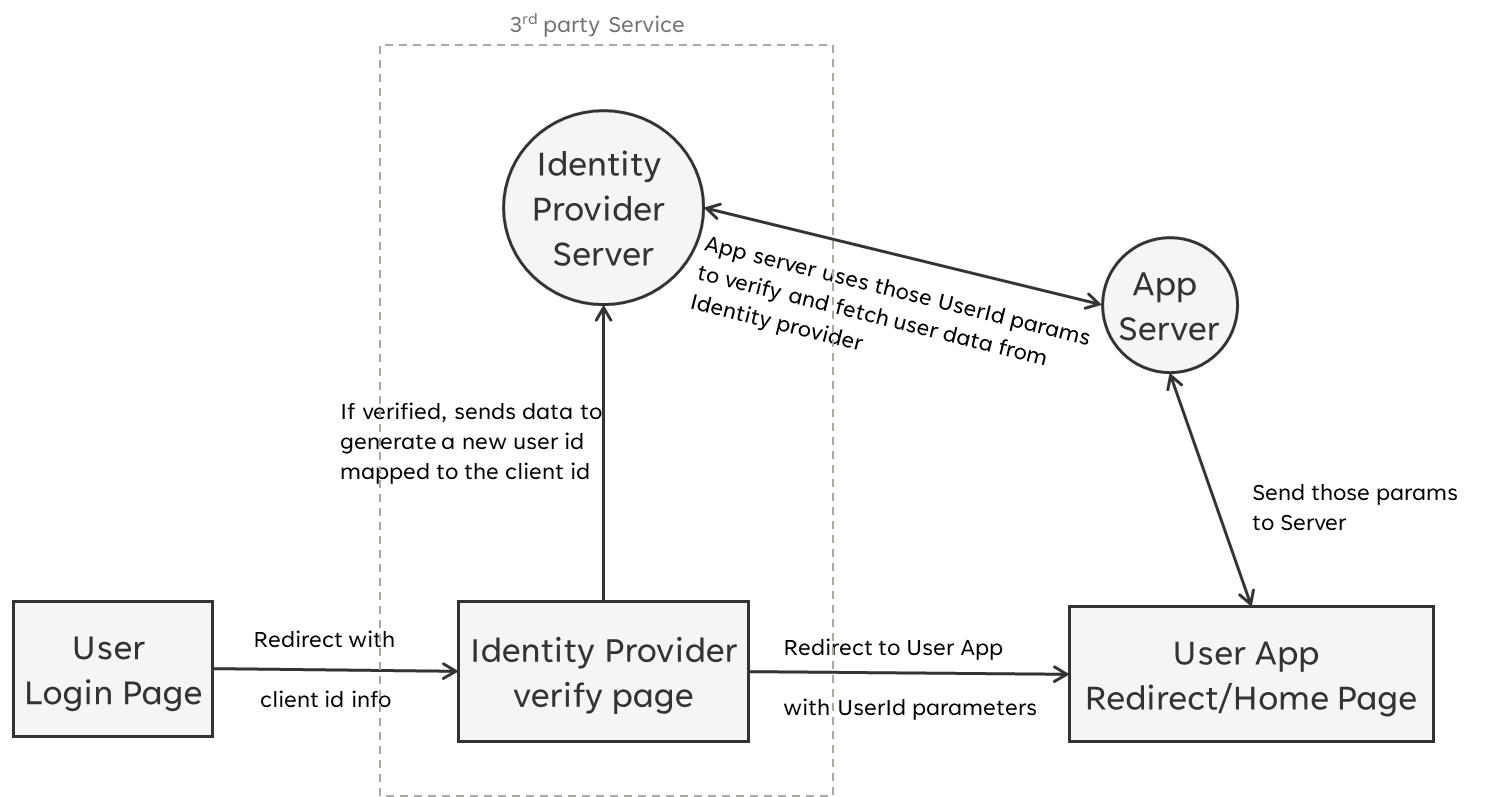
**1.4 Proposed system:**

In this proposal, In the proposed system, we will develop a robust and secure centralized server which also supports Single Sign-On (SSO), OAuth 2.0 principles to streamline authentication and authorization processes across the organization's services. With SSO, users will be able to access multiple services with a single set of credentials, enhancing user convenience and productivity.

The centralized servers will act as the trusted authority for user authentication, generating and managing secure tokens or session identifiers upon successful login. By integrating their client applications and web services with the centralized server, developers can ensure a consistent and reliable authentication experience for users. This server will handle access control and enforce security policies, allowing developers to focus on their core application logic rather than managing authentication mechanisms.

Moreover, the centralized server will provide a comprehensive administration interface, enabling authorized personnel to manage user accounts, permissions, and access policies from a centralized location. This centralization simplifies user management, enhances security, and provides a unified platform for monitoring and auditing access control activities.

By adopting a centralized SSO server, the organization can achieve improved security, reduced development effort, and enhanced user experience through seamless access to services with a single sign-on.



**Fig 1.2**: Proposed Single Sign-on Recipe Architecture

The centralized server enables users to authenticate once and access multiple services seamlessly. With additional features like central monitoring, client-service management, and rate-limiting, the system becomes more robust, enhancing security and operational efficiency. The library offers a strong foundation for implementing authentication recipes like Single Sign-on and improving the user experience in accessing services across the organization.

