Project: End World Hunger

Date: 3 November 2019

Group 2

James Giouroukelis - **460379993**

Alvin Ho - **450098927**

Alexandra Jina Lee - **440401072**

Ham Ranpara - **460146007**

Rona Wang - **460387840**

Abstract

Food waste is an unnecessary consequence of poor consumption management. It harbours a variety of worldly complications and is paradoxical in nature as it exists in conjunction with avoidable food insecurity statistics. This is due to a lack of food accessibility, where a buffer exists between consumers and produced food. End World Hunger seeks to remove the obstacle by providing a communication channel between the parties, thus, removing the food waste/insecurity paradox.

We investigate the effects of food waste and food insecurity, and provide statistical and scientific evidence through research. Then, we present our entire system architecture and reason with our choice of tools and structure. Finally, we review our system's effectiveness through our testing process.

Table of Contents

1. Introduction

1.1 Overview

End World Hunger (EWH) is a web application that connects food providers with consumers, driven by the goal of reducing food waste. Users can sign up as a food provider or a consumer. The platform allows food providers to post up their excess food, including details about serving quantity, pick-up location, and pick-up time. Consumers can view the food listings real-time and claim food from a provider. Consumers are limited by a daily food cap threshold, depending on whether they are an individual, organisation or a charity, to give all users a fair opportunity to claim food.

1.2 Main functionalities

1.2.1 Location-based search

Consumers can search for food items or providers centered around their current location or a custom location. A distance cap is implemented to limit the consumer's search results, depending on the user's subtype as an individual, organisation or a charity.

1.2.2 Food order dashboard

Both consumers and providers have access to a dashboard that contains a list of completed, pending, and expired orders. All details given during ordering of food is presented here.

1.2.3 Ratings, reviews, and reports

The platform supports a rating mechanism to promote the trustworthiness of users. After a food order is complete, food providers can leave ratings for consumers and vice versa. In extreme circumstances, food providers can report consumers and vice versa.

1.2.4 Leaderboard

The platform includes gamification features that encourage providers to accrue points and achieve awards for posting food. Food providers can achieve streaks for posting food listings over consecutive days and they can also be recognised on a leaderboard.

1.2.5 Admin management

The administrator has access to an admin dashboard which displays a list of all users. User reports also show here and can be actioned accordingly. This is where all user access is managed.

1.2.6 System architecture

In terms of software specifications and architecture, EWH is running on the Spring Boot framework, with a Model-View-Controller (MVC) architecture. The database is run on MySQL and supported by Hibernate as an ORM. The frontend is implemented with the React framework and other styling frameworks. The RESTFUL API is used to send HTTP Requests to handle data. External APIs associated with Google Maps are also used to assist with our location services.

1.3 Background research

One-third of the world's food is either wasted or lost, adding up to 1.3 billion tonnes a year (Oz Harvest, 2019). By increasing consumption of this otherwise wasted food, EWH would improve problems such as hunger, operating costs for charity organisations, and greenhouse gas effects. This project's model differs from other applications of the same market as we create free food exchanges.

21% of Australians have experienced food insecurity in the last 12 months (FoodBank, 2019) because families could not afford to buy an adequate amount of food. By registering on our application as an individual, lower income earners can receive gratuitous food. This will ease their monetary burden, therefore, diminishing Australian food insecurity.

Meal service charity organisations spend a large portion of their budget on purchasing food. Food Bank Australia's 2017-2018 Financial Report shows 24% of costs were for food procurement. If charities took part in our business, they would advantage from our free food

exchanges by decreasing, if not eradicating, all food purchasing costs. Then, they would have the opportunity to operate more efficiently, by focusing on other business aspects, such as expansion.

In Australia alone, 5 million tonnes of food a year end up in landfills (Oz Harvest, 2019). Decomposition of food releases a huge amount of methane and in a 100 year period, methane's effect on the atmosphere is 21 times worse than carbon dioxide's (Victorian State Government, 2019). Therefore, consuming a higher ratio of produced food would lessen the negative impact on the environment.

There are existing services, such as *Too Good to Go*, where their business models involve selling close-to-expiration food at a discount. EWH differs by encouraging providers to donate food at no cost. Food sellers' motivation behind this non-profit participation will exist because it would promote their social responsibility.

2. System architecture

2.1 Tool selection

In order to create a scalable and reliable application, several tools have been selected to be used in development. These include:

Amazon EC2: EC2 is a cloud platform provided by AWS (Amazon Web Services). This tool allows our EWH application to be available on the cloud, hence improving its up-time during normal operations. This also allows more room for scalability for any future opportunities and expansions of the app.

