# Food Brain – Documentation

## Outline

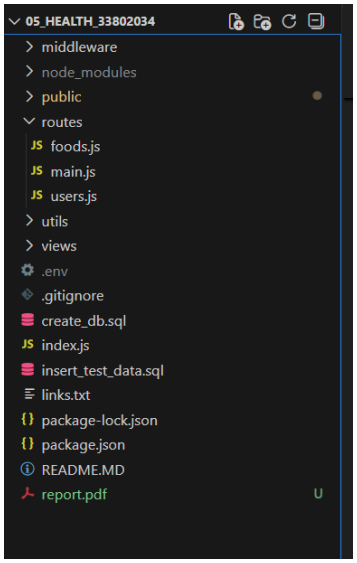
Food brain is a personal food logging app, allowing users to track their current diets by logging the food they eat every day. This is then presented to them by day, with the total number of food items and total calories consumed displayed nicely. If a user struggles with finding out about the nutritional value of the foods they are eating, they can also easily check this thanks to the implementation of Fatsecrets API, providing a vast library of food data.

Other than this, food brain features efficient data handling techniques, automatically saving and incrementing users' food data: During logging, it essentially gives users access to their own mini databases of favourite food items alongside their nutritional value, efficiently storing and logging them by day, denying wasteful food copies in the database whilst logging them effectively.

Food brain also provides basic and secure authentication systems, using bcryppt hashing, secure sessions, appropriate validation, sanitations and sql queries to further ensure secure and reliable data handling.

## Architecture

Food Brain uses a two-tier architecture consisting of an application tier and a data tier. The application tier is built with Node.js and Express, handling routing, authentication, validation, and communication with external services. EJS templates provide server-side rendering for all user-facing views. The data tier uses MySQL, accessed through a mysql2 connection pool defined in index.js, storing users, foods, and daily logs. FatSecret integration uses an OAuth 2.0 client-credentials flow implemented in utils/api.js to fetch nutritional data. Sessions are maintained using the default cookie-based session mechanism for user authentication.




## Data model

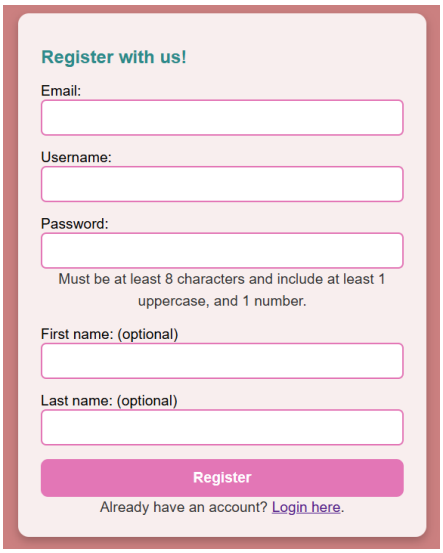
The data model centres around the users table, which stores account and authentication data. Each user maintains their own set of foods, while the food_log table links users to the foods they consume on specific dates, enabling daily summaries. I was planning on implementing more features to this app, such as fave_foods to add food logs more quickly and daily goals for users to achieve, however I ran out of time to implement these. auditlog records login attempts and security-related events for administrative visibility. This structure ensures efficiency and avoids duplication, foods created once per user and referenced through logs, supporting efficient querying of daily intake, calorie totals, and long-term dietary behaviour.

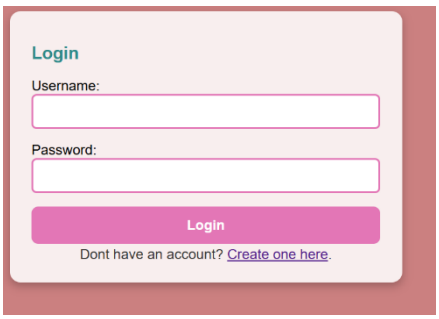
```
mysql> describe users;
+---------------+--------------+------+-----+-------------------+-------------------+
| Field         | Type         | Null | Key | Default           | Extra             |
+---------------+--------------+------+-----+-------------------+-------------------+
| id            | int          | NO   | PRI | NULL              | auto_increment    |
| username      | varchar(50)  | NO   | UNI | NULL              |                   |
| first_name    | varchar(100) | YES  |     | NULL              |                   |
| last_name     | varchar(100) | YES  |     | NULL              |                   |
| email         | varchar(255) | NO   |     | NULL              |                   |
| password_hash | varchar(255) | NO   |     | NULL              |                   |
| created_at    | timestamp    | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
| current_streak | int         | YES  |     | 0                 |                   |
| longest_streak | int         | YES  |     | 0                 |                   |
| last_goal_date | date        | YES  |     | NULL              |                   |
+---------------+--------------+------+-----+-------------------+-------------------+
```