**1.5 Objectives:**

The main objectives of the proposed system are:

* To develop a Single Sign-on Scheme based Authorization that can be used between multiple applications.
* Centralize Third-Party Access Management: Implement a secure and centralized mechanism for managing access permissions for external entities, such as partners, vendors, or clients. This allows for effective control, monitoring, and revocation of third-party access to organization's services.
* Develop a Library of Functions: Create a reusable library of functions, modules, or APIs that support the centralization objectives. This enables faster and consistent implementation of centralized authorization processes, third-party access management, and related security features across various services.

**CHAPTER 2**

**LITERATURE REVIEW**

**[1] "Information technology of user authentication in cross-platform systems" - V. Krylov, N. Volkova and Y. Kozina**

Authors proposed the information technology of user authentication for cross-platform systems has been developed, which is characterized by high reliability and efficiency. Additional research on the appearance of collisions while generating unique keys for user authentication have been carried out. In conditions of information technology growth, systems which are being developed to be used on multiple platforms, gain in importance and popularity. As a rule, such systems have client-server architecture and have strong security requirements for user authentication.

To develop the information technology of such applications, client-server architecture is used, that is, the network architecture in which tasks or network load is distributed between the service providers (servers) and the customers (clients). Typically, they interact through a computer network using network protocols and are located on different computers. Servers expect requests from the client programs and provide the latter with the resources in the form of data. Clients send requests for various operations containing the user object. Server in its turn works with the database and returns the required data to the clients.

The server component is the core of the entire information system. Almost all the functionality is implemented on the server side, namely creation of users, the authentication system, the system of messages and notes, and work with the database.

To provide access to all these features, the server must implement the mechanism, responsible for processing the HTTP requests from the clients and returning the required data with the help of REST API. In REST API, URI, the analogue of hyperlink in a Web browser, is used for requests generation to server. The server recognizes the URI and the HTTP request method, executes the appropriate actions, and then returns the required data to the client.

**[2] "A Single Sign-On Scheme for Cross Domain Web Applications Using Identity-Based Cryptography" - Y. Wang, Q. Wen, and H. Zhang.**

Authors proposed a novel scheme to satisfy the requirement of cross domain application single sign-on. Identity based encryption and signature have been applied in this protocol, which supports the progress of single sign on between heterogeneous target systems. A ticket can roam from single sign-on domain to the other. The communication between application servers, in this scheme, we use identity-based encryption to protect the data safety; the data transfer between user’s browser and servers we choose session key to prevent attacking.

In general case, when a user accesses the web application, it checks the user’s identity credential locally, that is the authenticate progress is accomplished in the system. Meanwhile, the SOA has been applied in more and more systems, we can separate the authentication and authority module from web applications. We proposed a ticket-based scheme to satisfy the single sign-on requirement between cross domain heterogeneous system.

**[3] "OAuth Web Authorization Protocol" - Barry Leiba**

This paper discuss about Internet identity management is an umbrella that covers several related problems, all which stem from our use of multiple Internet services that come from different providers and reside in different trust domains. For each domain, we have a separate identity and use separate authentication. Where NSTIC seeks to consolidate these identities through central management, software such as password managers tries to make it easier to manage authentication credentials for our various identities.

OAuth addresses this exposure by providing an alternative mechanism through which we can authorize specific actions, and only those specific actions, without giving unrestricted or permanent access. It has the target service create an access token that we can give out that allows only limited access we’ve authorized, perhaps for a limited time or on one-time basis.

The OAuth Working Group is developing a set of token types, that will allow implementations to choose different security characteristics that might be appropriate for different use cases and operational environments. Another working group document describes the “Threat Model and Security Considerations” in some detail.

**CHAPTER 3**

**SYSTEM REQUIREMENTS**

**3.1 Hardware Requirements:**

* Memory: 8GB RAM or above
* Processor: intel 8th gen, AMD Zen 2 or above
* Disk: Minimum 32 GB of free space
  1. **Software Requirements:**
* Operating System: Linux kernel v3.2 / Windows build 1809 or higher
* Processor: intel 8th gen, AMD Zen 2 or above
* Tools and Runtimes: NodeJS v14+, Git

**Chapter 4**

**SYSTEM DESIGN AND ARCHITECTURE**

**4.1 Functional Requirements:**

* **User Registration and Authentication**: The system should allow users to register for an account and authenticate themselves securely. It should support various authentication methods such as username/password, social login, or integration with external identity providers.
* **Standard Authentication Protocol**: A standard process must be followed across the system for authenticating the incoming HTTP requests. This process should follow fixed set of rules and protocols for the consistency.
* **Role-based Access Control**: The system should implement role-based access control (RBAC) to define and manage user roles and their corresponding permissions. Administrators should be able to assign roles to users and configure access control rules based on these roles.
* **Authorization Policies**: The system should provide a mechanism to define and enforce authorization policies for accessing specific resources or functionalities. This may include fine-grained access control based on user attributes, such as user type, department, or organizational hierarchy.
* **Single Sign-On (SSO)**: The system should support single sign-on functionality, allowing users to authenticate once and access multiple services within the organization without needing to re-enter credentials for each service.
* **Token-based Authentication**: The system should utilize token-based authentication, such as JSON Web Tokens (JWT), to securely transmit authentication information between the client and server. Tokens should be issued upon successful authentication and validated for subsequent requests.
* **Access Token Management**: The system should manage access tokens, including token issuance, expiration, and revocation. It should provide mechanisms for refreshing tokens or obtaining new tokens when they expire.
* **User Profile Management**: The system should allow users to manage their profile information, including updating personal details and other account settings.
* **Integration with External Services**: The system should provide integration capabilities with external services or identity providers, allowing for seamless authentication and authorization across different platforms.

