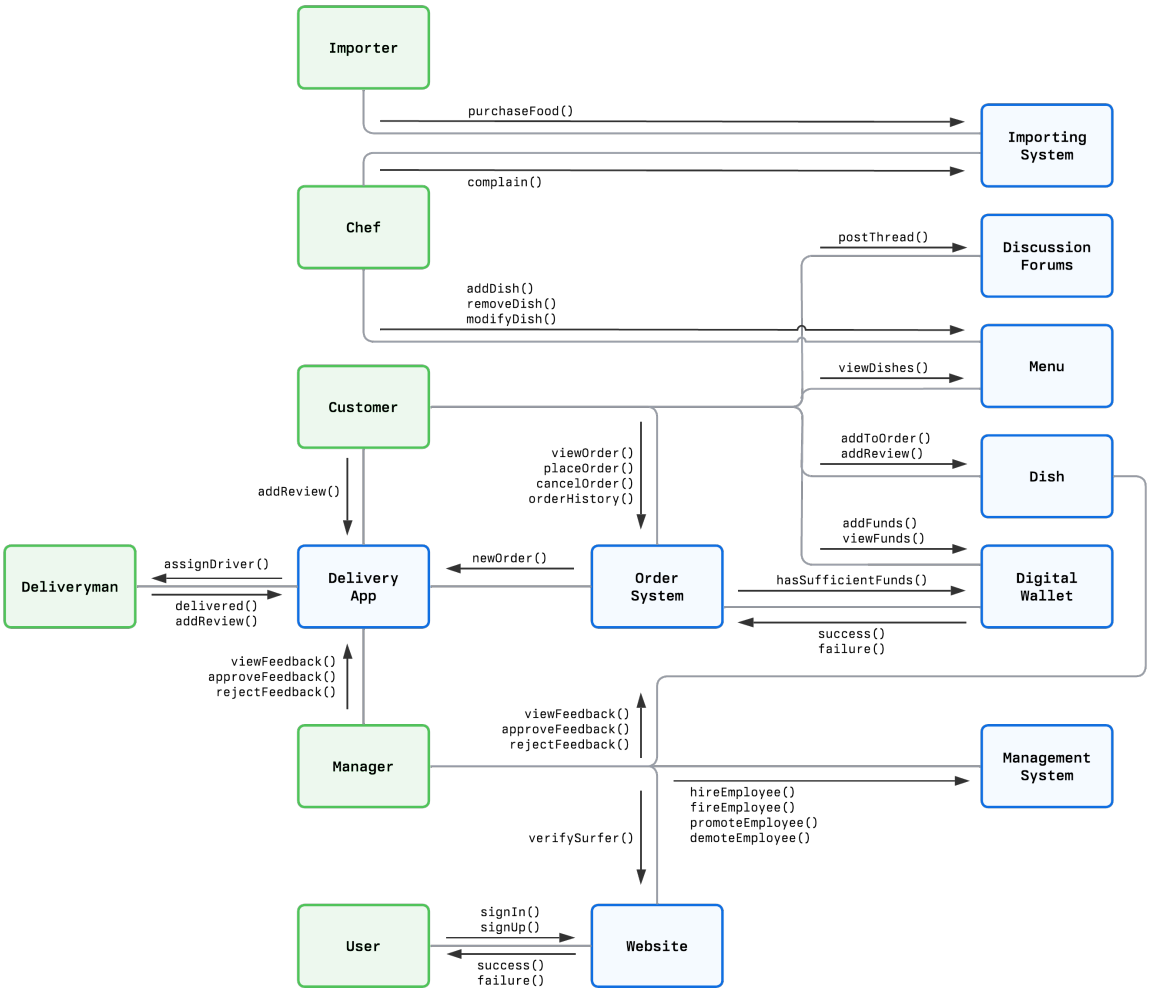
PROJECT PHASE 2
DESIGN REPORT

DISCLAIMER! This project, including supplemental documentation, is open source and hosted on GitHub under the MikeRomaa/csc322-project repository.

