



Computer Science/ Information Technology Project

COMP3900/9900 20T1

WAITLESS Report

Masterminds

Aarushi Gera	- z5228145
Anagha Thekkedath	- z5225775
Atul Vasudevan	- z5199180
Firzad Ahammed	- z5216888
James Gabor	- z5146610

Overview

Introduction

The Waitless software is a restaurant management system with the administrative, staff, kitchen and customer components working in full conjunction with each other, making the often tedious and stressful job of setting up, maintaining and managing a restaurant efficient, reliable and painless. The stress and unreliability of written and word-of-mouth orders used in traditional non-automated restaurants has been superseded by digitally stored orders, with the movement of customer requests to the kitchen now seamless. Management and updating menus, including customising pre-existing items with new ingredient options and adding completely new entries, is now a matter of clicks away on the administrative dashboard, annulling any need for the manual reprinting of a new menu. Customers can easily request custom menu items with specific remarks for the kitchen and order at their own discretion, not at the availability of wait-staff. By streamlining and automating the most essential aspects of any restaurant, the Waitless application allows the restaurant's focus to be on the setup and quality of their service, not on the minutiae of management.

The production and delivery of the Waitless software took place in 4 components, each with their own dashboard for use by their respective users: the Administrator, Customer, Kitchen and Staff dashboards. Each of the 4 components of the software had a different design philosophy and goal in its presentation and use.

- Administrative component
The administrative component is where the setup of the restaurant occurs, with admins being able to create menus, tables and view analytics. The goal was to create a fully customisable restaurant system, totally in control of the restaurant administration.
- Customer component
The customer component is placed at each table on a screen for people to see and interact with when seated. On this dashboard is the full restaurant menu, showing all items and their customizability. Customers can order items and finish their session from here. The design mentality was for an easily filtered and efficiently presented menu where all items are evenly spaced and all custom options are easily accessible for the customer.
- Kitchen component
The kitchen component is used as a display portal for incoming orders from tables. Here the chefs can select which items they wish to make and can send them out in the order best suited to them.

- Staff component

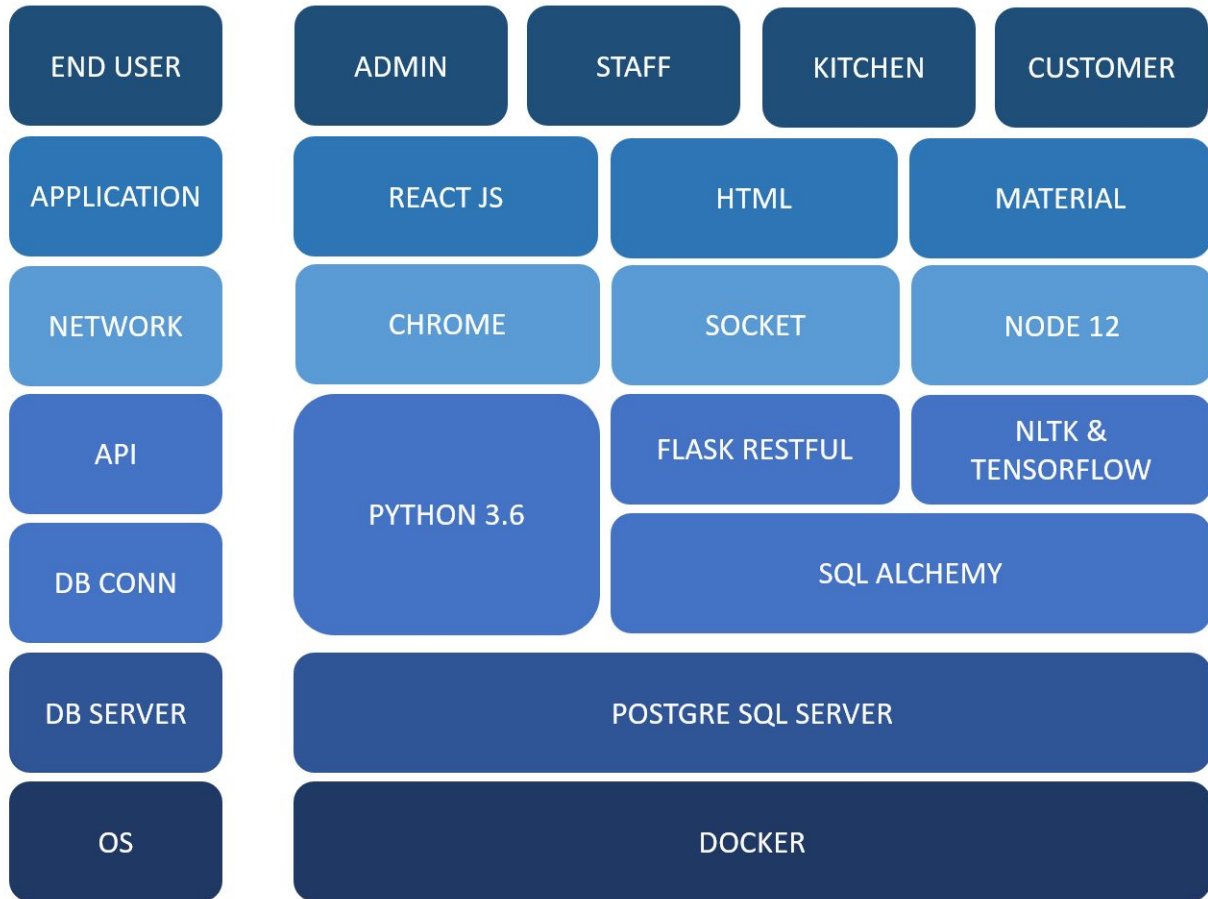
The staff component displays active and empty tables and allows staff to see when orders are ready to be delivered to the floor. Staff members can view previous order history for an active table as well as end a customer's session when they are ready.

The Waitless software's goal is to augment the workflow of a standard restaurant system, usually reliant on paper, memory and word of mouth, by using efficient and easily presented user interfaces and databases in conjunction with an automated table and order feed, giving the kitchen and waiting staff all the resources they need to painlessly manage the inflow and outflow of customers and their order requests.

For the customers, usage of the Waitless software will involve being seated at a table with a screen on it, set up by the administrator. When setting up the restaurant, the admin can select which table number to assign each table through the main entry screen of the program. Once a table number has been chosen, customers will enter and be greeted by an entry screen where they can start their order.

The development of the software took place over 4 main phases. The first was the design and planning where different features were proposed and a general procedure for the workflow throughout a customer's session was discussed. The second was the design of each of the 4 dashboards including a base UI and test data to mimic backend interactions. The third was the setup of a backend system to handle live database calls. Finally, all the components were integrated and the UI was revamped to fit any changes made along the way.