React: React is a component-based frontend framework. It is mainly used in developing our views/available pages in the EWH application. By utilising a component-based design, our web pages can be developed in a more dynamic and reusable manner, hence reducing total development time needed.

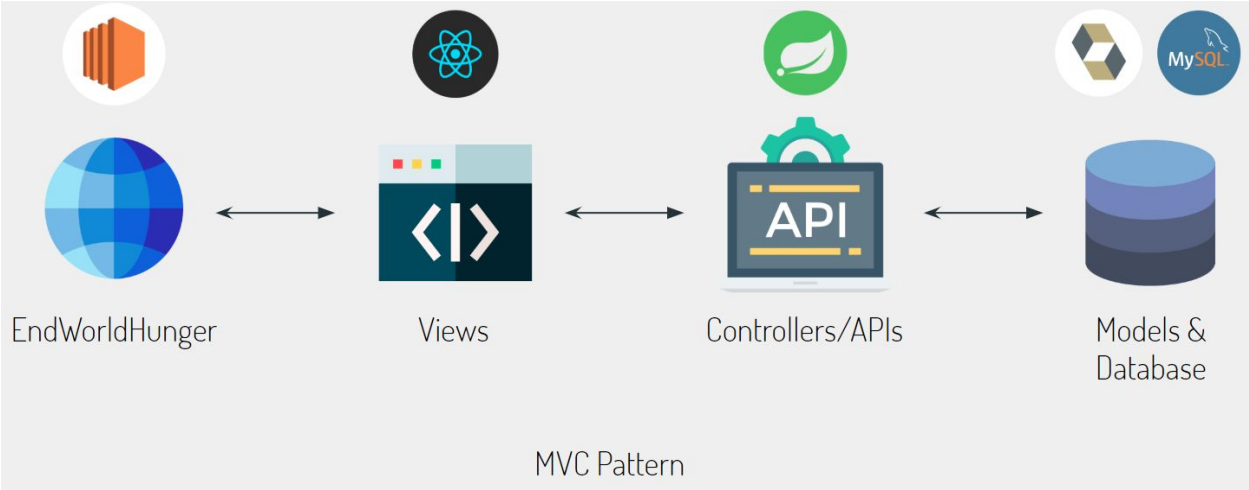
Spring: Spring is the main backend framework chosen. It is used to develop our internal APIs that are used to serve and compile data to our frontend pages. Spring itself is proven to be enterprise-ready with ease of creating testable and performant infrastructure.

Hibernate: Hibernate is an ORM (Object Relational Mapping) framework which allows easy and fast mapping between the database and existing models that reside in our

backend infrastructure. It is also compatible with any type of database (any relational or non-relational database) and other structural changes.

MySQL: MySQL is a relational database and is our main option for persistence storage. It is widely used and is an open-source tool which makes it tested and proven by the community.

