Coursework

Computer Science OCR A Level

Alex Constantin-Gomez

Kensington Aldridge Academy

Table of contents

INTRODUCTION **3**

ANALYSIS **4**

Problem identification 4

Stakeholders 7

Problem research 11

Proposed solution 14

DESIGN **16**

Problem decomposition 16

Solution description 17

Testing approach 32

DEVELOPMENT **33**

Part I: Developing the web application 33

Part II: Developing the kitchen client program 70

Part III: Extending the web API and database operations 112

Part IV: Finalising the kitchen client program 123

EVALUATION **130**

Testing to inform evaluation 130

Evaluating the success of the solution 146

Final product description 147

Maintenance and further development 150

Introduction

My project will consist of a restaurant ordering system, which will allow customers to place their orders on a mobile app and then be sent to a computer system inside the kitchen. The system is designed for my father, who manages and works in his restaurant, L'oriental. He would like this software built to automate and speed up the ordering process as well as other smaller features, such as recording the revenue produced every day.
The app will contain a form and the restaurant menu, allowing customers to place orders. The app sends a request to a web server, which is linked to a database. The program running in a computer inside the kitchen will communicate with the web server for new orders. The program will be able to view orders as well as other features, such as marking them as complete, moving them around in the queue and recording the revenue produced per order.

This document will provide a complete description of all the stages in the development of this system, including an analysis, the design and implementation of the project, as well as a final evaluation.

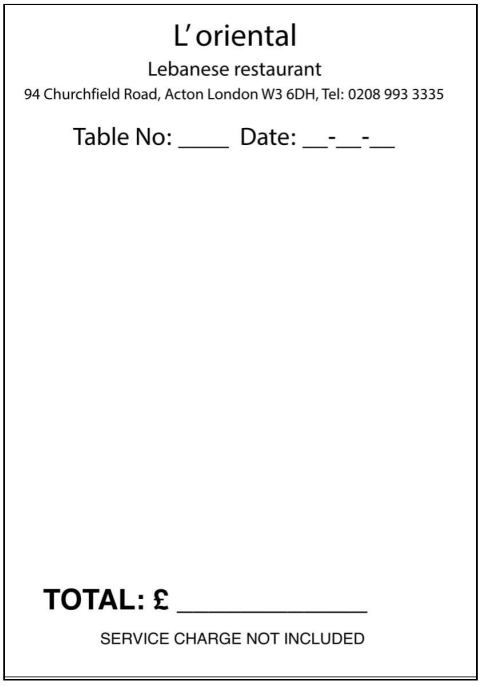
ANALYSIS

1.1. Problem identification

My father owns a restaurant in London, called L'oriental. He works there full time as a cook and the manager. The current ordering system is traditional: a waiter or waitress takes orders from customers sat a table who have a menu available on the table. The order is then communicated to the cooks in the kitchen. For customers that would like a takeaway order, the order is taken at the till rather than at a table.

Once the food has been prepared, it is served to the customer. If they eat inside the restaurant, once they are ready for payment, they are provided with a handwritten invoice, shown in figure 1.1. If the order is a take away, the payment is made upon the collection of the food and the handwritten invoice is given. Due to time constraints because of the busy environment, it is not feasible to rewrite a copy of the invoice to keep, meaning there is no record of the details of the order.

Only the revenue made from orders which are paid by card on a PDQ machine are formally recorded, meaning any orders paid by cash are not clearly recorded, which is one of the problems to be solved. For card transactions, a receipt is generated by the PDQ machine, however this only records the amount paid, and not the full details of the order (figure 1.2). The PDQ system does allow the customer as well as the kitchen staff to have a copy of the receipt, since two can be printed in very little time, unlike the handwritten invoice.


L' oriental

Lebanese restaurant

94 Churchfield Road, Acton London W3 6DH, Tel: 0208 993 3335

Table No: ____ Date: __-__-__

TOTAL: £ _____

SERVICE CHARGE NOT INCLUDED

Figure 1.1


L'ORIENTAL RESTAURANT

M520334509009387
TID26533777
AID : A0000000031010
Visa Debit
VISA DEBIT
4547
EXP 02/20
CONTACTLESS PAN.SEQ 01
SALE
MERCHANT COPY

AMOUNT £21.50
No CVM Used
20:05 24/11/17
AUTH CODE: 748800
RECEIPT 1352

Figure 1.2

Apart from the more technical problems described above, an obvious issue that arises in most restaurants is time management. For successful restaurants, it is crucial to be fast in order to process more customers and more orders to maximise profits.
Therefore, it is important to have a clear set of orders, each with its own priority in order to have a queue in mind. At L'oriental, the orders taken by the waiter/waitress (which are written down on a piece of paper) are pinned onto a wall-mounted holder inside the kitchen, shown in the figure 1.3. This acts as a reference in order to decide the timing of the preparation of the food. For this method to be as efficient as possible, the orders on the holder must be easy to understand. However, in some cases, this may cause confusion, for example, due to poor handwriting. Having the data shown on a screen will avoid this problem. Also, if you notice on figure 1.3, there are only 6 "spikes", which may not be enough space in some cases. A computer will be able to solve this problem very easily using basic data structures and a GUI.

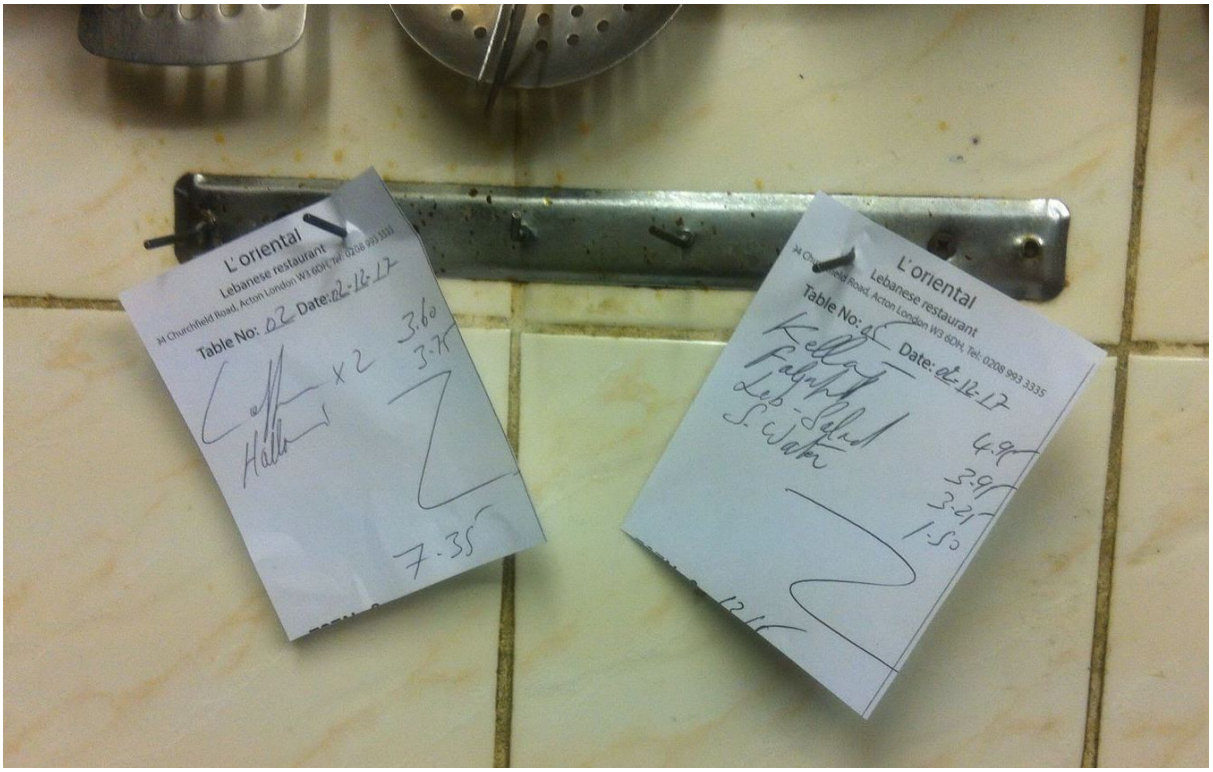


Figure 1.3

To solve this problem, sorting the orders into a clear queue which can be modelled on a computer makes the job easier for the chefs, saving crucial time. Seeing all the orders in an organised fashion allows the chef to prioritise orders more easily.

Justification for the computational methods to be used

The problems which have been described above can all be solved using a computational approach. For example, storing data about orders is amenable to a computational approach since computers are designed to work with large volumes of data easily. With the use of a database, orders can be easily stored and accessed. This will avoid having to store physical copies of invoices, which can get lost easily.

Organising the orders using the current system used at L'oriental has various drawbacks, as I have already discussed. Modelling the "holder" shown in figure 1.3 is definitely amenable to a computational approach, since a data structure could easily be designed using a computer. It could be as simple as a FIFO structure such as a queue, or a more flexible data structure such as a list. This will be further explored in the Design section.

To extend the idea of organisation and clarity, a computer can also be used, since most home/work computers run a graphical based operating system. Thus, a graphical user interface can be designed to organise all the information about orders (and the orders themselves) into a clear and compact way, such that all the information needed by the chefs are displayed on screen. This also removes the issue regarding poor handwriting, since digital fonts are mostly very easy to read.

Another feature of the system which could easily be solved by a computer is automatically calculating the total price of the order. This would require a simple algorithm which uses the prices in the menu to figure out the total price.

1.4. Proposed solution

Requirements for configuration:

My proposed solution has few hardware requirements from the point of the view of the restaurant manager, however it relies on customers having their own devices to place orders if they would like to.
From the restaurant's point of view, my proposed solution will require the following hardware configuration:

- A computer system, such as a laptop or a desktop computer, to run the kitchen client program.
- A web server, to run the web application.
- A monitor for the computer system, so the staff can view the graphical user interface and interact easily with it.
- A mouse (or a touchpad) to interact with the graphical user interface, for example, clicking buttons.
- A keyboard to input text information, for example, entering order details into a form.
- A network card in the computer system in order to connect to the Internet, in order to fetch any new orders.
- A router or modem to obtain access to the Internet (for the same reason stated above).

From the customer's point of view, the solution will require the following hardware devices:

- Any device with access to the Internet, such as a mobile phone, a tablet, a laptop, etc. in order to place the order through the web application.
- A web browser to view the web application.

The software required for my proposed solution to work is the following:

- The computer system must be running Windows operating system, version 7, 8 or 10.
- A database software, to store orders. This software will be decided and installed in the following sections.
- A web server framework, in order to run the web application. This will be decided and installed in the following sections.

Success criteria:

For a successful solution, the following criteria should be met:

- It should store order and customer data in a database
- It should have a simple and easy to understand user interface
- Customers should be able to submit their own orders
- Staff users should be able to manually add orders to the program
- There should be a way of categorising orders
- It should be able to move orders around in the queue
- It should be able to calculate daily revenue automatically

The success criteria list above cover all the essential functionality of the system, such as storing data in a database, adding new orders, categorising orders, alongside usability features such as moving orders up or down the queue and having a simple user interface.

DESIGN

2.1. Problem decomposition

Due to the nature of the problem analysed in the previous sections, it will be necessary to break down the system into smaller parts. This will allow me to approach each part of the system separately, making the overall task more manageable.
Decomposing the problem will also help me design the structure and architecture of the solution, since I will be able to analyse how and what the most efficient way of integrating the different parts of the problem together.