**User functionality**

1. Registration and Login

Users can create an account via /users/register and log in through /users/login. These also validate and sanitise user inputs while displaying useful error messages if a user gets stuck. Passwords are securely hashed using bcrypt (routes/users.js), and all authentication attempts,



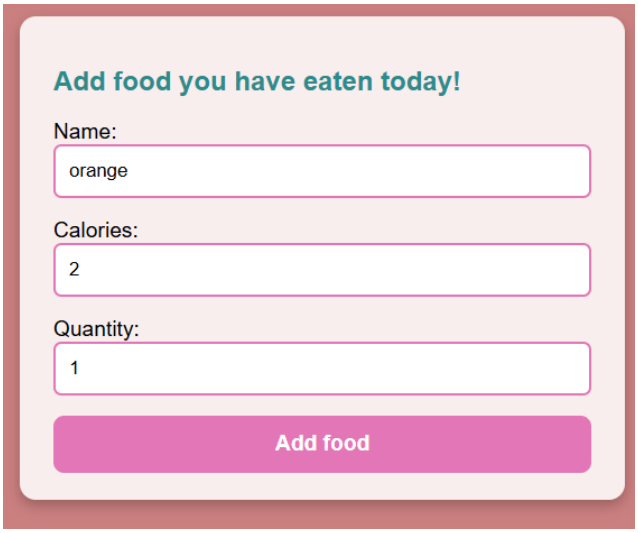



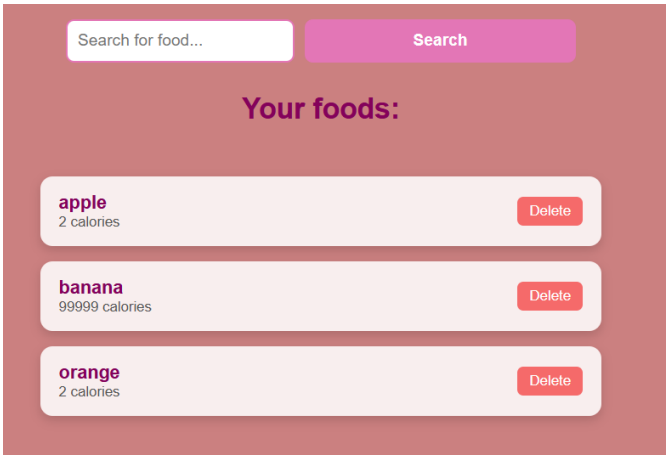

both successful and failed, are recorded in the auditlog table through helper methods in utils/dbQueries.js. Once logged in, users receive a session cookie that grants access to protected routes for a limited time.

2. Managing Personal Foods

Each user maintains a private list of foods they commonly consume along with a daily log of food and calorie intake.

- /foods/search displays the user's saved foods, with optional searching via query parameters.
- /foods/addfood allows users to add a new food item. When a food is added, the system automatically logs it for the current day, with the logic behind this implemented in utils/addfood.js.
- Entire food items can be removed from all logs using /foods/delete/:id

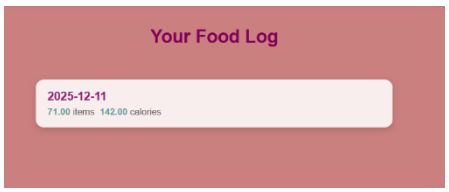


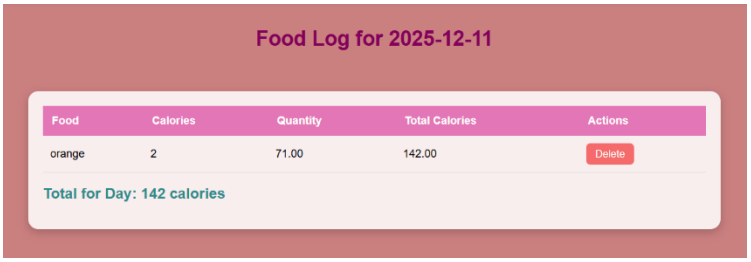




3. Daily Food Logging

Food Brain automatically tracks daily intake through the food_log table.

