FaceFillers-DB Technical Documentation

Table of Contents

User Documentation

FaceFillers-Electron is an Electron/NodeJS based mockup for the FaceFillers app package. This software utilises MariaDB, a fork of MySQL, to store information utilised by the User, Stores and Couriers involved. This documentation aims to explain and justify many of the features and design decisions involved in this software package.

Home Page

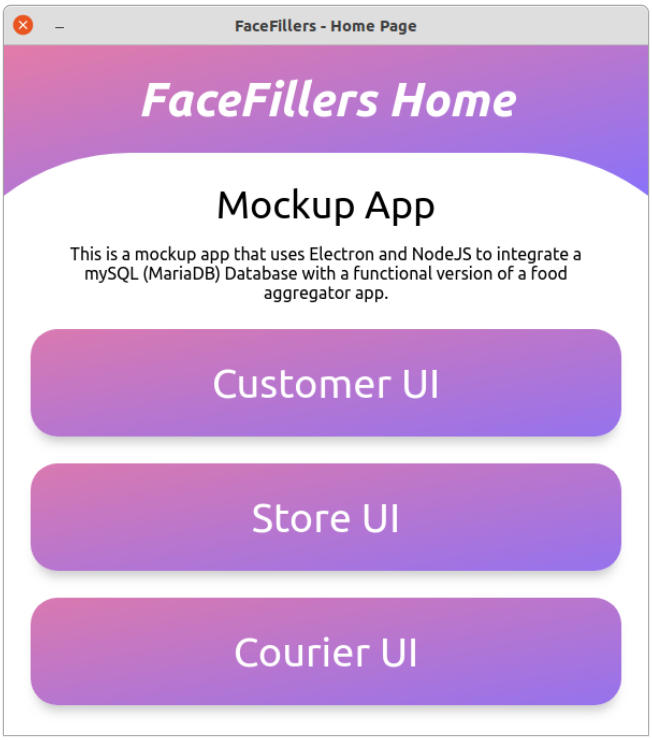

Figure 1. Home Page

This page, featured in Figure 1., contains a small description of the app, along with three buttons that lead to the other pages. This section is generic and is not aimed at Customers, Stores or Couriers, and wouldn't be included in a production application.

- "Customer UI" leads to the Customer Page.
- "Store UI" leads to the Store Page.
- "Courier UI" leads to the Courier Page.

This linking works by linking to separate HTML files, as Electron is effectively a chromium application running in a window.