These functional requirements form the core functionality of the centralized authorization system, ensuring secure and efficient authentication and authorization processes for users and administrators within the organization.

**4.2 Non-Functional Requirements of the Project:**

* **Security**: The system should prioritize robust security measures to protect user data and prevent unauthorized access.
* **Scalability**: The system should be able to handle increasing user and service demands without sacrificing performance.
* **Performance**: The system should provide fast response times and efficient resource utilization.
* **Availability**: The system should be highly available, with minimal downtime or disruptions.
* **Usability**: The system should have a user-friendly interface and intuitive navigation.
* **Compatibility**: The system should be compatible with different platforms, browsers, and operating systems.
* **Maintainability**: The system should be easy to maintain, update, and enhance.
* **Compliance**: The system should adhere to relevant laws and regulations regarding data privacy and protection.

**4.3 Actors Involved**

The system assumes there are 3 actors involved in the entire authorization process.

**Client Developer**: one who is developing the client application while also authenticating his client application with central auth server beforehand, so they can access all the services they need for their application. Here, the client application can also be a service which the other services and applications can use.

**Auth Server**: or simply called as Authorization Server or in some cases an Authentication Server, if the service provider decides develop authorization part himself, but generally we call it an “Auth Server”. In this project, the server is centralized so there exists only one server that deals with all the auth related stuff, unlike each service having their own auth process built in.

**Service Provider:** are the ones that develop and maintain the web service. This web service provides a public web API which a client application uses to access the resources it needs after being authorized by the auth server. Beforehand, the service provider must authenticate the service to the auth server.

Often there can be a single actor that takes multiple roles. For example, the developer of client applications, auth server and all services can be single individual.

**4.4 Software Development Kit**

The system or in technical term called “Software Development Kit” (SDK) is divided into 2 components, namely client developer libraries and Auth Server Libraries as shown in below figure.

Fig 4.1: Components of SDK

Each of these components or set of libraries provide Application Programming Interface (API) which are reusable functions written in a programming language.

**4.4 Methodology**

We use an authentication protocol that works based on tickets/keys to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Its designers aimed it primarily at a client–server model, and it provides mutual authentication—both the user and the server verify each other's identity.

We use existing cryptographic algorithms, both symmetric and asymmetric in the various phases of authentication. For example, AES-256 and RSA-512 algorithms respectively. While a hashing algorithm is used for key verification and storage. SHA-3 is the current latest and most secure hash algorithm.

The client authenticates itself to the Authentication Server (AS) which issues a key, which is time stamped and contains other metadata for AS to identify and verify the validity of the metadata and returns the base64 encoded result to the client application. the key expires at some point although it may be transparently renewed by the user's session manager while they are logged in.

Server that provides with services is then requested by the AS with a different protocol which also guarantees that request is sent to the right client. This protocol request uses http headers for specifying the exact type of service, which the service can declare and document as a part of their service API documentation.

**CHAPTER 5**

**SYSTEM IMPLEMENTATION**

The Process of implementing is not a straightforward one. To develop a library, we need to identify what are the parts of code that are reused frequently in between multiple implementations of similar requirements. Hence, A prototype is developed which simulates the working of multiple systems communicating between each other in a secure way. This is a case where each system considers rest all the systems as external or third party, hence they can’t be trusted to be given direct access to the data or services.

Initially it started with the implementation of database schema which stores the information about the applications, services, users which we progressively kept on updating through database migrations. The database is linked to a management server where REST APIs are written to securely manage/modify the data in the PostgreSQL database. Additional REST APIs are written for the authentication of incoming requests, processing them, and forwarding them back to appropriate service provider. These REST APIs are initially developed in Rust/Axum based web server, but later due to complexity of writing REST APIs, calling database queries within a low-level language has proven to be a burden, hence we started looking out for alternate efficient ways for much smoother development.  
  
After some investigation, we found out that we could separate the existing monolith server into two separate servers as they are independent. One being a proxy server written in Rust/Axum based server and other being a management/control server which also has power and direct access to control the database schema and data. The proxy server is only limited to read-only access of database. This proven to be effective and allowed us to work each parallelly.  
  
Finally, two sample services and applications are developed as part of simulating the system working together. One is a Custom written OAuth provider i.e. It implements OAuth practices or patterns to allow the other external applications authenticate through itself. And another one is a sample client application which is an already existing application that has a simple username and password scheme like authentication pattern, which we have migrated to use the OAuth provider to authenticate into the application.