- /foods/logs provides a summary of each day, showing total foods eaten and total calorie count.
- /foods/logs/:date shows the detailed log for a specific date, including individual items, quantities, and combined calories.
  Users may delete entries to correct mistakes or adjust their daily logs with /logs/:date/delete/:logId

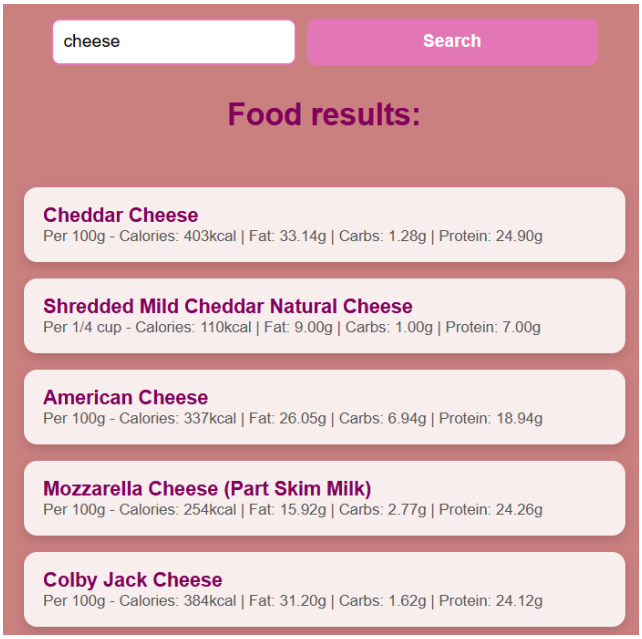



4. FatSecret API Integration

If users need nutritional information for foods not already in their list, the app offers integrated search through the FatSecret API.

- /foodslist?search_text=<term> sends a query using an OAuth2 client-credentials flow implemented in utils/api.js.
- Results are displayed through the foodsApi.ejs template, allowing users to search for select items and optionally add them to their personal food list with addfood.




5. Administration and Audit Logs

For transparency and security, you can  view audits via /users/audit.
This page retrieves entries from the auditlog table, recording timestamps, login outcomes, and attempted usernames. Access is restricted to authenticated users. Admin feature should be implemented but hasn't due to a lack of time.

**Advanced techniques**

Food Brain incorporates several techniques demonstrating secure development practices, API integration, and intelligent data-handling logic.

1. Secure Password Handling and Audit Logging

User authentication is implemented with bcrypt for password hashing, ensuring that plaintext credentials are never stored. Additionally, every login attempt, successful or not, is written to an auditlog table, improving traceability and security monitoring.
This is handled in routes/users.js, which both validates credentials and inserts audit entries through utils/dbQueries.js:

// routes/users.js

```
bcrypt.compare(req.body.password, user.password_hash, function (err, match) {
 if (!match) {
  dbQueries.insertAuditLog(req.body.username, 'incorrect password', next);
  return res.render('login', { error: 'Invalid credentials' });
 }
 // If match, create session
 req.session.userId = user.id;
 res.redirect('/foods/search');
});
```

This showing secure authentication design – stopped due to a lack of admin feature.

2. Manual OAuth 2.0 Client-Credentials Flow for FatSecret

Food brain manually implements the OAuth 2.0 client-credentials flow to obtain API tokens for FatSecret. The logic in utils/api.js constructs the request, encodes credentials, and retrieves the access token:

```
// utils/api.js
async function getAccessToken() {
 const basicAuth = Buffer.from(CLIENT_ID + ':' + CLIENT_SECRET).toString('base64');
 const response = await axios.post(TOKEN_URL, 'grant_type=client_credentials', {
  headers: { Authorization: 'Basic ' + basicAuth }
 });
 return response.data.access_token;
}
```

Routes such as routes/main.js then attach the token to authorised API calls:

```
// routes/main.js
const token = await getAccessToken();
request({
 url: FATSECRET_ENDPOINT + params,
 headers: { Authorization: 'Bearer ' + token }
}, handleResponse);
```

Implementing this was quite challenging.

**3. Parameterised Database Access**

All database interactions use parameterised SQL queries through mysql2, preventing SQL injection while maintaining efficiency. The global connection pool is initialised in index.js, and queries are defined in utils/dbQueries.js:

```
// utils/dbQueries.js

const sql = 'SELECT * FROM foods WHERE user_id = ? AND name = ?';

pool.query(sql, [userId, name], next);
```

This ensures user input is never interpolated directly into SQL statements, adhering to secure database-access best practices.

---

**4. Intelligent Add-Food and Daily-Log Logic**

The utils/addfood.js module implements a workflow that prevents duplicate food records and handles smart updates to daily logs. The logic checks whether a food already exists for the user, whether it has been logged today, and whether quantities should be incremented:

```
// utils/addfood.js

if (existingFood) {
  if (todaysEntry) {
    updateQuantity(todaysEntry.id, todaysEntry.quantity + qty);
  } else {
    insertFoodLog(userId, existingFood.id, qty);
  }
} else {
  const newId = insertFood(userId, name, calories);
  insertFoodLog(userId, newId, qty);
}
```

This approach avoids unnecessary database duplication and ensures consistent, predictable behaviour.

**AI Declaration**

There has been a very minimal use of AI during the creation of this web app. Mainly used for the structure of this very documentation.