Software Architecture

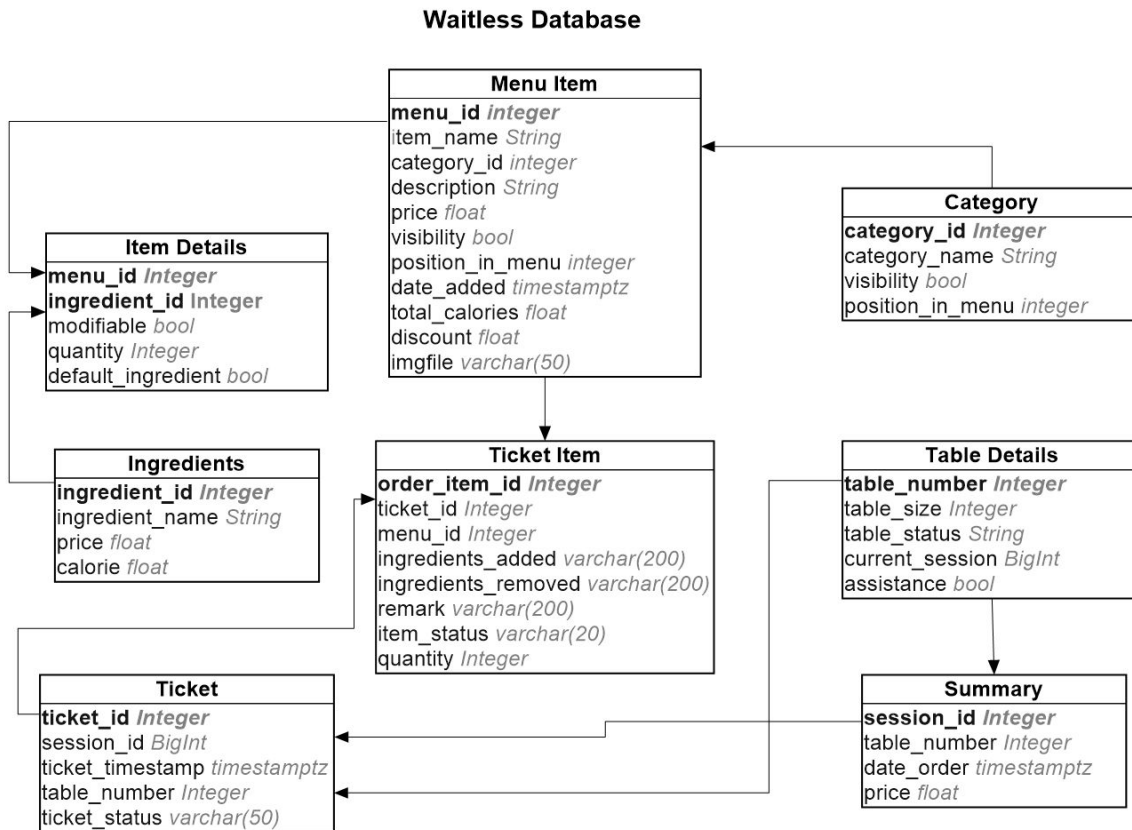


Operating system

The application setup is through docker, so the application can run on any operating system. We set all the required environment requirements for backend and frontend in the docker. See *user Manual* for more information on installation for different versions of docker.

Database

The application uses PostgreSQL database for storing data. We wanted to use the benefits of having a fixed schema and also reduce data redundancy so we chose to go with RDBMS. The database has 8 interconnected tables, together with storing static and dynamic data.



Backend

The server side is built using Python 3.6. It's a rest API run using the Flask-restful framework. The Model-View-Controller architecture is followed for the endpoints. SQLAlchemy ORM has been used to connect to the database. The models are defined as DB tables. For chatbot, we used NLTK, Tensorflow and TFLearn. The orders are sent to the kitchen via socket, for this flask-socketio was used.

Web page

We chose ReactJS, which is a javascript framework, to develop the client-side web application. The react application consists of components for every view and the server renders the view for which the data has changed. This makes the user experience seamless.

For styling, we used Material-UI which is compatible with ReactJS.

Features and Functionality

Waitless Technical Overview

The Waitless application is divided between 4 dashboards: administrator, customer, staff and kitchen; each operating simultaneously in conjunction with each other, communicating through a common backend server, passing necessary customer and table information between each other. The database tables used to store the restaurant information can be considered in two categories:

- *Menu details*
Included here are 4 tables containing all information of the restaurant menu; **category** (list of menu categories), **menu** (list of menu items), **ingredients** (further details of menu items) and **itemdetails** (custom modifications to menu items, eg. vegan, extra chilli etc.). These tables are created at the setup of the restaurant from an administrator and can be updated from only an administrator at any time.
- *Table details*
This category contains information about customer sessions in the restaurant, past and present. This includes **table_details** (list of tables in the restaurant and their current status), **summary** (summary of a table session including price), **ticket** (an order placed by a table) and **ticket_item** (further details on contents of a ticket)

On setup of the Waitless application, an administrator should first generate the data for *Menu Details* in the administrator dashboard. A demo database has been included in the submission for a sample menu, see the user manual for how to turn this on and off. Upon a customer being seated at the restaurant, they are assigned a *session_id* in the **summary** database which will track their order throughout their meal. Upon the customer orders, **ticket** and **ticket_item** entries are generated to be passed to the kitchen and staff dashboards. From here, the kitchen and wait staff will staff orders as ready and delivered respectively. Upon completion of a session, the **summary** table is updated with the final price.

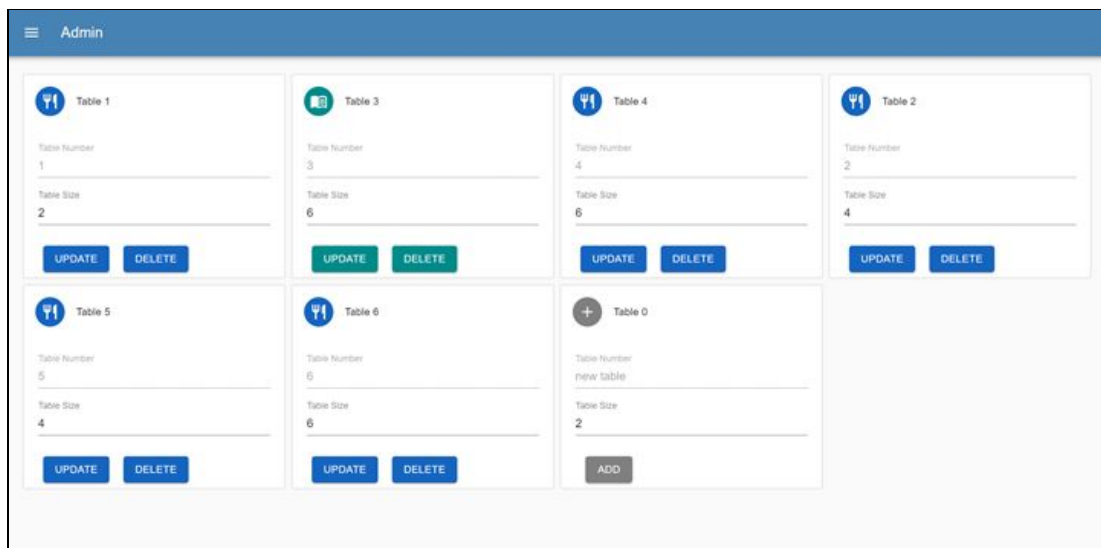
Specific features and technical details for each dashboard are found below.

Administrator Dashboard

The Administrative dashboard is split into 3 parts, Menu Settings, Table settings and Analytics. For all the screens we decided to go with simple material components. The administrative dashboard is designed as the central hub of the restaurant management, to be used for the creation and modification of restaurant organisation, including the menu and tables, as well as to view restaurant statistics over its active operation.

Table Settings

Here, the administrator can add, remove or modify pre existing tables in the restaurant. Table numbers are locked, however the size of each can be modified. The icons and colours of each table reflect the current status, seated, ordering or empty.



Creation of tables is done through a backend API POST command in `'/Tables'`.

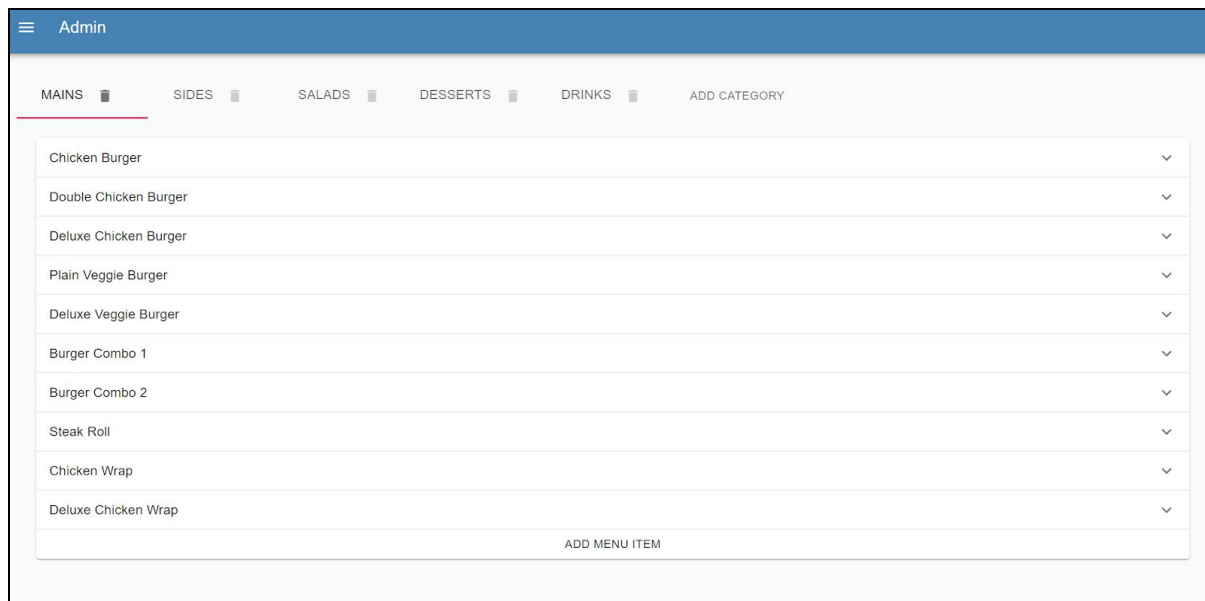
```
axios.post<Tables>(`Tables`,{'table_size': table_size}).then(  
  (res:ServerPostResponse) => {  
    setTables((tables) => [...tables, res.data]);  
  }  
)
```

In the frontend, the Admin dashboard sends only the table size, the backend initialises the status and assistance automatically in our Database Model..

