Table of Contents

Contributors

- Cordus Bailey
- David Jara
- Ben Mesiter
- Ethan Dunning
- William Francis
- Lucas Feazel
- Luke McMillian

Application Overview

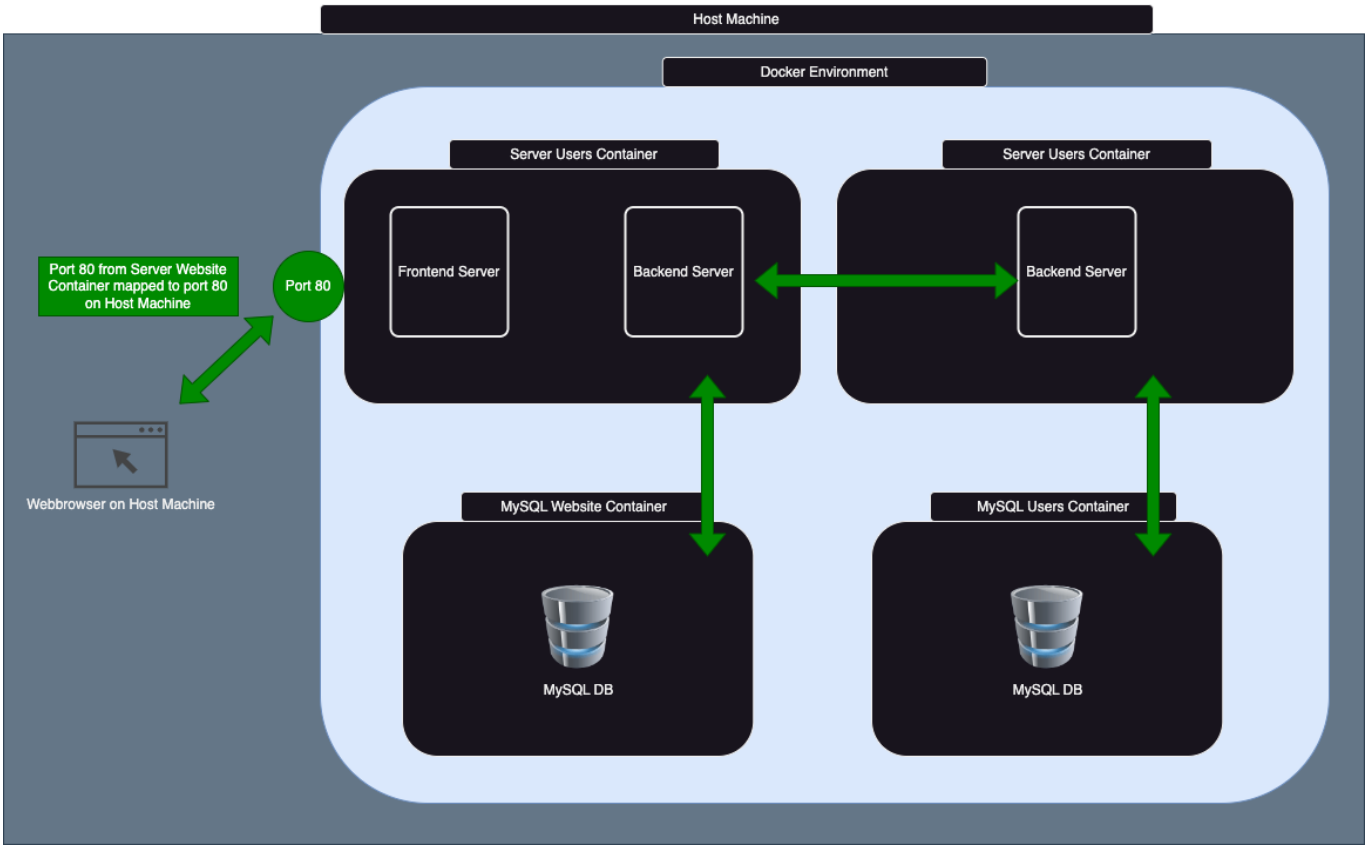
The Node.js-based web application is designed to handle user authentication, secure password storage, and database queries. The application comprises multiple containers for modularity and scalability.

