WeCasa

High Level Design Document

Team HAGS JP

Team Lead:

Allison Austin

Team Members:

Githel Lynn Suico

Haley Nguyen

Joshua Quibin

Judy Li

Matthew Chung

https://github.com/githelsui/WeCasa

Date Submitted: December 6, 2022

High Level Design Version Table

Version	**Description**	**Date**
1.0	Initial High Level Design Requirements - System Schematics - Breakdown of Components	10/5/2022
1.1	Content Improvements - Services - Security	10/14/2022
1.2	Content Improvements based on feedback - Frontend - Backend cleanup - Security - Error handling - Logging - Design Decision Justification	10/17/2022
1.3	Content Improvements based on feedback - HLD Diagram - Front-End HLD Design Decision Justifications - Front-End Error Handling, Security, and Logging	12/6/2022

Table of Contents

High Level Design Version Table **2**

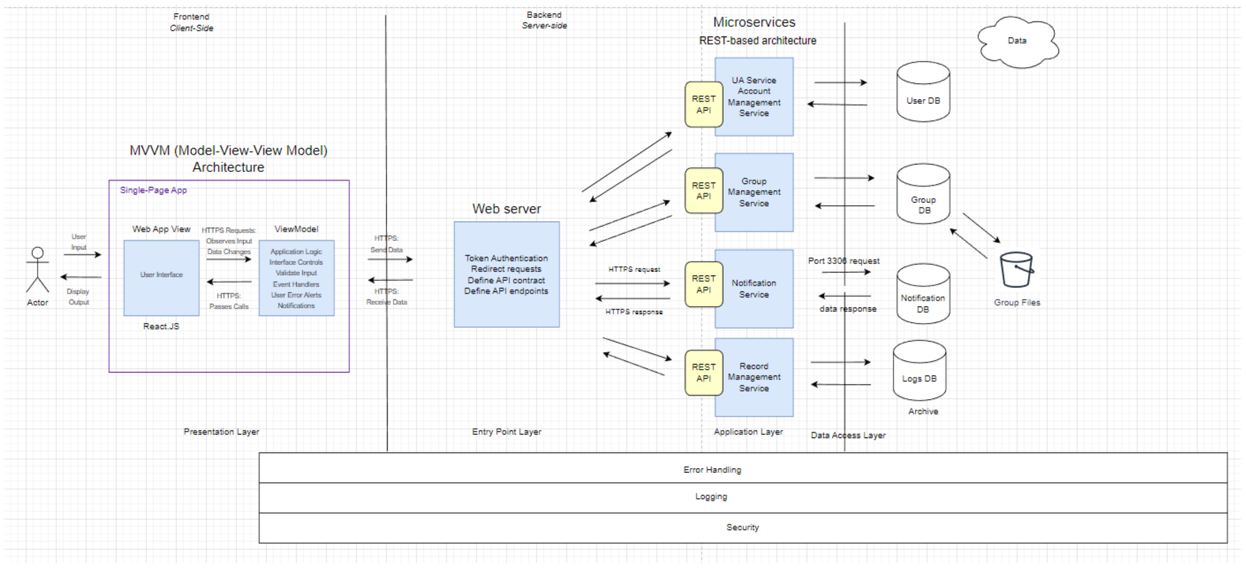
Table of Contents **3**

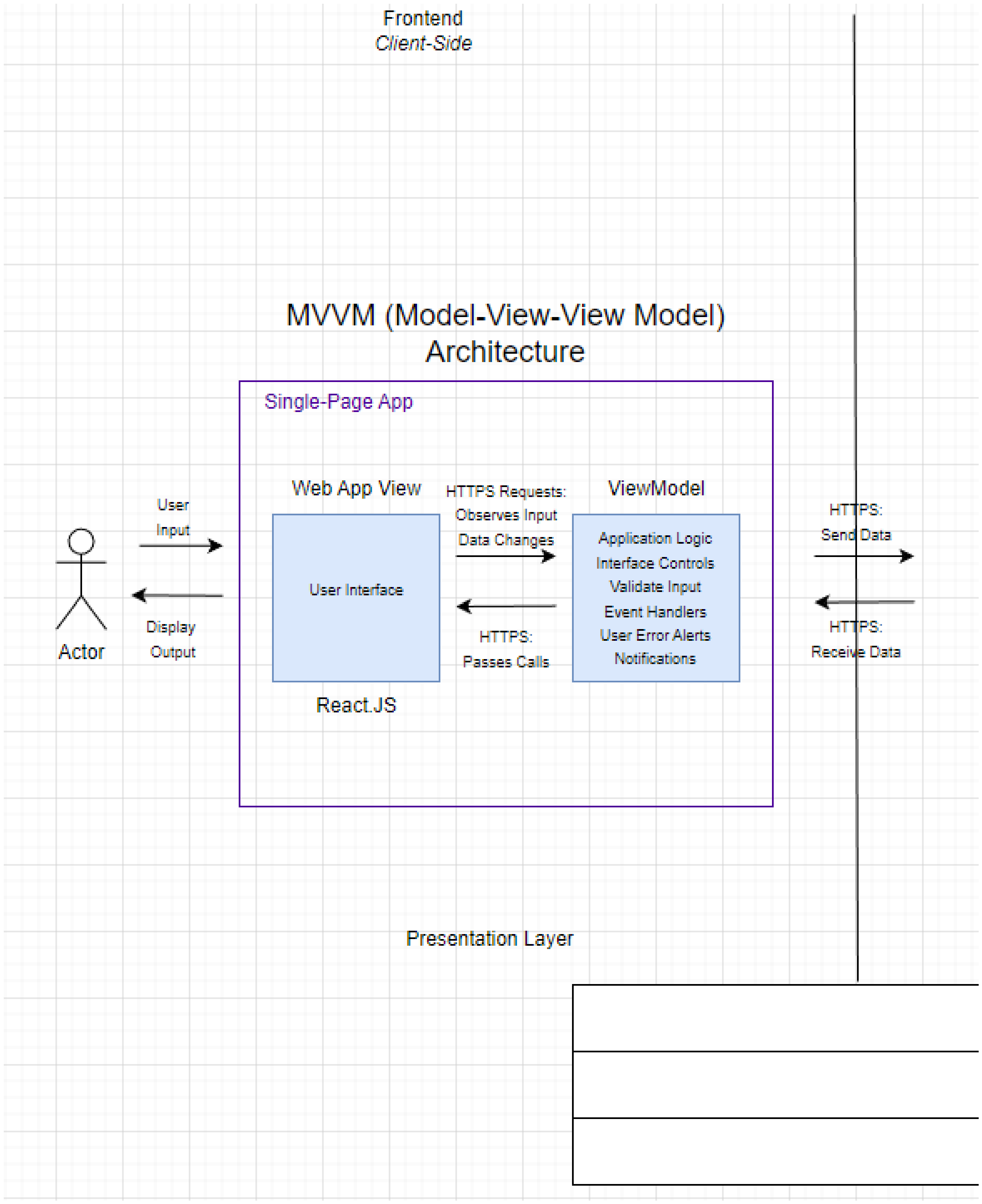
Software Design Overview **4**

- Front-End 5
  - MVVM (Model-View-View-Model) Architecture 6
  - Design Decision 6
- Back-End 8
  - Back-End Container 8
  - Web Server 9
  - Application Layer 9
  - Services 9
  - Data Access Layer 10
  - MariaDB 10
- Security 11
  - System-Wide Security 11
  - Front-End and Client-Side Security 11
  - Server-Side Security 12
  - Database Security 12
- Error Handling 13
  - Front-End and Client-side Error Handling 13
  - .NET Error Handling 14
  - Apache HTTP Server Error Handling 14
  - Database Error Handling 14
- Logging 14
  - Front-End and Client-side Logging 14
  - .NET Logging 14
  - Apache HTTP Server Logging 15
  - Database Logging 15

References **16**

Software Design Overview





Front-End

WeCasa will be implemented as a single-page application (SPA). SPAs load a single page that dynamically rewrites when a user interacts with it, allowing for better performance and user experience over the traditional multi-page application. SPAs are also advantageous as they can be easily converted into a mobile application. However, SPAs are more susceptible to cross-site scripting attacks and data exposure via API.