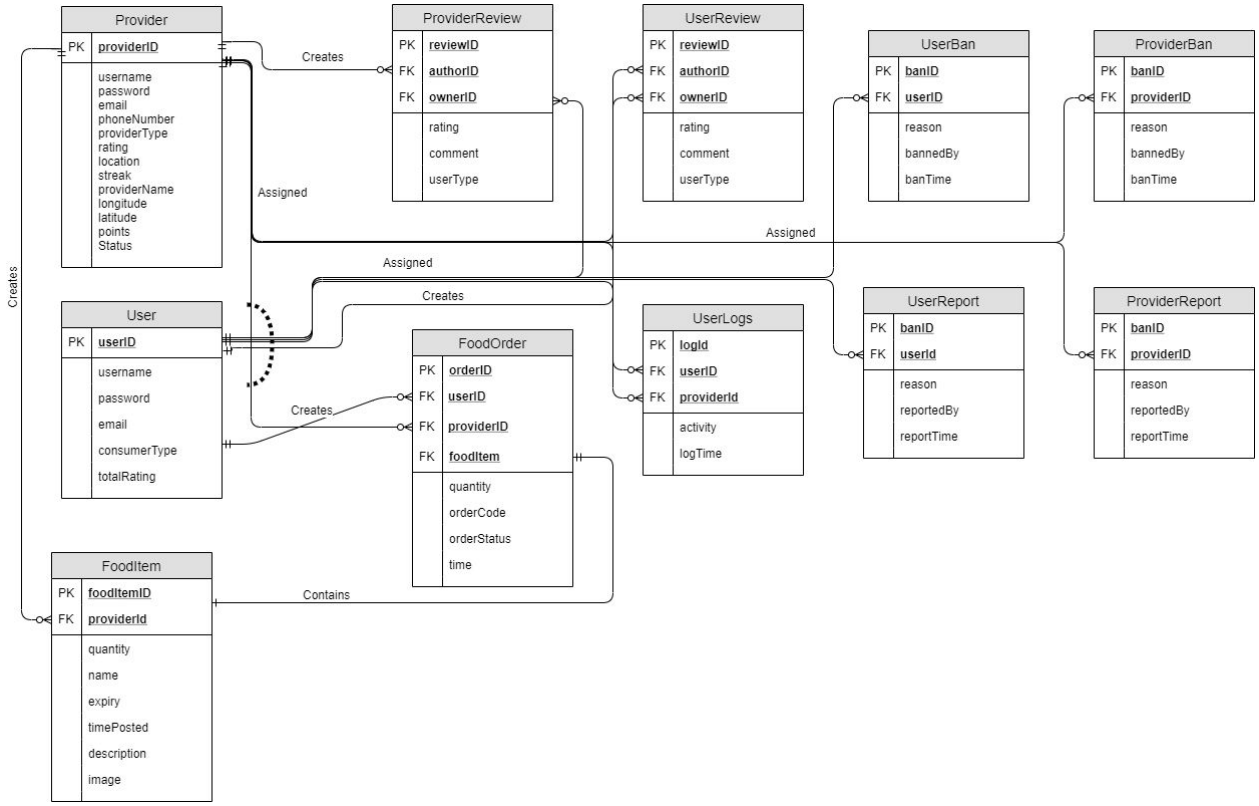
2.2 MVC architecture

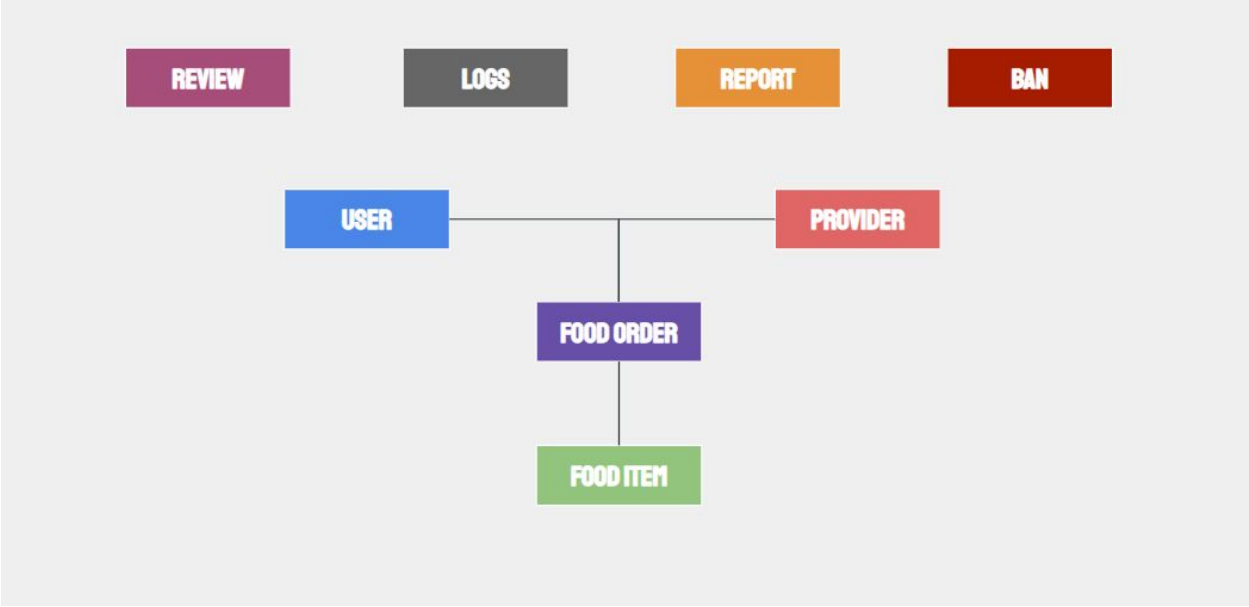


Our MVC solution is structured as above. Where our models (**M**) is defined in Spring, stored in our MySQL database, and is transferred/mapped using Hibernate ORM. The views (**V**) is developed using React framework displaying all the data and interface seen by our users. These views will then retrieve data through our internal APIs via the available URL endpoints defined in our controllers (**C**). These internal APIs are responsible for retrieving the request sent by the user's action in the frontend, compiling the required response through accessing necessary data from the database, executing business logic through services, and sending a response back to the frontend layer.