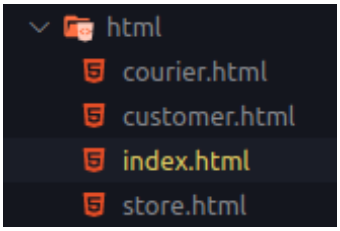

Figure 2. The folder layout for the available files

Customer Page

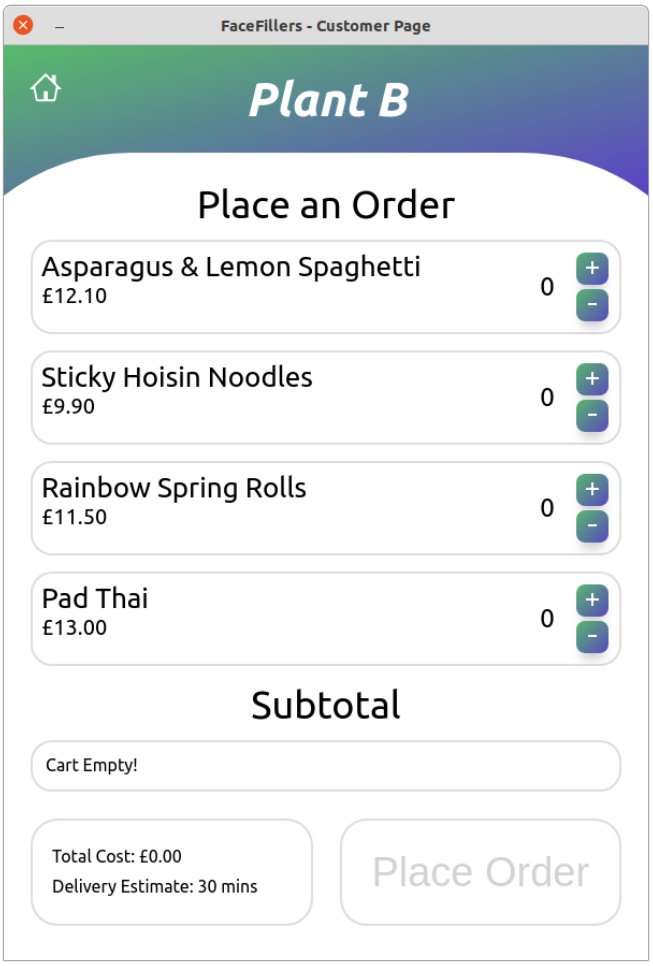

Figure 3. Customer Ordering Page

The customer ordering page is designed to allow an end-user to place an order with the store, choosing the items in their order, along with the quantity. This page features a cart system that collects the data required for the order to be posted, as well as a system to display the items available from the store, the price, the store name, and a system to total the cost in the cart. At the bottom of Figure 3., a delivery estimate is shown. This feature is planned but not feasible to implement for this mockup.

This system works like a traditional ordering system:

1. The user constructs their cart with the items they would like
2. The user is happy with the cart and total price, and places the order
3. The user is presented with a confirmation dialogue, and the order is placed

Not pictured is the user confirmation modal, which pops up once the order is placed. This modal prevents users from accidentally placing additional orders.

Store Page

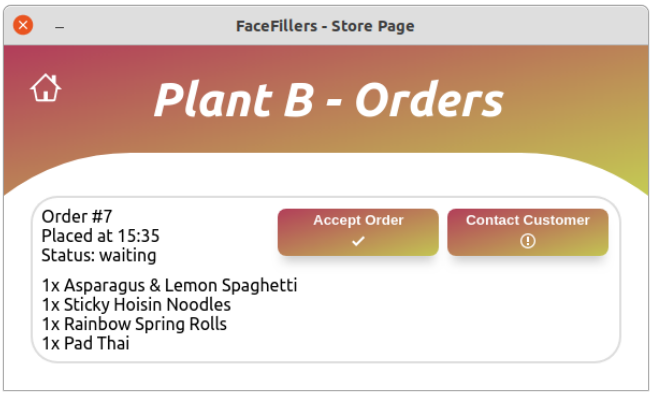

Figure 4. Store page showing an available order

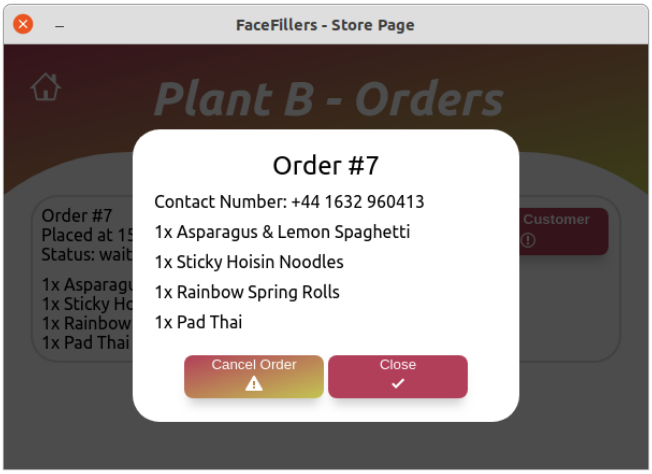

Figure 5. Customer Contact Modal

The store page features order discovery, showing available orders, accepted orders, and orders awaiting collection or delivery. This section has a few features:

1. Order fulfilment requirements in the card
2. A button to accept an order waiting for confirmation
3. A button for accepted orders to be completed, awaiting collection
4. A popup modal for contacting the customer
    a. Customer contact information inside the modal (Figure 5.)
    b. 'Cancel Order' button inside the modal (Figure 5.)

