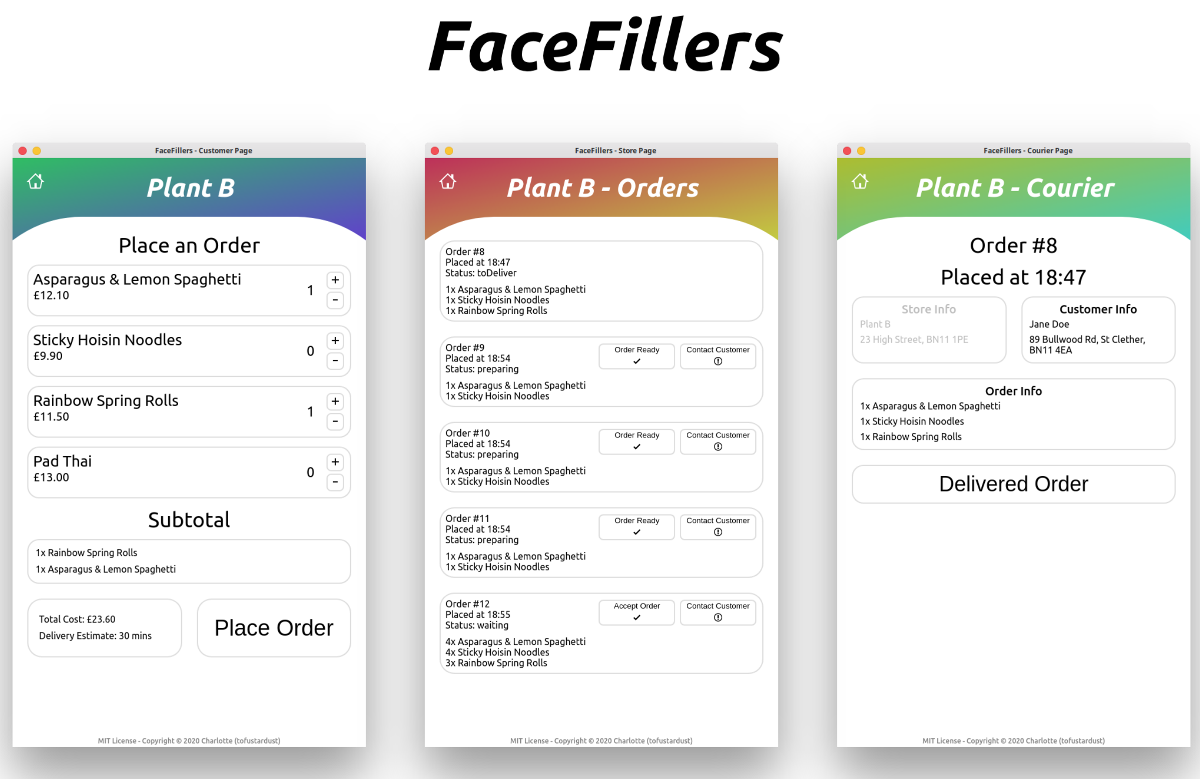
# FaceFillers Documentation


FaceFillers
FaceFillers - Customer Page
Plant B
FaceFillers - Store Page
Plant B - Orders
FaceFillers - Courier Page
Plant B - Courier
MIT License - Copyright © 2020 Charlotte (tofustardust)

Place an Order

Asparagus & Lemon Spaghetti £12.10 1
Sticky Hoisin Noodles £9.90 0
Rainbow Spring Rolls £11.50 1
Pad Thai £13.00 0

Subtotal

1x Rainbow Spring Rolls
1x Asparagus & Lemon Spaghetti

Total Cost: £23.60
Delivery Estimate: 30 mins

Place Order

Order #8
Placed at 18:47
Status: toDeliver
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles
1x Rainbow Spring Rolls

Order #9
Placed at 18:54
Status: preparing
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles
Order Ready ✓ Contact Customer

Order #10
Placed at 18:54
Status: preparing
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles
Order Ready ✓ Contact Customer

Order #11
Placed at 18:54
Status: preparing
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles
Order Ready ✓ Contact Customer

Order #12
Placed at 18:55
Status: waiting
4x Asparagus & Lemon Spaghetti
4x Sticky Hoisin Noodles
3x Rainbow Spring Rolls
Accept Order ✓ Contact Customer

Order #8
Placed at 18:47

Store Info
Plant B
23 High Street, BN11 1PE

Customer Info
Jane Doe
89 Bullwood Rd, St Clether, BN11 4EA

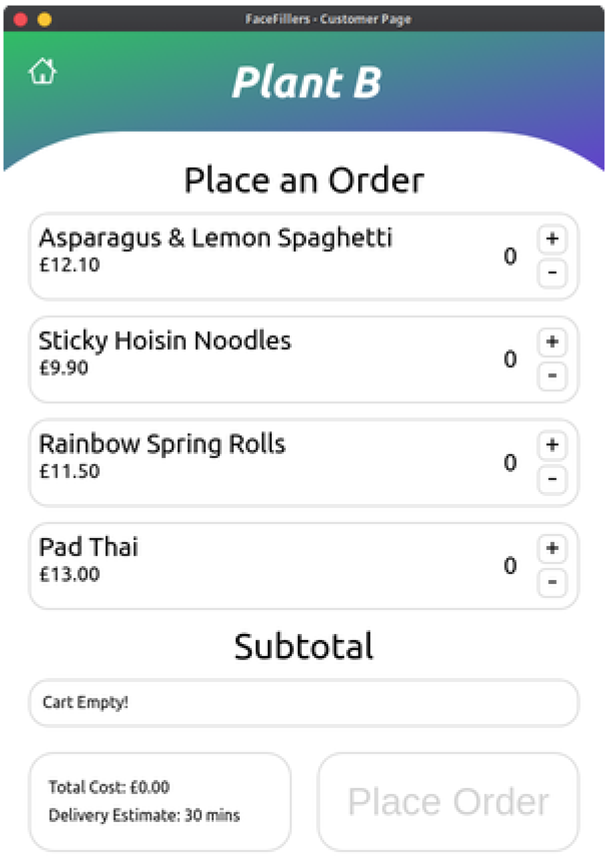
Order Info
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles
1x Rainbow Spring Rolls

Delivered Order

This app mockup is developed using Electron, Node.js, and some npm plugins (most importantly 'mysql' and 'Octicons'). Electron utilises HTML, CSS and JavaScript to function, using Chromium to display webpages as though they were part of a desktop or mobile application (Warcholinski, ca. 2018-2020). This was the perfect choice of framework for this application due to my prior knowledge about web development, and that Node.js allows for mysql to be integrated into the development of the app.