3. Technical implementation

3.1 Models





Note: In our implementation, our User model is based on a **Consumer** (Consumer and Provider). Therefore, we have both **Provider** and **User** which are the two most important models.

The Review model is broken down into UserReview and ProviderReview. Similarly, Report and Ban have been broken down into UserReport, ProviderReport, UserBan and ProviderBan respectively. The above chart succinctly describes all the Models implemented and created within our database upon startup.

We added foreign key constraints which is a requirement of any good database and backend. It also keeps objects connected, improves efficiency and creates ways to collect and display data as desired by our frontend.

The following describes a list of the constraints we placed upon our database and some more detailed information.

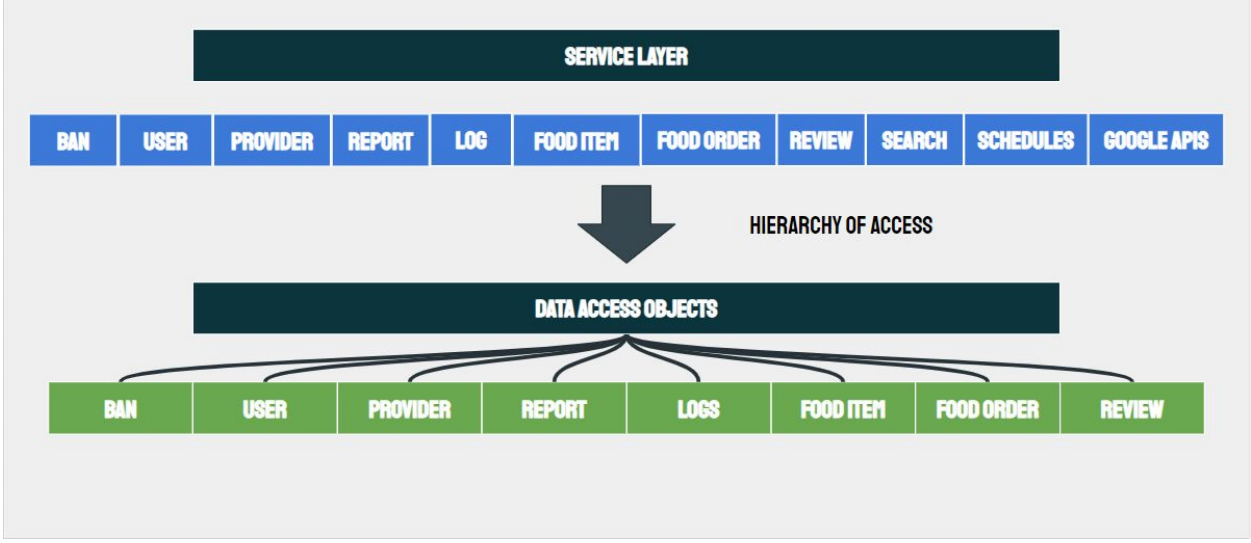
Foreign key constraints:

1. A User and a Provider can have many Reviews from and for them.
2. A User and a Provider can have many Logs.

3. A User and a Provider can be reported multiple times.
4. A User and a Provider can only be banned once. This indicates a One to One relationship.
5. A User can make multiple FoodOrders.
6. A Provider can have multiple FoodOrders.
7. A FoodOrder must contain a FoodItem and be connected to a User.
8. A FoodItem must be connected to a Provider; it cannot exist without originating from a Provider.

- The Review model has User ids and Provider ids as foreign keys.
- The Logs model has User ids and Provider ids as foreign keys.
- The UserReport model has User id as a foreign key and the ProviderReport model has Provider id as a foreign key,
- The UserBan model has User id as a foreign key and the ProviderBan model has Provider id as a foreign key,
- The FoodOrder model has User id, Provider id and FoodItem id as foreign keys.
- The FoodItem model has a Provider id and a FoodItem id as foreign keys.

3.2 Services and DAO's



The above contains a list of Services that are connected to Data Access Objects. The Service Layer is the only way the Controller can access the Models. In particular, the arrow shows the hierarchy of access and below the DAO's lies the Storage - Model layer. Some services abstracted and combined needed access points for example; the Ban service overlooked User bans and Provider bans. This design allowed us to separate and combine most of our logic to avoid repetitions.