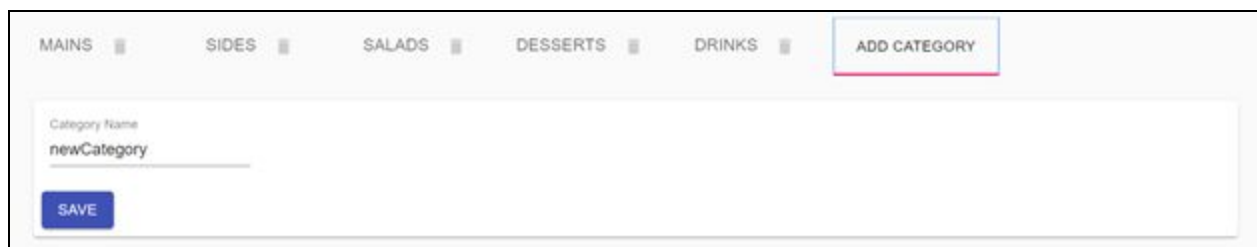
```
class TableDetails(db.Model):
    ...
    def __init__(self, size):
        self.table_size = size
        self.table_status = "Empty"
        self.assistance = False
```

Throughout a customer session, the fields *table_status* and *assistance* are updated, with the icon and colour of the tables filtered from the *table_status* field.

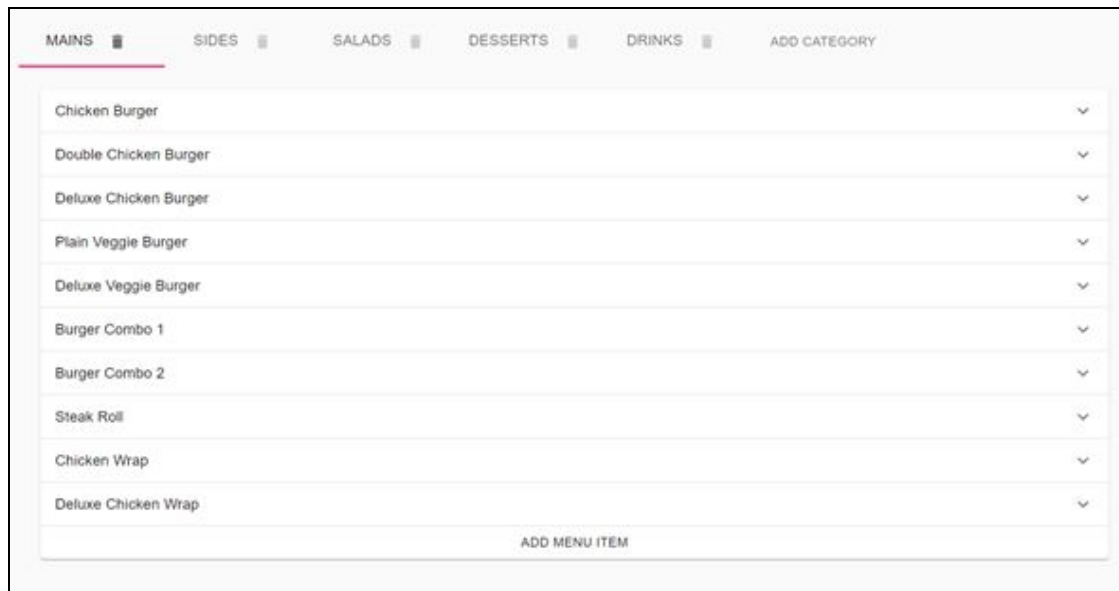
Menu Settings



In the menu screen, the administrator can create custom menus, complete with modifiable/persistent ingredients, images, pricing and categories. Addition and removal of menu categories is simple and intuitive, allowing for complex menus which can be modified quickly and easily by administrators.




Deleting a category can be done by clicking on the delete icon on the category tab. To prevent unintended data loss, pressing the delete button will bring up a prompt confirming if the deletion is intentional. Only after confirmation will the backend call be made.



All menu items are pulled at the load of the page and filtered as the administrator clicks through the categories as seen in the code below:

```
menu.filter((item)=>item.category===cm.category_name).map((item,index)=>(  
    <Item item={item} key={item.menu_id} updateMenu={updateMenu}/>  
    ))
```

This helps prevent unnecessary load on the backend server and allows for a faster user experience. On click of a menu item, the administrator will be greeted with details on the order item. From here, the item can be changed and updated. All previous settings for the menu item are retrieved and displayed. The administrator can elect to modify or delete the menu item. If it is removed, it will no longer be visible from the menu of the customer dashboard.



Menu Item Name

Chicken Burger






Price

12

\$12

Description

100% Australian grown chicken

Ingredient (100g serving)	Modifiable	Action
Glutton Free	<input checked="" type="checkbox"/>	
Chili	<input checked="" type="checkbox"/>	
100% Australian Chicken Patty	<input type="checkbox"/>	
Sesame Seed Bun	<input type="checkbox"/>	
Ingredient	<input type="checkbox"/>	

SAVE

DELETE

```

React.useEffect(()=>{
  if (!isNew){
    axios.get(`ItemDetails/`+String(item.menu_id)).then(
      (res: ItemDetailsJsonResponse)=> {
        setIngredients(res.data);
      })
    setName(item.item_name);
    setDescription(item.description);
    setPrice(item.price);
  }
}, [item, isNew]);

```

The above code updates the database with the new menu item input. Deletion of menu items is implemented through a soft delete, switching the visibility of the menu item to false. This implementation is seen in the code below:

```

axios.patch(`Menu/`+String(item.menu_id), {"visibility": false}).then((res)=>
    props.updateMenu()
)

class MenuItems(Resource):
    @marshal_with(menu_resource_fields)
    def get(self):
        """Return the list of all MenuItems."""
        return
Menu.query.filter(Menu.visibility==True).order_by(Menu.menu_id).all(), 200

```

When an administrator clicks the add menu item button, they are led to an input screen where they can customise the new menu item.

The screenshot shows a web form titled "Add New Menu Item". On the left, there is a large cloud icon with a white arrow pointing upwards, and below it is a blue button labeled "UPLOAD IMAGE". To the right of the upload area, there are three input fields: "Menu Item Name" (with the text "Untitled"), "Price" (with the text "0"), and "Description". To the right of the "Price" field, the text "\$0" is displayed. Below the "Description" field is a table with three columns: "Ingredient (100g serving)", "Modifiable", and "Action". The table has one row with the text "Ingredient" in the first column, a checkbox in the second column, and a "+" sign in the third column. At the bottom right of the form is a blue button labeled "ADD".

To upload a picture, the administrator can click on the cloud icon, select the file and click on upload.

```

let formdata = new FormData()
formdata.append(`file`,file)
axios.post(`Image/default.png`, formdata).then(
    (res: any)=>props.setImageFile(res.data.filename);

```

We are storing the uploaded image in the backend in a static folder. Allowed image types are: png, jpg, jpeg, gif.