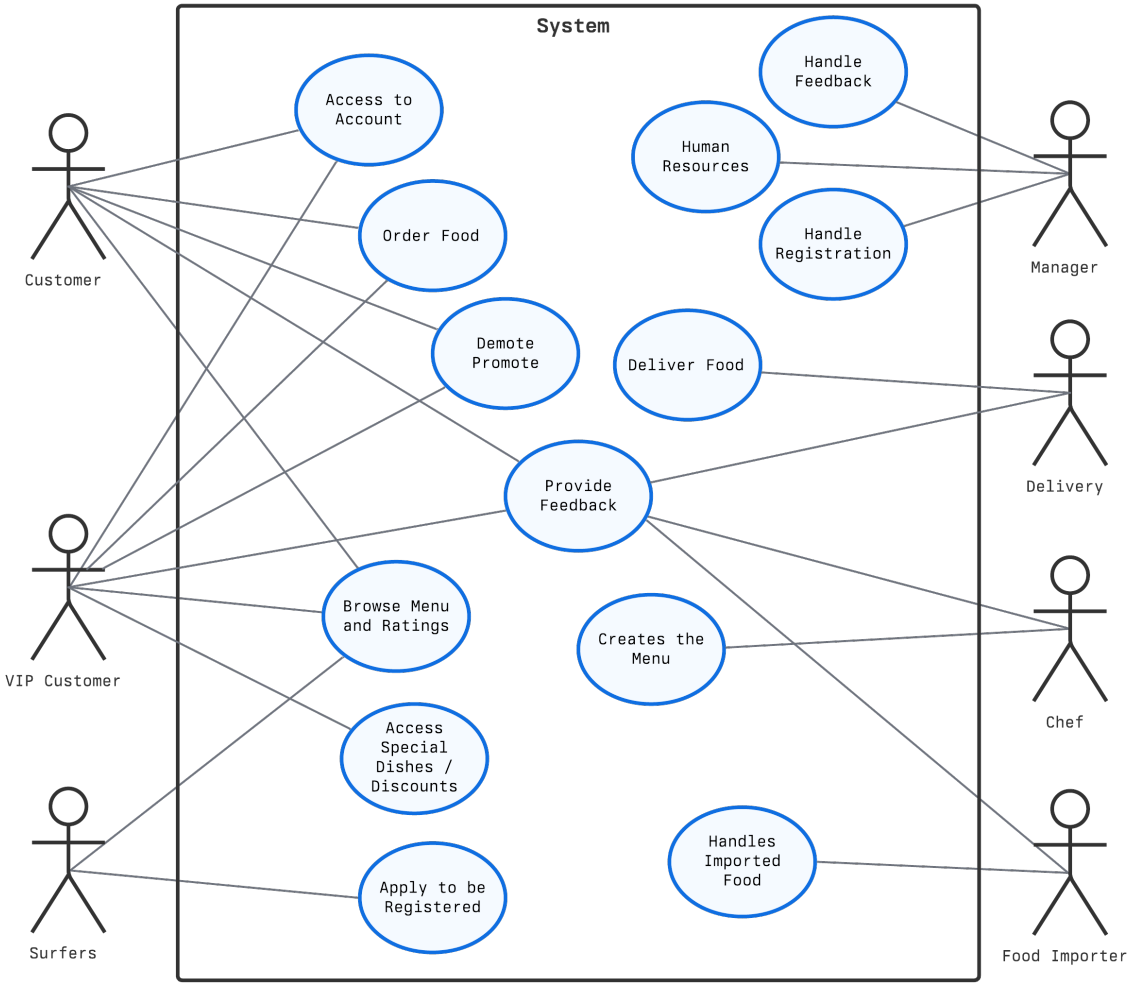
INTRODUCTION

The purpose of this document is to introduce more implementation-level details about how different parts of the **PlatePal** restaurant management system interact and communicate.



USE CASES

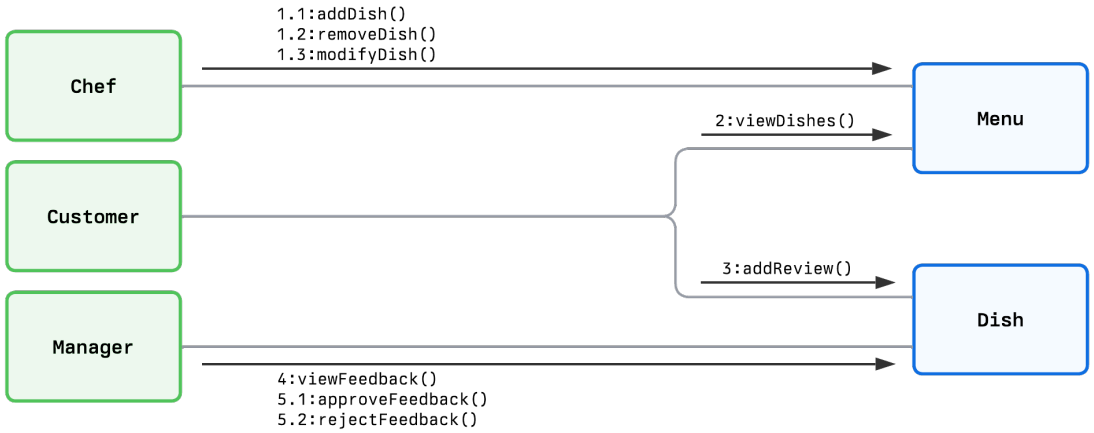
We first present an overall use-case diagram for our system. The main actors of the system are *customers*, *VIP customers*, *surfers*, *managers*, *deliverymen*, *chefs*, and *food importers*. We also present more intimate diagrams for selected use cases that offer a better idea of the flow of data throughout the system when that use case is exercised.



Menu Selection and Display

One of the biggest and most prominent use cases is the ability for chefs at restaurants to create menus and for customers to view, order, and review the dishes on that menu. This use case is the core functionality powering our business model, and without it our application would be just another human resources dashboard.

Below we present a communication class diagram showcasing the different message flows between classes within our system that power the menu selection and display functionality.



Chefs can either add dishes to the menu, remove dishes, or modify them. Each dish contains a name, optional thumbnail image, price, and description, all of which can be modified and set by the chefs.