For the database portion of the application I used MariaDB, a fork of MySQL created to act as an open-source, community developed alternative of the original MySQL. This means that it's functionally similar to standard MySQL, allowing for database queries, relational databases, and data validation, making it perfect for this use-case (MariaDB, n.d.).

## Customer Page


FaceFillers - Customer Page

Plant B

Place an Order

Asparagus & Lemon Spaghetti
£12.10
0

Sticky Hoisin Noodles
£9.90
0

Rainbow Spring Rolls
£11.50
0

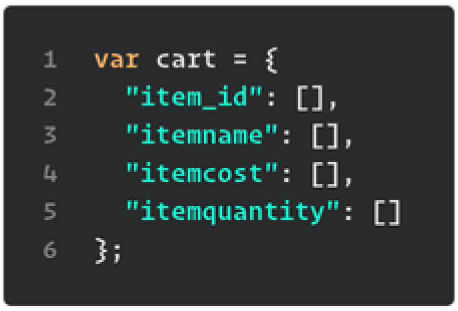
Pad Thai
£13.00
0

Subtotal

Cart Empty!

Total Cost: £0.00
Delivery Estimate: 30 mins

This page is arguably the most complicated section of the application, utilising both SQL queries along with a 'pure-js' cart system for constructing the completed order. This cart system was developed by myself alongside the rest of the application, and works by having a dictionary with 4 separate arrays inside, each containing information fetched from the database or entered by the user.



```
var cart = {
  "item_id": [],
  "itemname": [],
  "itemcost": [],
  "itemquantity": []
};
```

- item_id – an array of item IDs that are in the cart at the moment
- itemname – a human-friendly array of names to go with the IDs
- itemcost – an integer array containing costs of the item
- itemquantity – another integer array representing quantity

These four arrays are used in parallel, so that the name and ID of an item can be identified by using the same array index, using the name of each element to get specific details. This system works well with the query that creates an order in the orders table, along with the set of items it adds to the table order_items.

```
for (const i in cart.item_id) {
  if (cart.itemquantity[i] != 0) {
    document.getElementById("subtotal-item-container").innerHTML +=
      "<div class='subtotal-item'>" + cart.itemquantity[i] + "x " + cart.itemname[i] + "</div>";
  }
}
```

The controller script utilises this cart dictionary to fill information into the order subtotal box, showing item names and quantity should the quantity be greater than 0. This is done using the above section of code, which gets the quantity, adds an ‘x’, gets the name, then adds it to the div shown above.

```
// add order items to order_items tables with reference to order
for (const i in cart.item_id) {
  if (cart.item_id.hasOwnProperty(i)) {
    if (cart.itemquantity[i] != 0) {
      db.query(`INSERT INTO order_items(menu_item_id, order_id, quantity)
        VALUES (` + cart.item_id[i] + `, ` + result.insertId + `, ` + cart.itemquantity[i] + `);`);
    }
  }
}
```

```
function homebuttonModal() {
  document.getElementById("modalContainer").innerHTML = `
    <div class="modalBackground">
      <div class="modalContainer">
        <div class="modalText">Order placed successfully! Click the button to return to the home menu.</div>
        <a href="index.html" class="modalHomeButton">Home</a>
      </div>
    </div>
  `;
}

```

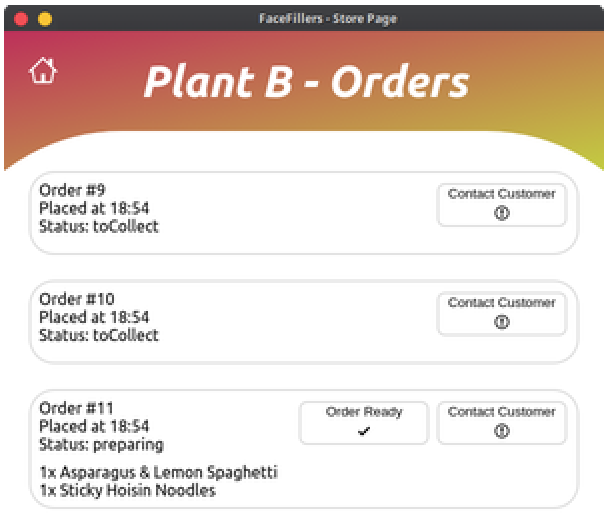
This snippet shows that it creates one order in the orders table, then adds one row to order_items for each item present in the cart, filling the quantity, ID and the order ID that it needs to reference. Following that, it clears the existing cart just to make sure that the order cannot be placed twice. Along with clearing the cart, the placeOrder function also pops up a modal showing a home button and a confirmation along with disabling all the buttons present on the page, stopping multiple orders from being placed in a stronger fashion.

```
function addItem(itemIDToAdd, itemNameToAdd, itemCostToAdd) {
  addinData = [itemIDToAdd, itemNameToAdd, itemCostToAdd];

  let cartOffset = 0; // add to first item if list empty
  for (let i in cart.item_id) {
    if (cart.item_id.hasOwnProperty(i)) {
      let element = cart.item_id[i];
      if (element === itemIDToAdd) {
        cartOffset = i; // found item!
        break;
      } else {
        cartOffset = cart.item_id.length; //didn't find item, append
      }
    }
  }

  // fill cart listing with content from addinData object
  let cartKeys = Object.keys(cart);
  for (const i in cartKeys) {
    const element = cart[cartKeys[i]];
    if (i != 3) { // only do math on the quantity object
      element[cartOffset] = addinData[i];
    } else {
      if (element[cartOffset] != null) {
        if (element[cartOffset] < 10) {
          element[cartOffset]++;
        }
      } else {
        element[cartOffset] = 1;
      }
    }
  }

  // Redraw cart & element
  document.getElementById("innerElementQuantity" + itemIDToAdd).innerHTML =
    cart.itemquantity[cartOffset];
  updateCart();
}
```