This section is designed so that stores can quickly get an overview of the status of all the relevant orders, including `waiting` orders, those that are waiting to be accepted, `preparing` orders, which are accepted and in preparation, and `toCollect` orders, which are waiting for a courier to collect them. These three statuses are those that are relevant to the operation of the store.

Courier Page

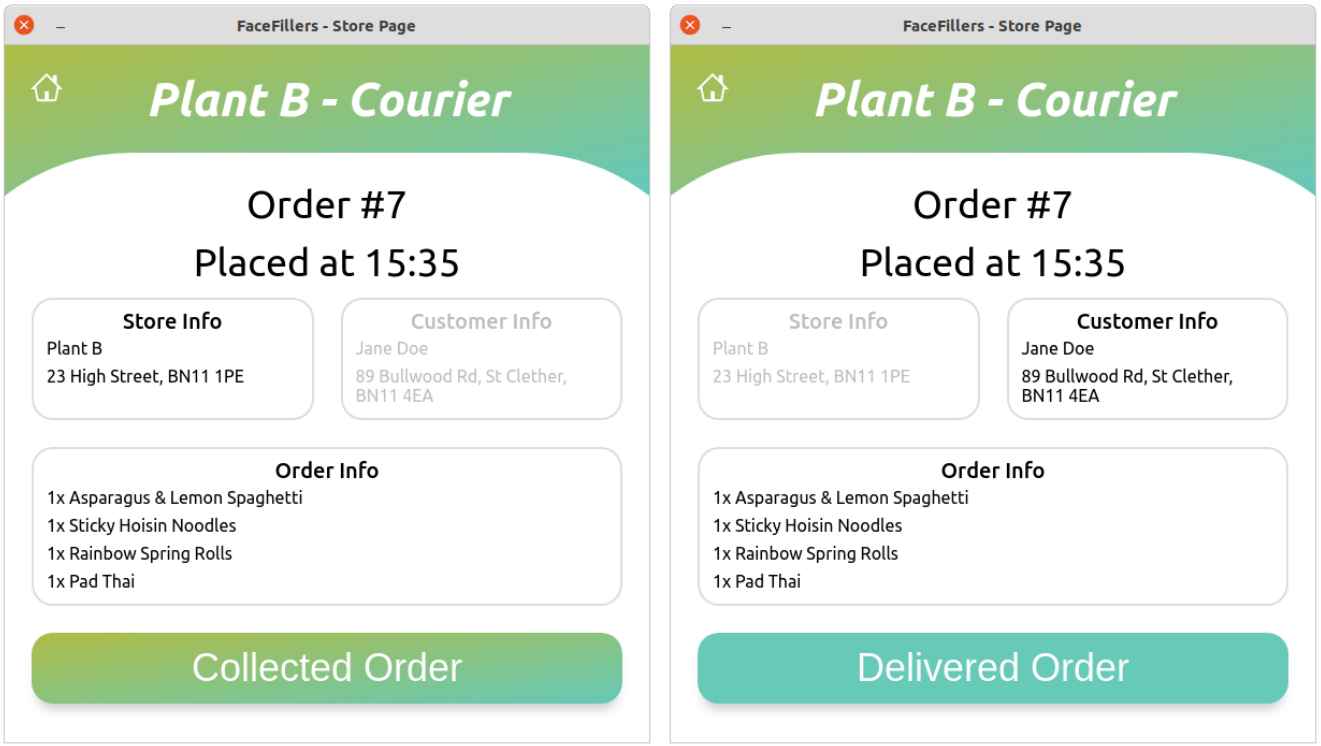

Figure 6. Courier Page, in use. Left: Assigned order. Right: Collected order

Figure 6. shows two states for the courier page. The left screenshot shows the assigned order, taken from the first order of the orders database that has the `toCollect` status, as assigned from the store page. The right screenshot shows a collected order, updating the status of the order to `toDeliver`, reached by pressing the 'Collected Order' button on the left side.