The DAO layer contained most of the HQL queries. There was usage of a SessionFactory which is a lightweight database connecter that needed to be set at the creation of each DAO implementation. It acts as a way to ensure that everything is carried out in transactions and in case of an operation failing within this transaction, it sends an internal server error back and rolls back the change. If all operations are successful, it commits everything and sends back a completion message.

Some snippet examples are outlined below and many more can be found in code;

Update query

```
"update Provider " +
"set status = :newStatus " +
"where id = :providerId ")
.setParameter( "newStatus", "Active")
.setParameter( "providerId", idX)
.executeUpdate();
```

This query replaces the status of a provider to display whether they are ‘active’ or ‘banned’. It uses the natural id of the provider to look for it efficiently.

Select query

```
"from FoodOrder " +
"where providerId.id like :id ")
.setParameter( "id", id)
.getResultList();
```

This Hibernate query selects related FoodOrders with a given providerId to display all of them to the Provider. It is used by the frontend to display what the provider can accept or decline.

Delete

```
User p = session.load(User.class, new Integer(id));
if (null != p) {
    session.delete(p);
    return true;
}

return false;
```

In some cases, we had to use SessionFactory methods in order to carry out deletions and additions. The above shows how we first try to obtain a user through their id. If the user we are trying to delete exists, we are able to delete them off the data store and we send a flag to the frontend to show that we indeed managed to delete the id sent back.

3.3 Controllers and internal API's

To meet business requirements several internal APIs have been developed to process and compose the necessary data to be displayed in the frontend. In order to connect to these controllers, the frontend views creates an HTTP request to the corresponding API endpoint typically in the format of ***/api/controllerName/operation/<optional parameters>***. The controllers and internal APIs provided are as follows:

User: Responsible for CRUD operations of our consumers (consumers of our system is considered the User). The operations includes login, registration, profile and removal of our customers.

Provider: Responsible for CRUD operations of our food providers (restaurants, supermarkets, etc.) side of the system. This includes login, registration, provider's page, edit provider details and leaderboard system. This leaderboard system is calculated through points gained by providers when an order is successfully completed. Top providers will be awarded and is viewable on their page where the users/consumers will frequent.

Food item: This controller manages any food listing that a provider would put up in their page. It ensures creation of this listing on the provider's page, the food search page, and orders page.

Food order: Users can create an order of the food items listed by our food providers. This controller manages order creation, and the corresponding food quantity available after the order is completed.

Review: To create trust between users and providers, they are able to review one another upon completing an order. The review controller allows them to leave an anonymous review on each of their page, and an average user and provider rating.

Search: This controller handles any querying and searching of food or provider based on user's preference and location. It filters out available food items and providers, while also allowing them to search via location using the haversine formula. This formula takes in longitude and latitude values and calculates nearest available restaurants for the user to choose from.

Report: Further adding on to the review controller, users and providers are able to file a report such that the reported user gets reviewed by the system's admins. For admins, this controller allows operations such as resolving a report (i.e. dismissed), these reports will get removed from the database.

Ban: If an admin deemed a report to be true however, they can further ban/suspend the reported user/provider. This locks the user/provider account temporarily or until the ban itself is lifted by the admin.

Frontend views	Backend APIs					
	Controller	Function/method	HTTP request type	URL endpoint + parameter	Request Body	Return value
Login Register Profile Admin	UserController	**getAllUsers**	GET	/api/user	-	User[]
		getUserById	GET	/api/user/{id}	-	User
		addUser	POST	/api/user	User (if id exists, update that user, else add as new)	New/Updated User
		updateUser	PUT	/api/user/{id}	User	Updated user
		deleteUser	DELETE	/api/user/{id}	-	Boolean successful/not
Login Register Provider page Admin Leaderboard	ProviderController	**getAllProviders**	GET	/api/provider	-	User[]
		getProviderById	GET	/api/provider/{id}	-	User
		addProvider	POST	/api/provider	Provider (if id exists, update that user, else add as new)	New/Updated Provider
		updateProvider	PUT	/api/provider/{id}	Provider	Updated provider
		deleteProvider	DELETE	/api/provider/{id}	-	Boolean successful/not
		getStreakLeaderboard	GET	/api/provider/leaderboard/streak	-	Provider[] (top 50)
		getPointsLeaderboard	GET	/api/provider/leaderboard/points	-	Provider[] (top 50)

Shown above is the available mapping and information about the controllers, their existing endpoints, operations, expected request and response (for more information the full mapping table can be seen from here).