Below is the class API code for posting and getting images:

```

class Image(Resource):
    def get(self, file_name):
        print(app.config['UPLOAD_FOLDER'], file_name)
        try:
            return send_from_directory(app.config['UPLOAD_FOLDER'], file_name)
        except Exception:
            return send_from_directory(app.config['UPLOAD_FOLDER'], ErrorFile)

    def post(self, file_name):
        data = parser.parse_args()
        ...
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            resp = jsonify({'filename' : filename})
            resp.status_code = 201
            return resp

```

The database stores the URL to these images which are uploaded server side.

```

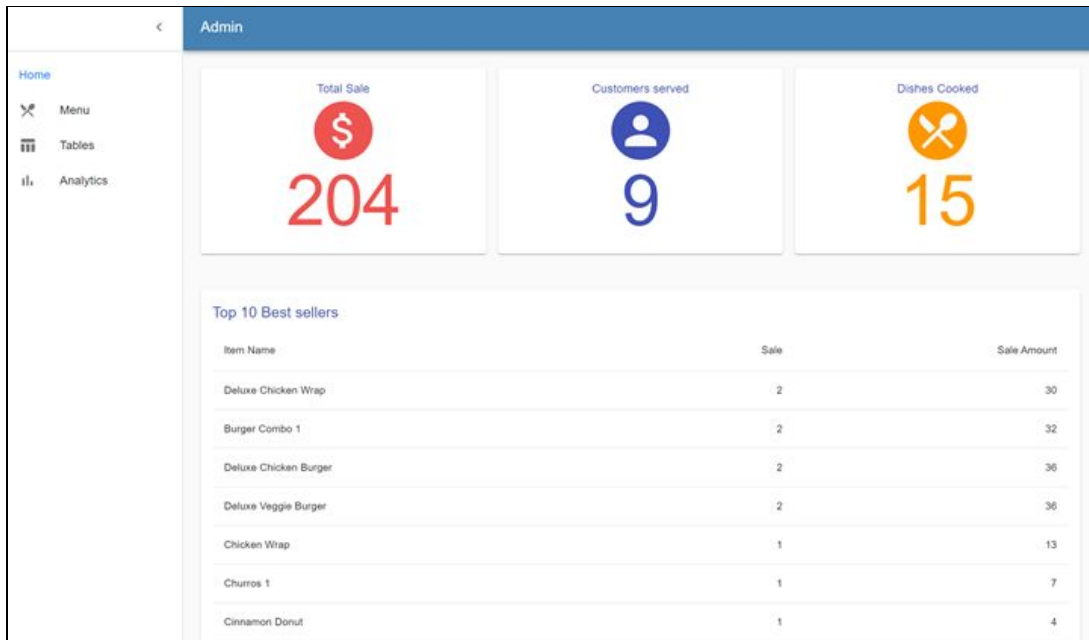
class ImageURL(fields.Raw):
    def format(self, value):
        return app.config['BASE_URL']+url_for('image', file_name=value)

```

Once the administrator provides a display image, name, description and price, the menu item is pushed into the server and can be displayed within the menu of the customer dashboard.

Analytics

The analytics screen is a summary screen to provide management with an overview of the restaurant's status as well as long-term financial statistics. Here, the administrator can see the total restaurant sales, total dishes cooked and the number of customers served. Past orders are consolidated into a list of the top sellers in the restaurant.



Below are the calls to the backend for the total sales, customers served, dishes cooked and best selling items:

```
axios.get(`Summary/Day`).then(
  (res: any) => {
    const summary = res['data'];
    setSale(summary['total_sale'])
    setCust(summary['total_customers'])
  }
)
axios.get(`DishSummary`).then(
  (res: any) => {
    const summary = res['data'];
    setMeal(summary['total_dishes'])
    setBestSellers(summary['top_10'])
  }
)
```

Customer Dashboard

The customer dashboard consists of the entry screen and the Menu screen. These are the screens which will be visible to the customer for placing an order from the list of items and initiating the payment process when the entire order is complete.

Customer Entry

The Customer Entry Screen is what the customer is greeted with once they are seated at a table.

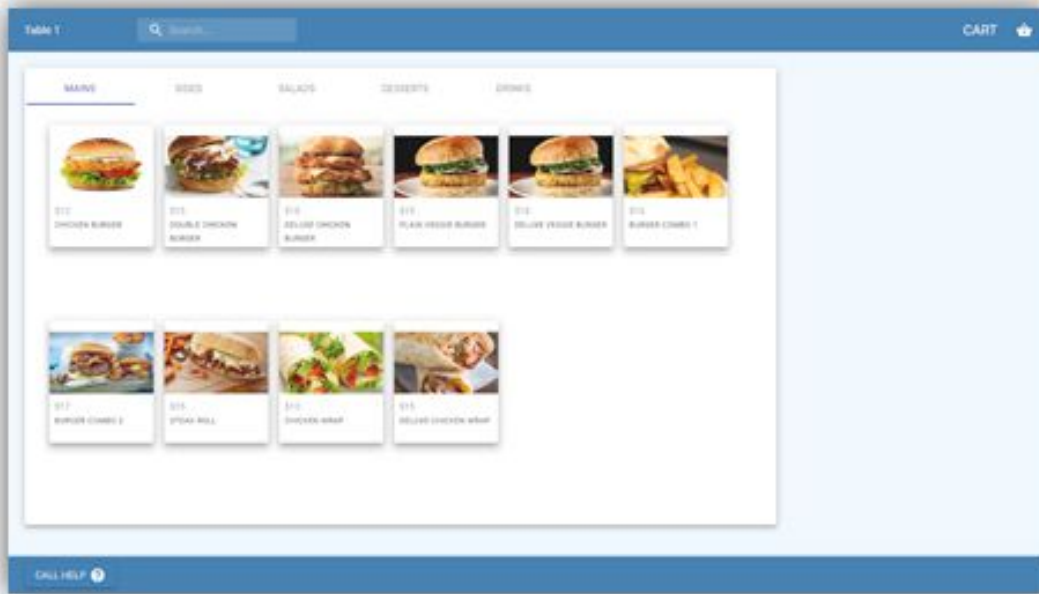


The Customer can see the table number on screen along with a 'Start Order' button. Once the Customer is seated, they can press the 'Start Order' button to start the session. The session and table details will be updated in the database.

On click of 'Start Order' the Customer Menu Screen will be visible to them. On this screen, the customer can choose items from the menu, place orders and initiate the payment process.

Menu

Once the customer begins a session, they can choose from a set of categories and menu items set by the admin. Each Category has a list of menu items under it. An image and price associated with each menu item is visible to the Customer under each Category.

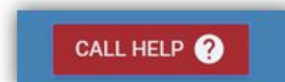


Search

The customer can filter items by the Search functionality. They can search for a menu item using keywords, or the entire menu item name. Once a query is entered the search results are visible under the respective categories.

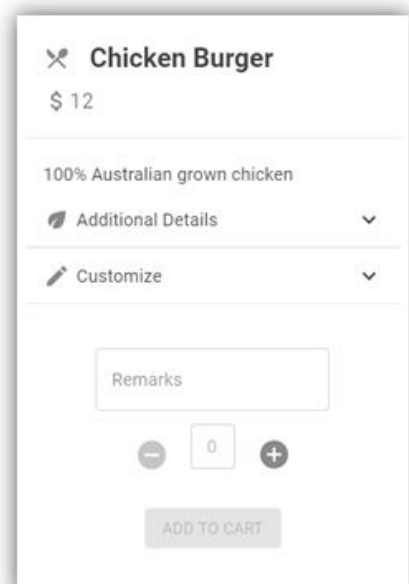
Assistance

If the Customer requires assistance during their session, they can click the 'Call Help' button. This will flag the wait-staff and have them come over to help.



Modify Order

On selection of an item, the item can be modified. The customer can view the description, 'Additional Details' and a list of the customizable ingredients of the menu item. The additional details panel contains the list of default ingredients present in the item. The customer can send remarks to the kitchen if they wish to do so. This can be used to alert the kitchen of any additional dietary requirements or requests. The quantity of the item can be selected in this component. Once modifications are done, the item can be added to the bucket, seen below.



Cart

The added menu items can be viewed in the bucket (cart). The customer can remove a menu item prior to placing the order by clicking the delete icon under 'Current Items'. The previously ordered items can be viewed in the bucket with a checkbox corresponding to it under 'Previous Orders'. This component is used to place an order with one or many items. Once all orders are complete, the customer can initiate the payment process.

> Hide

Current Items

Coke

\$4

Wedges x2

\$12

\$

Previous Orders

^

✓

Chicken Burger

\$12

✓

Burger Combo 1

\$16

PLACE ORDER

PAY BILL

Payment Screen

Once the Customer clicks on the 'Pay Bill' button in the Cart, they will be redirected to the 'Paying' screen until the staff assists the Customer with the Payment. This will prevent a new order from starting at the same table until the previous order has been cleared. It also allows the wait-staff to clear and prepare the table before the next customer arrives. Once the Bill has been paid, the screen will reset and the next customer will be greeted with the 'Customer Entry' screen.

A screenshot of a mobile application interface for 'Waitless Restaurant'. The screen displays 'Table 1' and a 'PAYING...' button. The background is a blurred image of a restaurant interior with warm lighting and people.