The difference between the two pages relates to the info displayed. The `toCollect` status side greys out the Customer Info section, as the more significant info is the Store address, along with the contents of the order itself. The `toDeliver` status side greys out the store info using the same philosophy. The two greyed sections are still legible, but are less intrusive.

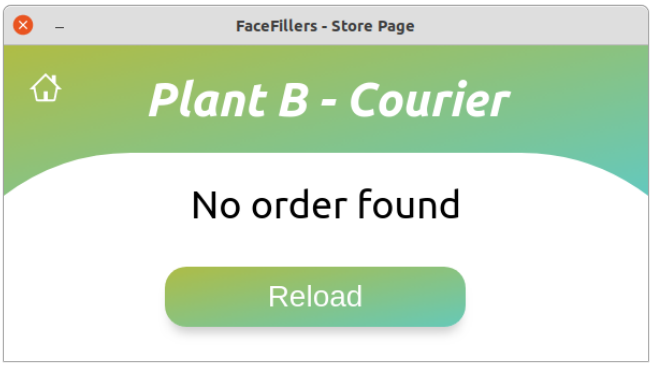

Figure 7. "No order found" screen

If no order can be found with the `toCollect` status, the app displays a refresh button along with a readout. This refresh button reloads the page (thanks to Electron), re-checking if an order is available. A real solution would use a callback to update this page whenever an order is available, but this works for a mockup.