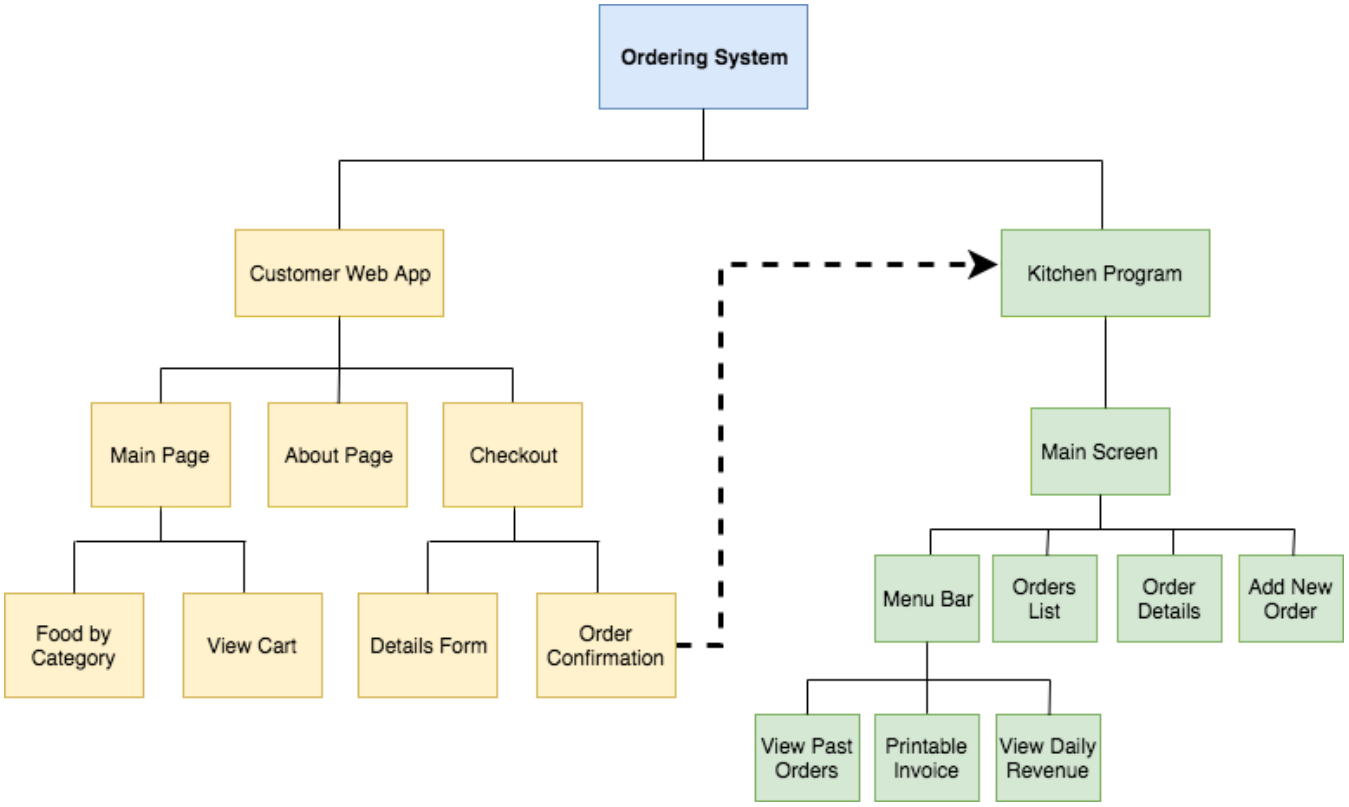


Figure 2.1

The diagram shown in figure 2.1 shows how I have broken down the problem into two main sections: the customer web application (represented by the orange boxes) and the kitchen client application (represented by the green boxes). The customer web application will then communicate with the kitchen client application via the Internet to receive the orders.
I have divided the project into these two parts because they can be developed separately up until I connect both modules via the Internet, which is represented by the dotted arrow.
The customer web app can be broken down into three main pages: the interactive menu, which displays all the food sorted by category and the state of the cart on the sidebar of the page, the about page, which will contain any extra information about the restaurant and the website, and the order confirmation page, which will send the order to the kitchen software.

The kitchen client program will consist of a main interface which displays the a list of the orders fetched from the web application and a panel which will show the details of the selected order. It will also have an option to view past orders from the orders database.

2.2. Solution description

Structure of the communication over the network

The flowchart below shows how the overall system lifecycle works and how both components of the system will work together over the Internet. The structure of the communication over the network will be based on a REST API (representational state transfer application programming interface) on the web app. The desktop program will poll the web app every certain amount of time (defined by the refresh rate variable) to check if there are any new orders to be shown on screen.

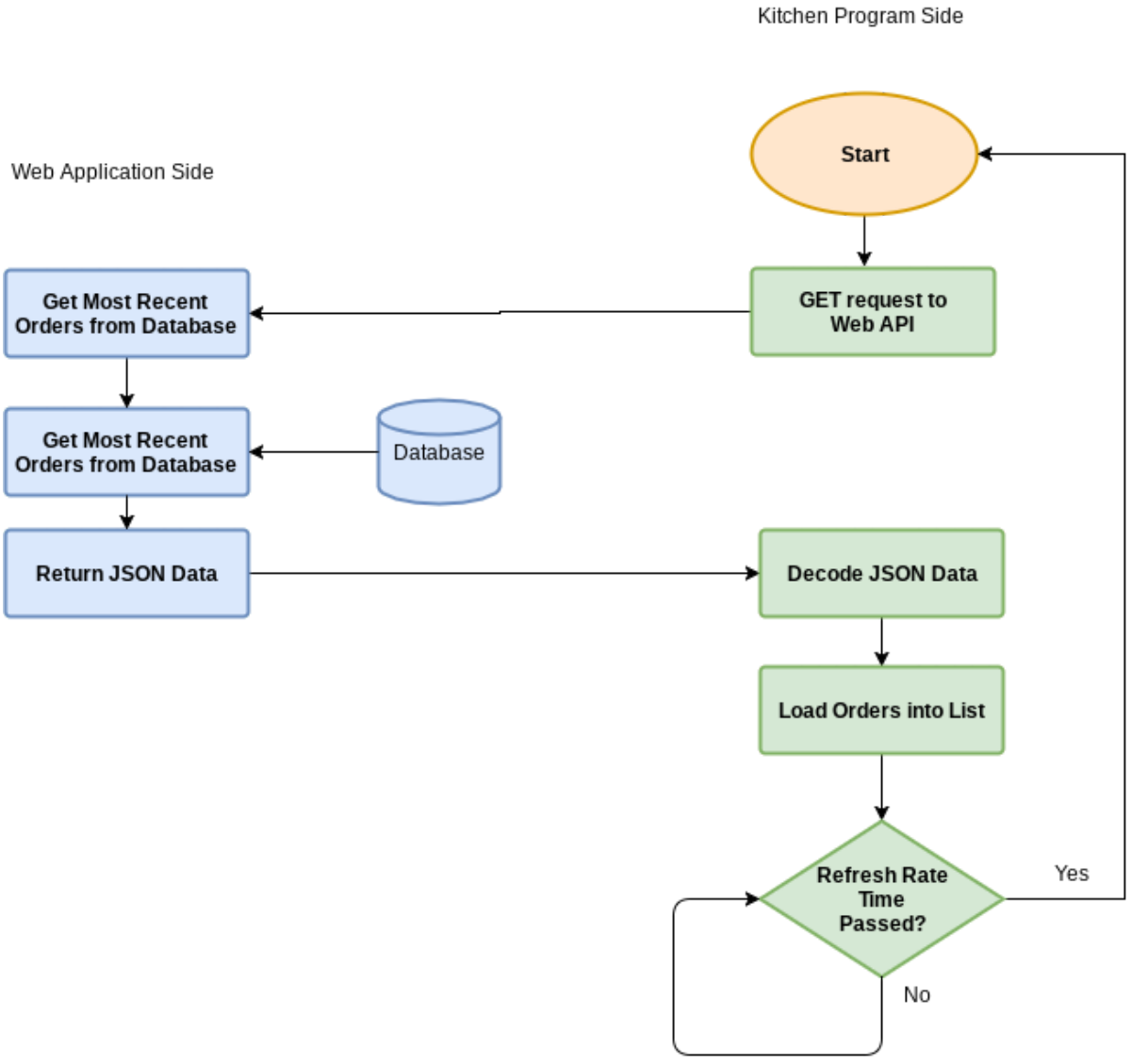


Figure 2.2

Justification of the chosen structure of the network

Since the majority of the orders will be created from the web application, it will be more efficient to have the database on the server side of the system. Also, it is much more straightforward and easier to implement a REST API on the web app to send data from the server to the desktop kitchen program rather than sending data from the desktop program to the server, as this may require websocket connections, which is excessive for such a basic data transfer. However, although polling may be seen computationally inefficient in some cases, for communication over a network I believe it is the simplest method to implement and it should work perfectly, since the data being communicated is only text and therefore there will not be a problem with bandwidth.
In addition, using a REST API is probably the best option since data structures such as classes can be easily converted (ie: serialised) into JSON data, which is the most common method of transmitting data over a REST API.

Algorithms to describe the solution:

Create new order from web application:

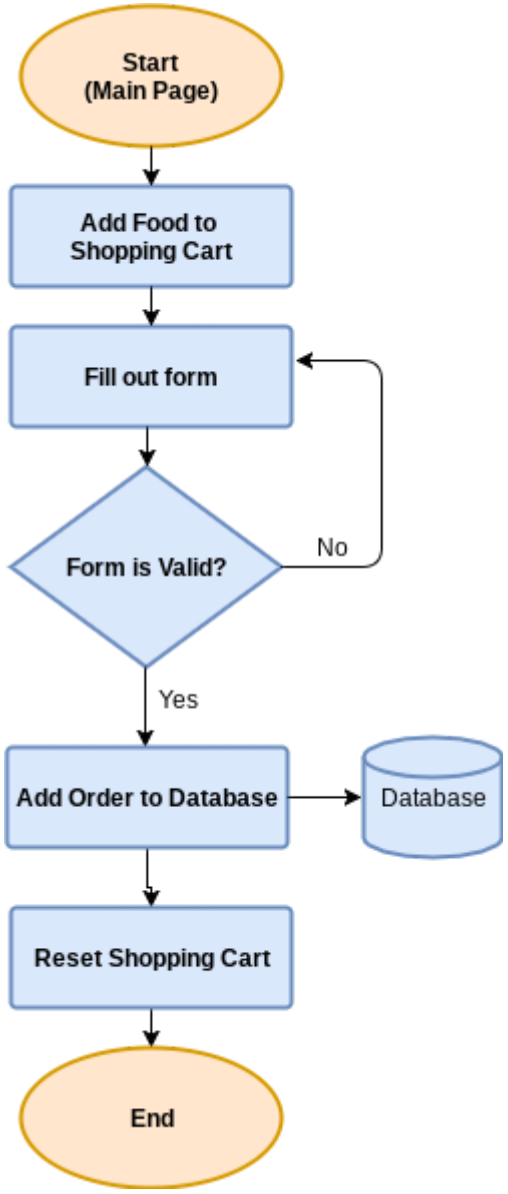


Figure 2.3

How the algorithm works:
The flowchart shows the general process of creating a new order and committing it to the database. The customer adds food items to the shopping cart by clicking the "Add" button next to each item on the menu screen. This adds the food to the cart, which is stored in a web session. Upon clicking the checkout button, they must fill out a form with their details. The form is validated, and added to the database. The shopping cart in the session is set to empty.

Justification for the algorithm:
This process is essential because it defines how the orders are added to the database. The algorithm requires the use of a web session to maintain the food items persistently throughout the use of the web application. The form validation is essential to make sure the database is consistent. In conclusion, this is a very common and straightforward algorithm used by many e-commerce websites to process and record orders.