This code outlines specifics on how items are added to the cart, which begins as a dictionary with empty arrays. The function gets the information it needs from the onClick function created in the initialisation forEach, then passes it to this function, which utilises that information to create a listing in the cart. This function checks if an item already has a listing in the cart, then either appends or edits the listing at the offset identified. The cart.item_id.length variable is used because the array starts at 0, which means that the length is always the last offset plus one.

The code block below that uses the Object.keys() function to turn the RowDataObject array into a true dictionary for easier iteration and modification. After that, it fills data into the cart, using a simple algorithm to add to the quantity up to the value of 10. If the quantity doesn't yet exist, it sets it to 1, as the HTML that's filled in has the quantity div set to 0 to begin with. This code is reused for removing an item, with items with quantity 0 being ignored by the order creation function.

## Store Page


FaceFillers - Store Page

Plant B - Orders

Order #9
Placed at 18:54
Status: toCollect

Order #10
Placed at 18:54
Status: toCollect

Order #11
Placed at 18:54
Status: preparing

1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles

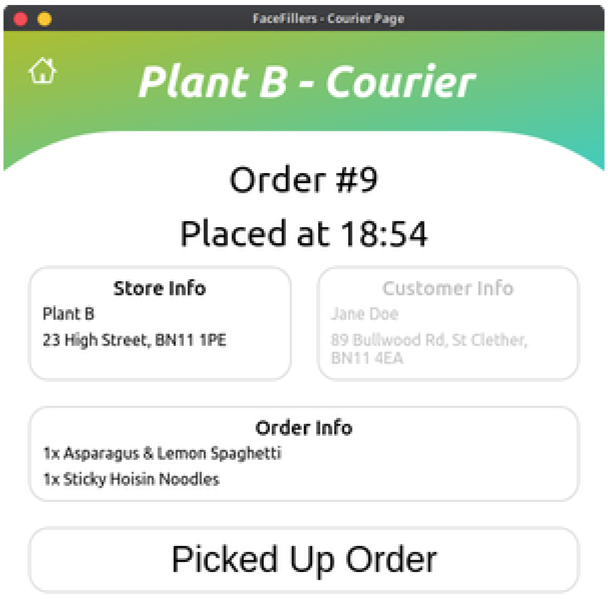
This page is simpler than the customer page, as it only involves displaying active orders, with a system to accept orders, mark orders as ready, and another modal that shows customer information for contact and order cancellation. Not included is a way to edit orders, however the majority of the functionality is here. The contact customer modal works a lot like the order confirmation modal on the customer page, this time showing the order ID, the customer contact number, the items in the order, and two buttons. The first button cancels the order (drop it from the table), and the close button closes the modal by clearing the container it's within.

```
function contactCustomer(orderNumber) {
  document.getElementById("modalContainer").innerHTML = /* HTML */;

  document.getElementById("modalText").innerHTML =
    `<div class="modalTitle">Order #` + orderNumber + `</div>`;

  document.getElementById("modalText").innerHTML += `
      <div class="modalText">Contact Number: ` + customerData.contact_number + `</div>`;

  for (const i in result) {
    if (result.hasOwnProperty(i)) {
      const element = result[i];
      document.getElementById("modalText").innerHTML += `
        <div class="modalText">` + ordersData[i].quantity + "x " + menuItems[i].item_name + `</div>`;
    }
  }
}
```

```
function closeModal() {
  document.getElementById("modalContainer").innerHTML = "";
}

function cancelOrder(orderNumber) {
  closeModal();
  db.query("DELETE FROM orders WHERE order_id =" + orderNumber + ";", function (err, result) {
    if (err) throw err;
    updatePage();
  });
}
```

This code shows a simplified version of what the Contact Customer button does, adding in a chunk of HTML for the modal itself, customer contact data, the order number, and a list of items and quantities. The two buttons utilise these functions to close and delete the row from the table.