3.4 External API's

Our project made use of two External APIs, both of which come from Google. This is because Google's APIs are readily available, well documented and are reliable from a business perspective.

3.4.1 Google Geocoding API

The first external API that we used is called Google Geocoding. It takes in an address and returns its respective longitude and latitude values. This was used so that any providers that use our service do not need to know the longitude and latitude values of their place of work, but rather only need to give us their address and we can work out the rest. The reason that we need the

longitude and latitude values is so that they can be used in our internal search API to calculate the distance between a user and multiple providers.

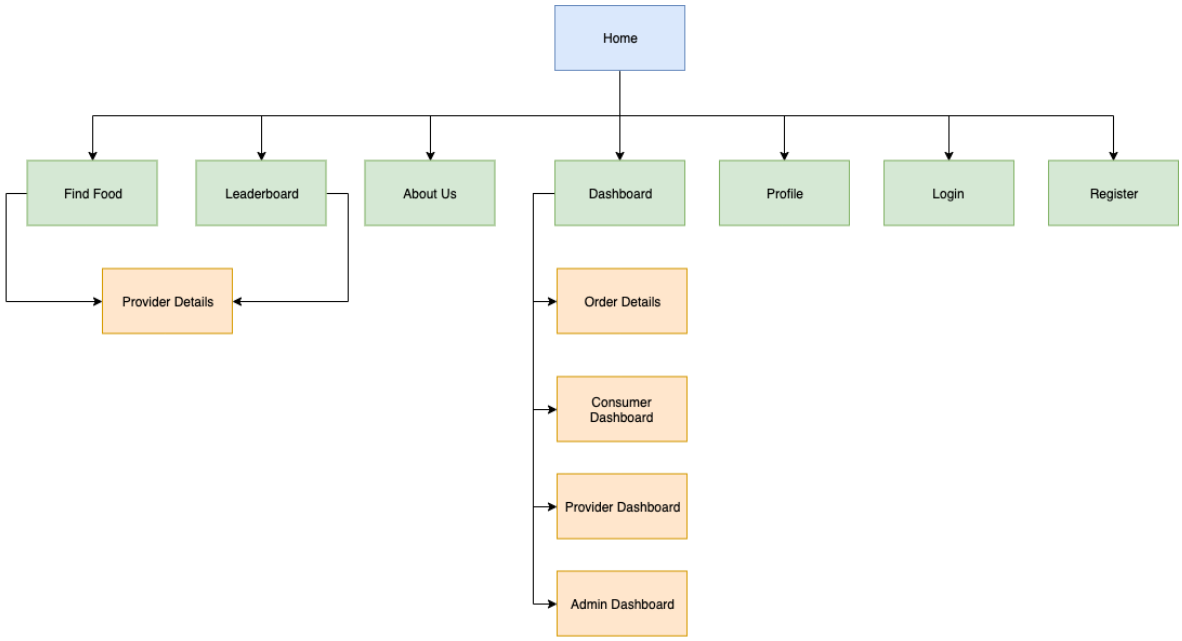
3.4.2 Google Places Autocomplete API

The second external API that we used is called Google Places Autocomplete. It provides autocomplete suggestions to improve the experience of all users who decide to type in a manual address instead of using their devices' location. It also has the added benefit of returning longitude and latitude values so that we can send the result straight to our internal search API.

3.5 Presentation layer (Views)

3.5.1 Frontend overview

The presentation layer is summarised by the sitemap shown in the figure below. The landing page is the Home page, where logged in users can navigate to the Find Food, Leaderboard, About Us, Dashboard and Profile pages; logged out or new users can navigate to the Leaderboard, About Us, Login and Register page. From the Find Food and Leaderboard pages, the user can also navigate to a Provider Details page that contains detailed information about a specified provider and the food listings they offer. The Dashboard page will display the Consumer Dashboard, Provider Dashboard or Admin Dashboard depending on the respective user. From the Dashboard page, the user can also navigate to an Order Details page that contains details about a specified order.