Create new order from kitchen program:

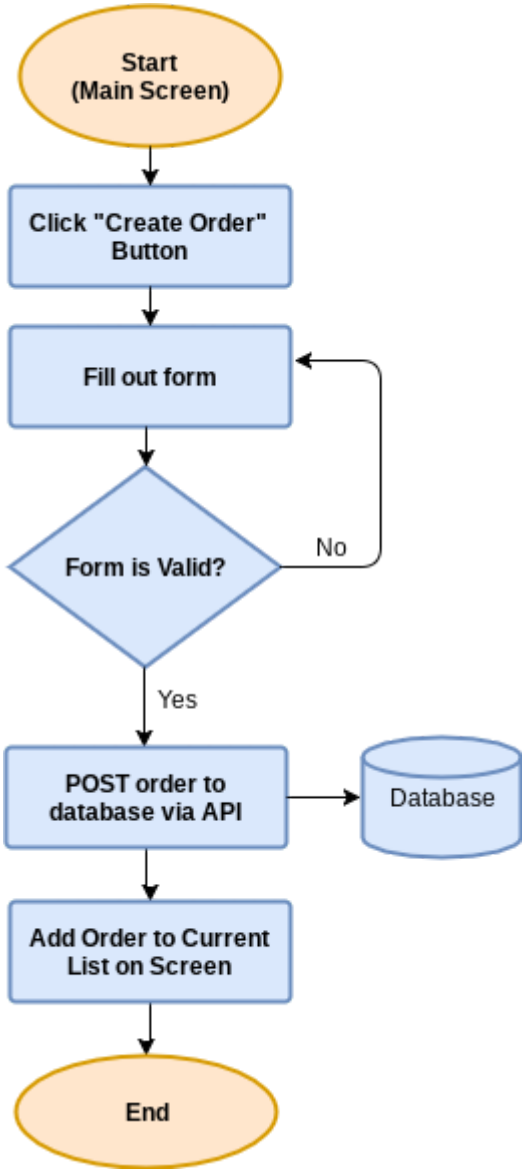


Figure 2.4

How the algorithm works:
The algorithm collects the order data by filling out a form manually. The form is validated to make sure the data inputted makes sense and is correctly formatted (i.e.: correct data types).
If so, the order is sent to the server using a POST HTTP request. The server receives and decodes the data, and instantiates an Order object based on the data. The order is then committed to the database on the server. If the order has been successfully received by the server, a response will be returned to the kitchen program to confirm the data has been received by the server. Then, the order is shown on screen.

Justification for the algorithm:
This algorithm will be very useful for the staff to record orders that are not submitted via the website, for example, via a phone call. Since the database is located on the server, the order must then be "posted" to the server for it to be added to the database, as explained in the structure of the system. The order is then added to the list of current orders shown in the kitchen desktop program.

Mockup of Main Screen

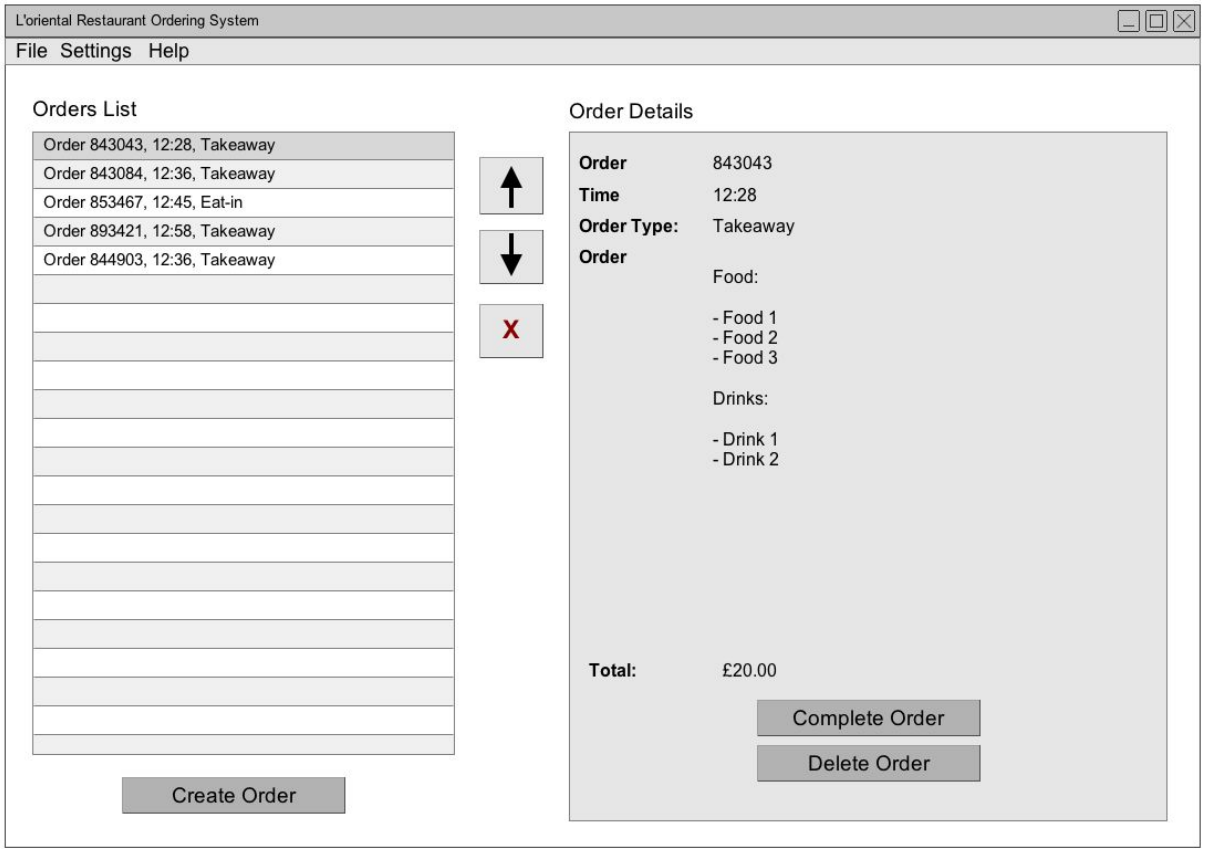


Figure 2.5

The user interface for the main screen, as sketched in figure 2.5, is the most important screen of the kitchen program. It is divided into three parts: the menu bar, the orders list and the details of the selected order.
The current (live) orders are received by the program and displayed in a table on the left-hand side of the window. Each row of the table shows the heading of the order, which consists of the order ID, the time submitted and the type of order (takeaway or eat-in). The position of the orders can be manipulated using the buttons on the centre of the screen, which shift a selected order up or down. They can also be deleted, before being prompted by the program to make sure an order is not accidentally deleted. Just under the table is a button to create a new order, which opens a form which required manually filling out, shown in figure 2.6.

The panel on the right-hand side displays the information associated to the order selected in the table. Upon selection, the panel is updated with the relevant information and rendered on screen. This section includes a button to mark an order as complete, as well as another button to delete an order. If no order is selected, the panel will be “greyed out” (disabled).

The menu bar contains different buttons to access the rest of the program, such as viewing past orders in the database or viewing the revenue of a given day. It also allows to access the program settings, which allows the user to change the polling refresh rate and other features such as the font size.

Mockup of the Create Order Form Pop-Up

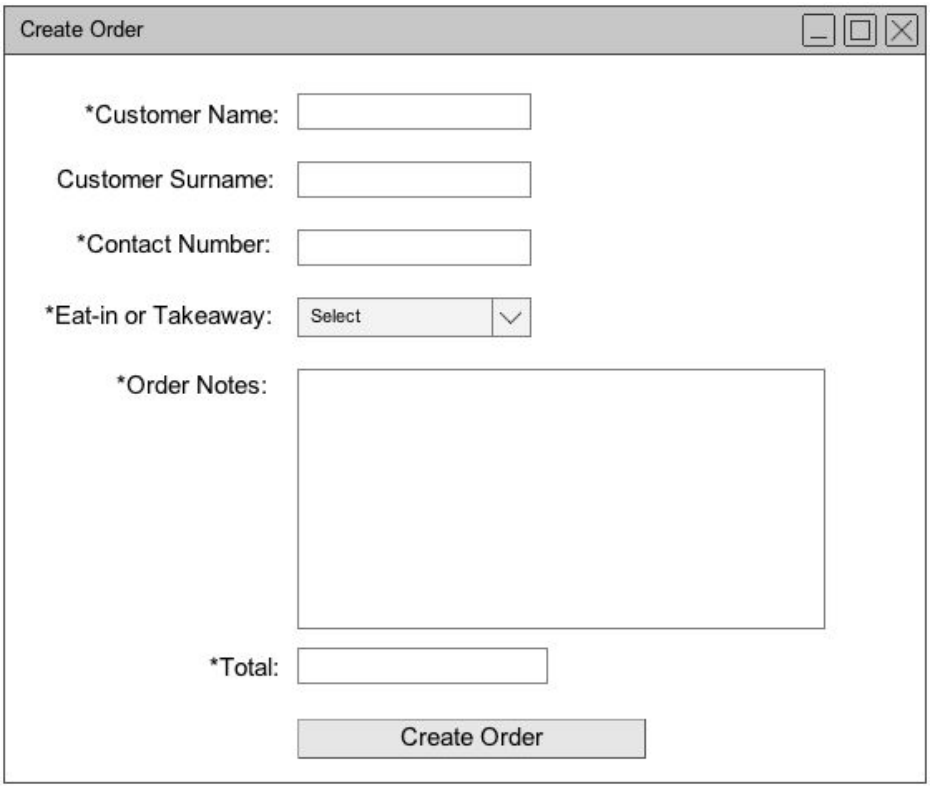


Figure 2.6

The form shown above (figure 2.6) appears upon clicking the “Create Order” button on the main screen. This form is intended to be used when kitchen staff would like to record an order which has not been submitted via the web application. For instance, this may be useful when taking orders from a phone call. It allows the user to enter a customer name (and surname), their contact number, the type of order and the order contents.

Once the “Create Order” button is pressed, the relevant validation will take place. If successful, the window will close and the order will be added to the orders list on the main screen.

Mockup of the window to search for past orders in the database:

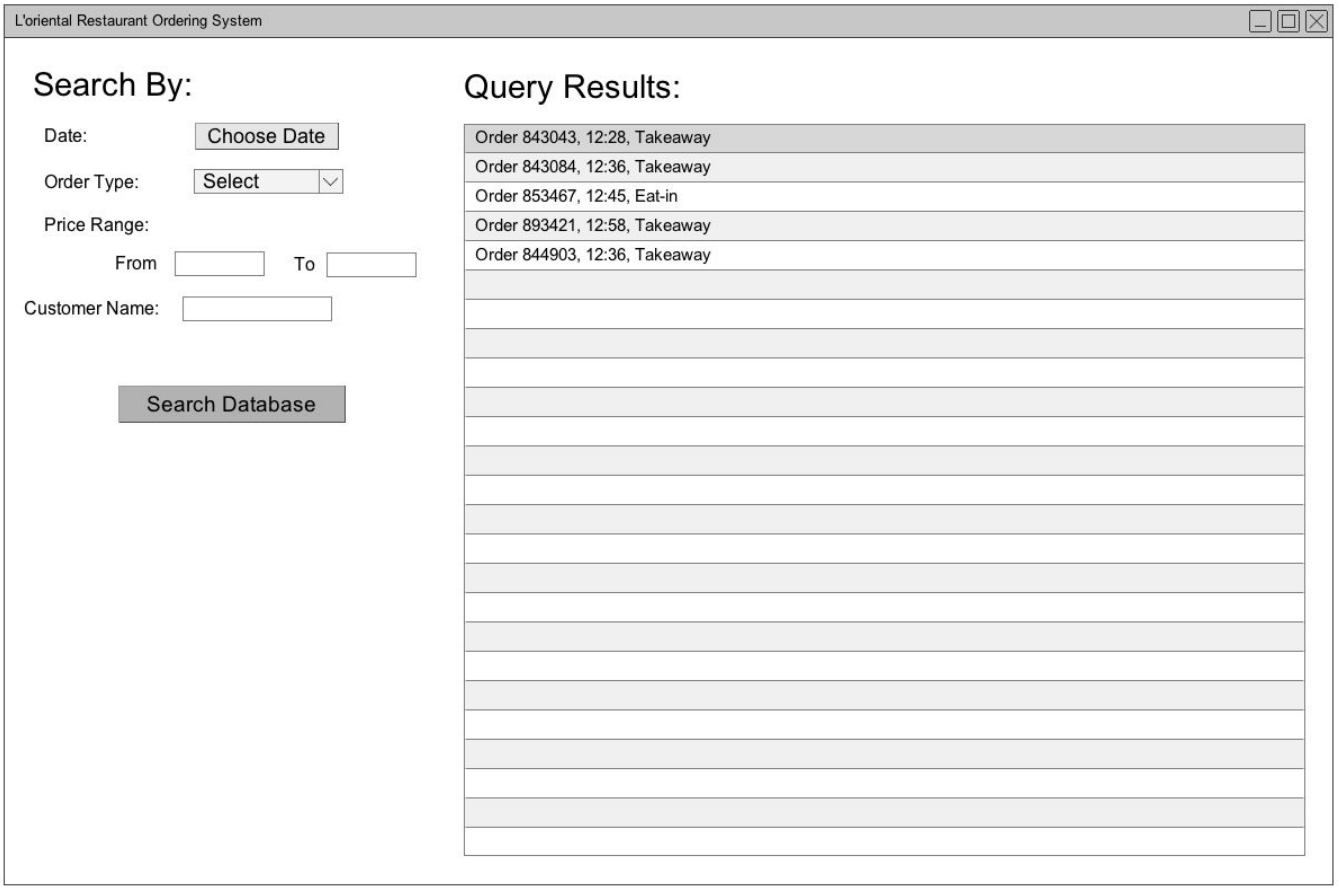


Figure 2.7

This window, which can be accessed via the menu bar on the main screen, allows the user to query the database for orders based on certain parameters, such as a range of dates, the type of order, the price range or the customer name, if known.
This will query the database using SQL and return results to the table shown on the right-hand side of the screen. Each row, just like in the main screen, shows the order ID, the time and date submitted and the type of the order. These orders can then be clicked, which opens a window containing the details of the order, shown in the next mockup.