## Courier Page



Order #9

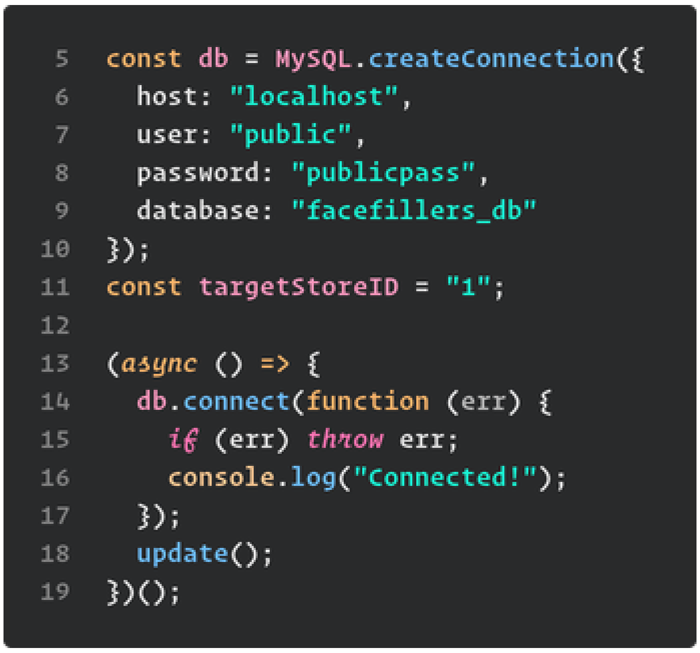
Placed at 18:54

**Store Info**
Plant B
23 High Street, BN11 1PE

Customer Info
Jane Doe
89 Bullwood Rd, St Clether, BN11 4EA

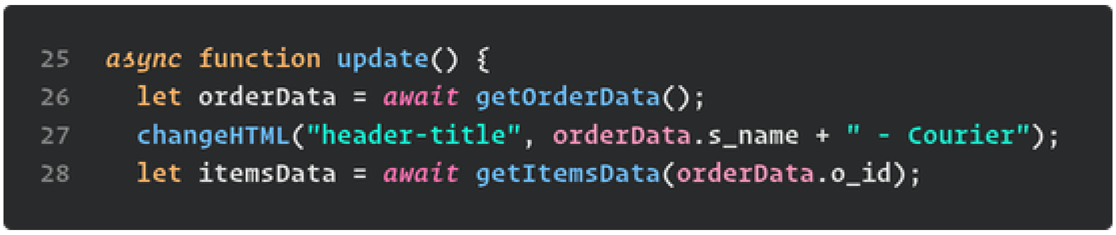
**Order Info**
1x Asparagus & Lemon Spaghetti
1x Sticky Hoisin Noodles

This page has the neatest implementation in JavaScript, using queries to assign data to objects with a large scope, preventing redundant queries. This implementation also involves asynchronous functions, which is the ideal way to get data from a database using the mysql plugin, as getting data back can take time.



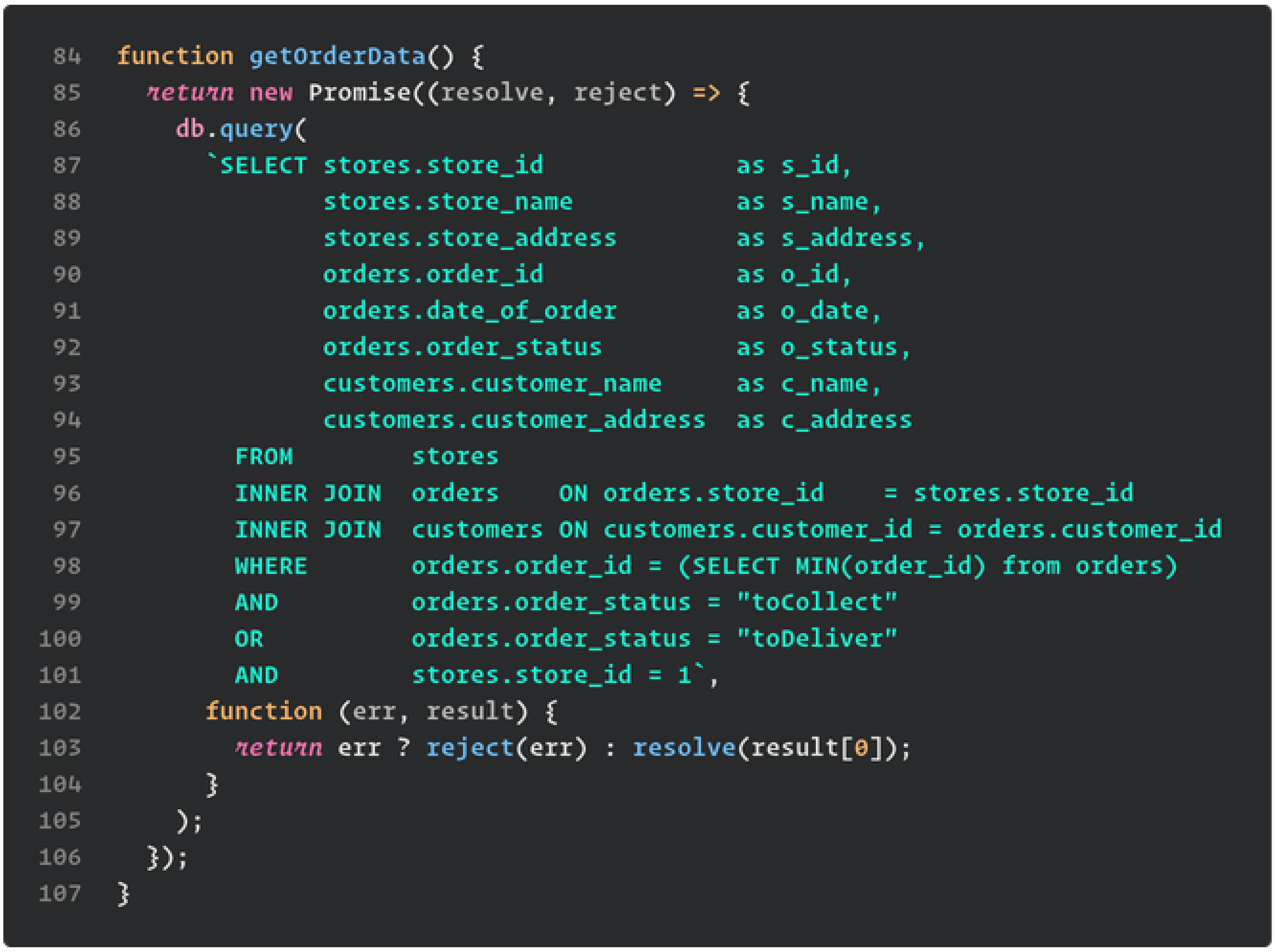
```
const db = MySQL.createConnection({
  host: "localhost",
  user: "public",
  password: "publicpass",
  database: "facefillers_db"
});
const targetStoreID = "1";

(async () => {
  db.connect(function (err) {
    if (err) throw err;
    console.log("Connected!");
  });
  update();
})();
```

