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

Date Submitted: October 5, 2022

High Level Design Version Table

Version	Description	Date
1.0	Initial High Level Design Requirements - System Schematics - Breakdown of Components	10/5/2022

Table of Contents

High Level Design Version Table **2**

Table of Contents **3**

Software Design Overview **4**

Front-End 4

MVVM (Model-View-View-Model) Architecture 5

Back-End 6

Entry Point 7

Web Server 7

Authentication Server 7

Application Layer 7

Data Access Layer 7

MariaDB 8

Security 9

System-Wide Security 9

Client-Side Security 9

Server-Side Security 9

Database Security 10

Error Handling 11

Client-side Error Handling 11

.NET Error Handling 11

NGINX Error Handling 11

Database Error Handling 11

Logging 12

Client-side Logging 12

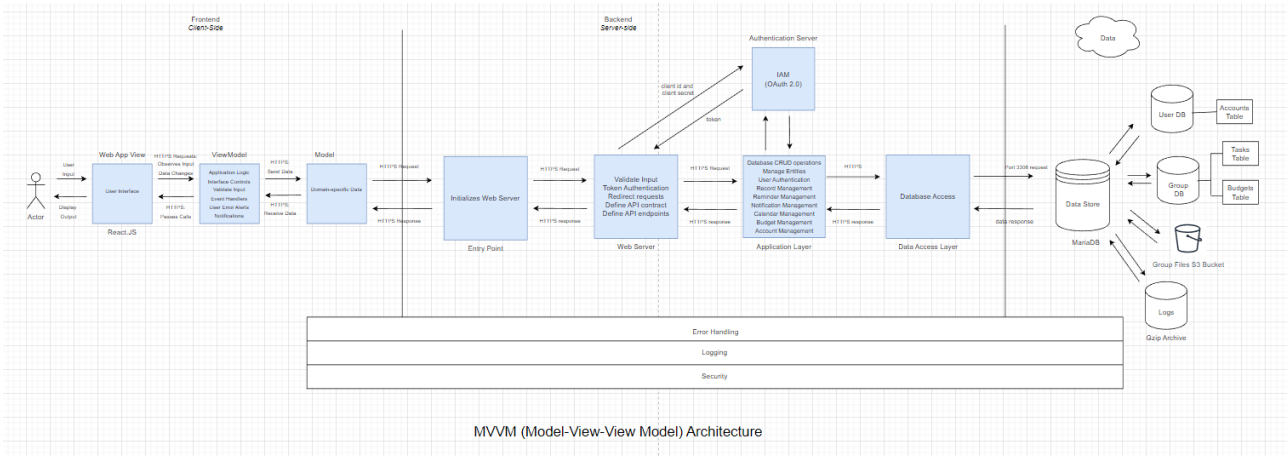
.NET Logging 12

NGINX Logging 12

Database Logging 12