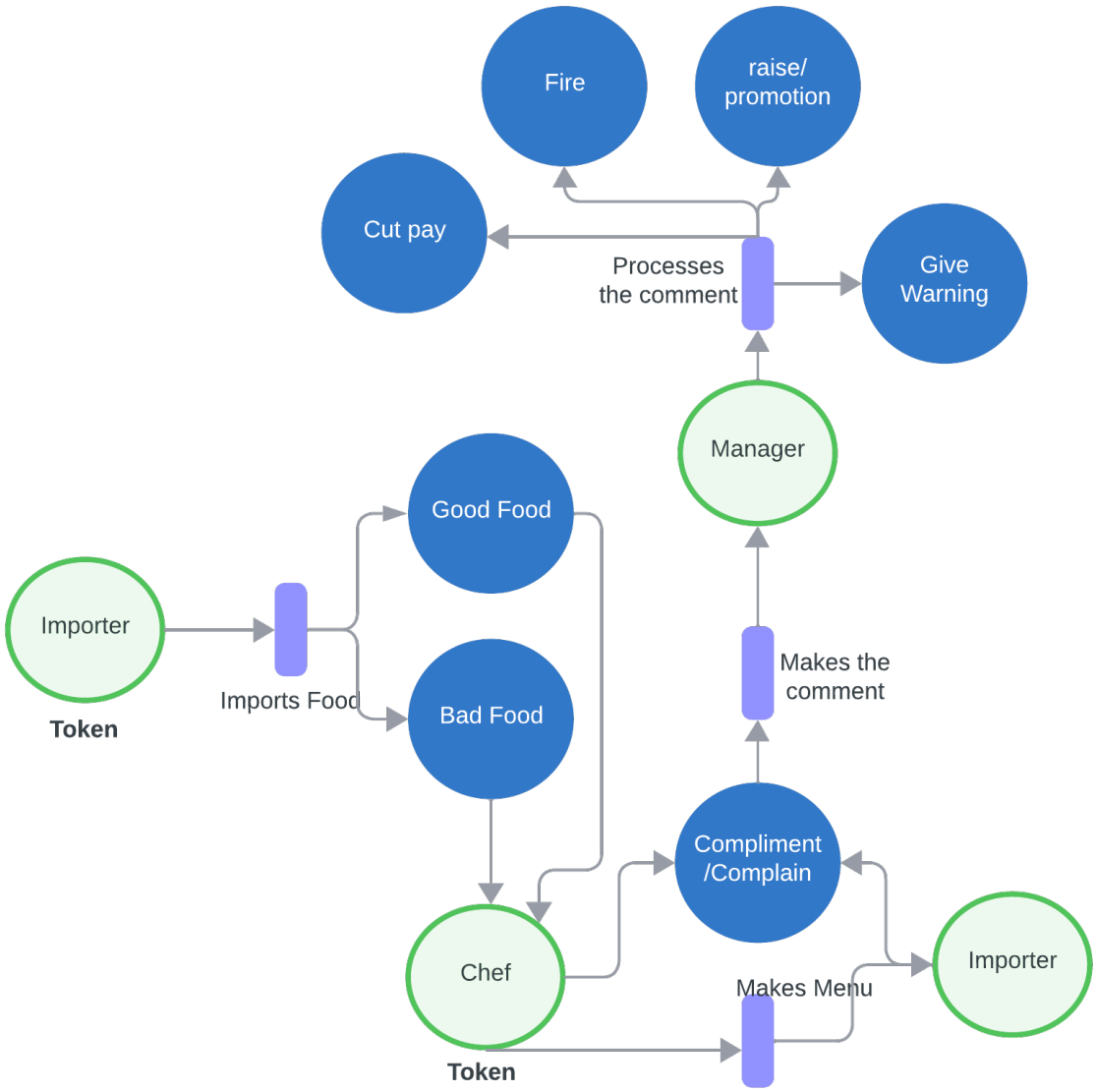
Customers can either view the dishes of a particular menu, or add their reviews for a particular dish. The dishes visible to a customer also depends on their VIP status, so different users may see different dishes given the same menu. The reviews left by customers include a rating from 1-5 and a comment about their thoughts on the dish.

Managers can then view the feedback left by customers, and approve or reject it depending on whether or not the feedback was left with merit or not. This way, spam feedback can be filtered out by the manager and not impact the chefs.

Human Resources / Management

The next major use case we will showcase is the human resources and overall management functionality. It is just as important as our menu system since without it, our application would just be another delivery application. The management functionality allows managers at restaurants to demote, promote, hire, and fire restaurant staff, as well as review feedback left by chefs and importers about the food companies the restaurant works with.