The code first has a section to create a db connection using the host, user, password, and database information required, then sets a constant of the target store ID (referring to the only store in the table). After that, a one-time asynchronous function is performed, connecting to the database and calling update(), another asynchronous function with the majority of the code inside.



```
async function update() {
  let orderData = await getOrderData();
  changeHTML("header-title", orderData.s_name + " - Courier");
  let itemsData = await getItemsData(orderData.o_id);
```

This function is also asynchronous, which allows for the await/promise system, which facilitates database queries in a much better way than nesting multiple queries.



```
function getOrderData() {
  return new Promise((resolve, reject) => {
    db.query(
      `SELECT stores.store_id             as s_id,
              stores.store_name           as s_name,
              stores.store_address        as s_address,
              orders.order_id             as o_id,
              orders.date_of_order        as o_date,
              orders.order_status         as o_status,
              customers.customer_name     as c_name,
              customers.customer_address  as c_address
        FROM        stores
        INNER JOIN  orders    ON orders.store_id    = stores.store_id
        INNER JOIN  customers ON customers.customer_id = orders.customer_id
        WHERE       orders.order_id = (SELECT MIN(order_id) from orders)
        AND         orders.order_status = "toCollect"
        OR          orders.order_status = "toDeliver"
        AND         stores.store_id = 1`,
      function (err, result) {
        return err ? reject(err) : resolve(result[0]);
      }
    );
  });
}
```

The getOrderData function uses a query that uses `INNER JOIN` to gather data from other tables as a pair of cross-table queries. This query allows for the orderData object to have only the children that it needs, by querying and assigning variables only when they're required for use in the rest of the script. This function also utilises `MIN()` and confirms that the order status is `toCollect` or `toDeliver` to get only the first order that is relevant to the courier.

The JOIN clause definitely simplifies this process, as a single query can serve the purpose of three separate queries, as well as allowing the database server itself to handle the selection of the right row with the right ID, rather than the javascript itself. This process is reused for the `itemsData` object additionally, returning an array rather than a single `rowDataObject`:

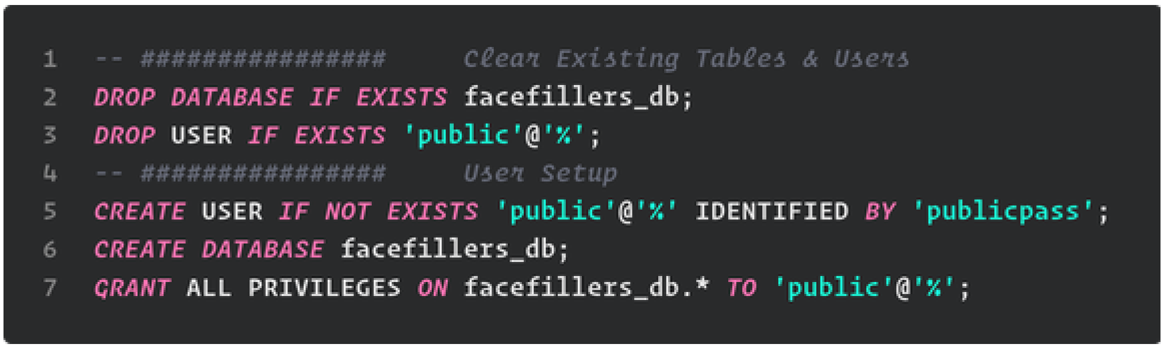
```
function getItemsData(id) {
  // more complex join query to get all the relevant data
  return new Promise((resolve, reject) => {
    db.query(
      `SELECT order_items.order_item_id as oi_id,
              order_items.quantity      as oi_quantity,
              menu_items.item_name      as oi_name
        FROM order_items
        INNER JOIN menu_items ON menu_items.menu_item_id = order_items.menu_item_id
        WHERE order_items.order_id = ` + id,
      function (err, result) {
        return err ? reject(err) : resolve(result);
      }
    );
  });
}
```

This function also uses a `JOIN` clause to gather data from the menu table, based on the items in the order. This allows for the two sets of information to be accessed as a single set, with the name and the quantity next to each other. This would otherwise require two queries, with two arrays, two `rowDataObject` objects, or two dictionaries in the script itself. This query also has a `WHERE` clause that makes sure that the order items selected are guaranteed to be for the relevant store (even if this is the only store that exists).

```
function resolveItemInfo(orderItems, menuItems, i) {
  return orderItems[i].quantity + "x " + menuItems[i].item_name;
}

function dateToTime(datetime) {
  return datetime.getHours() + ":" + datetime.getMinutes();
}

function changeHTML(id, s) {
  document.getElementById(id).innerHTML = s;
}

function addHTML(id, s) {
  document.getElementById(id).innerHTML += s;
}

function reload() {
  window.location.reload();
}

function pickedUp(id) {
  db.query("UPDATE orders SET order_status = 'toDeliver' WHERE order_id = " + id + ";");
  update();
}

function delivered(id) {
  db.query("DELETE FROM orders WHERE order_id =" + id + ";");
  update();
}
```