Mockup of the window to view the details of a past order

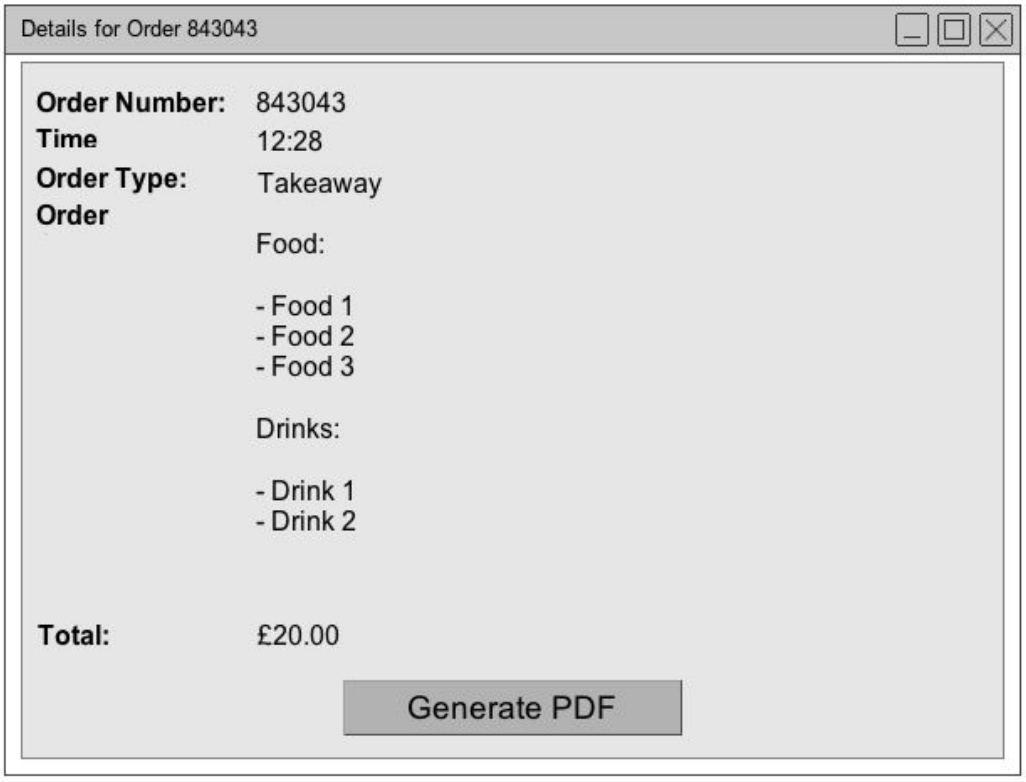


Figure 2.8

The window shown in figure 2.8 shows the details of a past order, similar to the details panel on the main screen. It also has a button to generate a PDF containing the information, for the user to save onto their hard drive, which can then be printed off. This can be useful when the user would like to keep a paper copy of an order stored digitally, on the database.

Mockup of the Home Page

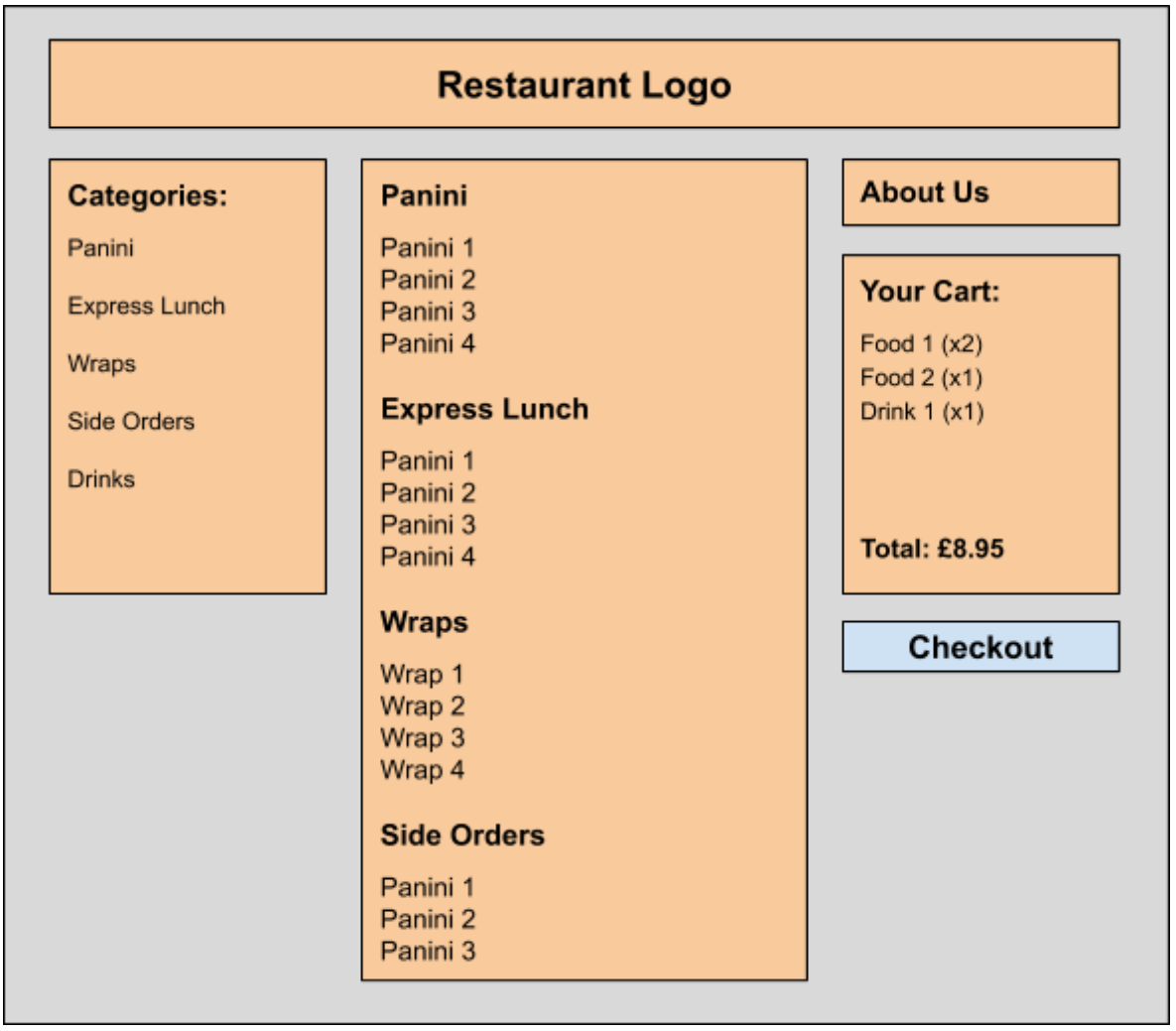


Figure 2.9

The home page of the website is split into three columns: the two “sticky” (fixed) sidebars and the main content section. The left sidebar has the links to each subheading in the restaurant’s menu, so users can browse the food easily. The right sidebar shows the status of the cart: the items in the cart, the quantity of each item and the total price of the order, with a button to proceed to the confirmation page.
The central column has the food items classified by category according to the subheadings in the left sidebar. Each food item has a description and a price, with a button to add it to the cart. This main column will be scrollable, unlike the fixed sidebars. As mentioned in the Analysis section, the website’s user interface is inspired by Just Eat’s website.

Mockup of order confirmation page

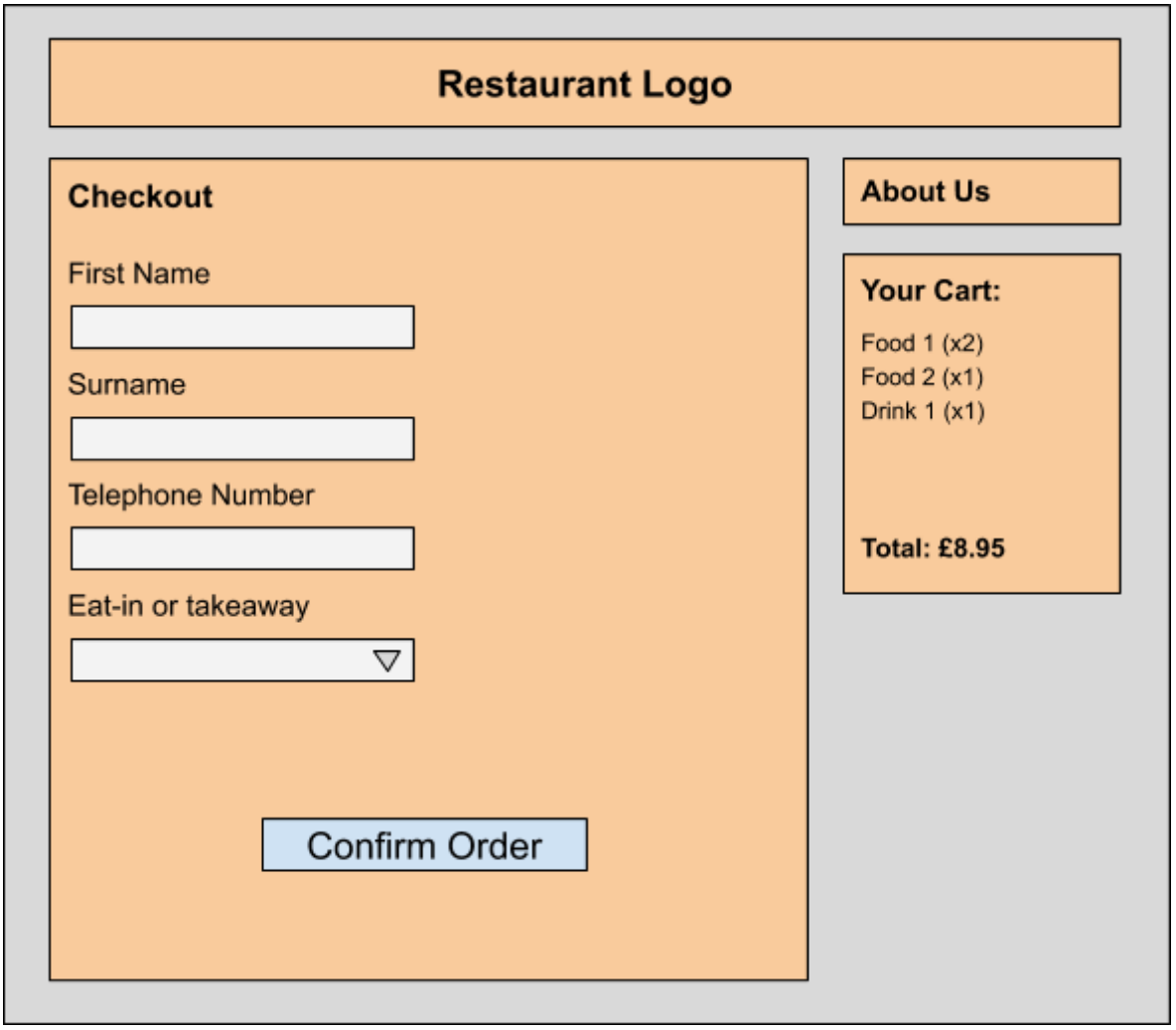


Figure 2.10

This page provides a form for the customer to input their details and select whether the order is for takeaway or to eat-in. This allows the backend of the web application to create a Customer object and link it with its corresponding Order object.
The Confirm Order button will submit the form via a POST request and the form data will be collected, in order to instantiate a Customer object and then create an Order object using the form data alongside the items in the cart.
If successful, the user will be redirected to a confirmation page to make the user aware that the order has been submitted.