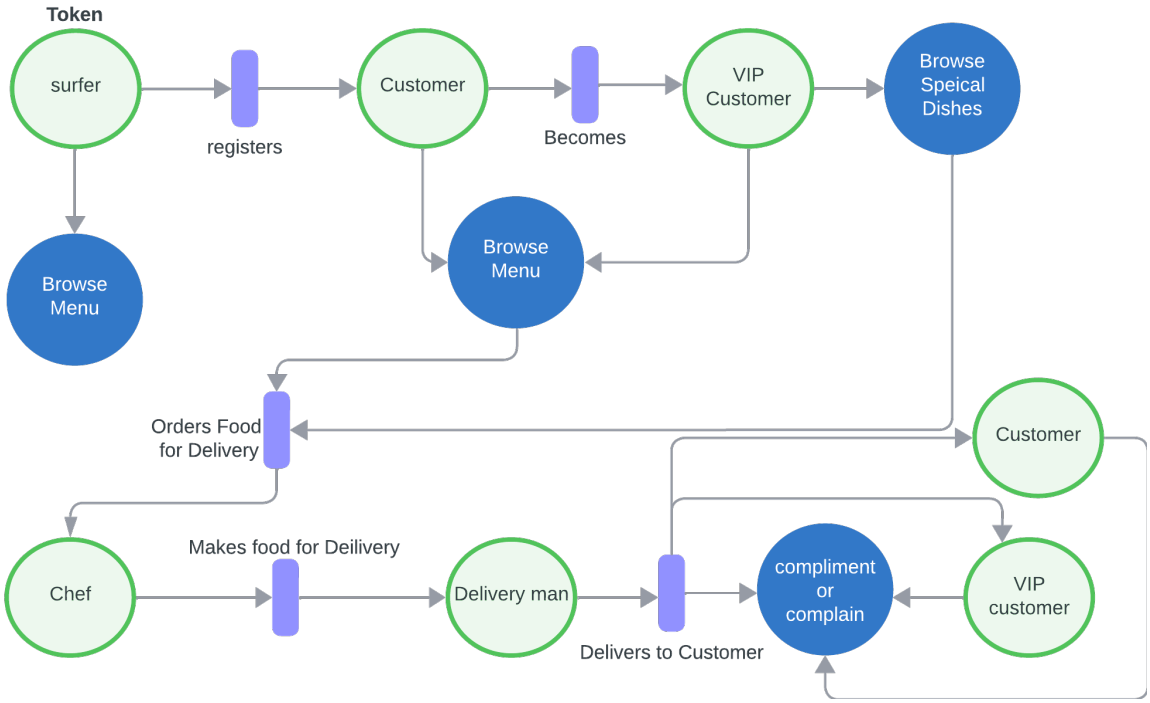
Below we present a Petri net diagram showcasing the logical flows between an importer and a chef making comments and a manager taking action on the comment.



Delivery System

Another crucial use case of our application is the delivery system. This allows customers to place orders from the comfort of their home, and the delivery drivers at the restaurant can then bring their order straight to them. In this system, drivers compete for orders with their fellow deliverymen, and can leave feedback on customers if they are exceptionally good or bad. Conversely, customers can also leave feedback on their delivery driver if they thought their services where exceptionally good or bad.

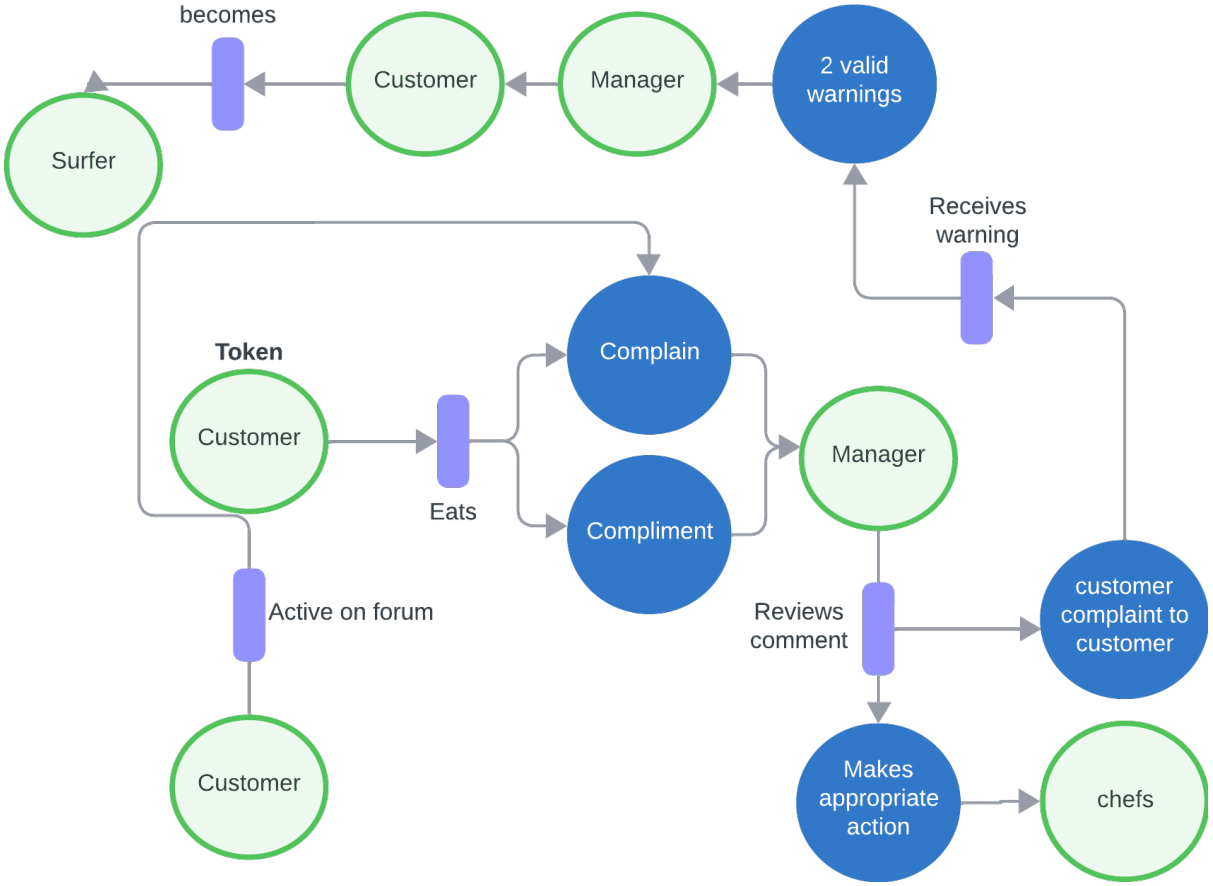
Below we present another Petri net diagram for the delivery system use case, showcasing the logical flows between a surfer signing on to the website, placing an order, and receiving the delivery.