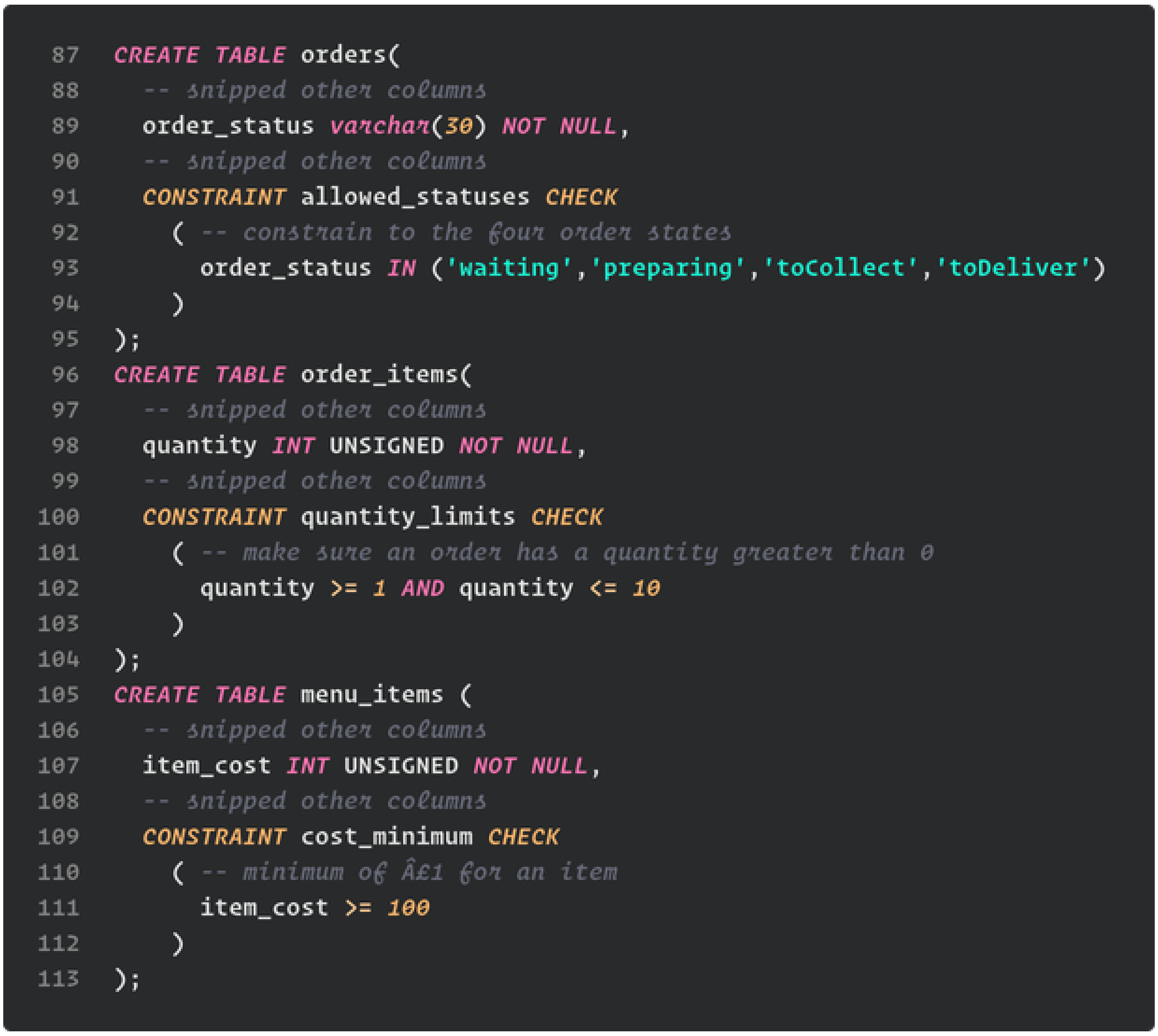
Finally is a set of non-asynchronous functions that serve simple purposes like adding HTML to the document, replacing the complex line “document.getElementByID(id).html +=” with a simpler, neater function called addHTML(id,s), as an example. Also included here are the functions that the big button performs, which either sets the order_status of the order in the table to “toDeliver” from “toCollect”, or deletes it from the table entirely. This could be replaced with an archived status, though for the purposes of this mockup that’s not necessary.

## SQL Configuration



```sql
-- ################     Clear Existing Tables & Users
DROP DATABASE IF EXISTS facefillers_db;
DROP USER IF EXISTS 'public'@'%';
-- ################     User Setup
CREATE USER IF NOT EXISTS 'public'@'%' IDENTIFIED BY 'publicpass';
CREATE DATABASE facefillers_db;
GRANT ALL PRIVILEGES ON facefillers_db.* TO 'public'@'%';
```

The MySQL tables and databases were set up to be used alongside Javascript by specifying user information with a known password, with privileges allowing that user to access the database itself. This screenshot shows an sql script which can be executed to drop the table, delete the user, then recreate all the information required.



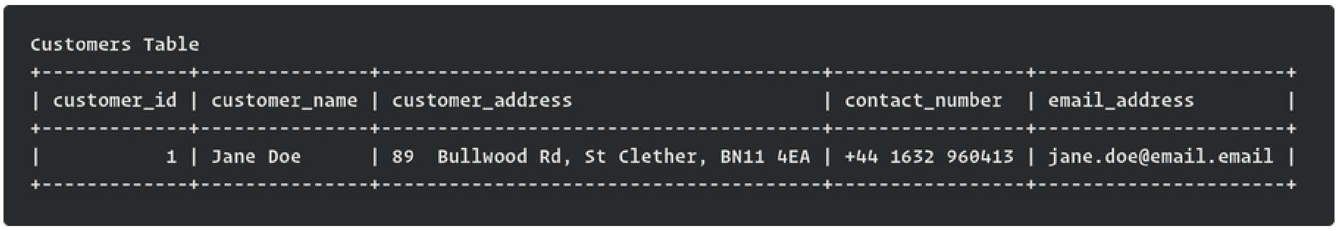
```sql
CREATE TABLE orders(
  -- snipped other columns
  order_status varchar(30) NOT NULL,
  -- snipped other columns
  CONSTRAINT allowed_statuses CHECK
    ( -- constrain to the four order states
      order_status IN ('waiting','preparing','toCollect','toDeliver')
    )
);
CREATE TABLE order_items(
  -- snipped other columns
  quantity INT UNSIGNED NOT NULL,
  -- snipped other columns
  CONSTRAINT quantity_limits CHECK
    ( -- make sure an order has a quantity greater than 0
      quantity >= 1 AND quantity <= 10
    )
);
CREATE TABLE menu_items (
  -- snipped other columns
  item_cost INT UNSIGNED NOT NULL,
  -- snipped other columns
  CONSTRAINT cost_minimum CHECK
    ( -- minimum of Â£1 for an item
      item_cost >= 100
    )
);
```

The tables themselves have many constraints, both specific constraints using `CHECK`, and general constraints like `NOT NULL` and `AUTO_INCREMENT`. The three `CHECK` constraints shown above each serve a different purpose:

- The first `CHECK` constrains the order status to four distinct strings, to stop invalid statuses from being set.
- The second `CHECK` makes sure that the quantity of an item in an order is in the valid range (nonzero and less than 11).
- The third `CHECK` sets a minimum cost for menu items at £1, designed to prevent entering the cost wrong, setting an item too cheap (as an arbitrary value), or setting an item without a cost.