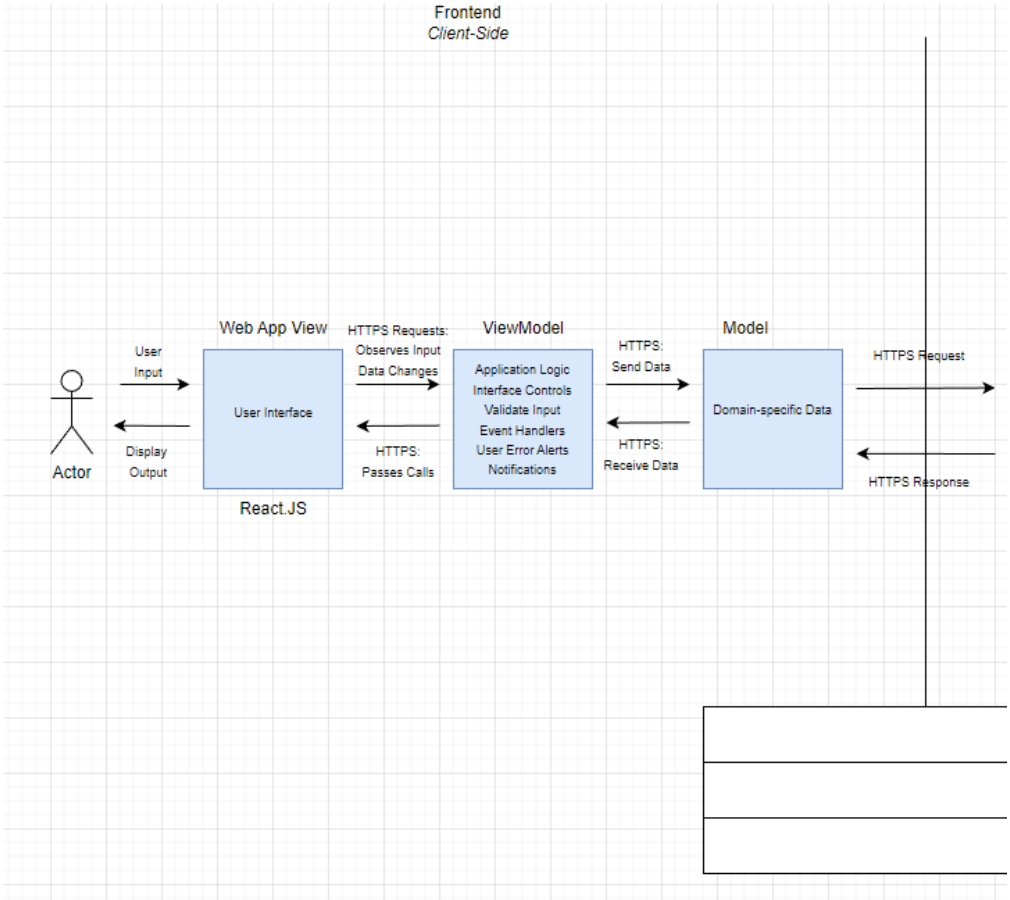
References **13**

Software Design Overview



MVVM (Model-View-View Model) Architecture

Front-End



MVVM (Model-View-View-Model) Architecture

- **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

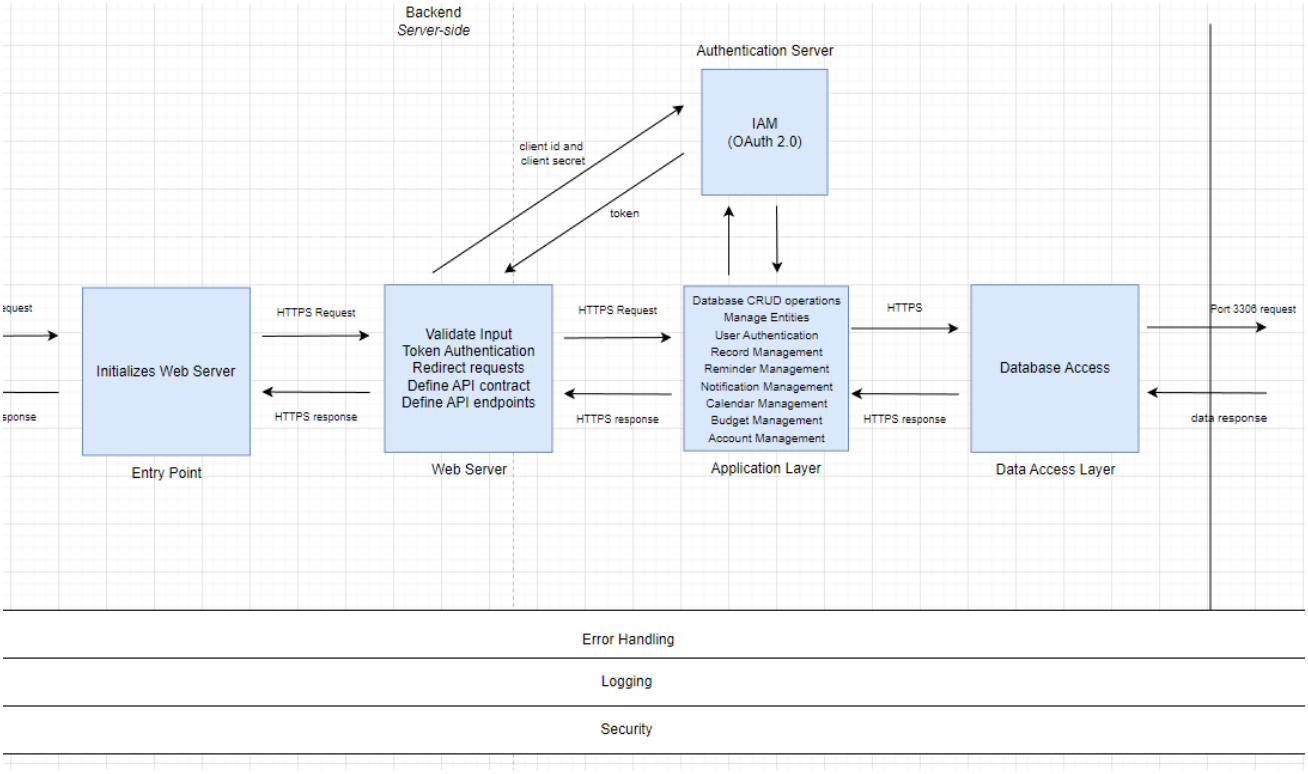
- **ViewModel**
    - The controls for the web app view are housed in the ViewModel, connecting the visual elements of the user interface to the handlers and classes on the client-side. This layer binds together the model with the view acting as a

controller for the user interface portion of the application. 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



Entry Point

The entry point to an application is the gateway between the user interface and the backend. This layer redirects the user to the correct version, user or administrative access, of the application by initializing the corresponding web server.