MVVM (Model-View-View-Model) Architecture

Design Decision

By implementing the MVVM pattern as our front-end architecture, our single-page application will be able to perform at its maximum efficiency and functionality. The decision to implement this specific architecture is based on our project goal's aim to deliver a responsive and efficient web application. Due to the separation of the user interface from the application logic, all of this abstraction will eliminate any redesigns or code overriding for future maintenance. Furthermore, the MVVM architecture allows faster performance as only necessary data will be filtered to be sent or received, instead of fetching everything again (Bonniwell, n.d.).

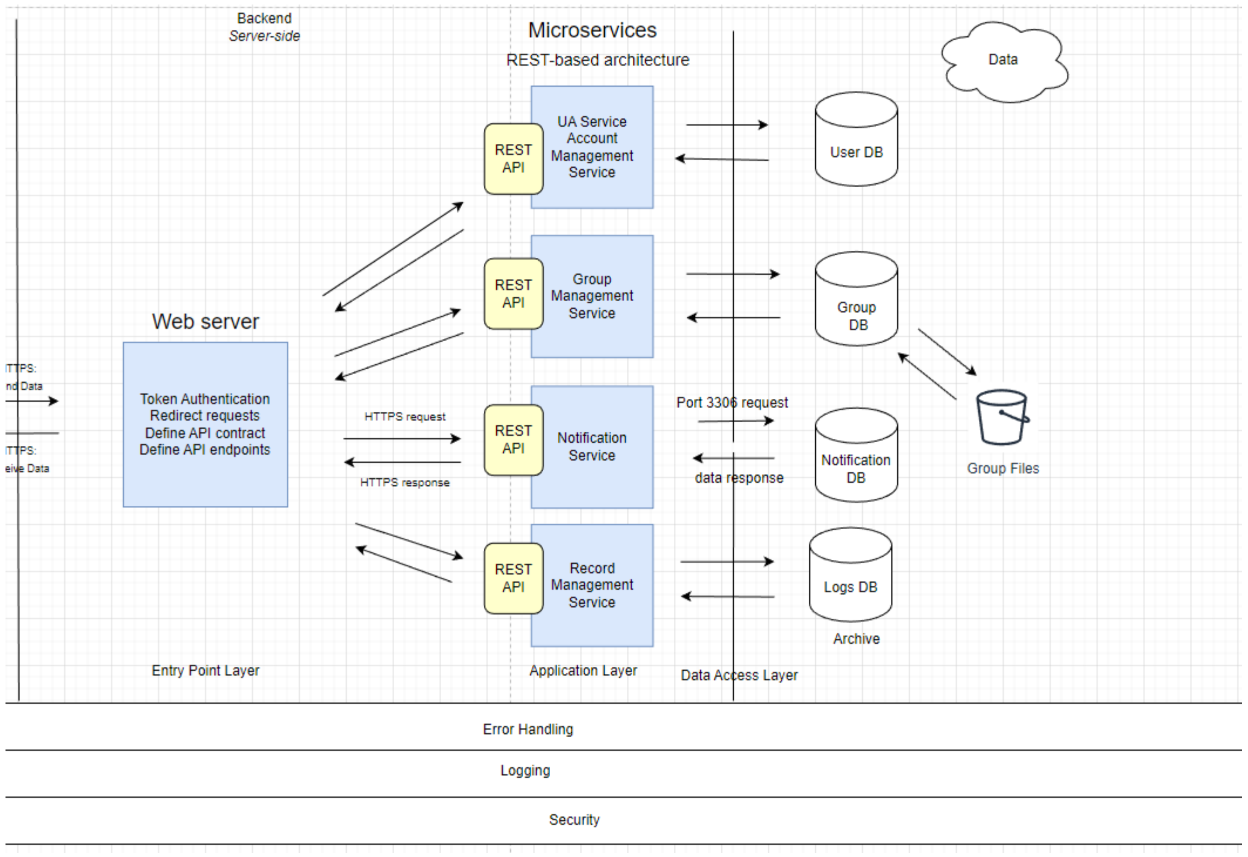
- **Web App View**
    - Clients will be expected to run our application through Google Chrome Web Browser (Version 104.X).