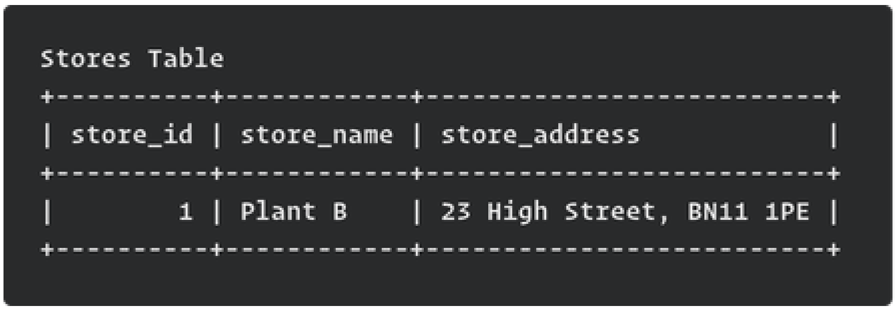
## SQL Reports

These reports are taken from MariaDB's console directly, performed on an in-use database made for testing. Some redundant data in the Order Items table has been removed.



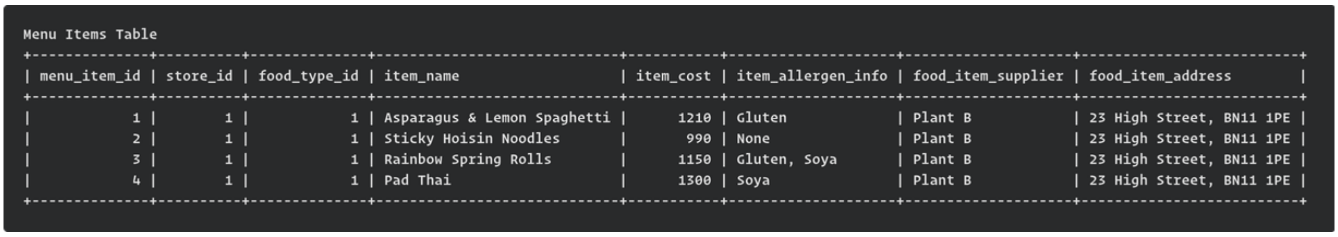
```
Customers Table
+-------------+---------------+------------------------------------+-----------------+----------------------+
| customer_id | customer_name | customer_address                   | contact_number  | email_address        |
+-------------+---------------+------------------------------------+-----------------+----------------------+
|           1 | Jane Doe      | 89  Bullwood Rd, St Clether, BN11 4EA | +44 1632 960413 | jane.doe@email.email |
+-------------+---------------+------------------------------------+-----------------+----------------------+
```

The customers table shows information about the customers, the key parts being the name, address, and contact number, which are used to contact the customer directly. The ID is also used for order data for a direct link. The rest of the data isn't used in the scope of this project.



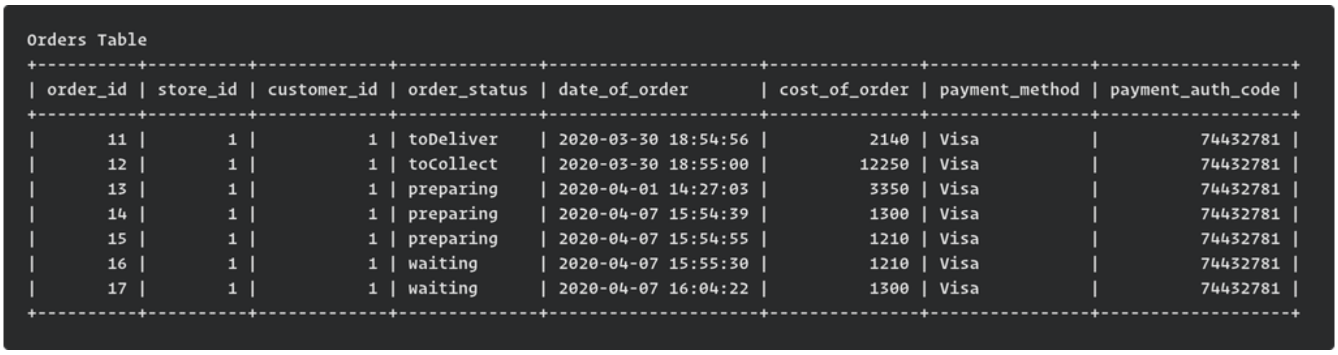
```
Stores Table
+----------+------------+-------------------------+
| store_id | store_name | store_address           |
+----------+------------+-------------------------+
|        1 | Plant B    | 23 High Street, BN11 1PE |
+----------+------------+-------------------------+
```

The stores table is very similar, containing key information about the name, address and most importantly, ID. The name is used on the title of each page in the applet, and the address is used for couriers to collect orders. The ID is used to connect this table with the Menu Items table and the Orders table.