Code Snippets

The code seen below from *client/src/components/Customer/CustomerEntry.tsx* is the database update done once the Customer begins an order. On start, a new session is created. At the end of the order, this session is closed and the table is updated with the amount paid. The status of the table is changed at the staff end to 'Seated'. Once the database table updates are done, the customer can start with the order placement.

```
function handleEntryCustomer(){
  axios.post<Summary>('Summary',
    {'table_number':table_number
      , 'date_order':new Date().toISOString()
        .slice(0,19).replace('T',' ')}))
    .then(
      (res)=>{
        const session = res.data
        updateCurrentSession(session.session_id)
        if (table_number !== null){
          axios.patch('Tables/status/'+table_number.toString()
            ,{'table_status':'Seated'
              , 'current_session':session.session_id})
            .then((res)=>{
              setPage(1)
            })
        }
      })
    }
}
```

The categories and menu items to be displayed are set by the admin and are retrieved in *client/src/components/Menu.tsx*. First the categories are retrieved from the Customer Table, followed by Menu item details from the Menu table.

```
React.useEffect(() => {
  if (current_category.length === 0){
    axios.get('Categories').then(
      (res:CategoryResponse) =>{
        //Parse response and set state
      }
    )
  }
})
```

```

    if (menu.length === 0){
      axios.get('Menu').then(
        (menuItem:MenuResponse)=>{
          //Parse response and set state
        }
      )
    }
  })
}

```

Once the customer clicks on an item to modify the menu details which we get from the database are taken to the Modify order menu at *client/src/components/ModifyOrder.tsx*. When we initially make a get call for the menu item details we do not retrieve the ingredient details present in the Item Details table for each menu item. Here we retrieve the ingredients and use it for 'Additional Details' and 'Customize'.

```

    axios.get('ItemDetails/'+menuD.menu_id.toString()).then(
      (res:ItemDetailsJsonResponse) =>{
        //Parse the response
      }
    )

```

After the customer has selected the order items into the bucket, the order is placed on click of 'Place Order'. The Ticket table is first updated with the session id and the table associated with the ticket. A ticket consists of multiple items and this ticket is what the kitchen views. A ticket comprises multiple ticket items and we update the TicketItem table with the details associated with each ticket. Once the order the items are moved from the Current Items panel to the Previous Orders panel.

```

    axios.post<Ticket>(`Ticket`, { 'session_id': current_session,
      'table_number': table_number }).then(
      (res: TicketPostResponse) => {
        //Parse response
        Promise.all(itemList.filter(ticket_item =>
          !ticket_item.ordered).map((ticket_item) => (
            axios.post<TicketItem>(`TicketItem`, {
              /* List of items details to be posted */
            })).then(
              (res: TicketItemPostResponse) => {
                //Parse response and set state
              }).catch(error => console.log(error))
            ))
        ).then(() => {

```

```

        //clear the bucket and set flag
    })
}
)

```

At the backend, the menu item is added into the database. This is done in *server/core/views/ticket_item.py*.

```

class TicketItem(Resource):
    @marshal_with(ticket_item_resource_fields)
    def post(self):
        args = parser.parse_args()
        new_item = TicketItemModel(''set values for each field to be
updated'')
        db.session.add(new_item)

```

As mentioned above, the Customer can use the 'Call Help' button for Staff Assistance.

```

function addAssistanceTable() {
    axios.patch('Tables/Assistance/' + table_number.toString(), {
'assistance': !assistance_click })
    setAssistanceClick(!assistance_click)
}

```

Once the assistance button has been triggered by a customer, the page polls the database for updates from the staff end, turning the button off if a staff member elects to turn off the assistance button from their dashboard.

Kitchen Dashboard

The Kitchen Display System (KDS) is designed as a bump screen which displays all the necessary information in a very minimalistic design. Since the kitchen side is always busy with the food preparation, the system is designed to reduce the number of clicks necessary to view orders and have the majority of information automated.

The KDS consists of mainly two parts:

- Order Ticket Filter

The order ticket filter is used to filter the order tickets and ordered items for a more focused view. This is especially helpful if different areas of the kitchen cook specific types of food and want to get a focused view of the order tickets without showing any of the items unrelated to their department. This helps declutter the view for an effective service.

The filter bar lists the different categories as created by the admin. Selecting a category filters out all the other items from order tickets and displays only the tickets that are relevant to the selected category.

- Order Tickets

Order tickets basically consist of the items and relevant details for that particular order. Each order ticket displays the order number, the table number, time elapsed and the item details along with modified ingredients and remarks.

The order tickets are arranged according to the time at which the orders were placed. New incoming orders are automatically added to the bump screen without the need of any user interaction.

Tickets are colour coded to easily convey the time urgency of the order. Three colors are used to highlight the time elapsed since the order was placed:

- Green

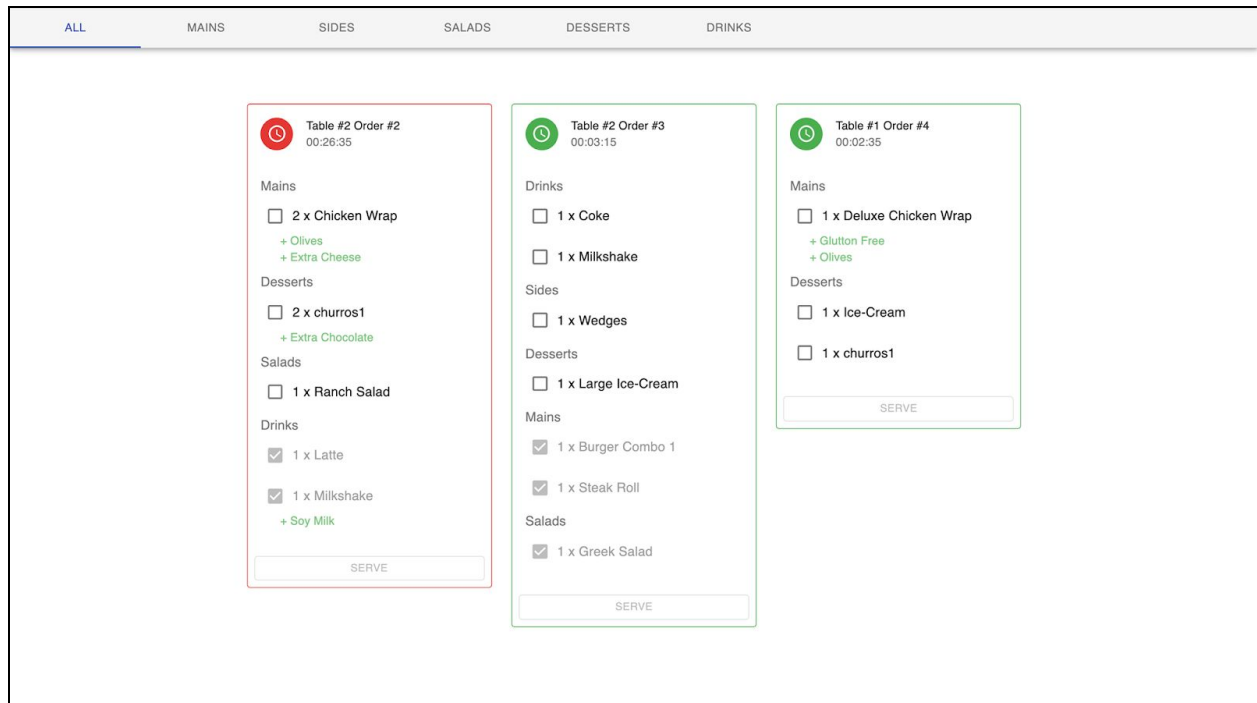
Green represents newly placed orders where the time elapsed is less than 10 minutes.

- Yellow

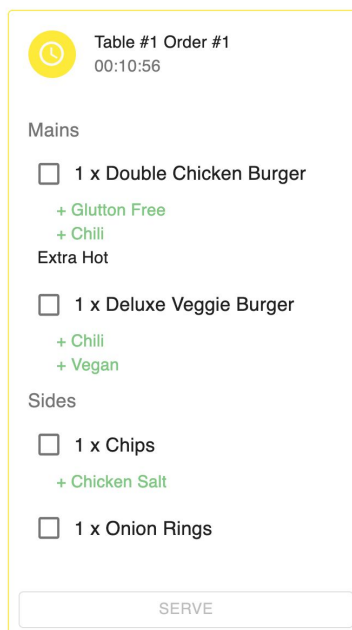
Yellow represents orders which have been placed for just over 10 minutes.

- Red

Orders marked in red need to be attended to immediately as the elapsed time exceeds 15 minutes. Red order tickets have the highest priority and requires immediate attention



Above is a sample of the kitchen with two currently active tables, with table 2 having placed two separate orders over their session. We can see the flexible nature of the ticket boxes, as well as the colour used to convey urgent and low-priority orders.