**5.1 Technologies used:**

* **Rust**: Rust is a modern systems programming language known for its emphasis on safety, concurrency, and performance. With strong memory safety guarantees and zero-cost abstractions, Rust enables the development of secure and efficient software. We chose Rust for our project to develop the most core part of project i.e. Proxy server which requires faster cold start and high performance
* **Axum**: Axum is a Rust web framework designed for building high-performance, asynchronous web applications using async/await syntax. With its focus on simplicity and composability, Axum simplifies the development of scalable server-side code. We integrated Axum into our project for its low latency and its asynchronous nature through Tokio framework which can handle multiple connections in parallel.
* **SQLx**: A Rust library that offers a type-safe and efficient approach to interact with databases using SQL queries. With support for multiple database backends, it reduces the risk of SQL-related bugs and provides seamless database integration in Rust projects. This is used in our proxy server to query the database.
* **Node.js**: Node.js is a runtime environment that enables server-side execution of JavaScript code with an event-driven and non-blocking I/O model. It is widely used for building scalable and real-time applications due to its large ecosystem and efficient handling of concurrent requests. We chose Node.js for our project to leverage its server-side JavaScript capabilities, enabling us to develop responsive and scalable applications that can handle high concurrency and real-time communication.
* **TypeScript**: TypeScript, as a superset of JavaScript, uses static typing to enhance code maintainability and catch errors during development. By leveraging TypeScript in our project, we can improve code documentation, identify, and prevent potential bugs at compile-time, and deliver high-quality and reliable code. Its widespread usage in modern web development further validates its effectiveness in building scalable and robust applications.
* **Next.js**: Next.js, a React framework, offers automatic code splitting and server-side rendering for efficient and performant web applications. By utilizing Next.js in our project, we simplified development, achieved faster page loads through server-side rendering, and optimized the user experience by loading only essential code. Its performance, scalability, and developer-friendly features made it an ideal choice for our project.
* **Mantine UI**: Mantine UI is a user interface (UI) component library that provides pre-designed and customizable UI components for building web applications. It offers a collection of ready-to-use UI elements, such as buttons, forms, and navigation components, helping developers create visually appealing and consistent user interfaces.
* **Fastify**: Fastify is a highly efficient and low-overhead web framework for Node.js. It focuses on speed and performance by leveraging asynchronous programming and optimized request/response handling. Fastify is designed to handle a large volume of requests efficiently and is suitable for building high-performance web applications. We selected Fastify for our project because as It allowed us to write server code in much simpler TypeScript and integrate with other features.
* **PostgreSQL**: PostgreSQL is a powerful and open-source relational database management system. It is widely used for data-intensive applications and provides excellent performance and reliability. We opted for PostgreSQL as our chosen database management system because of its robustness, advanced features, and excellent performance.
* **Prisma**: Prisma provides an intuitive and type-safe way to interact with databases using a powerful Object-Relational Mapping (ORM) layer. Prisma supports PostgreSQL and enables developers to write database queries in a concise and type-safe manner. Prisma was chosen for our project due to its ability to simplify database access and provide a type-safe and intuitive ORM (Object-Relational Mapping) layer.

**5.3 Database schema:**

The Database that stores main data about every application and service and their owners are stored in a central PostgreSQL database. The schema is designed to be as simple as possible to avoid bringing up complexity for this simple implementation. The PostgreSQL database is connected with an Object-Relational Manager (ORM) to automatically manage database migrations and help write queries in target programming language. In this case, we chose Prisma for its type-safety feature where we declare the schema within a separate file whose syntax is declared by the prisma itself rather than the database.