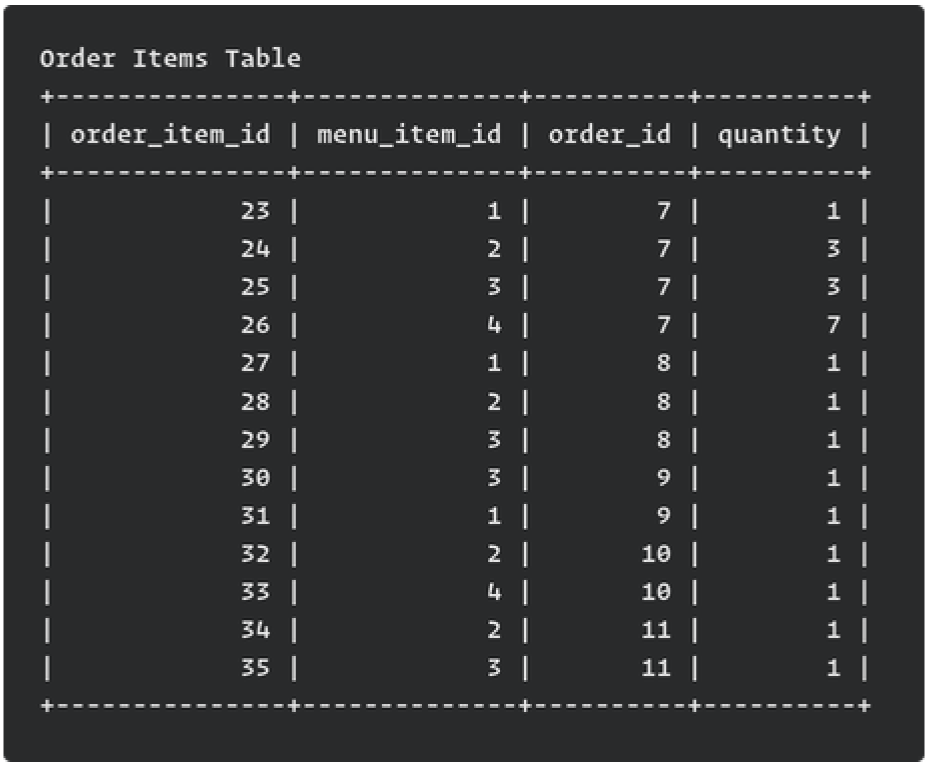
```
Menu Items Table
+--------------+----------+--------------+----------------------------+-----------+---------------------+--------------------+--------------------------+
| menu_item_id | store_id | food_type_id | item_name                  | item_cost | item_allergen_info  | food_item_supplier | food_item_address        |
+--------------+----------+--------------+----------------------------+-----------+---------------------+--------------------+--------------------------+
|            1 |        1 |            1 | Asparagus & Lemon Spaghetti |      1210 | Gluten              | Plant B            | 23 High Street, BN11 1PE |
|            2 |        1 |            1 | Sticky Hoisin Noodles      |       990 | None                | Plant B            | 23 High Street, BN11 1PE |
|            3 |        1 |            1 | Rainbow Spring Rolls       |      1150 | Gluten, Soya        | Plant B            | 23 High Street, BN11 1PE |
|            4 |        1 |            1 | Pad Thai                   |      1300 | Soya                | Plant B            | 23 High Street, BN11 1PE |
+--------------+----------+--------------+----------------------------+-----------+---------------------+--------------------+--------------------------+
```

This table contains all of the information about the menu for each store, with a foreign keys for which store each row refers to, as well as a foreign key for what food type they are. This table is important for ordering as it contains cost, allergen info, food type, and the item name, allowing customers to make a purchasing decision based on that.



```
Orders Table
+----------+----------+-------------+--------------+---------------------+---------------+----------------+-------------------+
| order_id | store_id | customer_id | order_status | date_of_order       | cost_of_order | payment_method | payment_auth_code |
+----------+----------+-------------+--------------+---------------------+---------------+----------------+-------------------+
|       11 |        1 |           1 | toDeliver    | 2020-03-30 18:54:56 |          2140 | Visa           |          74432781 |
|       12 |        1 |           1 | toCollect    | 2020-03-30 18:55:00 |         12250 | Visa           |          74432781 |
|       13 |        1 |           1 | preparing    | 2020-04-01 14:27:03 |          3350 | Visa           |          74432781 |
|       14 |        1 |           1 | preparing    | 2020-04-07 15:54:39 |          1300 | Visa           |          74432781 |
|       15 |        1 |           1 | preparing    | 2020-04-07 15:54:55 |          1210 | Visa           |          74432781 |
|       16 |        1 |           1 | waiting      | 2020-04-07 15:55:30 |          1210 | Visa           |          74432781 |
|       17 |        1 |           1 | waiting      | 2020-04-07 16:04:22 |          1300 | Visa           |          74432781 |
+----------+----------+-------------+--------------+---------------------+---------------+----------------+-------------------+
```

The orders table is critical to preparation and delivery, containing the status of the order along with lots of other info, including which customer it's for, the total cost of the order, which store the order is assigned to, along with the date that the order was placed.


```
Order Items Table
+---------------+--------------+----------+----------+
| order_item_id | menu_item_id | order_id | quantity |
+---------------+--------------+----------+----------+
|            23 |            1 |        7 |        1 |
|            24 |            2 |        7 |        3 |
|            25 |            3 |        7 |        3 |
|            26 |            4 |        7 |        7 |
|            27 |            1 |        8 |        1 |
|            28 |            2 |        8 |        1 |
|            29 |            3 |        8 |        1 |
|            30 |            3 |        9 |        1 |
|            31 |            1 |        9 |        1 |
|            32 |            2 |       10 |        1 |
|            33 |            4 |       10 |        1 |
|            34 |            2 |       11 |        1 |
|            35 |            3 |       11 |        1 |
+---------------+--------------+----------+----------+
```

This table is a set of numbers, two foreign keys that refer to which menu item and which order the order item is supposed to be. The important data here is quantity, which impacts how many items are ordered in a specific order.

## Reference List

Warcholinski, M. (ca. 2018-2020) *What is Electron JS?*. Available at: https://brainhub.eu/blog/what-is-electron-js/ (Accessed: 4 April 2020).

MariaDB (n.d.) *About MariaDB Server*. Available at: https://mariadb.org/about/ (Accessed: 4 April 2020).