Above is a close up on the format on the order tickets themselves. The first division is on category, each able to be filtered by the app bar at the top of the page. Customised

ingredients appear in green and custom remarks appear as black text under the order item.

The ordered items are divided into sections based on the category. Each item will also show the modified ingredients in green. Any remarks or notes to the kitchen staff from the customer will be displayed beneath the corresponding item.

Once the order items have been prepared, the staff can select the completed items and mark them as ready to be served. This would send an alert to the staff dashboard, notifying them that an order is ready to be delivered to that particular table. Once all the items in the corresponding order ticket are marked as done, the order ticket is automatically removed from the bump screen.

In the implementation of the KDS, a major design priority was a real time efficient updating system which would immediately show an order as soon as it was placed by a customer. To ensure maximum efficiency, sockets were used to interact with the backend.

```
useEffect(() => {
  // Socket registered to listen to new tickets added to the system
  socket.on('ticketsUpdated', () => {
    updateTickets();
  })
}, []);
```

Above is an extract from *client/src/components/Kitchen/Kitchen.tsx*, where the frontend UI connects to a socket defined on the backend, updating the displayed active ticket whenever an item is added from a customer order. The socket labelled *'ticketsUpdated'* is an API flag whenever an item is changed, used to reduce the load on the server.

```
class TicketFlag(Resource):
    def get(self):
        socketio.emit('ticketsUpdated', broadcast=True)
        return [], 200
```

The socket call is emitted from *server/core/views/ticket_item.py*. After the flag has been set, the *updateTickets()* handle is called, contacting the backend API to pull all active orders.

```
class ActiveTicketMenuItems(Resource):
    def get(self):
        """get all ticket details"""
        tickets = TicketModel.query.filter(
            TicketModel.ticket_status == 'Active').all()
        ticket_json = []
```

First, the ticket table is queried for all tickets which are flagged as active. Once a ticket is completed, their status is set to 'Delivered', keeping it in the database for analytic access

later for administrative purposes. Next, each ticket item within each ticket is queried by its ticket id:

```
for t in tickets:
    ticket_items = TicketItemModel.query.filter(
        TicketItemModel.ticket_id == t.ticket_id).all()
    ticket_total = []
```

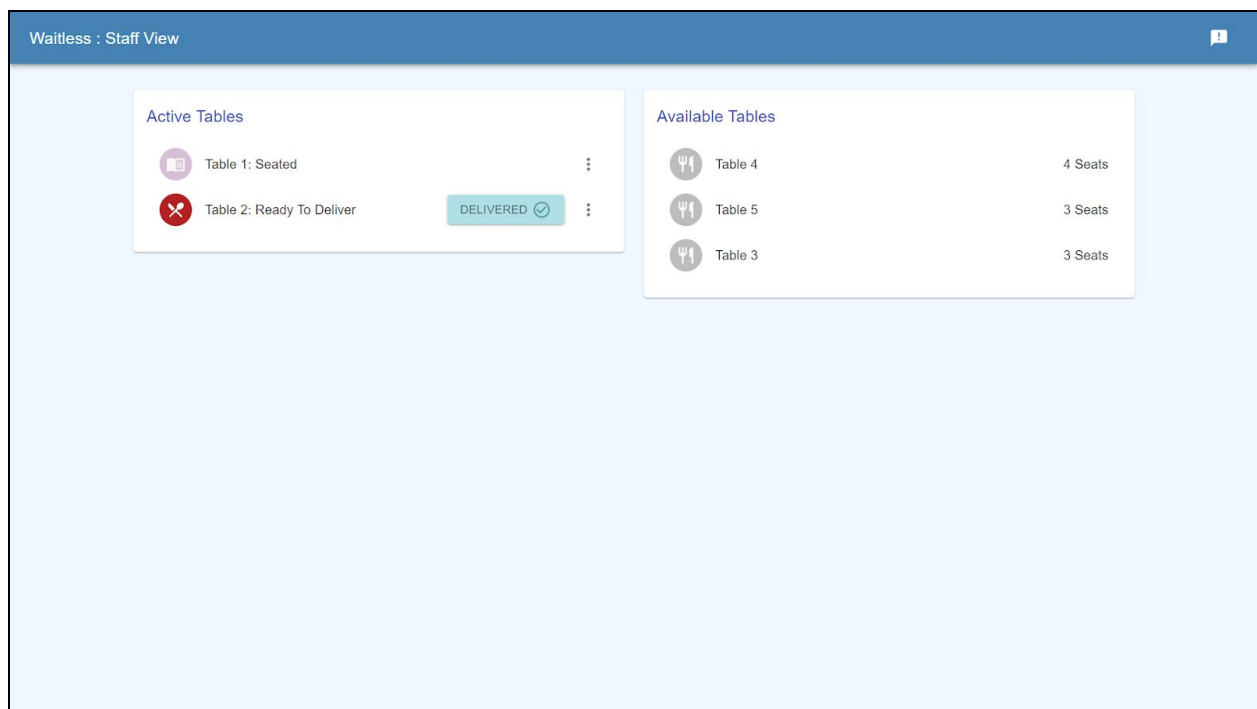
Then for each ticket item, then menu details are pulled. This will allow additional ingredients to be displayed to the kitchen.

```
for ticket_item in ticket_items:
    menu_entry = MenuItemById().get_no_marshall(
        ticket_item.menu_id)[0]
    menu_marshall = marshal(menu_entry, menu_resource_fields)
    ticket_item_marshall = marshal(ticket_item, ticket_item_resource_fields)
    ticket_marshall = marshal(t, ticket_resource_fields)
    ticket_total.append(**ticket_marshall,**ticket_item_marshall, **menu_marshall)
ticket_json.append(ticket_total)
```

The marshal commands convert the class objects pulled through the *flask_restful* database models, as defined in the class files in *server/core/models/** into json, After these are iterated, the full list of active menu items are returned.

Staff Dashboard

The design mentality for the staff dashboard was minimalism and automation, having the busy wait staff needing as few clicks as possible, with the essential data and functionalities being displayed and refreshed automatically, with more details readily available. The display of each table's status is the forefront of the dashboard display, being the most essential piece of data for staff to view. The screen is split into two sections; one for the feed of currently active tables and their status, the other for the feed of free tables. As customers are seated, order food and pay, tables will move between the two columns. When an item of food is ready to be delivered, staff can press a single button to flag that the order has been sent to the table. If a staff member requires to view the previous orders, they can click on the details of the table and view its ordering history. This history will also contain the current status of each order item, either in preparation or delivered.



Above is a sample image of the staff application where there are 5 tables in the restaurant, 2 with active seating. Table 1 is seated and in the process of ordering, table 2 has ordered and the food is prepared and ready to be delivered in the kitchen.

The live feed of the tables was implemented through backend server calls utilising flask-sqlalchemy and axios API. The backend is continually polled over a constant interval for any changes to the status of the restaurant tables. This polling pulls only the overview of the table details, including its number, whether its active or vacant and status. Further details such as previous order history are polled only when a staff member clicks on the button for more information, keeping server calls to a minimum.


```

useEffect(() => {
  const interval = setInterval(()=>{
    axios.get(`Tables`).then(
      (res: ServerResponse) => {
        /*
         parse the response and update table data
        */
      }
    )
  }, 1000)
  return () => clearInterval(interval)
})

```

Seen above is the API call from the javascript frontend in *client/src/components/Staff/Staff.tsx* to parse the table data from the 'Tables' page in the backend. Updates are called every 1000ms (1 second). On the backend, when pulling the details for a specific table, including the past and current order details, the database is polled and filtered for the current session.

```

class TicketItemsBySession(Resource):
    def get(self, session_id):
        """get all ticket details for a given table session"""
        tickets = TicketModel.query.filter(
            TicketModel.session_id == session_id).all()
        ticket_json = []

```

Above is the first part of the backend code for the API request for ticket items by a session, taken from *server/core/views/ticket_item.py*, used by the staff end to display the full order history for an active table. **TicketModel** one of the *sqlalchemy* database objects, with all tables in the Waitless database having a python model class replicating its columns. After the **ticket** table is queried for the session, each ticket is subsequently iterated:

```

for t in tickets:
    ticket_items = TicketItemModel.query.filter(
        TicketItemModel.ticket_id == t.ticket_id).all()
    ticket_total = []