Reviews and Discussions

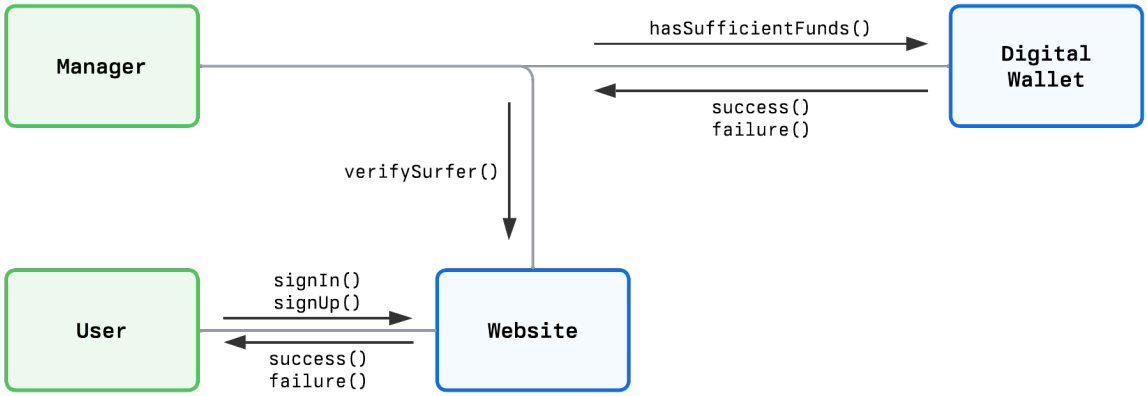
Another cornerstone feature of our application is the ability for customers to leave reviews for dishes, participate in discussion forums, and report their fellow forum-dwellers for bad behavior. We believe that this not only improves the quality of feedback that the restaurants receive from customers, but also adds a community aspect to traditionally bland ordering experiences.

Below we present another Petri net diagram for the delivery system use case, showcasing the logical flows between customers submitting complaints and compliments to the restaurant, where the manager can then review the feeedback and take appropriate action. Customers with two complaints to their name will be removed from the system and reverted to being a surfer.



Surfing

The final use case we present is the surfing functionality. When a user initially registers for the website, they are unable to place orders until they top up their digital wallet with sufficient funds and are approved by a manager. This avoids spam orders from being submitted to a restaurant, since the manager can manually reject any accounts they suspect to be spam.



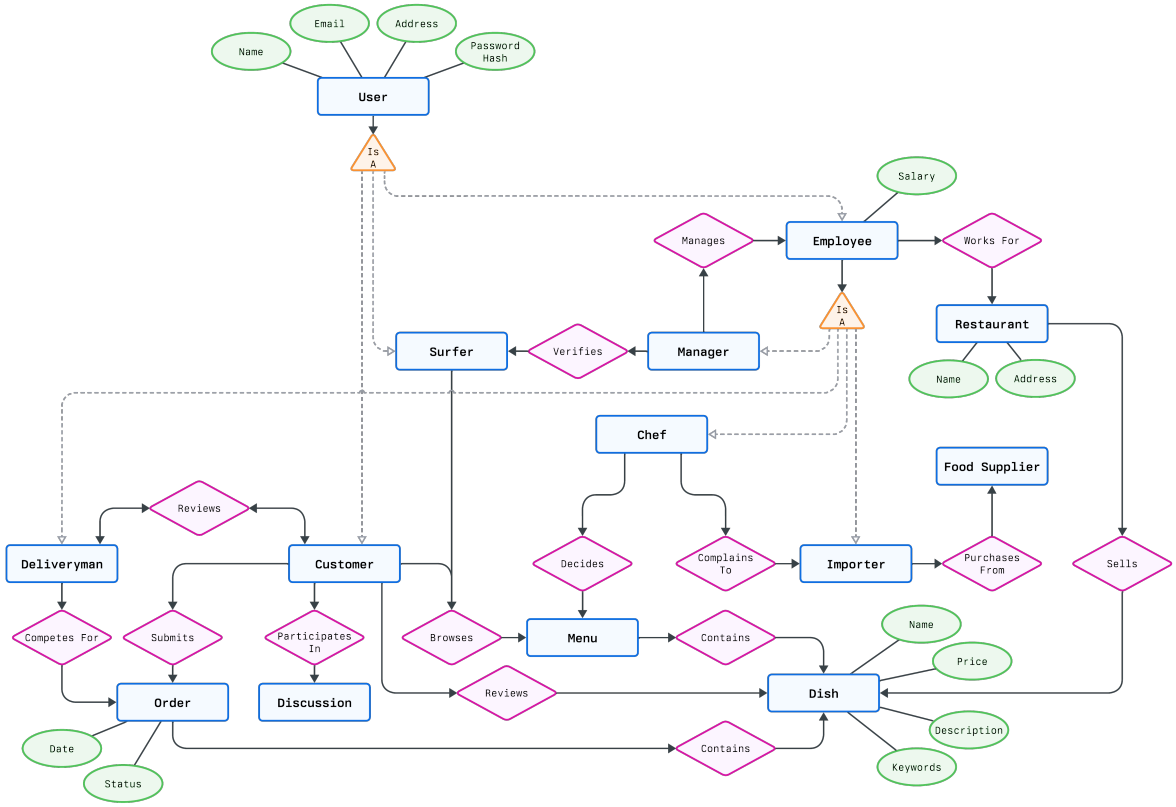
When a manager receives an account sign-up request, they can first check the digital wallet of the account to ensure it has sufficient funds. They can then chose to either verify the surfer and allo them to place orders, or simply ignore the account.