Database entities

Entity	Attribute	Data Type	Description
ORDER	orderID	INTEGER	A unique ID given to each order (primary key)
	timestamp	DATE-TIME	The time and date of the order submission
	type	STRING	Identifies whether the order is eat-in or takeaway
	customerID	INTEGER	Foreign key to the Customer entity

Entity	Attribute	Data Type	Description
CUSTOMER	customerID	INTEGER	A unique ID given to all customers (primary key)
	name	STRING	The name of the customer
	surname	STRING	The surname of the customer
	telephone	STRING	A numeric string which holds the telephone no.

Data structures

Firstly, the current orders to be sent to the program will be stored in a list rather than a queue (First In First Out structure) to satisfy different priorities the chef may have. This will allow the chef to freely move the orders up and down the queue, using the arrow buttons. Also, deleting orders (if required) can be done in any position, rather than following the First In First Out algorithm.

On the webserver, the orders will be collected as a class defined in Python, since the web server framework I will be using is Flask, which is based on Python. Once the form data has been collected, an instance of the Order class will be created. This will then be committed to the database and serialised to be used in the REST (representational state transfer) API to be fetched by the kitchen program.

Pseudocode for the Order Class:

```
class Order
	private id
	private timestamp
	private category
	private contents

	public procedure new(orderContents, orderCategory, orderTotal)
		id = generateUniqueId()
		timestamp = getCurrentTime()
		category = orderCategory
		contents = orderContents
		total = orderTotal
		completed = false

	public procedure getId()
		return id

	public procedure getTimestamp()
		return id

	public procedure getCategory()
		return id

	public procedure getContents()
		return id

Endclass
```

Pseudocode for the Customer Class:

```
class Customer
	public id
	public name
	public surname
	public telephone

	public procedure new(custName, custSurname, custTelephone)
		id = generateUniqueId()
		name = custName
		surname = custSurname
		telephone = custTelephone
endclass
```

Justification for the Order and Customer classes:

Taking an object-oriented approach towards creating orders and customers is a suitable method of dealing with these data structures. It allows orders and customers to be treated as single entities with their own attributes and methods, which provides a more robust codebase and an easier time reading and understand the code.

Form validation algorithm for the checkout page on the website:

- First name must be a string of up to 30 characters
- Surname must be a string of up to 35 characters (on the kitchen program, this is not required)
- Telephone number must be a numeric string following the UK format (mobile or home)
- Order category will be selected to be eat-in or takeaway

Pseudocode for Form Validation:

```
function ValidateForm()
	IF firstName.length > 30 OR firstName.length == 0 THEN
		return "Invalid field"

	IF surname.length > 35 OR surname.length == 0 THEN
		return "Invalid field"

	IF dropDownBox.value == null THEN
		return "Invalid field - you must select an option"

	pattern = Regex("^((\+44\s?\d{4}|\(?\d{5}\)?)\s?\d{6})|((\+44\s?|0)7\d{3}\s?\d{6})$")
	IF pattern.Matches(telephoneNumber) == true THEN
		return "Phone number is not correctly formatted"

	return "Validated"
```

In order to detect whether the inputted telephone number matches the UK format for mobile and landline numbers, I will be using regular expressions: a sequence of characters that define a search pattern. This will be implemented using an external “regex” library.
The regular expression I will be using is

^((\+44\s?\d{4}|\(?\d{5}\)?)\s?\d{6})|((\+44\s?|0)7\d{3}\s?\d{6})$

To ensure this regular expression is what I am after, I have designed some test cases in the next page, which will accept or reject certain phone numbers.

Designing test cases for the regular expression

This regular expression is able to detect to both mobile and landline phone numbers. Here are some tests I have carried out to ensure it supports the multiple formats.

Valid matches:

String = "02073135800" MATCH INFORMATION Match 1 Full match 0-11 `02073135800` Group 1. 0-11 `02073135800` Group 2. 0-5 `02073`	String = "+442073135800" MATCH INFORMATION Match 1 Full match 0-13 `+442073135800` Group 1. 0-13 `+442073135800` Group 2. 0-7 `+442073`
String = "07847937875" MATCH INFORMATION Match 1 Full match 0-11 `07847937875` Group 1. 0-11 `07847937875` Group 2. 0-5 `07847`	String = "+447847937875" MATCH INFORMATION Match 1 Full match 0-13 `+447847937875` Group 1. 0-13 `+447847937875` Group 2. 0-7 `+447847`

Invalid matches:

String = "012345" Reason: The number is not long enough.	String = "abcdefgh@%$" Reason: The string contains non-numeric characters.

Evaluating these test cases, the regular expression works and meets different types of UK formats of telephone numbers, so I shall be definitely using it in my code.

2.3. Testing approach

Unit testing:
In unit testing, each module in the system is tested to make sure that it functions correctly. I have chosen to use this method because I will be developing the web application module separately to the kitchen program module. This means I can fully test whether the website works before moving on the the second part of the system.

Integration testing:
The integration test makes sure that all the units of the system work together. Once all the modules function individually (as described in unit testing), it is crucial to test the networking aspect of the system, to ensure both modules work together. Therefore, I will be be developing the network aspect of the system alongside iterative integration testing. Due to the importance of the networking features, I believe this will be the most thorough and repeated testing method.

White box testing:
In whitebox testing, the aim is to consider all of the possible paths through the algorithm in order to test that it is operating as expected. I will used this to exhaustively test the system and make sure it functions correctly, alongside test data which is described below.

Use of normal, invalid and boundary data for testing
To test the validation algorithms used in the forms, I shall be using valid, invalid and boundary data to ensure the form collects data correctly. For valid data, I shall be using an expected value which should be accepted by the form. For invalid data, I shall be using a value of a different data type, an incorrect length... For boundary data, I shall be using valid values which are on the border of being invalid, for example, inputting a string of maximum length allowed in that entry.

Post-development testing
Once the development stage has been completed, further testing shall be carried out to test the functionality and robustness of the solution. The usability features will also be tested by the target stakeholder, who will then provide feedback. These testing stages will be required to make sure the success criteria defined in the Analysis section are met, and if not, considered for future development.

DEVELOPMENT

Part I: Developing the web application

Choosing a front-end framework to implement the website UI

The website's user interface design, as shown in mockups in the Design section, is very simplistic. Therefore, to avoid building the HTML and CSS templates from scratch, which could be very time consuming, I have decided to use the Bootstrap framework, which is a very popular open source HTML, CSS and JavaScript front-end library. It includes all the CSS code for HTML components to look user-friendly and appealing.
In addition, it is responsive to the size of the web page, meaning it will resize perfectly for mobile devices, which is perfect for this project, because most customers will probably prefer to use the mobile device to place orders.

Choosing a Python framework to implement the website back-end

As mentioned previously, I shall be using the Python programming language to implement the server back-end, mainly because it is the language I am most comfortable using. When researching for web frameworks for Python, I had two options: Django or Flask. After further research, I have decided to use Flask because it is more minimalistic and it suits smaller web servers, such as the one I am planning to implement. Django, on the other hand, is more rigorous to set up because it has more overheads and requires more dependencies. It is also recommended for larger scale applications, such as social networks and blogging sites.

Web template engines

To make the content of the website dynamic, Flask uses a web template engine called Jinja2. These engines allow for data found in the back-end to be passed into the front-end and rendered to the user. This is depicted visually below, in figure 3.1.
As it will be demonstrated later, Flask uses a function called *render_template()* which returns a HTML response to the client alongside any variables, which are then shown using template tags, which have the format: *{{ variable_name }}*.
As well as this, Jinja2 has more complex features, such as loops and if statements. These programming constructs have the following format: *{% if condition %}{% endif %}*. These constructs will be very useful when it comes to dynamically rendering the items in the menu, rather than "hard-coding" the HTML code for every item, which would be very time consuming.

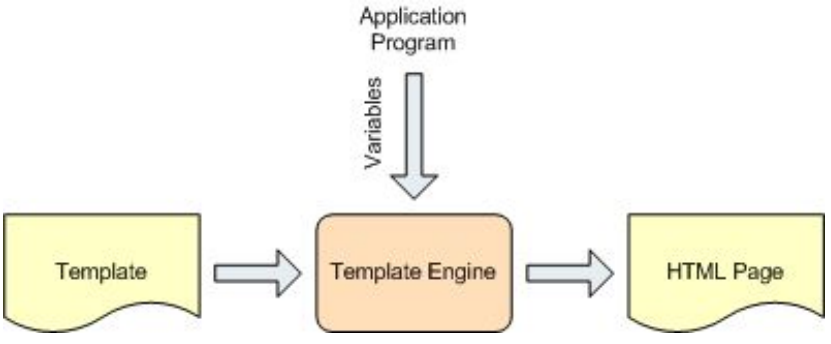


Figure 3.1

Creating the base template

Most of the times, many HTML components in a website are found on every single page of the application, for example, the top navigation bar. Rather than repeating the HTML code for the navigation in every HTML file, Jinja2 has a feature which allows HTML templates to inherit from a base template using the following tag: *{% extends "base.html" %}*.
This feature will allow me to link all the Bootstrap CSS and JavaScript files only in base.html, as well as the navigation bar. Hence, all the other templates will inherit from base.html and have the link tags too.

```
1  <!doctype html>
2  <html lang="en">
3      <head>
4          <title>{{ title }} - L'Oriental Restaurant</title>
5          <meta charset="utf-8">
6          <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
7
8          <!-- Bootstrap CSS file -->
9          <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
10     </head>
```

The code above shows the <head> part of the base.html, which will be inherited by all the other pages. The <title> tag includes a *{{ title }}* variable, which will be passed into by the Flask views, which will be developed later. It also contains the link to the Bootstrap CSS file.

```
14     <body style="background-color: #fff1e0;">
15         <!-- Logo header panel -->
16         <div class="row">
17             <div class="col">
18                 <nav class="navbar navbar-light navbar-fixed-top" style="background-color: #f9821e;">
19                     <div class="container">
20                         <a href="{{ url_for("menu") }}"><span class="navbar-brand mb-0 h1">L'Oriental Restaurant</span></a>
21                     </div>
22                 </nav>
23             </div>
24         </div>
25         <br>
26
27         <!-- Main page contents go in here -->
28         <div class="container">
29             {% block content %}
30             {% endblock %}
31         </div>
32
33
34         <!-- JavaScript libraries for Bootstrap to work correctly -->
35         <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
36         <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"></script>
37         <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
38     </body>
39 </html>
```

The code above shows the <body> of the HTML page, which is where all the content is to be rendered onto the browser. A light orange background colour is given to match L'Oriental's theme. A header navigation is also coded into base.html so that is appears throughout the website. It is formatted by Bootstrap using the *navbar* class, shown on line 18. The navigation bar will only contain the name of the restaurant, which is an anchor tag that links back the home page, which is the menu. This is done on line 20, using another template tag, called *{{ url_for() }}*. This just gets the URL for the view function called menu(), which is defined in the Flask code.

On line 28, a container is defined to hold the content of the other templates. This is achieved using the *{% block block_name %} {% endblock %}* construct. Finally, all the JavaScript files are linked in for Bootstrap to work correctly.
In order to actually render any HTML files, the Flask application must be initialised first, which is the next step in the development.