```

If a customer at a table has put in multiple orders (which may contain multiple items), this will be iterated here. Next, each ticket item for each ticket is appended to a json object then returned

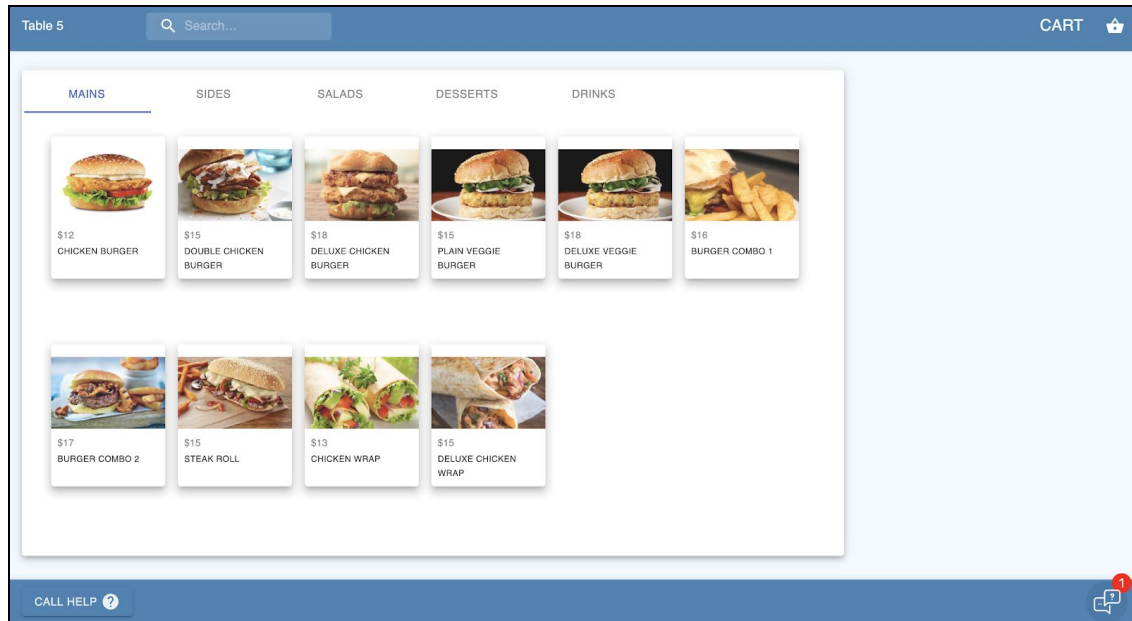
```
for ticket_item in ticket_items:
    menu_entry = MenuItemById().get_no_marshal(ticket_item.menu_id)[0]
    menu_marshall = marshal(menu_entry, menu_resource_fields)
    ticket_item_marshall = marshal(ticket_item, ticket_item_resource_fields)
    ticket_marshall = marshal(t, ticket_resource_fields)
    ticket_total.append(**ticket_marshall,**ticket_item_marshall, **menu_marshall)
ticket_json.append(ticket_total)
```

Similar methodologies are in place for other server API calls. The staff dashboard will also poll the backend when flagging an order as delivered, when switching off customer assistance and when finalising the payment for a table, thereby ending the session.

In order to prevent malicious attacks of customers ordering then simply leaving, completely ending the session is the responsibility of the staff after the customer flags that they are ready to end their session with the 'Pay now' button. Once a customer presses the 'Pay now' button, they are taken to a waiting screen outside of the menu. On the staff screen, the status of their particular table is changed to paying. The customer would then move up to the register where they can pay. Once the staff members have cleaned and organised the table for the next guest, they can click on the table details and press the 'Finish order' button. The design goal here was to give control to the staff when finalising orders and clearing restaurant tables. No customer will be able to sit and enter the menu while a staff member has not finished the order, ensuring payment is received and the table is ready for the next customer.

Chatbot

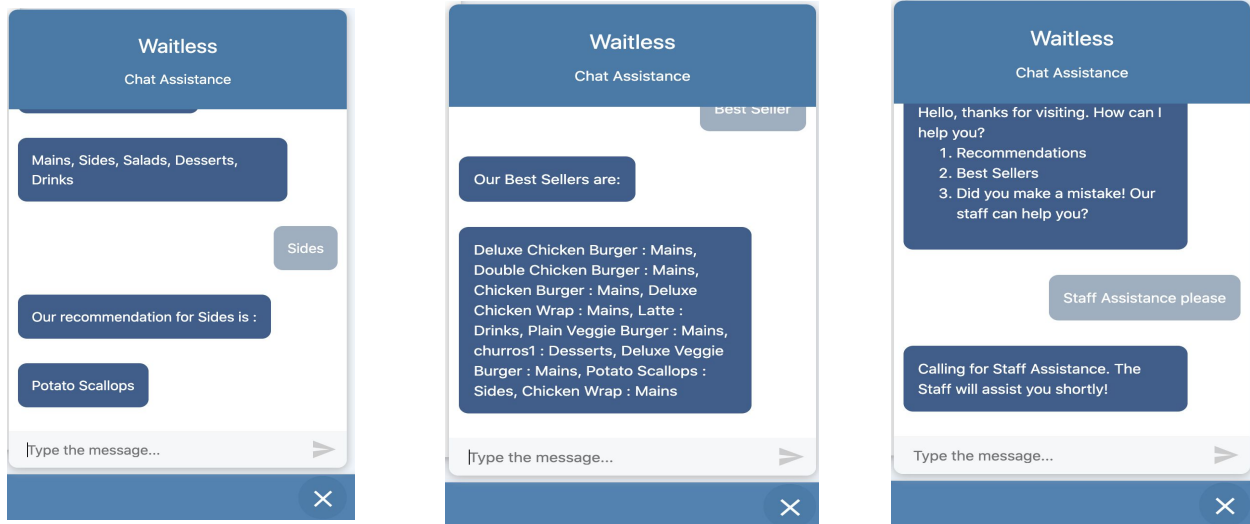
The chatbot is an ai system for assisting customers during their order, seen in the bottom right corner of the Customer dashboard.



The chat bot widget is designed as a user-friendly environment for the customer to interact with. Once this widget is clicked, a chat session is initiated between the customer and the assistant 'Jenny'. Once the chat session is initialised, the customer is greeted with 3 different interaction options:

- Recommendations
The *recommendations* command polls the backend for the restaurant's prior order history for a category specified by the customer, selecting its menu items which have sold the most over the restaurant's lifetime. After entering the recommendations command, the customer can type the category they wish to get the top selling items from.
- Best Sellers
The *best seller* option polls the backend for the full order history of the restaurant in all categories, filtering on the top 10 best selling options. The data displayed is live and will continually update as customers come through the restaurant.
- Staff Assistance
The *staff assistance* option is an alternative to the call for help button, notifying the staff dashboard that the table requires assistance

Screenshots below outline the functionality of the above 3 chatbot features respectively.



Backend functionality of the chatbot utilises AI to handle customer requests and parse live order data. Potential user intentions are found in *server/core/chatbot/intents.json*, defining the possible customer input commands for the chatbot to parse.

In *server/core/chatbot/pre_processing.py*, the input commands are tokenised and their intents determined.

```
for intent in data["intents"]:
    for pattern in intent["patterns"]:
        wrds = nltk.word_tokenize(pattern)
        words.extend(wrds)
        docs_x.append(wrds)
        docs_y.append(intent["tag"])

    if intent["tag"] not in labels:
        labels.append(intent["tag"])

words = [stemmer.stem(w.lower()) for w in words if w != "?"]
words = sorted(list(set(words)))
```

After the inputs have been tokenised and the intents determined, the function **create_model()** is called, utilising tensorflow and tflearn libraries to predict the desired Chatbot functionality

```
def create_model(words, labels, training, output):
    tensorflow.reset_default_graph()
    net = tflearn.input_data(shape=[None, len(training[0])])
    net = tflearn.fully_connected(net, 8)
    net = tflearn.fully_connected(net, 8)
    net = tflearn.fully_connected(
        net, len(output[0]), activation="softmax")
    net = tflearn.regression(net)
    model = tflearn.DNN(net)
```

The model can then predict the results for output:

```
results = model.predict([input_processing.bag_of_words(inp, words)])[0]
```

With the user's intent determined, the required information is polled from the backend and returned via an API POST command from *server/core/views/table.py*

```
def post(self):
    args = parser.parse_args()
    message=args.get('message')
    chatbot_response, item_list = chatbot.chat(message)
```

The Chatbot frontend in the Customer dashboard will then receive this return statement and can display the backend output within the widget.