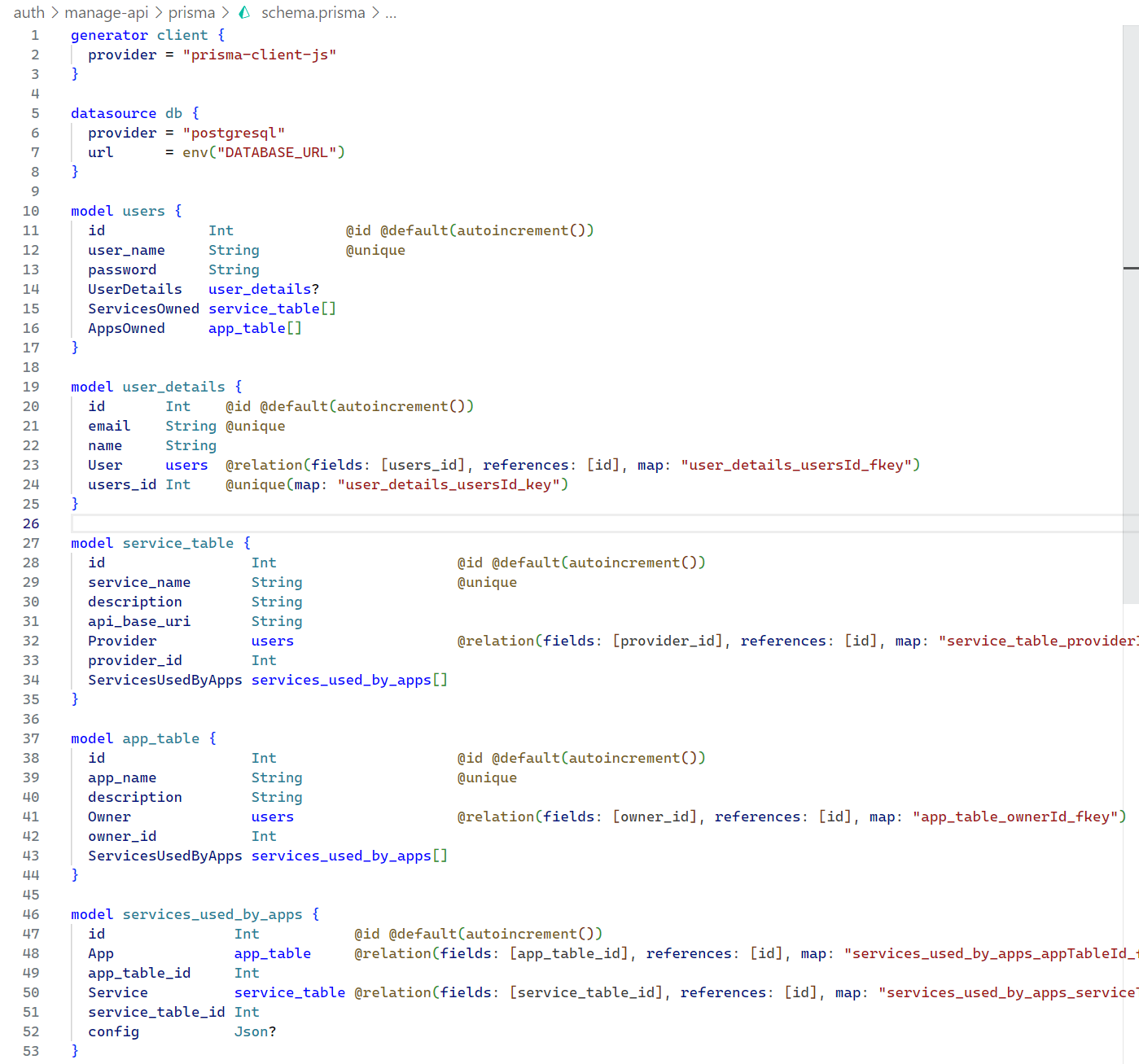


Fig 5.1: prisma.schema file

And Here's a consolidated table summarizing the descriptions of all the tables:

|  |  |
| --- | --- |
| **Table Name** | **Description** |
| *users* | Contains user information such as ID, username, password, associated user details, and owned services and apps. |
| *user\_details* | Stores additional details for each user, including email, name, and a relationship with the *users* table. |
| *service\_table* | Represents services available in the system, stores information like id, name, description, API base URI, and provider. |
| *app\_table* | Represents applications in the system. Contains app information such as ID, name, description and owner. |
| *services\_used\_by\_apps* | Establishes a many-to-many relationship between *app\_table* and *service\_table*, including the additional configuration between each service used by an app. |

Table 5.1: Central Database Table Description

These tables define the structure and relationships of entities within the system, facilitating data storage, retrieval, and efficient management.

**5.3 Proxy Server Implementation**

The job of a proxy server in the current implementation involves redirecting a client request to appropriate service after processing the request and fetching the response back from the service and forward back the response to client along with response headers and body.

We developed a custom protocol upon the HTTP protocol where we declare custom headers which contain the metadata for processing the request while HTTP body as left unprocessed to ensure the privacy. The custom headers include the name of service, redirect endpoint of the request for which the base URL is fetched from the database and additional headers that should be appended to the request before forwarding it to the appropriate service. A bearer authorization token is also necessary to verify the access of the client application to the service.

The following code is taken from the proxy server implementation which is written in Rust Programming language and Using Asynchronous frameworks like Tokio with which Axum server is built upon. This brings low-latency, better memory management and error protection due to the robust nature of the tech stack.