Chrome Browser System Requirements	
Windows	OS: Windows 7, Windows 8, Windows 8.1, Windows 10 or later CPU: Intel Pentium 4 processor or later that's SSE3 capable
Mac	OS: macOS High Sierra 10.13 or later
Linux	OS: 64-bit Ubuntu 18.04+, Debian 10+, openSUSE 15.2+, or Fedora Linux 32+ CPU: Intel Pentium 4 processor or later that's SSE3 capable
Android	OS: Android Marshmallow 6.0 or later

    - The single-page web application (SPA) will use React.JS as its front-end user interface. This is the layer the user interacts with and initially observes user inputs. The user will interact with this layer, sending the inputs to the ViewModel layer.

Web App View Requirements
○ React.JS 18.0.0 Stable Release ○ JavaScript ES6, CSS ○ Npm.js 8.19.2 Stable Release for Extra React.JS Packages ○ Node.js v18 Stable Release

- **ViewModel**
  - The controls for the web app view are housed in the ViewModel, connecting the visual elements of the user interface to the handlers and classes on the client-side. This layer binds together the model with the view acting as a controller for the user interface portion of the application. This component communicates requests to the web server. The following operations are handled in the ViewModel layer:
    - User Input
    - Input Validation
    - Interface Controls
    - User Error Alerts
    - Event Handlers
    - Notifications
- **Model**
  - The Model component will encapsulate domain-specific data and client-side classes. This component holds Model Objects from the client-side which will be binded by the ViewModel and presented to the user in the View layer. This disconnection between the View component and Model component is important for smoother, efficient operations.



Back-End

The backend of the application will follow the microservices architecture. It will be implemented as service-oriented hosted on the AWS (Amazon Web Services) cloud platform. Each feature will be split into one of four modules: User Management, Group Management, Notifications, and Records; Each service will have access to one database. The microservices architecture is more scalable, stable, and has faster deployments compared to a monolithic architecture which is expensive, and less flexible. The downsides include higher resource usage, complex communication between services, and cumbersome global testing. Using AWS will provide our application with limited free cloud infrastructure, and higher availability. However, using cloud computing services means a third party has access to WeCasa's sensitive information, and the application is completely reliant on AWS.

Back-End Container

We will use an Amazon Elastic Computing Cloud (EC2) t3.micro instance to host the following components: web server, authentication server, application layer, and the data access layer. Amazon EC2 provides computing, storage, and networking for our workloads.

This service provides static IPv4 addresses (elastic IP addresses), security groups, and its own virtual private clouds.

Web Server

We will be using Apache HTTP Server (Version 2.4+) to host our application. This server handles all HTTPS requests coming from the user via a web browser. When the user interacts with the interface, the Apache server will accept the request and process the request by routing it to the Application Layer. After processing the request, the server will send a response to the requests with a .html file containing the HTML and Javascript code that will be executed in the browser.

The web server will also be responsible for the identity and access management (IAM) of users before they can access the network. This module will authenticate two types of users, administrators and customers, and grant the correct authorizations. The OAuth 2.0 PKCE (Proof Key for Code Exchange) will be implemented. SPAs cannot store client secrets securely on the frontend because the source is entirely available in the browser. Thus, PKCE is the most secure flow for SPAs because it prevents client secret storage on the frontend. This flow also prevents code injection and cross-site request forgery (CSRF) attacks.