External Libraries

Material UI - A library for building UIs with React [1]. This was used for developing and designing the front end for all the screens.

Axios - A promise based HTTP client for browser and node.js [2]. This is used to send asynchronous HTTP requests to REST endpoints. In this project it is used with React for get and post calls.

Socket IO - A javascript library for real time web applications [3]. This is used by the client and the server to talk to each other in real time. In this project we use it for the staff to contact the kitchen in real time, and also for the customer to contact the staff for assistance.

Moment - A javascript library for parsing, validating, manipulating and formatting dates [4]. We have used it in the kitchen UI for formatting the dates.

Tensorflow - A free open source library used in Machine Learning applications [5]. Here we are using it for model building and prediction.

Natural Language Toolkit - Suite of libraries and programs for natural language processing [6]. It is used for processing the user input and chatbot response word by word.

TFLearn - A deep learning library built on top of TensorFlow [7]. This supports TensorFlow in model building and prediction.

Numpy - Library for Python programming language to support multidimensional arrays and matrices [8]. This has been used in the chatbot.

Implementation Challenges

Several design and implementation challenges were faced along the development of Waitless, requiring joint discussion and effort to overcome. The biggest being the widespread impact of the domestic lockdown from the COVID-19 pandemic, severely hindering our ability to interact effectively and giving us each personal emergencies to handle. Without any weekly face to face interaction, coordination between code changes and feature implementation was difficult and often required additional team calls.

The format and structure of the database backend was a challenge over the middle 3 weeks of the project, with on the fly changes needed as new feature implementations led to necessary alterations to the database structure. Once backend calls were linked in the kitchen side, several refactoring changes were put in place, meaning other changes were necessary across the customer and staff backend calls.

In the final week, it was found that there were issues with the sockets with connections dropping over continual use of the kitchen side. This was found to be an issue with listening sockets continually being created over the course of a session and was patched before demonstration.

Packaging the project was the final difficulty with networking and port issues in our first attempt of compiling the application as a docker container. With time and effort, a complete package was put together, significantly reducing the complexity of dependencies and additional packages.

All base requirements outlined in the project assessment have been met. Some additional features which were in the proposal were omitted from the implementation due to technical and practical clashes with the design.

The features which have been omitted in the release but are present in the proposal are:

- Ratings - The rating of each item based on historical data and user input
- Sorting active tables in the staff dashboard based on status/priority
- The order of menu categories and menu items being customisable by the administrator
- The feature to close the session of a table if the Customer forgets to do so. Our current payment implementation supersedes this design
- The staff cannot select the area on the floor where the customer would like to sit electronically, we elected this to be a physical process.

User Manual Documentation

Installation

1. Install a Docker Engine (Docker Desktop recommended) for the respective system.

<https://docs.docker.com/engine/install/>.

For older Mac and Windows versions, use Docker Toolbox if the system doesn't support Docker Desktop.

<https://github.com/docker/toolbox/releases>.

If Docker toolbox is required, additional setup steps are required before proceeding, see below

Docker will set up the required environment after downloading the respective packages.

2. After installation of docker, navigate into the root directory of the project: *capstone-project-masterminds*. Start the Docker Engine by running the Docker Desktop application and run the following command to build and set-up the containers for the web application.

```
docker-compose up --build
```

This will set-up the environment for the application using the required libraries, build and run the **client**, **backend** and **database** servers. By default, a demo database script is run to populate the menu and tables, see options below to revert to a clean database.

3. After the docker container **react-app** is running and outputs *"Compiled successfully"*, visit the Waitless homepage.

<http://localhost:3000/>

This url will host the deployed systems where all the four areas of the application can be accessed from the homepage.

4. To exit the server, **Ctrl + C** to gracefully exit.

This will only stop the servers for client, backend and database servers and preserve anything in the database.

5. To clear the created containers including the database, type the following command.

```
docker-compose down
```


This command clears the server containers from the system, but the cache is still maintained.

6. To clear the docker environment along with any cached data completely from the system, use the following command:

```
docker system prune -a
```

Docker Toolbox

If the target system is running an older version of windows and Docker Toolbox is required, additional steps are required.

1. Install Docker Toolbox and virtualBox
<https://github.com/docker/toolbox/releases>.
<https://www.virtualbox.org/wiki/Downloads>
2. Due to the virtualisation of docker toolbox, the address *localhost* is not available within or outside of docker toolbox containers. A different IP needs to be used. First, navigate to *capstone-project-masterminds/client/src/pathing.tsx*, comment line 1 and uncomment line 2. It should now look like:

```
//export const net_path = 'http://localhost:5000';  
export const net_path = 'http://192.168.99.100:5000';
```

3. Navigate to *capstone-project-masterminds/server/core/__init__.py*, comment line 11 and uncomment line 12. It should now look like:

```
#app.config['BASE_URL'] = "http://localhost:5000"  
app.config['BASE_URL'] = "http://192.168.99.100:5000"
```

4. Navigate in a command prompt or the docker-toolbox command prompt to the root directory of the project
5. Run the command

```
docker-compose up --build
```

to begin the installation and run. Wait until the **react-app** container outputs *"Compiled Successfully"*

6. The Waitless application can be accessed at
<http://192.168.99.100:3000/>
7. For shutting down and cleaning, see the above section from step 4

Install Options

By default we have included a demo restaurant setup, including 4 tables and a sample menu populated with items. This is free to be modified and was put in place to save time for demonstration. To run the code fresh with no tables and an empty menu, the initial database script needs to be changed.

1. Navigate to and open *capstone-project-masterminds/docker-compose.yml*
2. On line 16, the initial sql script is listed. *Initial.sql* will run the container with a demo restaurant while *initial_base.sql* will be for an empty restaurant
3. Comment line 16 and uncomment line 17. It should now look like:

```
#- ./database/initial.sql:/docker-entrypoint-initdb.d/initial.sql  
- ./database/initial_base.sql:/docker-entrypoint-initdb.d/initial_base.sql
```

4. Now run **docker-compose down** to clear the current session, then re-run **docker-compose up --build** to recreate the container with an empty restaurant.

To revert back, undo the comment changes and run the previous two commands again.

Troubleshooting

There is a known bug in some older systems of windows which run Docker Desktop where the virtualisation container's internal clock through Hyper-v becomes out of sync, resulting in the internal containers time being wrong. If the kitchen order tickets have time out of sync, it is due to this bug within the Hyper-v virtual box. This can be fixed by resetting the internal clock of the virtual box.

1. Open the Hyper-v manager application (type hyper-v manager in the windows bar)
2. Under the list of virtual machines, the DockerDesktopVM should appear. Click this and navigate to the right hand bar which appears. Click on settings in this right hand bar
3. In the new window, click on *Integration Services* under *Management*
4. Deselect *Time Synchronisation*, hit apply and reselect *Time resynchronisation*, apply again.
5. Close out of the Hyper-v manager and rerun the docker application. The virtual container's times should now be in sync

References

- [1] Material-ui.com. 2020. Material-UI: A Popular React UI Framework. [online] Available at: <<https://material-ui.com/>> [Accessed 28 April 2020].
- [2] npm. 2020. Axios. [online] Available at: <<https://www.npmjs.com/package/axios>> [Accessed 28 April 2020].
- [3] Socket.IO. 2020. *Socket.IO*. [online] Available at: <<https://socket.io/>> [Accessed 28 April 2020].
- [4] Momentjs.com. 2020. *Moment.js | Home*. [online] Available at: <<https://momentjs.com/>> [Accessed 28 April 2020].
- [5] TensorFlow. 2020. *Tensorflow*. [online] Available at: <<https://www.tensorflow.org/>> [Accessed 28 April 2020].
- [6] Nltk.org. 2020. *Natural Language Toolkit — NLTK 3.5 Documentation*. [online] Available at: <<https://www.nltk.org/>> [Accessed 28 April 2020].
- [7] Damien, A., 2020. *Tflearn | Tensorflow Deep Learning Library*. [online] Tflearn.org. Available at: <<http://tflearn.org/>> [Accessed 28 April 2020].
- [8] Numpy.org. 2020. *Numpy — Numpy*. [online] Available at: <<https://numpy.org/>> [Accessed 28 April 2020].
- [9] GitHub. 2020. *Kingbar1990/React-TypeScript-Flask-Boilerplate*. [online] Available at: <<https://github.com/kingbar1990/React-TypeScript-Flask-boilerplate>> [Accessed 28 April 2020].