Also included in the diagram is a basic overview of the authentication system. Users can either **sign in** or **sign up**, and they will receive a success or failure state depending on if their password matched or not.

ENTITY RELATIONSHIPS

Below is an entity-relationship diagram for the entire system.

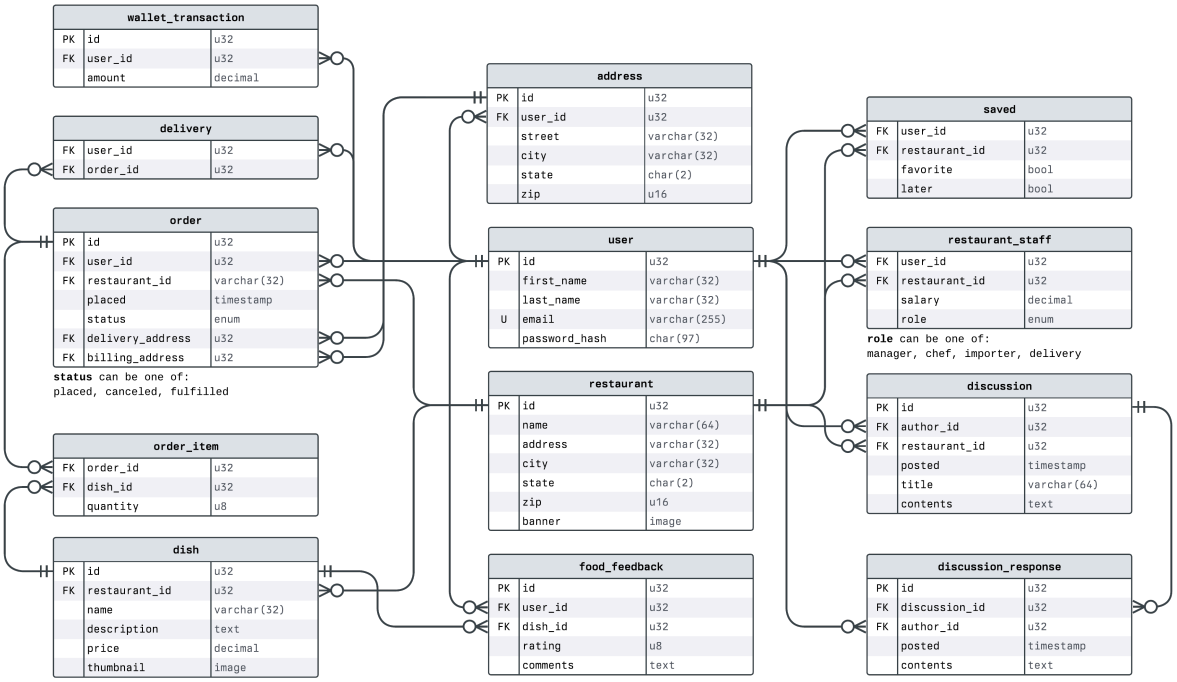
- Blue rectangles represent entities, which include actors and other object types that will eventually build up our database tables.
- Magenta diamonds represent relationships between those entities. This includes "has-a" relationships, as well as any actions taken by one entity onto another.
- Green ovals represent fields or attributes on an entity. These will become individual columns in our database and allow us to describe individual entities.
- Orange triangles represent inheritance ("is-a") relationships between entities. For example, employees, surfers, and customers are all users and are represented by the same database table, but may have different relationships with other entities.



DETAILED DESIGN

As part of our detailed design implementation document, we present the database schema for the application. All columns, data types, primary keys, foreign keys, and relationships are represented in a database entity-relationship diagram.

Each box is a distinct database table, and each row is a column within that table with the specified name and type. Lines represent many-to-many relatinships between junction tables and their respective surrogate keys.



An example of a query we may want to perform against this database is finding the subtotal for a given order when a user is checking out. This is done with a very simple **JOIN** operation as follows:

```
SELECT
    SUM(order_item.quantity * dish.price) AS subtotal
FROM
    order
JOIN
    order_item
ON
    order.id = order_item.order_id
JOIN
    dish
ON
    dish.id = order_item.dish_id
WHERE
    order.id = %(id)s
```

When a user places the order, we can then use the **subtotal** we got from the previous query in order to subtract the quantity from their account's balance. However, we also need to check if they have sufficient balance to begin with. Since our table stores deducations as negative values and balance top-ups as positive values, we can do that with this simple query:

```
SELECT
    SUM(amount) AS balance
FROM
    wallet_transaction
WHERE
    user_id = %(id)s
```

To then subtract the necessary balance when placing the order, we can use an **INSERT INTO** clause as follows:

```
INSERT INTO
    wallet_transaction
    (user_id, amount)
VALUES
    (%(user_id)s, %(amount)s)
```

Note that the use of %(<field>)s denotes a parameter inserted into the query by the database adaptor. This helps to defend our applicaiton against SQL injection attacks, since users are not able to execute arbitrary SQL queries as would be possible with the naïve string templating approach.