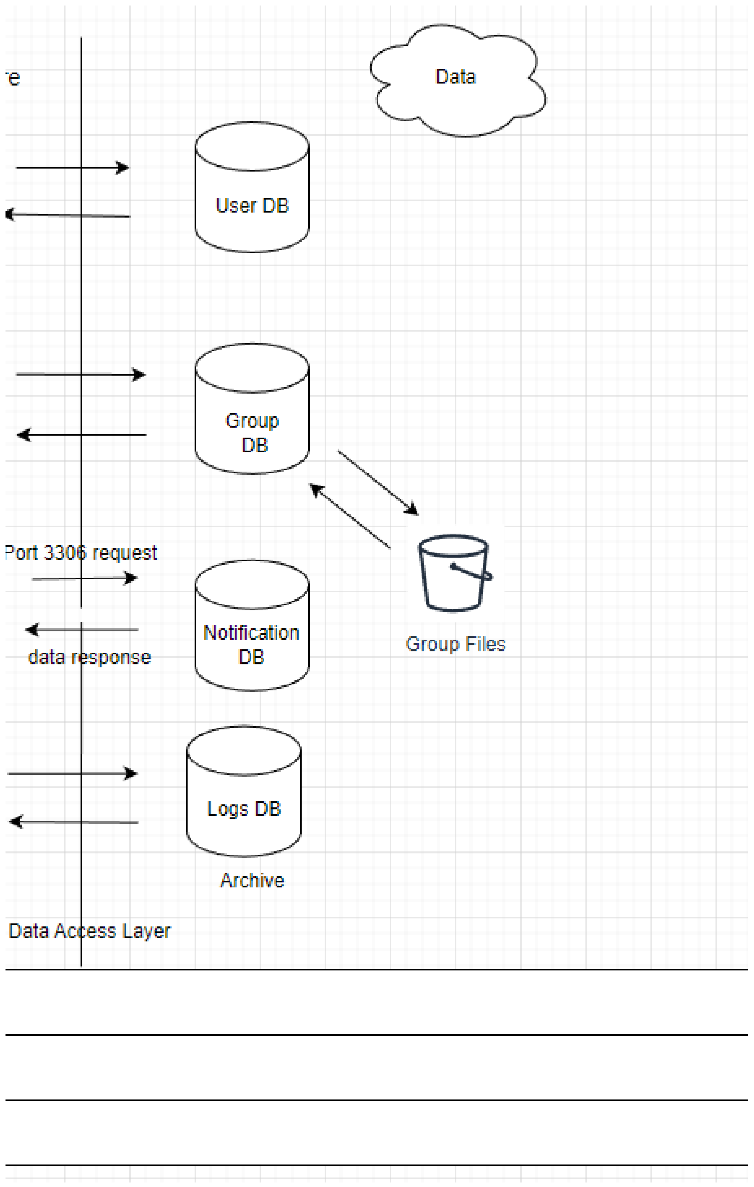
Application Layer

This layer implements the business logic of each specific feature, and/or communicates with the database if necessary. The programming language used in this layer is Javascript. For retrieving data from the database, the Application Layer will route the request to the Data Access Layer.

Services

We plan to implement a Service-Oriented Architecture (SOA) into our design. This will reduce dependency from our features, as well as provide usability for future development. As of now, our services will include User Management, Group Management, Notifications, and Records. These services represent well-defined business functionalities, and some features of the application will rely on one or more services.

Our SOA will be implemented with Representational State Transfer (REST) APIs which communicate through HTTPS URL requests with the GET, PUT, DELETE, and POST verbs.



Data Access Layer

This layer uses methods to open connections to the MariaDB database (Version 10.x), write SQL queries, and format the retrieved data to send back to the Application Layer for further use. This layer ensures loose coupling allowing modifications in the database or application layer without affecting other components.

MariaDB

MariaDB (Version 10.x) will be used and it will interact with the Data Access Layer. MariaDB is a fork of the MySQL relational database management system and will handle all of our data needs.

- **Relational Data**
  - Most of our application data will be stored in relational databases hosted by MariaDB's Management System. These databases will hold tables of user data, group data, and data related to features of our application (budget bars, grocery lists, calendar events, notification settings).
- **Gzip Archive**
  - Logs generated from users, app features, and databases will be stored in an archive storage engine provided by MariaDB. This data store will contain unstructured, compressed files in gzip format. This allows for efficient, read-only access to WeCasa logs.
- **S3**
  - Files uploaded to WeCasa will be stored in Amazon S3 buckets using MariaDB's S3 Storage Engine. S3 provides scalable storage for key-based objects in the cloud, which makes it ideal for handling the file upload feature.

Security

System-Wide Security

The application will employ well-established standards throughout our application to eliminate security risks:

- **Hypertext Transfer Protocol Secure (HTTPS):** HTTPS is a protocol used to encrypt and authenticate data in transit using the TLS protocol. This stops unregistered third parties from intercepting and modifying transported communication. This protocol will be applied to all communication between the frontend and backend components.