To follow the modular route, this website uses CSR(client side rendering) to achieve dynamic page loading on the frontend. Rather than switching to a completely different html file to view a different page, the html of the new page is injected into the index.html for the frontend to display. This allows us to be modular with our webpages and to keep consistence across the different pages using our universal styles.css file.

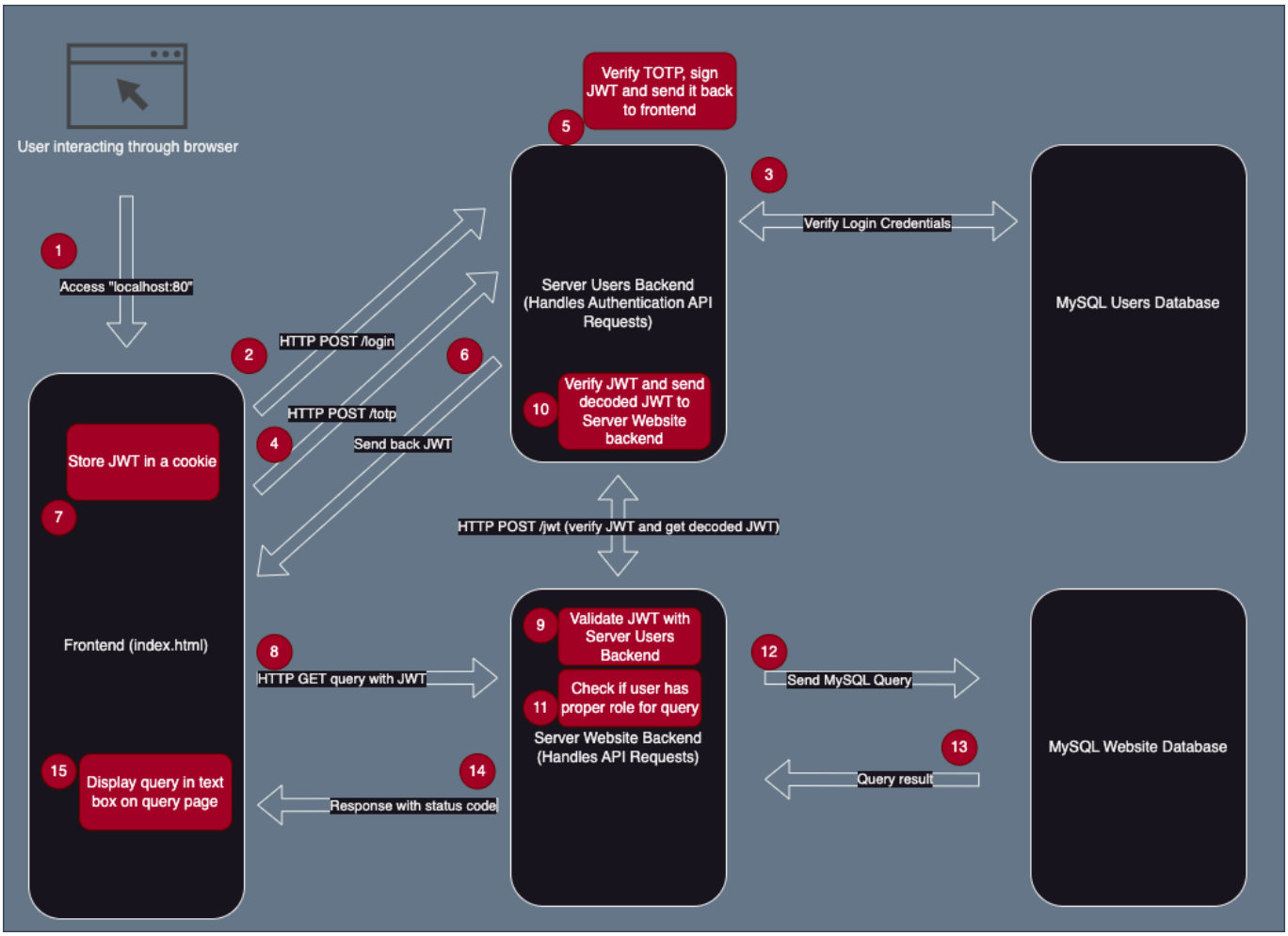
The main page consists of a navbar or header, a page-content section where the new html is injected, and a footer for the page.