Initialising the Flask application and creating the home page

To create the Flask application, I wrote all the code in a Python file called app.py. The basic structure of app.py is shown below. The flask library is imported first. The application is initialised using the Flask class defined by the flask library. The final two lines tell the application to run. All the code in between represents view functions: they return responses to the user based on the URL path they have request. For example, if they request the / path, which is the default path, the menu page will be returned. The path can be specified using the *@app.route()* decorator, which is found just above the function definition. The menu view simply renders the menu.html file (shown next) and passes the title variable with value “Menu” into the file. This value will then be replaced for the *{{ title }}* tag in base.html.

```python
from flask import Flask, render_template

# Initialise the Flask application
app = Flask(__name__)

# The home (menu) page of the web app
@app.route('/')
def menu():
    return render_template("menu.html", title="Menu")

# This runs the application
if __name__ == '__main__':
    app.run(debug=True)
```

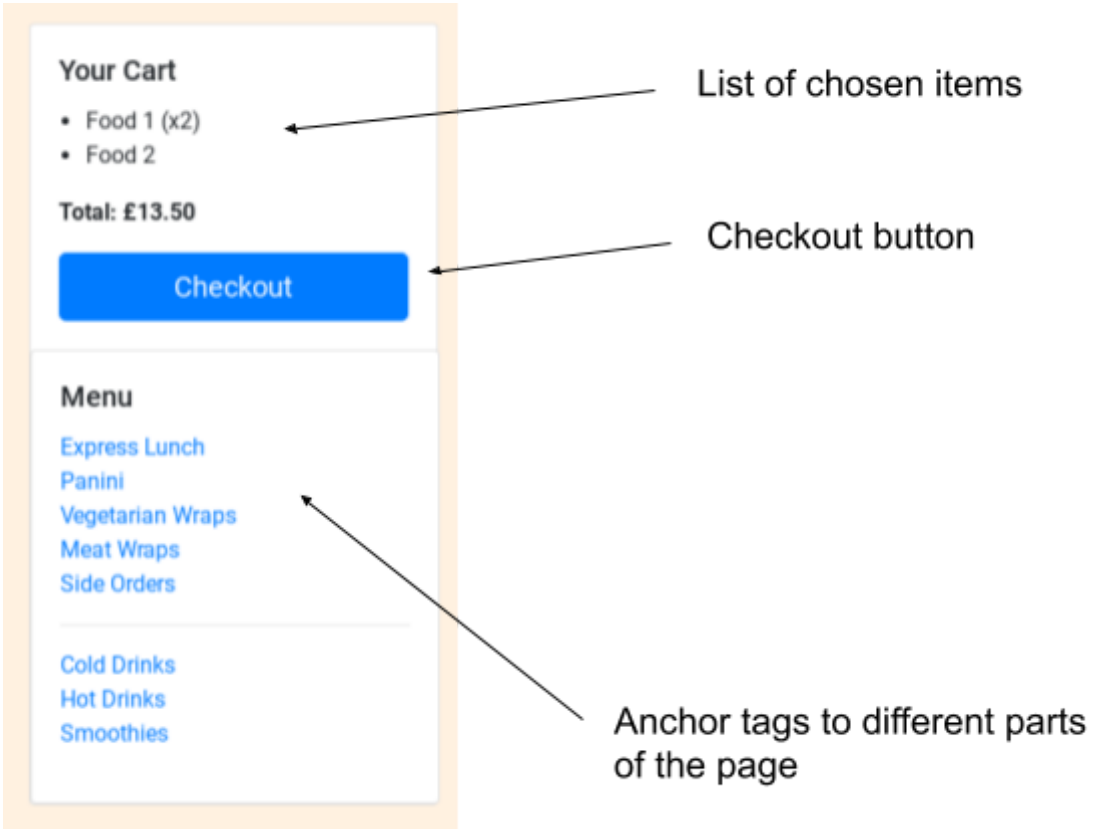
All the templates are found in the *templates* folder, which Flask will automatically know where to find. To achieve the look which I designed, I used following HTML code in menu.html:

```html
<div class="col-sm-4" style="position: fixed; right:0;">
        <div class="card" style="width: 18rem;">
            <div class="card-body">
                <h5 class="card-title">Your Cart</h5>
                <div class="card-text" >
                    <ul style="padding-left:8%;">
                        <li>
                            Food 1 (x2)
                        </li>
                        <li>
                            Food 2
                        </li>
                    </ul>
                    <p><b>Total: £13.50</b></p>

                </div>
                <a href="{{ url_for("checkout") }}" role="button" class="btn btn-primary btn-lg btn-block">Checkout</a>
        </div>
```

This code shows the status of the cart, using some placeholder data for now.

```
77              <div class="card" style="width: 18rem;">
78                  <div class="card-body">
79                      <h5 class="card-title">Menu</h5>
80                      <div class="card-text">
81                          <ul style="list-style: none;padding-left:0;">
82                              <li><a href="#expressLunch">Express Lunch</a></li>
83                              <li><a href="#paninis">Panini</a></li>
84                              <li><a href="#vegetarianwraps">Vegetarian Wraps</a></li>
85                              <li><a href="#meatwraps">Meat Wraps</a></li>
86                              <li><a href="#sideorders">Side Orders</a></li>
87                              <hr>
88                              <li><a href="#colddrinks">Cold Drinks</a></li>
89                              <li><a href="#hotdrinks">Hot Drinks</a></li>
90                              <li><a href="#smoothies">Smoothies</a></li>
91                          </ul>
92                      </div>
93                  </div>
94              </div>
95
```

This code shows the list of links to the different categories in the restaurant menu.
This list and cart status box are both found on a sidebar, and produce the following output.



Loading the menu into the web application

The restaurant's menu has 61 different items which may be ordered by customers, and are also classified by categories. Manually writing the code for all the items into the HTML would be very time consuming, especially since each item has its own information modal (popup).

To approach this problem, I shall create a CSV file holding each menu item, in the following format:

```
Chicken salad,EXPRESS LUNCH,Freshly grilled breast fillet with Lebanese salad,6.00
```

To read the CSV file and load it into a Python data structure, I will create a file called food_items.py within a folder called orders. I have decided to use a dictionary data structure to store all the items, where the keys of the dictionary represent the category. The dictionary will be stored under the `FOODS` identifier.

The procedure to load all the menu items into `FOODS` is shown below. It works by checking the second column of each item (row) and adding it to the dictionary. Once all the items have been loaded into the dictionary, it will remain a constant.

```
1  import csv
2
3  # dictionary to hold all menu items by category
4  FOODS = {   "Express Lunch":[], "Vegetarian Wraps":[], "Meat Wraps":[],
5              "Panini":[], "Side Orders":[], "Hot Drinks":[], "Cold Drinks":[],
6              "Smoothies":[]  }
7
8
9  # Load food items from csv file into python dictionary
10 def get_food_items(csv_file):
11     with open(csv_file, "r") as file:
12         reader = csv.reader(file)
13         for row in reader:
14             if row[1] == "EXPRESS LUNCH":
15                 FOODS["Express Lunch"].append((row[0], row[2], row[3]))
16             elif row[1] == "PANINI":
17                 FOODS["Panini"].append((row[0], row[2], row[3]))
18             elif row[1] == "VEGETARIAN WRAPS":
19                 FOODS["Vegetarian Wraps"].append((row[0], row[2], row[3]))
20             elif row[1] == "MEAT WRAPS":
21                 FOODS["Meat Wraps"].append((row[0], row[2], row[3]))
22             elif row[1] == "SIDE ORDERS":
23                 FOODS["Side Orders"].append((row[0], row[2], row[3]))
24             elif row[1] == "HOT DRINKS":
25                 FOODS["Hot Drinks"].append((row[0], row[2], row[3]))
26             elif row[1] == "COLD DRINKS":
27                 FOODS["Cold Drinks"].append((row[0], row[2], row[3]))
28             elif row[1] == "SMOOTHIES":
29                 FOODS["Smoothies"].append((row[0], row[2], row[3]))
30
31
```

To read a CSV file, Python has a built-in library which allows this. The code is self-explanatory: a for loop is used to iterate through all rows in the CSV file, and it checks what the value of the second column is. It then appends it to the list indexed by each key in the `FOODS` dictionary.

Finally, since the dictionary only needs to be loaded once from the external CSV file, a function decorator called `before_first_request` can be used to ensure that the items are loaded from the CSV file once, which occurs when the app.py script is interpreted.

```
14 # Loads menu items from csv file at server startup
15 @app.before_first_request
16 def load_menu_items():
17     get_food_items("food_items.csv")
```

Rendering the menu items onto the HTML templates

Once all the menu items have been loaded into the `FOODS` dictionary, it is very simple to dynamically render all the items onto the HTML template. This can be done by passing the dictionary and the list of categories (the dictionary keys) into the template and use some loop tags in the template engine to display each item.

The updated menu function simply passes the list of categories as the keys from the dictionary, along with the dictionary itself.

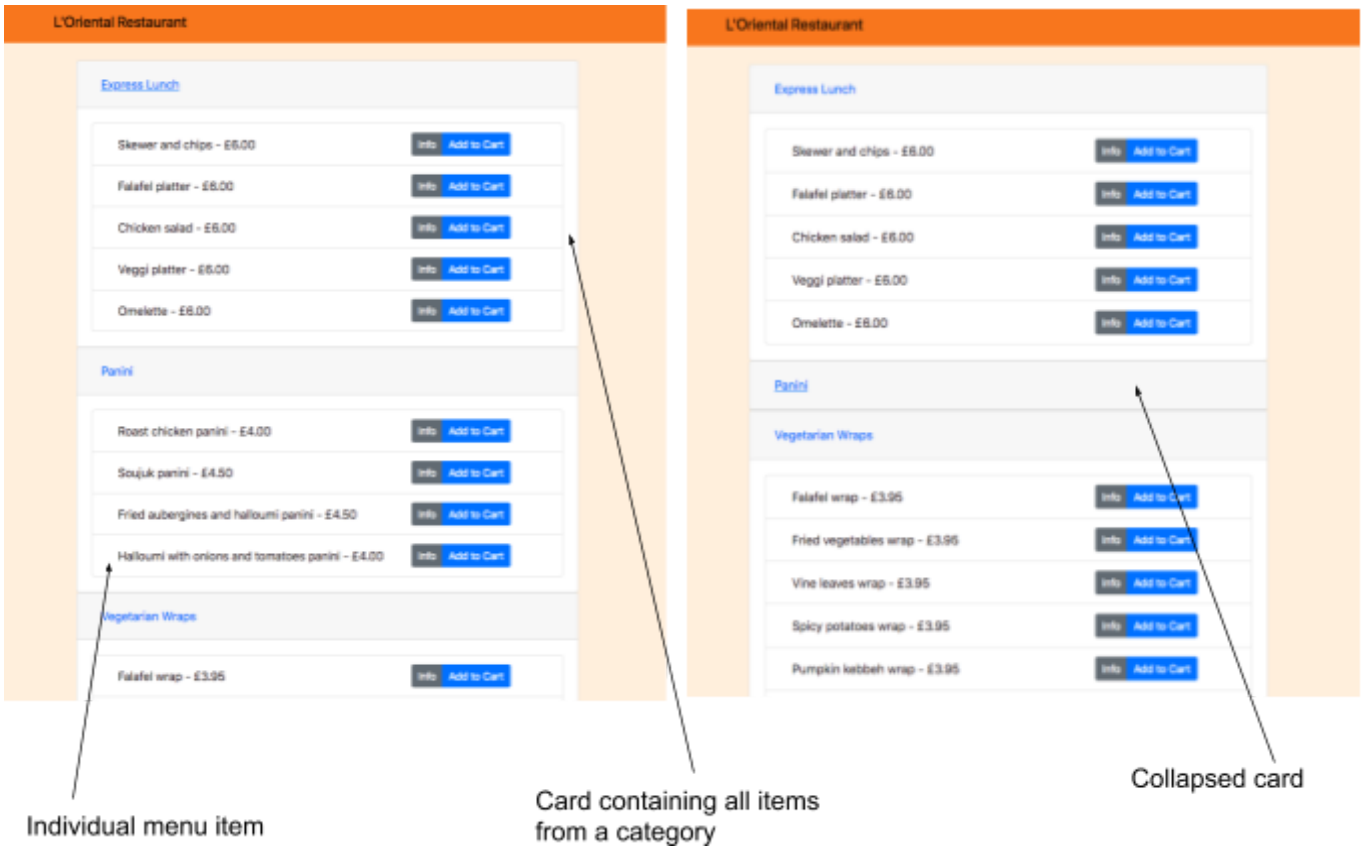
```
26 # The home (menu) page of the web app
27 @app.route('/')
28 def menu():
29     return render_template("menu.html", title="Menu", categories=FOODS.keys(), foods=FOODS)
```