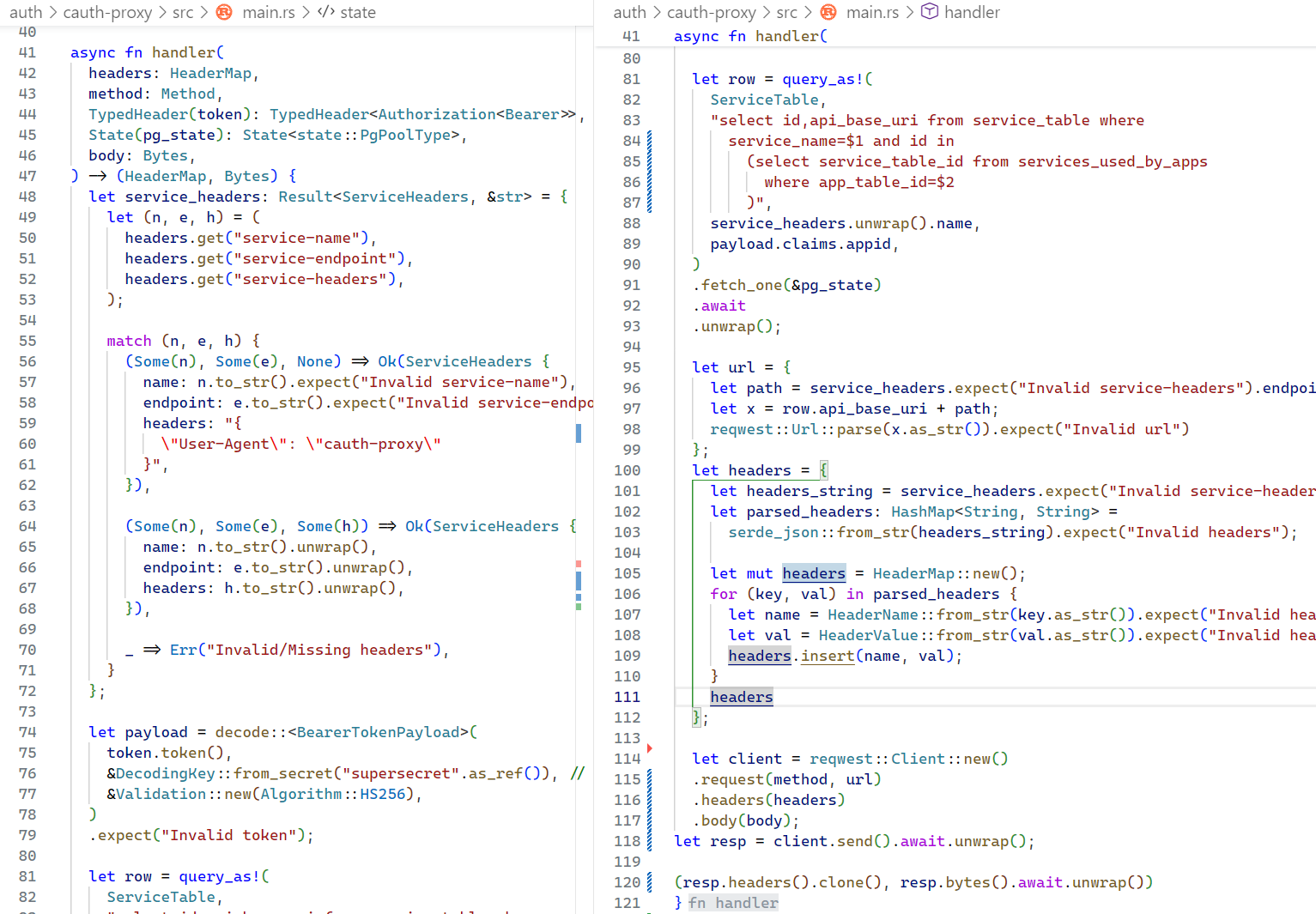
****

Fig 5.2: Proxy Server Request Handler Rust Code

* The *handler* function, which is triggered upon calling /query endpoint takes input parameters including headers, method, token, pg\_state, and body.
* It parses the service headers from the incoming request headers object, which is where client application also declare custom headers before sending it here.
* The bearer token is decoded using a secret, providing access to its payload, which has information about the *appid* which is used to identify the application itself.
* An SQL query is executed using the decoded token where it fetches data from *service\_table*.
* The fetched data is used to construct a URL, combining the endpoint from the headers with the *api\_base\_url* from database. Thus, hiding the API from public view.
* The proxy server then queries the data from the service using the constructed URL and constructs a new Response object using the response from this service query.
* The Response object is then forwarded to the client which handled by the Axum library.

**5.4 Access Management API**

The access management API manages the database directly and it is where we declare the schema for the database through Prisma as mentioned earlier. The database indirectly dictates the behavior of Proxy Server. This is REST API which has various endpoints for managing and authenticating the users accessing this API, adding, and modifying the services and applications added into the database. This API is written in TypeScript programming language using Node.js runtime and uses Fastify library to build the server for the APIs to be accessible across the network.

The following tables provides the description of each endpoint available as part of Access Management API.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | /login | Handles user authentication by checking the provided credentials and Generates a JWT token if valid. |
| POST | /register | Handles user registration by creating a new user account. Hashes the password and returns a JWT token upon successful registration. |
| GET | /whoami | Retrieves the user's profile information using the JWT token. |

Table 5.2: /auth endpoints for Access Management API

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | /service | Creates a new service associated with the authenticated user using the data from request body and returns the created service details. |
| POST | /app | Creates a new app associated with the authenticated user using the data from the request body and returns the created app details including the appid |
| DELETE | /service/:id | Deletes a service with the specified ID. Requires authentication. Returns the deleted service details. |
| POST | /link/:app/:service | Links a service to an app and returns the created link details, including app and service info. Additional configuration is also stored within this link. |

Table 5.3: /manage/create endpoints

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| GET | /:etype/:ename | Retrieves information about service or app entity based on the provided entity type and name. Returns the entity details if found. |
| GET | /all\_owned\_services | Retrieves all services owned by the authenticated user. |
| GET | /all\_owned\_apps | Retrieves all apps owned by the authenticated user. |
| GET | /all\_public\_services | Retrieves information about all public services. Does not require authentication. |
| GET | /app\_access\_token | Generates an access token for the specified app owned by the authenticated user. Returns the generated token and its issuance timestamp. |

Table 5.4: /manage/get endpoints.

The /get/app\_access\_token endpoint generates a token such that only the current server and the proxy server can decode and verify the data, this is a secure way of communicating the data between two different components that are known to interact with each other at some point but until then they are mostly get transported around publicly. The token generated here should be safely managed to avoid leaking just like all the remaining secure tokens.

A screenshot of a computer

Description automatically generated with medium confidence

Fig 5.3: Sample Postman query for all owned apps by a user

**5.5 Access Management Dashboard**

The Access Management Dashboard is a frontend application developed using an Access Management API. It provides a user interface to manage and control access permissions and privileges within a system. Users can utilize the dashboard to view, add, edit, and remove access rights for different resources and entities. The dashboard offers a centralized platform to easily manage user access, ensuring proper security and authorization throughout the system. It enhances the overall security and governance of the application by providing a user-friendly interface for access management operations.

**A screenshot of a computer

Description automatically generated**

Fig 5.4: Creating a new application through Dashboard frontend

**5.6 Sample Service: Info API**

This is the main component or service that provides the ability to perform Single Sign-on authentication. The data from this service is a central repository that provides a valid identity for a registered user just like how other OAuth based Social Logins work (Google, Facebook, GitHub are some the sample existing OAuth identity providers).

This service is a sample component to prove the working of current centralized authentication server implementation, hence it’s registered under a user as his service which other applications can access it by registering themselves within the centralized auth server.

Below are the schema tables for the database used by this service:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | Int | Primary key and auto-incrementing ID |
| name | String | User's name |
| email | String | User's email address (unique) |
| password | String | User's password |
| createdAt | DateTime | DateTime |
| dob | DateTime | DateTime |
| about | String | User's personal description (optional) |

Table 5.5: Table schema for user\_info

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | Int | Primary key and auto-incrementing ID |
| app\_id | Int | Foreign key referencing an app's ID |
| scope | String | Access scope for the app |
| user\_id | Int | Foreign key referencing a user's ID |

Table 5.6: Table schema for app\_data\_access\_info

The user\_info model stores user-specific details such as their name, email, password, creation timestamp, date of birth, and personal description. This data can be accessed by external applications after registration with this service. This table also handles user authentication.