NodeJS Application Structure and Data Flow

The structure for this application is depicted below in the diagram. This diagram shows how the differnt docker containers are layed out and how they communicate with each other. This provides the structure of our application by showing how everything is linked together.



The data flow for this application is shown in the diagram below. There are numbered steps to help guide through the process of how data is passed around the system. The diagram goes into detail on how the differnt containers interact and what they are doing with each other. This gives deeper insight on how the data gets from one point to the next and where checks are conducted to verify the user has logged in or has the proper role to conduct a query.



Dependencies Modules

The web application depends on the following modules:

1. **MySQL** - Database interaction.
2. **Express** - Backend framework.
3. **unirest** - Simplified HTTP requests.
4. **cors** - Cross-Origin Resource Sharing.
5. **bcrypt** - Password hashing and comparison.
6. **crypto** - Cryptographic functions.
7. **jsonwebtoken** - JWT token generation and verification.
8. **express-session** - Session management.

Adding Pepper to Passwords

The pepper used by the web server is defined by an environment variable called `PEPPER`. This is configured in the Docker Compose file for the `server-users` container:

```
environment:
    - HOST=0.0.0.0
    - PORT=80
    - MYSQLHOST=mysql-users
```

```
    - MYSQLUSER=root
    - MYSQLPASS=example
    - PEPPER=ef79
```

Bcrypt Integration

The `bcrypt` library is used for password hashing and comparison, enhancing security with salt and pepper. The dependency is reflected in the `server/package.json` file:

```
"dependencies": {
    "bcrypt": "^5.1.1",
    "express": "4.18.2",
    "mysql2": "2.3.3"
}
```

JWT Integration

The `jsonwebtoken` (JWT) library is used to generate and verify JSON Web Tokens for user authentication. JWTs are issued upon successful login and TOTP verification, stored on the frontend in cookies, and validated on the backend for secure communication.

Generating JWT in backend

Once a user logs in and provides a valid TOTP code, the backend generates a JWT and sends it to the frontend. Below is an example of the Javascript for this process

```
// Get information about current user account
const { username, email, role } = results[0];

// Generate a JWT containing the username, email, and role
const token = jwt.sign(
    { username, email, role },
    JWTSECRET,
    { expiresIn: '1h' } // Token expiration time
);

//If created token successfully send 200 response with the token
return response.status(200).json({
    message: 'TOTP verified successfully',
    token: token
});
```

Storing JWT on frontend

Once the JWT is generated, signed, and sent to the frontend to be stored in a cookie. Below is the relevant JavaScript code for this process:

```
// If response is 200, success login
if(response.status == 200) {
    return response.json().then(data => {

        // Get token from response and save as JWT constant
        const JWT = data.token;

        // Sets JWT token value into jwt cookie
        document.cookie = `jwt=${JWT}; path=/;`;

        // Navigate to where the user should go
        window.navigateTo('/home');
    });
}
```

Retrieving JWT for Queries

The JWT is retrieved from the cookie on the frontend when making authenticated requests, such as querying the backend. The following code snippet demonstrates this:

```
// Access the cookie and get the JWT
const JWT = document.cookie
    .split('; ')
    .find(row => row.startsWith('jwt='))
    ?.split('=')[1];

// Make a query request with the JWT stored in the cookie
fetch("http://" + parsedUrl.host + "/query", {
    method: "GET",
    credentials: "include", // Needed to pass cookies from session
    headers: {
        'Authorization': `Bearer ${JWT}` // JWT token from cookie
    },
    mode: "cors",
});
```

Backend JWT Verification

The backend validates the JWT sent with requests to ensure the user is authenticated and authorized. The following code verifies the token:

```
const authHeader = request.headers['authorization'];
if(!authHeader) {
    return response.status(401).send("Missing Authorization Header");
```

```
}

const JWT = authHeader.split(' ')[1];
if(!JWT) {
    return response.status(401).send("Invalid Authorization Header
Format");
}

// Verify the JWT using the secret stored in the backend
const verificationResult = await verifyJWT(JWT);
if(!verificationResult || !verificationResult.role) {
    return response.status(403).send("Token verification failed");
}
```

This process ensures only authenticated users with valid tokens can access restricted resources.