The updated HTML code for menu.html is the following:

```
{% for category in categories %}
    <div class="card">
        <div class="card-header" id="{{ category|lower_strip }}">
            <a name="{{ category|lower_strip }}"></a>
            <h5 class="mb-0">
                <button class="btn btn-link collapsed" data-toggle="collapse" data-target="#collapse{{ category|lower_strip }}" aria-expanded="true" aria-controls="collapse{{ category|lower_strip }}">
                    {{ category }}
                </button>
            </h5>
        </div>
        <div id="collapse{{ category|lower_strip }}" class="collapse show" aria-labelledby="{{ category|lower_strip }}" data-parent="#accordion">
            <div class="card-body">
                <ul class="list-group">
                    {% for item in foods[category] %}
                        <li class="list-group-item d-flex justify-content-between align-items-center">

                            <div class="col-sm-8" style="text-align: left;">
                                {{ item[0] }} - £{{ item[2] }}
                            </div>

                            <div class="col-sm-4" style="text-align: right;">
                                <div class="input-group float-sm-right">
                                    <div class="btn-group float-sm-right" role="group">
                                        <button type="button" data-toggle="modal" data-target="#{{ item[0]|lower_strip }}Modal" class="btn btn-secondary btn-sm"> Info</button>
                                        <input type="submit" class="btn btn-primary btn-sm" name="{{ item[0] }}" value="Add to Cart">
                                    </div>
                                </div>
                            </div>

                        </li>
                    {% endfor %}
                </ul>
            </div>
        </div>
    </div>
{% endfor %}
```

A for loop tag is used to iterate through every category to render each card element, provided by Bootstrap. Within each card is the name of the category (which can be clicked to collapse the card, as shown previously) and another for loop to iterate through each menu item in the current category from the dictionary. Each menu item then shows the name of the food, the price, and two buttons: an info button and an Add to Cart button, which will be used later to add items to the cart.

The code above produces the following output:



To provide a better user experience, I shall be adding extra information to each food item (if applicable), which the user can read by clicking the “Info” button next to each item. To achieve this, I shall be using another element made available by Bootstrap, called modals. These are simply popups which contain text, and can be triggered when a HTML button is pressed.

Each food item will have its own modal, and therefore, it will be more efficient to dynamically produce each modal rather than hard-coding each of the 61 information modals. To do this, I have created a HTML file called food_modals.html which will then be loaded into the menu.html template.

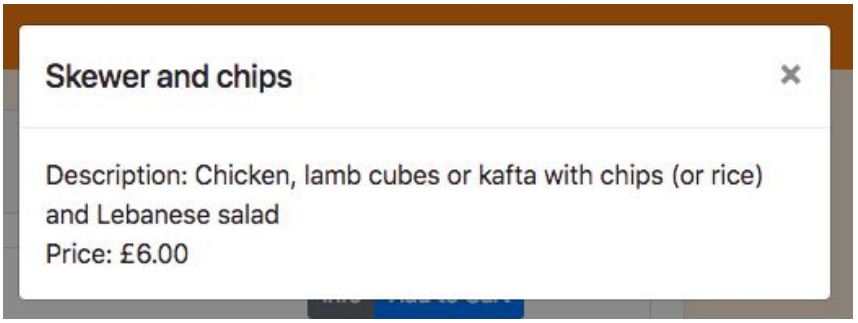
```
{% block modals %}
        <!-- Food Info Modals -->
        {% for category in categories %}
            {% for item in foods[category] %}
                <div class="modal fade" id="{{ item[0]|lower_strip }}Modal" tabindex="-1" role="dialog"
                     aria-labelledby="{{ item[0]|lower_strip }}ModalLabel" aria-hidden="true">
                    <div class="modal-dialog" role="document">
                        <div class="modal-content">
                            <div class="modal-header">
                                <h5 class="modal-title" id="{{ item[0]|lower_strip }}ModalLabel">{{ item[0] }}</h5>
                                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                                    <span aria-hidden="true">&times;</span>
                                </button>
                            </div>
                            <div class="modal-body">
                                {% if item[1] == "" %}
                                    No further description available.
                                {% else %}
                                    Description: {{ item[1] }}
                                {% endif %}
                                <br>
                                Price: £{{ item[2] }}
                            </div>
                        </div>
                    </div>
                </div>
            {% endfor %}
        {% endfor %}
{% endblock %}
```

Each modal will show the name of the food item as the title, and it will provide a further description if it exists in the CSV file. If the extra description column for a food item is empty in the CSV file, it will simply show “No further description available”. Finally, the price is shown, once again.

To include this code in the menu template, I added this to the end of menu.html:

```
101     <!-- Include the modals from the food_modals.html file -->
102     {% include 'food_modals.html' %}
```

When clicking the “Info” button for the Skewer and chips item, the following is shown on screen:



The Cart system

To add, remove and store food items temporarily, a cart system must be built. For this solution, I have decided to use web sessions. A web session is a small server-side storage designed to persist throughout the user's interaction with a web application. In Flask, web sessions can be created and modified very easily. Flask, by default, creates a global variable called session which is a dictionary. Therefore, each element in session can be identified using an identifier (the key).

Modeling the cart as a class would be an ideal situation however objects cannot be directly stored in web sessions. To create a class-based cart system, there would be a lot of serialisation and deserialisation of data, which would make the code repetitive and inefficient. Therefore, I have decided to *avoid an OOP approach* in this particular case.

The cart object will simply consist of a list of tuples, each tuple representing an item. The tuples will be structured in the following manner: `(item_name, quantity, price)`.

A few functions will be required to manage the cart:

- Check if a new item is already in the cart
- Add an existing item to cart
- Calculate the total price of the cart

Once again, to make the structure of the solution neat and easy to follow, I have created a cart.py file which will hold all the code related to cart management. The implementation of the three functions is shown below:

```
# updates the quantity of an item
def add_existing(food_name, quantity, sesh):
    for idx, item in enumerate(sesh['cart']):
        if food_name == item[0]:
            sesh['cart'][idx] = (item[0], str(int(item[1]) + int(quantity)), item[2])

# checks if an added item is already in the cart, works along add_existing()
def check_existing(food_name, sesh):
    for item in sesh['cart']:
        if food_name == item[0]:
            return True
    return False

# calculates total price from items in cart
def calculate_total(sesh):
    total = 0
    for item in sesh['cart']:
        total = total + float(item[2]) * float(item[1])   # multiplies price of item by its quantity
    return total
```

The `add_existing()` method takes in three parameters: the food name, the quantity and the current session. It uses the built-in `enumerate()` function to iterate through the session dictionary and checks for the food name in the cart. It then simply replaces the tuple with an updated one, containing the new quantity.

The `check_existing()` function takes in three parameters: the food name and the current session. It simply returns true or false based on whether the food item exists or not in the cart. This function will be used alongside `add_existing()`.

The `calculate_total()` function simply loops through the cart and calculates the total price by multiplying the item quantity by its price, and adding it to the total variable.
Finally, to add an item to the cart, I have created a new route function called `add_item()` which handles a POST request created when clicking the “Add to Cart” button next to each item in the menu page.
To submit POST requests, I create a form HTML element in each menu item. In addition,m I must also add an input box so the user can specify the desired quantity (by default, it will be 1). Therefore, the updated HTML code looks as shown below:

```
22      <div class="card-body">
23          <ul class="list-group">
24              {% for item in foods[category] %}
25                  <li class="list-group-item d-flex justify-content-between align-items-center">
26
27                      <div class="col-sm-8" style="text-align: left;">
28                          {{ item[0] }} - £{{ item[2] }}
29                      </div>
30
31                      <div class="col-sm-4" style="text-align: right;">
32                          <div class="input-group float-sm-right">
33                              <form action="{{ url_for("add_item") }}" method="POST">
34                                  <div class="btn-group float-sm-right" role="group">
35                                      <button type="button" data-toggle="modal" data-target="#{{ item[0]|lower_strip }}Modal" class="btn btn-secondary btn-sm"> Info</button>
36                                      <input type="submit" class="btn btn-primary btn-sm" name="{{ item[0] }}" value="Add to Cart">
37                                  </div>
38                                  <input type="number" name="quantity" class="form-control form-control-sm" style="max-width: 40px;" min="1" value="1">
39                                  <input name="food" value="{{ item[0] }}" hidden>
40                              </form>
41                          </div>
42                      </div>
43
44                  </li>
45              {% endfor %}
46          </ul>
47      </div>
```

As seen in line 33, a form element is created which sends a POST request to the `add_item()` view. An extra input box called “quantity” has been added for the user to specify the quantity, as well as a hidden item which simply contains the name of the food. This will be useful when processing the form, in order to determine what food item was added.

Showing the cart status on the sidebar

To show the contents of the cart on the right sidebar, we must pass the cart object and the total price of the cart into the menu template. Then, this can be dynamically rendered onto the template using some if statement and for loop tags.

To do this, we get the cart object from the session and calculate the total price. Then, these two variables are passed into the template:

```
6  # The home (menu) page of the web app
7  @app.route('/')
8  def menu():
9      cart = session['cart']
10     total = calculate_total(session)
11     return render_template("menu.html", title="Menu", categories=FOODS.keys(), foods=FOODS, total=total, cart=cart)
12
```

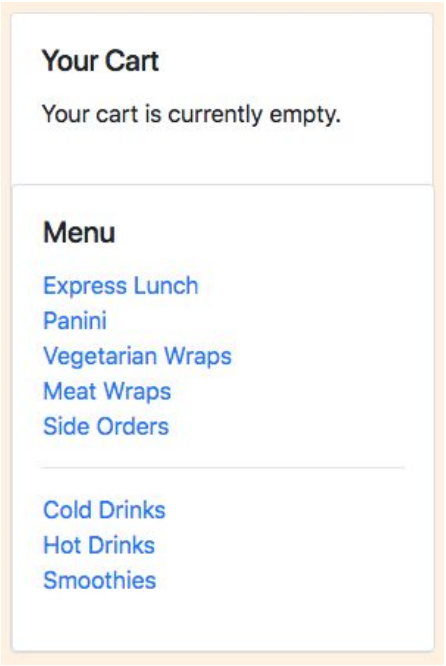
To display this on the menu template, I have updated the HTML code from the top part of the sidebar to the following:

```
56  <div class="col-sm-4" style="position: fixed; right:0;">
57          <div class="card" style="width: 18rem;">
58              <div class="card-body">
59                  <h5 class="card-title">Your Cart</h5>
60                  <div class="card-text" >
61                      {% if total != 0 %}
62                          <ul style="padding-left:8%;">
63                              {% for item in cart %}
64                              <li>
65                                  {{ item[0] }} {% if item[1] != "1" %}(x{{ item[1] }}){% endif %}
66                              </li>
67                              {% endfor %}
68                          </ul>
69                          <p><b>Total: £{{ total|price_format }}</b></p>
70                      {% else %}
71                          <p>Your cart is currently empty.</p>
72                      {% endif %}
73                  </div>
74                  {% if total != 0 %}
75                  <a href="{{ url_for("checkout") }}" role="button" class="btn btn-primary btn-lg btn-block">
76                      Checkout
77                  </a>
78                  {% endif %}
79          </div>
80
```

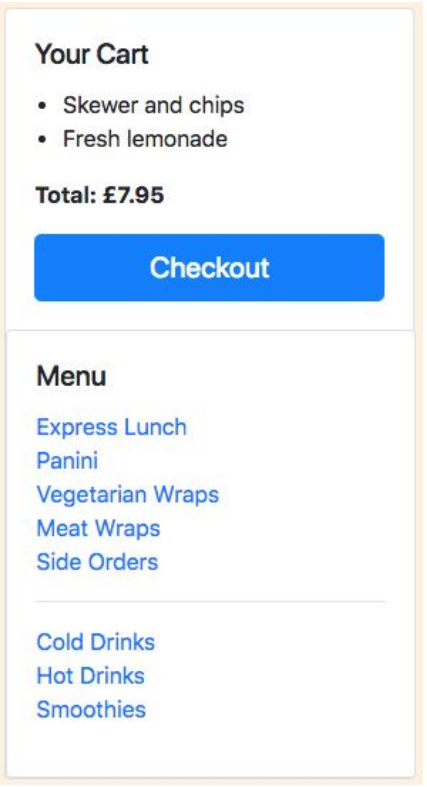
It first checks whether or not the the cart is empty by checking the price. If the total price is zero, then the cart must be empty (since there are no free items on the menu). If it is not zero, each item is shown in a list item element, including its quantity if it's not one. It then also shows the total price of the order, which uses a template filter called `price_format` which shows the price as a float with two decimal places. Otherwise, it tells the user that their cart is empty. A checkout button is shown if the cart is not empty too, which will redirect the user to the checkout page, implemented later.