Front-End and Client-Side Security

- **Web Application View**
  - **Disable HTML Markups:** By disabling certain HTML elements, this prevents attackers from submitting malicious data, acting as form validation.
- **View Model**
  - **Authentication of React.JS Application:** Ensure the domain header has a realm attribute containing the list of valid users and prompts for login when accessing restricted data.
  - **User Input Sanitation**: This is a method of filtering user inputs to prevent unwanted parties from accessing the web server, database, and other assets. This practice prevents injection and cross-site scripting attacks. React.JS does this by default by escaping JSX values before rendering them.

Server-Side Security

We intend to use Apache HTTP Server (Version 2.4+) as our web server to handle all HTTPS requests in the application. The following layers are to be handled by the Apache web server and the methods of security that will be imposed per level:

- **Web Server Layer**
  - **Authentication:** This layer redirects to an authorization module where an authorization token is generated after the user validates with their username, password and a code verifier.
  - **Authorization:** The authorization module verifies the authorization token and code verifier before returning an ID token, access token and a redirect URL.
  - Disable unwanted Apache HTTP Server modules and server_tokens
  - Control buffer size limits and resources
  - Disable unwanted HTTP methods
  - Utilize ModSecurity as an open-source module for a web application firewall
- **Application Layer (Business Logic)**
  - **Authorization:** The web server makes a request to the correct API with an access token. If this token is valid (it contains the correct claim value), then access to the service is allowed.
  - Ensure the Low Level Design of each feature is detailed and concise
  - Ensure developers and QA testers share the same understanding of the domain that the application serves
  - Avoid making assumptions about user behavior and account for all edge cases
  - and is used as a security clearance before accessing each service. Access tokens will be encrypted with Transport Layer Security (TLS) protocol
- **Data Access Layer**
  - **Authorization:** This layer will contain access lists to ensure a service is allowed to access a certain database.
  - **Granting Permission:** Deny access to all direct query types and update application’s connection string

Database Security

Our database will follow industry standards and best practices in order to mitigate security threats and damage to our network infrastructure:

- **Auditing**
  - MariaDB supports database audit mechanisms and audit logging for the following operations:
    - When a user connects to, fails to connect to, or disconnects from the database

        - When a user executes a query
        - When a query accesses a table
        - When a query modifies the database configuration
        - When a user's Audit Filter contains a Logging Filter that disables audit logging, all audit logging for the user's activity will be skipped.
- **Authentication**
    - MariaDB supports authentication which is performed by database user accounts, specified by user name, host name, and authentication plugins (password validation).
- **Encryption**
    - **Data at Rest** - MariaDB supports data-at-rest encryption, which secures data on the file system. The server and storage engines encrypt data before writes and decrypts during reads, ensuring that the data is only unencrypted when accessed directly through the server.
    - **Data in Transit** - MariaDB supports data-in-transit encryption, which secures data transmitted over the network. The server and the clients encrypt data using the Transport Layer Security (TLS) protocol, which is a newer version of the Secure Socket Layer (SSL) protocol.
- **Privileges**
    - MariaDB supports privileges, also referred to as grants, that are used to enforce security. Privileges are assigned with a GRANT statement to allow access to database objects and to define what the user is authorized to do with the database object. The REVOKE statement is used to remove privileges.
    - Privileges can be aggregated into ROLES to make it easy to GRANT a common set of privileges to a number of users.

Error Handling

An error may be thrown from any level of our application architecture, however the process to handle them will remain contained within the layer it originated from. The following are the different levels within our application and how that layer will handle these errors.

Front-End and Client-side Error Handling

At the highest level of our application architecture, an error may occur within the React.JS components and libraries that serve as our View, ViewModel, or Model layers. In order to handle these client-side errors, the *react-error-boundary* library will be used. Following the concept of error boundaries in React.JS, this library allows the application to display a fallback component that will be shown as an interface when an error is thrown. Furthermore, for error prevention, we will use input validation from the client-side.

Essentially, the user’s input will be monitored, ensuring their submissions obey the set of requirements that may prevent any errors.

.NET Error Handling

Based on the back-end framework we intend to use, .NET, the error handling process will be broken down into three components: tracing, error handling, then debugging. The first aspect, with tracing, either client level or server level, the program execution will be tracked or recorded. Depending on the error, a specific HTTPS status code will be sent from the server.