Roles and Permissions

User roles define the level of access a user has within the system. Currently, the application supports two roles:

1. **user**: Default role assigned to all users in the `mysql-users` database.
2. **admin**: Elevated role with additional permissions.

Role-Based Access in Queries

After validating the JWT, the backend extracts the role from the decoded token and checks it against the list of allowed roles for the requested operation. Below is an example of how roles are verified for the query endpoints:

```
// Verify the JWT with the server-users backend
const verificationResult = await verifyJWT(JWT);
if(!verificationResult || !verificationResult.role) {
    return response.status(403).send("Token verification failed");
}

// List roles allowed to use this query
const allowedRoles = ['admin', 'user'];
console.log("User role:", verificationResult.role);

// Check if the user has the correct role
if(allowedRoles.includes(verificationResult.role)) {
    // Conduct the query
}
```

The following table shows the required role needed to use each of the exist queries:

Query Endpoint	Required Role

Query Endpoint	Required Role
"/query"	"user" only
"/query2"	"user" or "admin"
"/query3"	"admin" only

Endpoints

Backend Server Users

Endpoint Name	Purpose
`/login`	Verifies user credentials.
`/totp`	Verifies TOTP codes during login.
`/jwt`	Checks the validity of JWT tokens for queries.

Backend Server Website

Endpoint Name	Purpose
`/query`	Allows users with the `user` role to query a table.
`/query2`	Allows users with the `user` or `admin` role to query a table.
`/query3`	Restricted to users with the `admin` role for database queries.

Docker Compose

The Docker Compose file defines and configures services for the application, including website, backends, and MySQL databases.

Handling Website Crashes

To ensure the MySQL databases are fully operational before starting the servers, health checks are implemented:

MySQL Health Check

```
healthcheck:
    test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
    interval: 10s
    timeout: 5s
    retries: 3
```

Server Dependency on Database Health

```
depends_on:
    [Name of corresponding database container]:
        condition: service_healthy
```

Application Structure Overview

Containers

The application consists of four containers:

1. **server-website**: Handles the frontend.
2. **server-users**: Handles user authentication.
3. **sql-users**: Manages user account data.
4. **sql-website**: Manages general application data.

server-website

This container contains the frontend, structured for dynamic content loading into `index.html`. It includes:

- **index.html**: Main webpage.
- **index.js**: Manages interactions and dynamic content loading.
- **styles.css**: Centralized styling.
- **/pages**: Contains folders for different contents/pages (e.g., `home`, `login`, `register`, `totp`). Each folder includes an `.html` and `.js` file for page-specific functionality.

server-users

The `server-users` container is the backend for user authentication, utilizing Express, JSON Web Tokens, and bcrypt for secure processes. It defines API endpoints for login, registration, and TOTP verification.

sql-users

The `sql-users` container runs a MySQL server with the `users.sql` file to create a database and a `users` table. The table includes:

Username	Password	Role	Salt	Email

- **Username**: Primary key.
- **Password**: Hashed with bcrypt, salt, and pepper.
- **Salt**: Randomly generated during registration.
- **Email**: User's linked email.

sql-website

The `sql-website` container runs a MySQL server with the `website.sql` file to create multiple queryable tables. Example tables:

things

thing1	thing2	thing3	thing4

super_secrets

secret1

normal_secrets

secret1

Steps Required to Access the Information

1. Run the Docker containers using `docker compose up`.
2. Open a browser and navigate to *localhost*.
3. Click the *login* button.
4. Enter valid credentials to sign in.
5. Generate a TOTP code using the standalone Node.js script.
6. Enter the TOTP code on the redirected page.
7. Select a query page from the home page.
8. Click the *Query* button to execute queries (requires appropriate permissions).
9. Terminate the Docker containers with `docker compose down`.