The *app\_data\_access\_info* model contains information about the access scope and the relationship between the current user and the client application. It includes the client app's id as stored in the central database, the access scope, the user\_id which is foreign key to the user\_info table itself.

The following are routes provided by this service:

|  |  |
| --- | --- |
| **Route** | **Description** |
| POST /eaquerydata | Retrieves app data based on the provided EA (External App) token. Validates the token and retrieves the associated app data from the database. |
| POST /accesstokengenerate | Generates an EA token for the authenticated user. Associates the token with the provided scope and appId and stores it in the database for future verification. |
| POST /checkappaccess | Checks if the authenticated user has access to the specified app. If access is granted, generates, and returns an EA token for future requests. |
| POST /info | Updates the user's information with the provided data. Requires authentication. |
| GET /info | Retrieves the user's information. Requires authentication. |
| POST /auth/login | Authenticates a user by verifying their email and password. Generates and returns a JWT token upon successful login. |
| POST /auth/register | Registers a new user with the provided email, password, and name. Generates and returns a JWT token for the newly registered user. |

Table 5.7: List of API routes provided by the Info API

These routes handle various operations related to user information, authentication, and access control. They interact with the Prisma ORM to perform database operations and utilize JWT tokens for authentication and authorization purposes.

The service also provides an Frontend UI to quickly showcase the working of API functionalities, where users can log in to normally using email and password.

**5.7 Sample Application: Fiction Logs**

This is another component which is made to register as an application or client that’s accessing another service. In this case, we use “Info API” as the service accessed by this application. The Application itself is a existing application built by one of the team members as an experiment which has typical JWT based authentication through username and password mechanism.

We have migrated this username and password recipe to OAuth recipe provided by the Info API as part of identity provider API endpoints where a button named “Login with InfoAPI” is provided. Upon clicking this button user is redirected to a external application permission (EAP) grant page

A screenshot of a login screen

Description automatically generated with medium confidence **A screenshot of a computer

Description automatically generated with low confidence**

Fig 5.5: UI for “Login with InfoApp”

**CHAPTER 6**

**DEPLOYMENT AND TESTING**

For deployment, we currently only prioritized only on deployments within local network using Localhost, which is also how the testing is done for all the sample applications after being integrated with central access management server.

Node package manager’s pnpm is used to manage all the multiple projects together. Additional scripts were written to quickly have the servers and applications running.

A picture containing text, font, screenshot

Description automatically generated

Fig 6.1: List of all the commands for all the servers and applications

Once all the servers are running, We used Postman API testing tool to send the requests to various endpoints.

A screenshot of a computer

Description automatically generated with medium confidence

Fig 6.2: Sample query to Info API through Proxy Server

**CHAPTER 7**

**ADVANTAGES & DISADVANTAGES**

**7.1 Advantages of Proposed System**

There are several advantages of using a centralized server for cross-application web API authorization:

* **Centralized Authorization**: Implementing a centralized server simplifies access control management by centralizing the authorization process. This eliminates the need for separate authorization implementations in each service, ensuring consistent and streamlined authorization across the organization.
* **Enhanced Security**: By leveraging a centralized server for authorization, the proposed solution improves security. It enables robust authentication mechanisms and enforces security policies from a central point, reducing the risk of unauthorized access and potential data breaches.
* **Streamlined User Experience**: With a centralized server for authorization, users can enjoy a simplified and seamless experience. They only need to authenticate once to access multiple services, eliminating the need for multiple login credentials. This reduces password fatigue and improves overall usability.
* **Scalability and Flexibility**: The system is designed to accommodate the organization's growth and changing needs. It can easily integrate new services into the centralized authentication system, providing scalability and flexibility as the organization evolves.
* **Centralized Management and Control**: Administrators gain efficient management and control over user accounts, access permissions, and security policies through a single interface. This centralized management capability simplifies administrative tasks and enhances control over the organization's resources.
* **Cost Efficiency**: By utilizing a centralized server, the organization can reduce development effort and costs. The centralized server provides a library of functions and standardized authentication mechanisms, eliminating the need for individual service-level authentication implementations. This results in cost savings for the organization.

By leveraging a centralized server for cross-application web API authorization, organizations can benefit from improved security, streamlined user experience, simplified management, scalability, flexibility, and cost efficiency.

Top of Form

**7.2 Disadvantages of the Proposed System**

There are a few potential disadvantages to using a centralized server for cross-application web authorization:

* **Initial Implementation Complexity**: The implementation of a centralized server for cross-application web API authorization may involve complexities during integration with existing services and configuring the necessary authorization mechanisms. This complexity can pose challenges during the initial setup phase of the system.
* **Dependency on Centralized Server**: The proposed system relies on a centralized server for authorization processes. In the event of server downtime or performance issues, access to all associated services may be disrupted, impacting user experience and productivity.
* **Potential Single Point of Failure**: Since the system relies on a central server, it becomes a potential single point of failure. If the server is compromised or experiences a security breach, it can lead to unauthorized access to multiple services, potentially compromising sensitive data.
* **Compatibility Issues**: Integrating the centralized authorization system with existing services may require adjustments and updates to ensure seamless integration. Compatibility issues can arise if the services have different authorization protocols or if there are legacy systems that are not easily compatible with the proposed system.
* **Increased Dependency on External Components**: The reliance on a centralized server for authorization introduces a dependency on external components such as libraries or frameworks. This dependency can introduce risks such as compatibility issues, versioning conflicts, and potential limitations imposed by the external components.