Database Documentation

Database Design

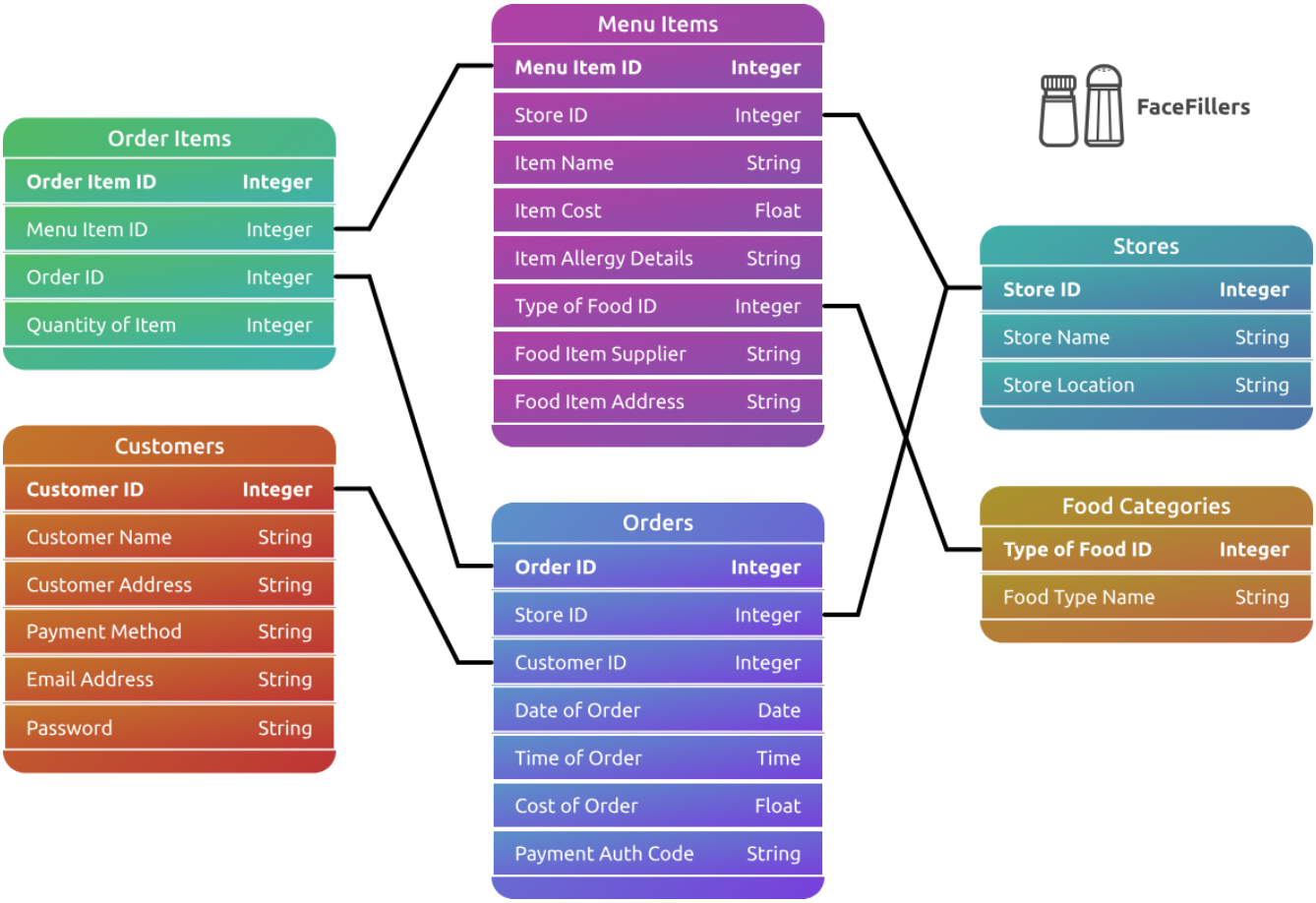

Figure 8. Database Schema

This schema outlines the overall structure of the MariaDB database. This database is normalised to third normal form, with various constraints applied to each table.

- customers Table
- food_types Table
- menu_items Table
- order_items Table
- orders Table
- stores Table

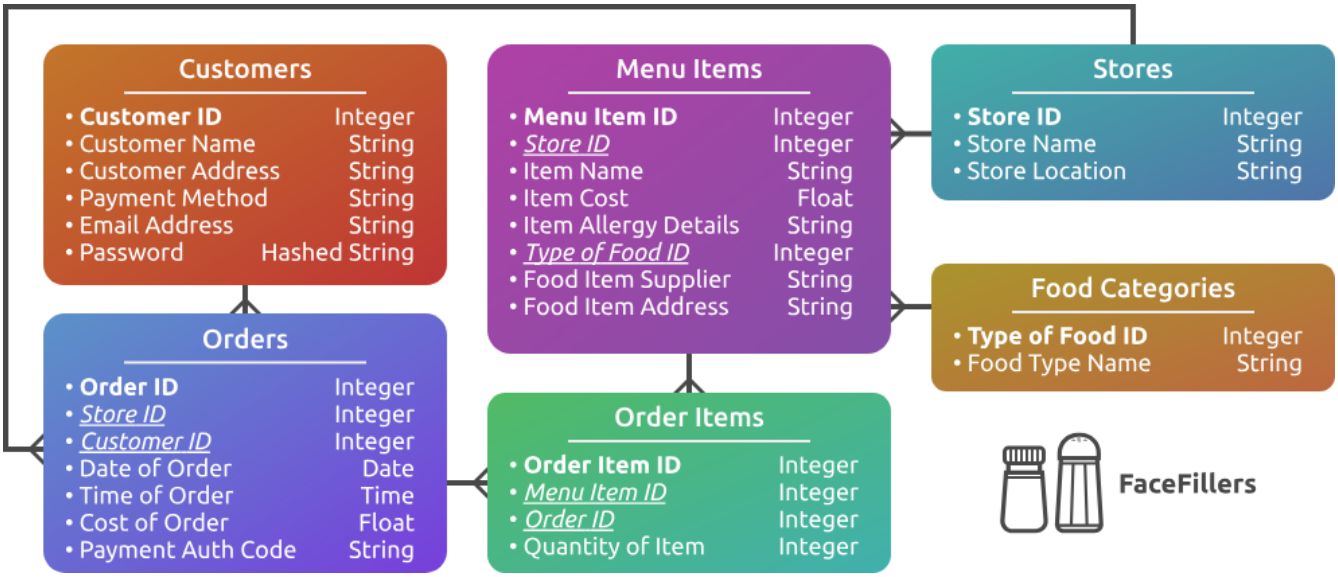

Figure 9. Entity Relationship Diagram

Further illustrating the structure of the database, this ER diagram shows the relationship between each table. Described here is:

- Customers can have many orders, but each order has a single customer reference.
- Orders have many order items, but each order item is assigned to one order.
- Each menu item has many order items, but each order item refers to a single menu item.
- Each store has many menu items, but each menu item refers to a single store.
- Stores have many orders, but an order refers to a single store.
- There are many menu items with the same food category, but each menu item refers to one food category.