Apache HTTP Server Error Handling

The Apache web server will handle all HTTPS requests coming from the client-side and user interface. When the Apache web server fails to retrieve a request, we will implement the following error handling process. When the requested content is not available, Apache will have a custom error page to redirect certain error codes to a custom page for the user to view.

Database Error Handling

Error handling for the databases will be done through the MariaDB Management System as well as proper utilization of try catch blocks in our API.

Logging

Front-End and Client-side Logging

The client-side logging library, *loglevel*, will be used to provide more control over the logs we send from our React.JS components. Loglevel will be used to replace console.log and provide log levels and filtering for a more productive workflow. The plugin loglevel-plugin-remote also allows for asynchronous logging by sending logs to a server remotely. The error boundary component will be utilized in order to avoid crashes and instead use handler logic with a fallback UI.

.NET Logging

Based on the back-end framework we intend to use, .NET, the following built-in logging providers will be used for our application: Console and EventSource. With the Console provider, this will log the outputs to the console. Most importantly, the EventSource provider will allow us to write to a cross-platform event source. .NET organizes logging levels as trace, debug, information, warning, error, and critical starting from least severe to most severe. .NET logging will also be done asynchronously.

Apache HTTP Server Logging

The Apache HTTP Server we intend to use has some built-in logging configurations the team will use for the logging of the web server layer of the application. The web server layer will utilize the core logging modules in order to configure the logging of errors, metrics, and the application's processed requests. The Apache HTTP Server error log is set by the ErrorLog directive, which sends diagnostic information and records any errors that are thrown while requests are being processed (*Log Files - Apache HTTP Server Version 2.4*, 2022). Similar to our Frontend, the LogLevel directive in Apache HTTP Server will allow us to specify a log security level for each of our modules. For all requests to access the server, we will use the CustomLog to configure the location and content of the log. Apache HTTP Server 2.4 supports asynchronous logging, which will be used in both the application and data access layer (*Overview of New Features in Apache HTTP Server 2.4*, 2021).

Database Logging

MariaDB Management System has the ability to keep four types of built-in logs with its platform. The application will monitor error logs, general query logs, binary logs, and slow query logs. Error logs relate to the error handling process of the database layer. It will be always enabled so every critical error will be logged in it. General query logs record every successful connection between client and server. Binary logs will record each instance data is altered in the system and slow query logs monitor each query that takes too much time to execute- both which will be used for our key performance indicators. Logging will be done asynchronously to avoid any blocks in our workflow.

References

Access Token. (2022). Oauth.com. *https://www.oauth.com/oauth2-servers/access-tokens/access-token-response/*

ASP.NET - error handling. Tutorials Point. (n.d.). Retrieved October 2, 2022, from https://www.tutorialspoint.com/asp.net/asp.net_error_handling.htm#:~:text=Error%20handling%20in%20ASP.NET,points%20to%20analyze%20the%20code

Bonniwell, J. (n.d.). *The 5W's of MVVM*. Benefit Administration Solutions. Retrieved November 8, 2022, from https://www.sagitec.com/blog/the-5ws-of-mvvm

Chethiyawardhana, M. (2021, December 16). React error handling and logging best practices. Medium. Retrieved October 2, 2022, from *https://blog.bitsrc.io/react-error-handling-and-logging-best-practices-4444c57cd666*

Google Enterprise and Education Help. (2022) support.google.com *https://support.google.com/chrome/a/answer/7100626?hl=en#:~:text=To%20use%20Chrome%20browser%20on,or%20later%20that%27s%20SSE3%20capable*

Log Files - Apache HTTP Server Version 2.4. (2022). Apache.org. *https://httpd.apache.org/docs/2.4/logs.html*

MariaDB Security Documentation. (2022) mariaDB.com *https://mariadb.com/docs/security/*

Overview of new features in Apache HTTP Server 2.4 - Apache HTTP Server Version 2.5. (2021). Apache.org. https://httpd.apache.org/docs/trunk/new_features_2_4.html

S3 Storage Engine. (2022). MariaDB KnowledgeBase. *https://mariadb.com/kb/en/s3-storage-engine/*

Stoplight. (2022). Stoplight. *https://stoplight.io/api-types/soap-api*

Proof Key for Code Exchange. (2022). oauth.net. https://datatracker.ietf.org/doc/html/rfc7636