Another interesting query we may want to perform is identifying the progress that a customer has made towards VIP status. This condition is met by either spending $500 or placing 50 orders at a single restaurant. We can build upon our previous query for order subtotal and use a common table expression (CTE) to pre-compute all order totals and then simply filter the results.

```
WITH subtotals AS (
    SELECT
        SUM(order_item.quantity * dish.price) AS subtotal
    FROM
        order
    JOIN
        order_item
    ON
        order.id = order_item.order_id
    JOIN
        dish
    ON
        dish.id = order_item.dish_id
    WHERE
        order.user_id = %(user_id)s
    AND
        order.restaurant_id = %(restaurant_id)s
)
SELECT
    CASE
      WHEN SUM(subtotal) > 500 OR COUNT(subtotal) > 50
      THEN true
      ELSE false
    END AS vip_status
FROM subtotals
```

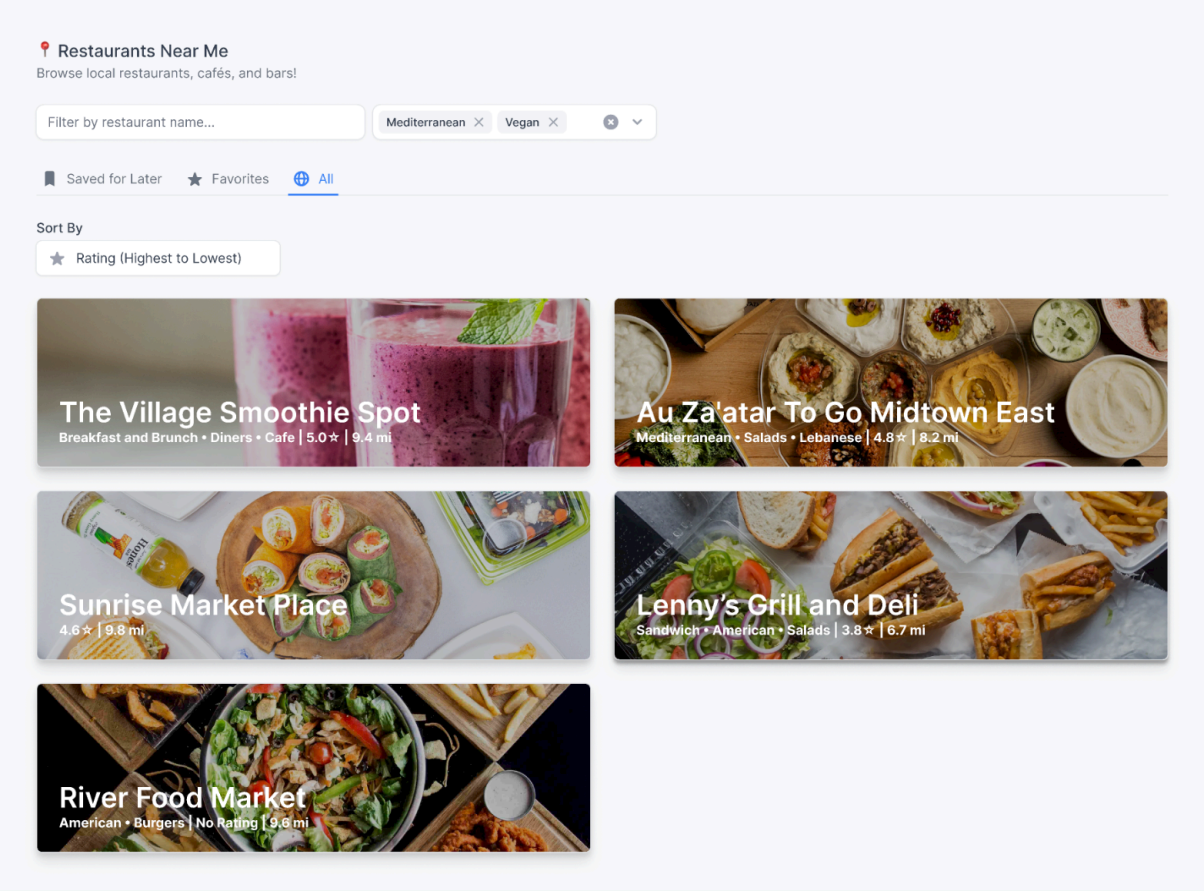
GUI MOCKUPS

Homepage / Browse

The following mockup represents a rough idea of what the homepage would look like to customers and surfers who are browsing for restaurants.

Users are able to filter by both restaurant name and tags (e.g. cuisine, dietary restrictions), sort the results by rating, distance, or price, and navigate to individual restaurants' pages to place their orders. Additionally, users can create "Saved for Later" and "Favorites" lists for restaurants that they want to check out in the future or have liked in the past.

The background images would be configurable by the restaurants to customize their appearance to best match their brand identity.