These two diagrams illustrate the overall design of the database, outlined further in the next headings. Many of the tables contain foreign keys, facilitating customer information to relate to an order, for example. This allows for the primary functionality of the application without relying on orders storing customer data itself, leading to data redundancy.

customers Table

```
CREATE TABLE customers (
  customer_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_name varchar(50) NOT NULL,
  customer_address varchar(100) NOT NULL,
  contact_number varchar(30) NOT NULL,
  email_address varchar(64) NOT NULL,
  customer_password varchar(256) NOT NULL,
  PRIMARY KEY (customer_id)
);
```

- `customer_id` - This is the primary key for the table, relating to the unique ID that each customer has. Set to `AUTO_INCREMENT` so that new customers get a unique ID. A production system should replace this with an algorithm, as sequential IDs can be a security risk.
- `customer_name` - This relates to the full given name of the customer. Courier uses this for delivery, and the store uses this for contact.
- `customer_address` - The delivery address for the customer. useful for the Courier.
- `contact_number` - Contact number for the customer. Used by the store and courier for contact.
- `email_address` - Proof of concept email address contact info.
- `customer_password` - Placeholder SHA-256 hash of a password. Currently not implemented, merely a proof of concept.

food_types Table

```
CREATE TABLE food_types(
  food_type_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  food_type_name varchar (50) NOT NULL,
  PRIMARY KEY (food_type_id)
);
```

- `food_type_id` - Primary key for the table, set to `AUTO_INCREMENT` so that each food type gets a unique ID.
- `food_type_name` - String of the name of the food type.

This table is primarily used for the categorisation of menu items and stores, based on the food types they offer. This isn't implemented in this mockup, but the database functionality is available.

menu_items Table

```
CREATE TABLE menu_items (
  menu_item_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_id INT UNSIGNED NOT NULL,
  food_type_id INT UNSIGNED NOT NULL,
  item_name varchar(50) NOT NULL,
  item_cost INT UNSIGNED NOT NULL,
  item_allergen_info varchar (100) NOT NULL,
  food_item_supplier varchar (50) NOT NULL,
  food_item_address varchar (100) NOT NULL,
  PRIMARY KEY (menu_item_id),
  FOREIGN KEY (store_id) REFERENCES stores(store_id),
  FOREIGN KEY (food_type_id) REFERENCES food_types(food_type_id),
  CONSTRAINT cost_minimum CHECK
    ( -- minimum of £1 for an item
      item_cost >= 100
    )
);
```

- `menu_item_id` - The primary key of this table. Also auto increments for ease of usage.
- `store_id` - This is a foreign key that relates the menu item to the store. See below.
- `food_type_id` - This is a foreign key that assigns a food type to the menu item. See below.
- `item_name` - A string for the name of the item.
- `item_cost` - An integer containing the cost of the item, multiplied by 100 to prevent floating point storage.
  - `£10` would be stored as `1000`
  - `£13.12` would be stored as `1312`
- `item_allergen_info` - A long string allowing for the menu listing to display allergen info. This can be left blank by entering `None`.
- `food_item_supplier` - Placeholder info for information about the supplier of the listing.
- `food_item_address` - Placeholder info for information about the supplier of the listing.