3.5.2 React

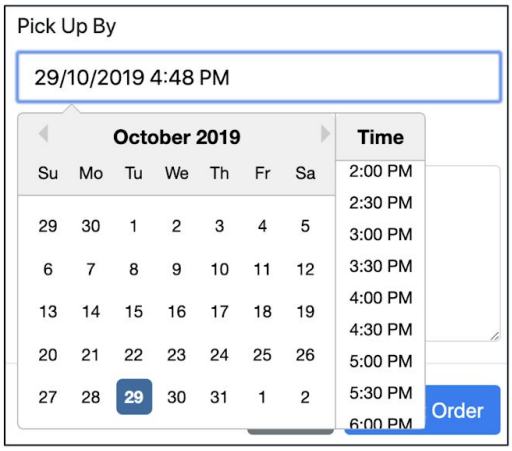
The frontend presentation layer is mainly implemented using the React framework, which is a JavaScript library that is commonly used across the industry. As it is written in JavaScript, it allows for dynamic rendering, which shifts workload to the browser and minimises time-consuming interactions between client and server. In React, the programmer can create custom components, which can then be reused throughout the application, allowing for more maintainable and higher quality code. A list of components that were used to create our application are described below.

Component	**Description**
AboutUs	An 'About' page that describes what End World Hunger is
AddFoodModal	A modal for providers to add a new food listing
AdminActiveSwitch	The buttons that manage the Active/Locked status of a user
AdminSearchResults	The list of results when admin searches for users
App	The root component, which handles routing to all other components
Dashboard	A wrapper to display a dashboard based on user type
DashboardAdmin	A dashboard that allows admin to manage user activity
DashboardConsumer	A dashboard for consumers to manage their order
DashboardProvider	A dashboard for providers to manage their order requests
FoodRequestModal	A modal for consumers to request food and make a food order
Home	The landing welcome page
Leaderboard	A leaderboard of providers based on their points or streak
Login	The login page

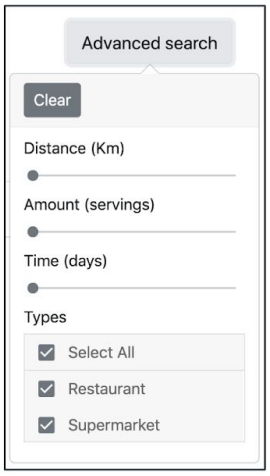
Navbar	The top navbar
OrderDetails	The order details page
Profile	The profile page
ProfileActivity	A list of the user's activity
ProfileEditModal	A modal to edit the user's profile details
Provider	The provider details page
ProviderBadge	A badge that recognises the award level of a provider based on their points and streak
ProviderFoodTable	A table containing the provider's food listings
ProviderReviews	All reviews written about the provider
Register	The register page
ReportModal	A modal that allows users to report other users
ReviewModal	A modal that allows users to review other users
SearchFoodList	The list of results when searching for food
SearchRestaurantList	The list of providers when searching for providers
SearchResults	A search page

3.5.3 Public libraries

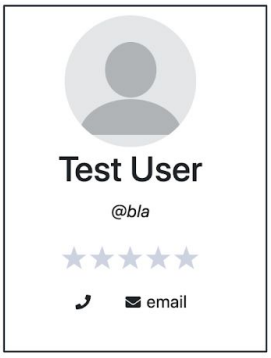
For styling, we used Bootstrap, which is an open source CSS framework and Font Awesome, which is a font and icon toolkit. React also comes with a vibrant ecosystem of public libraries and tools, which can be downloaded as node modules and managed through the npm JavaScript package manager. Some examples of notable node modules that we used are shown in the figure below.



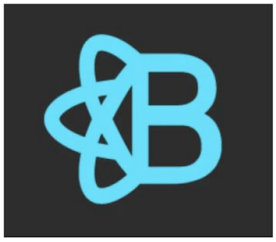
React-datepicker



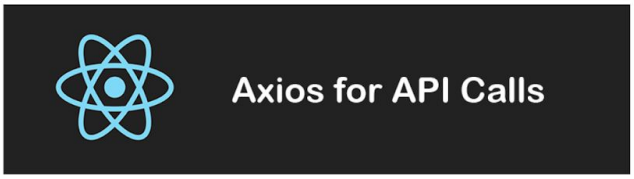
React-mutli-select



React-rating
React-star-rating



React-bootstrap



Axios

React-datepicker was used to create a Datepicker component. React-multi-select was used as a form input to select the provider type for the advanced search function. React-rating and react-star-rating were used to implement the rating function for consumers and providers to rate each other. React-bootstrap was used to easily use Bootstrap as components in React. Axios was used to manage API calls from the client to the server.