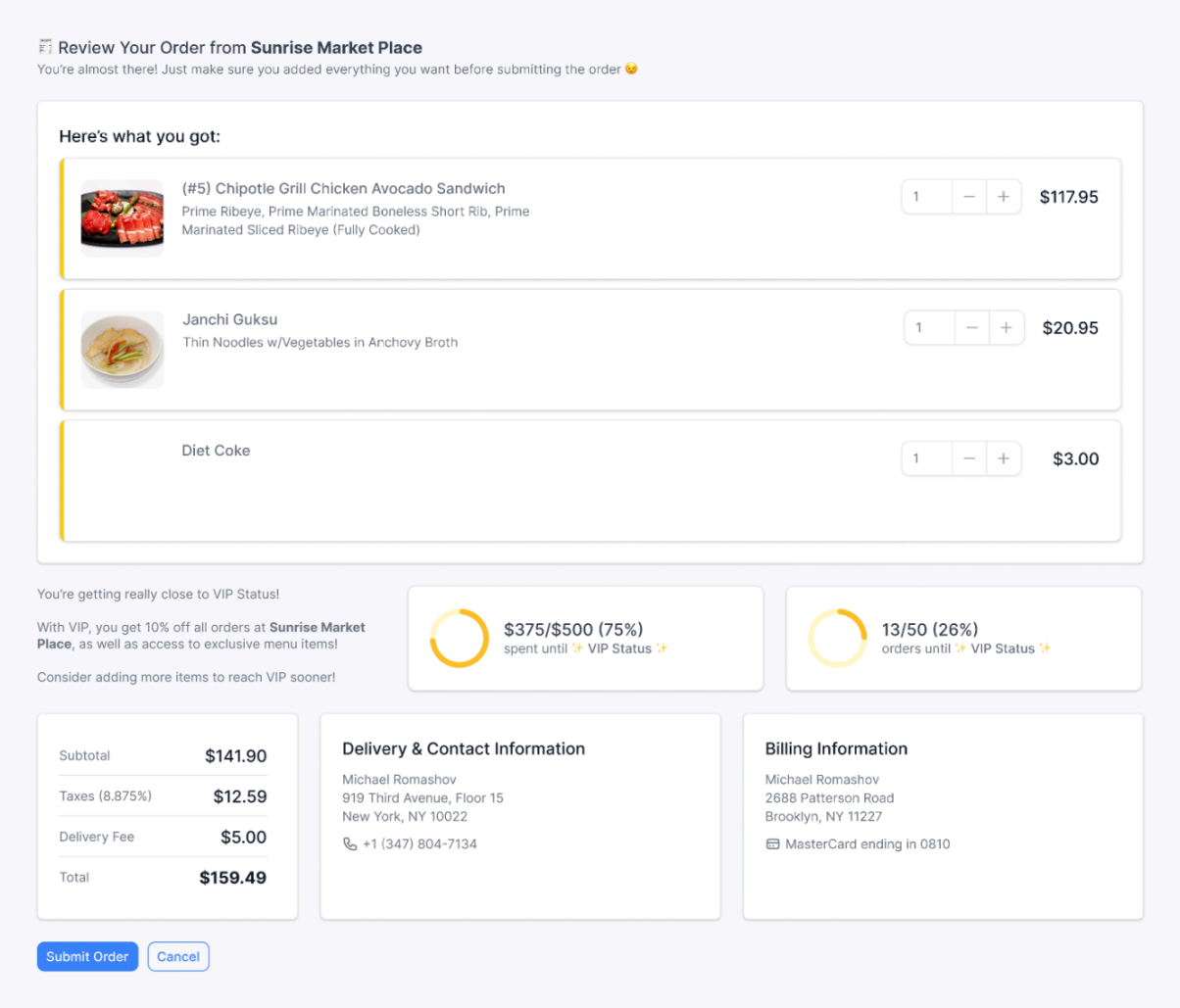


Confirm Order

The next mockup is a depiction of what a user sees when checking out. Orders are made at a specific restaurant (meaning you cannot pick dishes from different restaurants) and are listed along with thei rname, description, quantity, and price. Users have the option of adding more quantity of any item, as well as removing them entirely.

Users are also shown their progress to VIP status. This incentivizes them to spend more money and place more orders, since it gives them discounts and access to exclusive items in the future.

At the bottom, users confirm their delivery, contact, and billing information for the order. These will ideally come directly from the user's profile, with the option of modifying them when placing the order.



MEETING MEMOS

Apr 15, 2024

In the current state of our project, several factors were discussed regarding this project. First, Next.js will be one of the frameworks used due to its ability to create full stack web applications efficiently. Second, the project will result in a web application that will satisfy all aspects of the project specifications. Next, given the circumstances and the sheer amount of users, their actions, as well as the interactions between everyone, the actual programming part of the project will start soon as a result of that. As of now, there are no concerns regarding teamwork.

Apr 18, 2024

The diagrams required as part of the phase two report were discussed. This included identifying the major parts of the system (database, web service, frontend), the different actors and use cases, and specific message paths taken between classes and actors. This resulted in the creation of several system and use case diagrams that display the discussed behavior in a visual manner. There was a slight crunch due to the somewhat last-minute effort, but everything was completed to a high level of quality.

Apr 19, 2024

There was a short meeting leading up to the deadline for submitting the phase two report. The document was reviewed for accuracy and completion. No major issues were identified, and the document was later submitted for review by the professor.