Additional info:

- The foreign key `store_id` relates the menu to an individual store. This allows for a store to have lots of menu items assigned to it for content delivery. References the stores Table.
- The foreign key `food_type_id` allows for an individual menu item to have a food type. This should possibly allow for multiple IDs to be referenced, but in this implementation each menu item has a single food type (Indian, Italian, etc.). References the food_types Table.

order_items Table

```sql
CREATE TABLE order_items(
  order_item_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  menu_item_id INT UNSIGNED NOT NULL,
  order_id INT UNSIGNED NOT NULL,
  quantity INT UNSIGNED NOT NULL,
  PRIMARY KEY (order_item_id),
  FOREIGN KEY (menu_item_id) REFERENCES menu_items(menu_item_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  CONSTRAINT quantity_limits CHECK
    ( -- make sure an order has a quantity greater than 0
      quantity >= 1 AND quantity <= 10
    )
);
```

This table relates to the child items contained within each order. This is essential as an order has many child items, so having this table prevents an order from having a variable size.

- `order_item_id` -
- `menu_item_id` - Foreign key relating this child to an item available on the menu.
- `order_id` - Foreign key relating this child to the order itself.
- `quantity` - A simple integer that stores the quantity in the order.

Additional info:

- The foreign key `menu_item_id` relates this item to an item available on the menu. References the menu_items Table.
- The foreign key `order_id` relates this order child to the order itself. References the orders Table.
- The constraint here makes sure that the quantity is within 1 and 10. This prevents excessively large orders and orders of 0 quantity.

orders Table

```
CREATE TABLE orders(
  order_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_id INT UNSIGNED NOT NULL,
  customer_id INT UNSIGNED NOT NULL,
  order_status varchar(30) NOT NULL,
  date_of_order DATETIME NOT NULL,
  cost_of_order INT UNSIGNED NOT NULL,
  payment_method varchar(30) NOT NULL,
  payment_auth_code INT UNSIGNED NOT NULL,
  PRIMARY KEY (order_id),
  FOREIGN KEY (store_id) REFERENCES stores(store_id),
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
  CONSTRAINT allowed_statuses CHECK
    ( -- constrain to the four order states
      order_status IN ('waiting','preparing','toCollect','toDeliver')
    )
);
```

- `order_id` - Primary key for this table, auto incrementing for ease.
- `store_id` - Foreign key relating an order to the store. See below.
- `customer_id` - Foreign key relating the order to a customer. See below.
- `order_status` - This string stores the status of the order, constrained into four categories. This status allows for the order to be tracked. (not included in the schema)
- `date_of_order` - This is a datetime that stores when the order was placed for timing and display to the store and courier.
- `cost_of_order` - Displayed to the store, showing the amount of money that was transferred in the order.
- `payment_method` - Not implemented, placeholder for payment info.
- `payment_auth_code` - Not implemented, placeholder for payment info.

Additional info:

- The foreign key `store_id`
- The foreign key `customer_id`
- The constraints at the end make sure that the order status fits into one of four categories. These categories are essential to the operation of the application, with two referring to the store page, and two referring to the courier page.

stores Table

```
CREATE TABLE stores (
  store_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_name varchar (50) NOT NULL,
  store_address varchar (100) NOT NULL,
  PRIMARY KEY (store_id)
);
```

- `store_id` - Primary key for the store, auto incrementing to be unique.
- `store_name` - Displayed in the app when the store is selected, as well as on the store page and courier page.
- `store_address` - A string used to show the address of the store to the courier. Essential for delivery.