It is important to consider these potential disadvantages and address them effectively during the implementation and ongoing management of the centralized server for cross-application web API authorization to mitigate any risks and challenges that may arise.

**CHAPTER 8**

**COST EFFECTIVENESS**

Here are the cost-effective benefits of the proposed centralized server for cross-application web API authorization, tailored to the project requirements:

* **Reduced Development Costs**: By utilizing a centralized server and a pre-built library of functions for authentication and authorization, the project minimizes development effort and associated costs. The standardized authentication mechanisms eliminate the need for individual authentication implementations in each service, resulting in cost savings.
* **Streamlined Management Costs**: The centralized authentication system simplifies user account management, access permissions, and security policies. This streamlines administrative tasks and reduces maintenance and support costs associated with managing authentication across multiple services.
* **Enhanced Productivity**: With a centralized server, users can authenticate once and access multiple services seamlessly. This eliminates the need for users to remember and enter credentials for each service individually, improving productivity and reducing time spent on authentication processes.
* **Scalable and Cost-Effective Growth**: The modular and scalable design of the centralized server allows for easy integration of new services. This scalability minimizes the need for significant architectural changes or redevelopment efforts, resulting in long-term cost savings as the organization grows and adds new services.
* **Risk Mitigation**: Implementing a robust centralized server for authorization helps mitigate the risk of data breaches and unauthorized access. By enforcing security policies from a central point, the project reduces the likelihood of security incidents that can be costly to address, both financially and reputationally.

By considering these cost-effective benefits, the proposed centralized server for cross-application web API authorization offers potential cost savings through reduced development effort, streamlined management, enhanced productivity, scalability, and improved security. These factors contribute to the project's overall cost-effectiveness and make it a valuable investment for the organization.

**CONCLUSION**

In conclusion, the proposed project aims to develop a centralized server for cross-application web API authorization that enhances authentication and authorization processes across an organization's services. The project offers several advantages, including improved user convenience, enhanced security, streamlined service management, and potential revenue generation.

By implementing a centralized server for authorization, users can access multiple services seamlessly without the need for repeated logins. This improves the overall user experience and increases productivity. The system also provides centralized authentication and authorization processes, ensuring consistency and simplifying access control management across services.

Additionally, the proposed project includes features such as central monitoring, client-service management, rate-limiting, and the potential for introducing paid services. These features enhance security, enable efficient control and monitoring of access, and provide opportunities for generating revenue.

To meet the project requirements, the system will need a robust server infrastructure, secure authentication mechanisms, effective token management, centralized monitoring capabilities, and seamless integration and compatibility with client applications and web services.

Overall, the proposed project offers a comprehensive solution for implementing a centralized server for cross-application web API authorization, addressing the limitations of existing systems. It provides a solid foundation for improving user experience, security, and service management within the organization. By adopting this solution, the organization can achieve improved operational efficiency, reduced development effort, enhanced security, and a more streamlined and user-friendly authorization system.

**FUTURE ENHANCEMENTS**

Future enhancements for the project can include the following:

* **Multi-factor Authentication (MFA)**: Implement additional layers of security, such as MFA, to strengthen the authentication process. This can include methods like SMS-based verification, biometric authentication, or hardware tokens, providing an extra level of protection against unauthorized access.
* **Single Sign-On Federation**: Extend the system to support federation with external identity providers, allowing users to authenticate using their existing accounts from trusted third-party services like social media platforms or enterprise identity providers. This expands the user base and simplifies the authentication process for users.
* **Advanced Access Controls**: Enhance the authorization capabilities by introducing advanced access control mechanisms like attribute-based access control (ABAC) or role-based access control (RBAC). This provides more fine-grained control over user access permissions based on specific attributes or roles.
* **Integration with Identity and Access Management (IAM) Systems**: Integrate the system with an IAM solution to create a comprehensive identity management framework. This enables centralized user provisioning, deprovisioning, and user lifecycle management, ensuring consistent access control and security across the organization's services.
* **Continuous Monitoring and Threat Intelligence**: Implement real-time monitoring and threat intelligence capabilities within the system to detect and respond to security threats promptly. This includes monitoring user activities, analyzing access patterns, and utilizing machine learning algorithms to identify potential security risks and suspicious behavior.

These future enhancements will further enhance the security, usability, and management capabilities of the system, making it more robust and aligned with evolving industry standards and user expectations.

**REFERENCES**

1. IEEE Papers:
   1. Y. Wang, Q. Wen and H. Zhang, "A Single Sign-On Scheme for Cross Domain Web Applications Using Identity-Based Cryptography", 2010, pp. 483-485,

doi: 10.1109/NSWCTC.2010.120

* 1. B. Leiba, "OAuth Web Authorization Protocol," in IEEE Internet Computing, vol. 16, no. 1, pp. 74-77, Jan.-Feb. 2012,

doi: 10.1109/MIC.2012.11.

* 1. V. Krylov, N. Volkova and Y. Kozina, "Information technology of user authentication in cross-platform systems", 2017, pp. 952-954, doi: 10.1109/IDAACS.2017.8095227.

1. Websites
   1. <https://en.wikipedia.org/wiki/Representational_state_transfer>
   2. <https://en.wikipedia.org/wiki/Kerberos_(protocol)>
   3. <https://oauth.net/2/>
   4. <https://auth0.com/docs/get-started/auth0-overview>
   5. <https://www.rfc-editor.org/rfc/rfc7519>
   6. <https://auth0.com/blog/what-is-an-authentication-server/>