3.6 Scheduled tasks using Spring

Our project makes use of Spring's scheduled tasks which can be run periodically, adhering to a schedule. The reason for this is to automatically clean up/maintain the database at critical points to ensure smooth operation of the service.

3.6.1 Reset monthly leaderboard

Providers both get points for picking up and posting food in order to adhere to our gamification of the service. These points are tracked and are publicly available on a leaderboard in order to reward those businesses that give back the most to the community. At 1:01 am on the 1st of each month, all Providers have their points cleared from the database in order to reset the monthly leaderboard.

3.6.2 Reset daily limit

Users have a daily limit based on their role (Individual, Charity, Organisation) in order to prevent the hogging of food servings. Each day at 1:01am, all Users have their daily limit reset from the database.

3.6.3 Clean up food items

Food items that are posted by Providers have a quantity and an expiry date. Each day at 1:11am, any food items that have no quantity left or are expired are removed from the database.

3.6.4 Clean up food orders

Food orders that are created have a quantity and an expiry date. Each day at 1:21am, any food orders that have a 0 quantity or are expired are removed from the database.

3.7 Deployment on EC2

Our project consists of the React frontend server, the Spring Boot backend server and a MySQL database server. All three of these servers are deployed on an Amazon EC2 instance. The only one of these servers that is public facing is the React frontend server to ensure the security of our service is maintained. The React server will then communicate to the Spring Boot server internally which will also communicate with the MySQL server internally. This is seen by visiting our React frontend server at http://3.24.79.135:3000/.

4. Testing

During implementation, we employed two forms of testing to ensure a robust and reliable service. The first was using unit tests within our p application to ensure that all of our models could be created correctly and could be saved correctly to the database via our services. The second form of testing that we employed was using Postman. Postman is a free application that allows for easy construction of requests and the ability to view their respective responses. The use case for this application within our testing was to test each of our Controller/API endpoints to make sure they are reachable and behave exactly as expected.

5. Individual implementation

5.1 Food order (Rona)

- Claim food
    - Set time and quantity
    - Unique order ID
    - Send email confirmation to user
- Provider dashboard to see who has claimed food from them
    - Provider can 'complete' order once picked up by consumer
- Users leave reviews after order completed

5.2 Awards system (only for businesses) (Jina)

- Providers
    - Get points for listing
        - With expiry dates
        - Can have streaks
- Leaderboard
- All static pages & help with other parts

5.3 Search (James)

- Multi-parameter search functionality
    - Location based search (Google maps API)
        - Results: restaurants
    - Food based search
        - Results: food listing
    - Filter results

5.4 Food provider (Hamesh)

- Provider page
    - With provider's listings
    - With provider rating/comments
- Providers post food listings
    - Pick up location
    - Pick up time
    - Servings
    - Notes/policies
    - Name

5.5 User management (Alvin)

- Registration

- Login
- Profile
    - Edit
    - User history/activity
    - Rating
- Admin dashboard
    - View reported users
    - View banned users

6. Conclusion

End World Hunger connects food providers and consumers with the goal to minimise food waste. Our project team chose to combat food waste because minimising this waste would tackle food insecurity, environmental issues, and operating costs for charities.

EWH is hosted as a web application using the Spring framework and structured around a Model-View-Controller architecture, with a MySQL database in conjunction with Hibernate, a React frontend, and deployed using Amazon EC2. Testing was carried out using JUnit and Postman. The project was divided into five components with each component allocated to a team member, as described in Section 5.

By encouraging corporate social responsibility and complimentary food, we progress into a world with less food waste and revitalise our appreciation of food.

References

Australian Ethical Investment Ltd. (2019). Food waste: a growing Australian problem. Retrieved from Australian Ethical: https://www.australianethical.com.au/news/food-waste-growing-australian-problem/

FoodBank. (2019). Hunger in Australia. Retrieved from FoodBank: https://www.foodbank.org.au/hunger-in-australia/?state=nsw-act

Food Bank. (2018). Foodbank South Australia Annual Report 2017-2018. Retrieved from Foodbank: https://www.foodbank.org.au/wp-content/uploads/2019/03/FBSA-Annual-Report-2017-18.pdf

Oz Harvest. (2019). Food Waste Facts. Retrieved from Oz Harvest: https://www.ozharvest.org/what-we-do/environment-facts/

Victoria State Government. (2019). Love Food Hate Waste. Retrieved from Sustainability Victoria: https://www.sustainability.vic.gov.au/campaigns/love-food-hate-waste