Example Configuration & Reports

Provided in the software repo is a script that constructs the database and fills in some example info. This info provides key functionality to the mockup app, including a store, 4 menu items, a food type, and a placeholder customer. These reports below are generated in the `mariadb` CLI application.

```
MariaDB [facefillers_db]> select * from customers;
+-------------+---------------+---------------------------------------+-----------------+----------------------+--------------
| customer_id | customer_name | customer_address                      | contact_number  | email_address        | customer_passw
+-------------+---------------+---------------------------------------+-----------------+----------------------+--------------
|           1 | Jane Doe      | 89  Bullwood Rd, St Clether, BN11 4EA | +44 1632 960413 | jane.doe@email.email | 4F9A3F1F35376F
+-------------+---------------+---------------------------------------+-----------------+----------------------+--------------
1 row in set (0.001 sec)
```

```
MariaDB [facefillers_db]> select * from food_types;
+--------------+----------------+
| food_type_id | food_type_name |
+--------------+----------------+
|            1 | Vegan          |
+--------------+----------------+
1 row in set (0.000 sec)
```

```
MariaDB [facefillers_db]> select * from menu_items;
+--------------+----------+--------------+---------------------------+-----------+-------------------+-------------------+-------------------------+
| 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 |
+--------------+----------+--------------+---------------------------+-----------+-------------------+-------------------+-------------------------+
4 rows in set (0.001 sec)
```

```
MariaDB [facefillers_db]> select * from stores;
+----------+------------+--------------------------+
| store_id | store_name | store_address            |
+----------+------------+--------------------------+
|        1 | Plant B    | 23 High Street, BN11 1PE |
+----------+------------+--------------------------+
1 row in set (0.001 sec)
```

Production Tools

GitHub

GitHub is essential to modern software projects, in my opinion, as it provides cloud hosting, backups, version control, issues (ticketing), static site hosting, documentation and more. For this reason, I used GitHub alongside this project. The repo is publically available, licensed under the MIT license, a FOSS license that allows for permissive use and reuse and modification.

To use git alongside this project, type this into bash (or an equivalent shell):

```
$ sudo apt install git
$ git clone https://github.com/tofustardust/FaceFillers-Electron
```

This will install git (if it's not already installed) and clone the repository into a folder in the current directory. This repository contains all the relevant information for getting this working, requiring a few other things to work fully.

NodeJS & Plugins

NPM, or Node Package Manager, is a powerful package manager for NodeJS, an extensive library for JavaScript that facilitates more complicated applications and simpler development. NPM in

particular allows the usage of packages, small or large software packages that automatically install dependencies and can work together.

```
$ npm install nodejs
$ npm install mysql --save-dev
$ npm install octicons
```

- `nodejs` is used alongside mysql and Electron to load the packages in the first place.
- `mysql` provides the frontend database functionality, linking up with MariaDB, a serverside piece of software.
- `octicons` is a free set of icons provided by GitHub, utilised in this project for the home button on each subpage. This parralels font-awesome and other similar icon packages, with a small footprint and easy integration.

Electron

Electron is a frontend system that utilises Chromium, Google's open source browser framework, to display desktop apps. It runs on top of NodeJS, requiring only a single line to install:

```
$ npm install electron --save-dev
```

I used electron for this project as the `mysql` node package and Electron/Node integration made accessing the database from a frontend incredibly simple. As well as this, I'm somewhat competent in web design, allowing for the frontend to look good and work well. This package also makes testing and redistribution very easy, as chromium has the built in developer tools that allows for inspection, console entry and source viewing, and the redistribution simply requires a package file for installation:

```
$ npm install && npm start
```

A single line initialises the repository and starts the application itself.

MariaDB

MariaDB is a fork of MySQL that is community developed and commercially supported. It's fully compatible with standard MySQL queries, allowing for it to work in place of a (now unsupported) MySQL server. I used this package due to the FOSS nature, being licensed under GNU's GPLv2, a permissive copyleft license that guarantees the software will remain free and open source. As well as this, it's easily accessible in apt, the standard package manager on Debian Linux distributions:

```
$ sudo apt install mariadb-server
```

After installation, usage is very simple. I provided an SQL script in the repository that creates the user, database, all the tables, then fills in some example data. The server is started as soon as the package is installed, so using `sudo mariadb` lets you gain full access to the server. Copying the script into the command line results in full functionality when used with the frontend app.

The frontend app utilises credentials to log into the database, hardcoded into the script and the application. This wouldn't happen with a production database, with the queries instead being handled by other services preventing direct access, but for the mockup it's fine. The standard credentials are:

- host: "localhost",
- user: "public",
- password: "publicpass",
- database: "facefillers_db"