Web Server

We will be using Nginx web server (Version 1.22+) 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 Nginx 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.

Authentication Server

The authentication server is responsible for the identity and access management (IAM) of users before they can access the network. This server will authenticate 2 types of users, administrators and customers, and grant the correct authorizations. The OAuth 2.0 Client Credential flow will be implemented:

- When the browser initiates an authentication request, the web server redirects the user to a sign in or sign up page.
- Once the user signs in, an access code is generated and used to allow the user to call protected resources and APIs.
- Access tokens will be encrypted with Transport Layer Security (TLS) protocol and will expire after 3 hours.

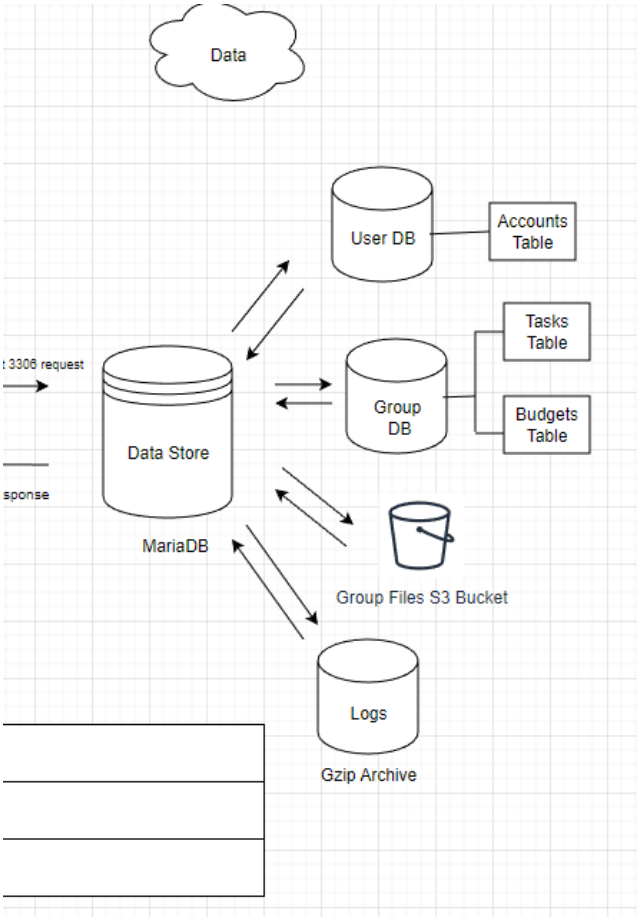
We will be using NGINX modules to validate client id, client secret and access tokens.

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.

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.



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.

Client-Side Security

- **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.
- **Log In/Log Out:** Users must authenticate with a username and password through the authentication server before accessing the application network.
- **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.
- **Disable HTML Markups:** By disabling certain HTML elements, this prevents attackers from submitting malicious data, acting as form validation.

Server-Side Security

We intend to use Nginx (Version 1.22+) as our web server to handle all HTTPS requests in the application. The following layers are to be handled by the Nginx web server and the methods of security that will be imposed per level:

- **Web Server Layer**
    - Disable unwanted Nginx 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)**
    - 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
- **Data Access Layer**
    - **SPROC (Stored Procedures):** Prevents injection attacks
    - **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.

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.

NGINX Error Handling

Nginx web server (Version 1.22+) will handle all HTTPS requests coming from the client-side and user interface. When the Nginx web server fails to retrieve a request, we will implement the following error handling process. When the requested content is not available, Nginx 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

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.

.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.

NGINX Logging

The Nginx platform 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 Nginx Open Source and Nginx Plus in order to configure the logging of errors, metrics, and the application's processed requests.

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. 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.

References

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

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

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

S3 Storage Engine. (2022). MariaDB KnowledgeBase.

https://mariadb.com/kb/en/s3-storage-engine/

NGINX Documentation. (2022). Nginx.com. https://docs.nginx.com/

OAuth 2.0. (2022). https://oauth.net/2/