The cart status display works correctly with that code, and below are four possible outputs:

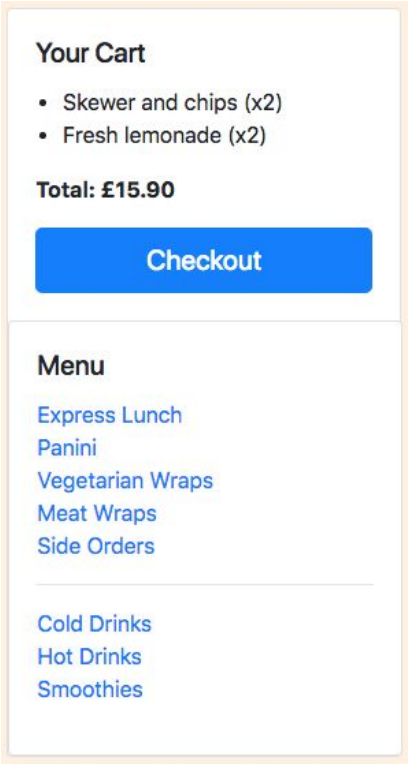
No items in the cart:



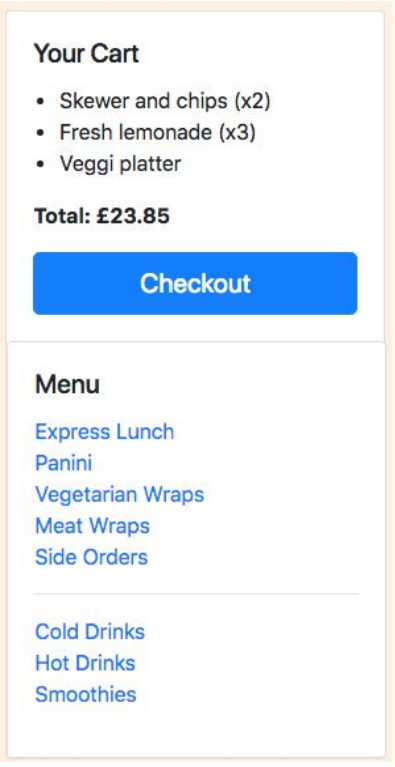
Item(s) which have a quantity of 1:



Item(s) which have a quantity greater than 1:



Items of different quantities:



Fixing the non-existent session error

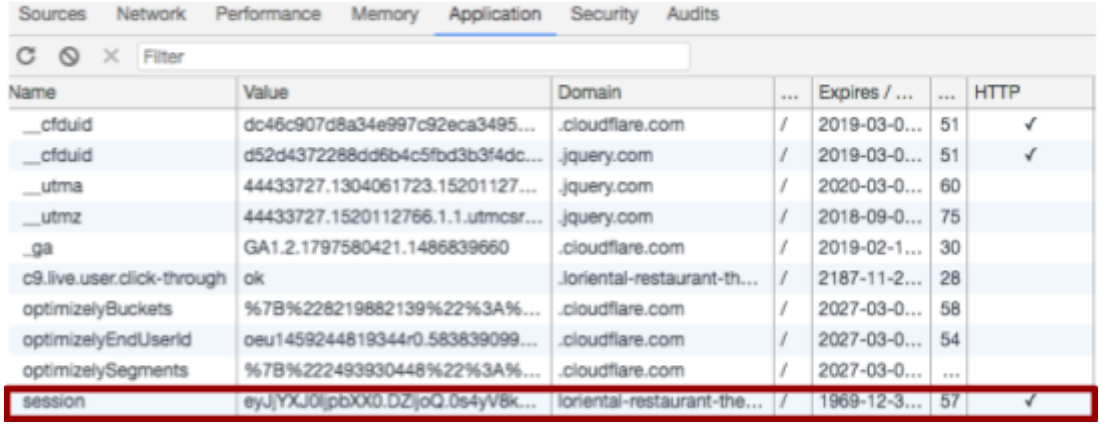
When running the server and viewing the menu page, an error occurs, resulting in an Internal Server Error (HTTP code 500).

It is a Python KeyError, which occurs when a dictionary is accessed with an invalid or missing key. In this case, the key is “cart” for the session dictionary, however it does not exist. To fix this, I used a try-except block, a construct which attempts a block of code (the try block) but if an error is produced, another block of code is executed (the except block).

```
26 # The home (menu) page of the web app
27 @app.route('/')
28 def menu():
29     try:
30         cart = session['cart']            # try to get cart from the existing session and calculate total
31         total = calculate_total(session)
32     except:                               # the except block will run if the cart does not exist
33         session['cart'] = []              # therefore, create an empty cart and set the total to 0
34         cart = session['cart']
35         total = 0.0
36     return render_template("menu.html", title="Menu", categories=FOODS.keys(), foods=FOODS, total=total, cart=cart)
```

In the try block, the code remains unchanged: the cart object is retrieved from the session dictionary and the total is calculated using the session. If an error is raised (i.e.: the KeyError), an empty cart is created in the session dictionary and the total is set to 0.

To test whether this works correctly, the session object can be deleted manually from the browser. Upon loading the website for the first time after the server is run, a session is produced, highlighted below.



Name	Value	Domain	...	Expires / ...	...	HTTP
__cfduid	dc46c907d8a34e997c92eca3495...	.cloudflare.com	/	2019-03-0...	51	✓
__cfduid	d52d4372288dd6b4c5fbd3b3f4dc...	.jquery.com	/	2019-03-0...	51	✓
__utma	44433727.1304061723.15201127...	.jquery.com	/	2020-03-0...	60	
__utmz	44433727.1520112766.1.1.utmcsr...	.jquery.com	/	2018-09-0...	75	
_ga	GA1.2.1797580421.1486839660	.cloudflare.com	/	2019-02-1...	30	
c9.live.user.click-through	ok	.loriental-restaurant-th...	/	2187-11-2...	28	
optimizelyBuckets	%7B%228219882139%22%3A%...	.cloudflare.com	/	2027-03-0...	58	
optimizelyEndUserId	oeu1459244819344r0.583839099...	.cloudflare.com	/	2027-03-0...	54	
optimizelySegments	%7B%222493930448%22%3A%...	.cloudflare.com	/	2027-03-0...	...	
session	eyJjYXJ0IjpbXX0.DZjoQ.0s4yV8k...	loriental-restaurant-the...	/	1969-12-3...	57	✓

Once deleted, I refresh the page and another session is produced, meaning the new code is working as expected.

Name	Value	Domain	...	Expires / ...	...	HTTP
__cfduid	dc46c907d8a34e997c92eca3495...	.cloudflare.com	/	2019-03-0...	51	✓
__cfduid	d52d4372288dd6b4c5fbd3b3f4dc...	.jquery.com	/	2019-03-0...	51	✓
__utma	44433727.1304061723.15201127...	.jquery.com	/	2020-03-0...	60	
__utmz	44433727.1520112766.1.1.utmcsr...	.jquery.com	/	2018-09-0...	75	
_ga	GA1.2.1797580421.1486839660	.cloudflare.com	/	2019-02-1...	30	
c9.live.user.click-through	ok	.loriental-restaurant-th...	/	2187-11-2...	28	
optimizelyBuckets	%7B%228219882139%22%3A%...	.cloudflare.com	/	2027-03-0...	58	
optimizelyEndUserId	oeu1459244819344r0.583839099...	.cloudflare.com	/	2027-03-0...	54	
optimizelySegments	%7B%222493930448%22%3A%...	.cloudflare.com	/	2027-03-0...	...	
session	eyJjYXJ0IjpbXX0.DZlk6Q.Wsw8e...	loriental-restaurant-the...	/	1969-12-3...	57	✓

Removing items from the cart

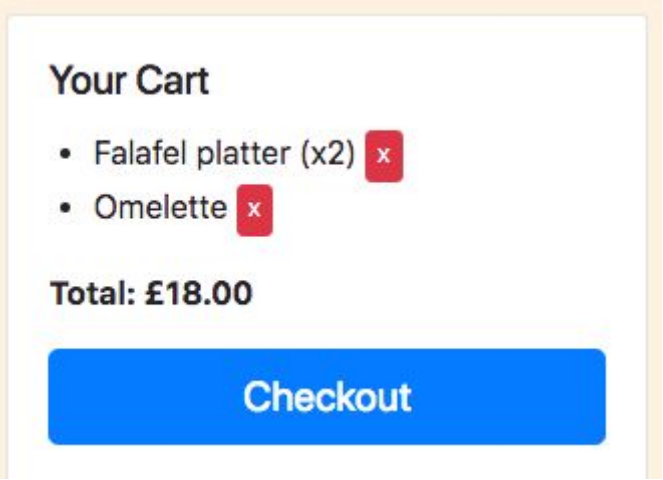
Currently, there is no way of removing an item from the cart. This feature is essential in order to provide a better user experience. For example, the customer may change their mind on a certain food, or simply clicked the wrong food.

To remove an item from the cart, I have added a Remove button next to the food item in the cart status display, on the sidebar. The user should be able to click this and the food item (of any quantity) will be deleted from the cart.

```
59      <h5 class="card-title">Your Cart</h5>
60      <div class="card-text" >
61          {% if total != 0 %}
62              <ul style="padding-left:8%;">
63                  {% for item in cart %}
64                  <li>
65                      <form action="{{ url_for("remove_item") }}" method="POST">
66                          {{ item[0] }} {% if item[1] != "1" %}(x{{ item[1] }}){% endif %}
67                          <input type="submit" class="btn btn-danger btn-sm" value="X" style="font-size: 60%; padding: 5px;">
68                          <input name="food" value="{{ item[0] }}" hidden>
69                      </form>
70                  </li>
71                  {% endfor %}
72              </ul>
73              <p><b>Total: £{{ total|price_format }}</b></p>
```

The code above shows the updated template code for the menu page. Inside the for loop when iterating through the items in the cart, a form is created for each list item. The same text is shown, however a red button containing an “X” is shown next to each item, representing the remove button. It has some extra CSS styling, as specified in the style attribute. When the button is clicked, the form will be sent to the remove_item function, implemented next.

The new code produces the following output:



An “X” button is added next to each item in the <li> tag

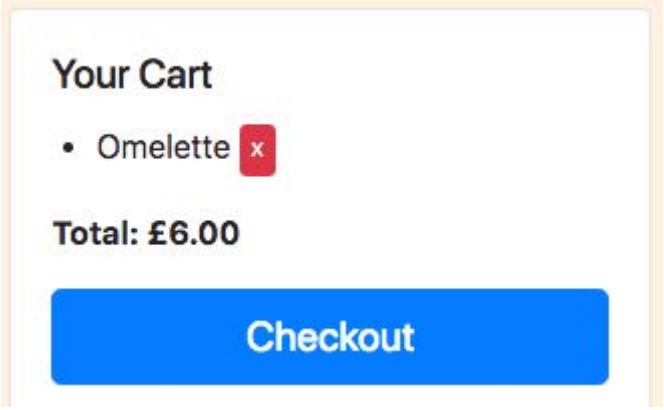
To process the deletion of the item, a new view function must be created, of similar structure to the `add_item()` view that currently exists. It will be called `remove_item()`.

```
59  # Handles the "Remove" button click for the items in the cart
60  @app.route('/remove', methods=['POST'])
61  def remove_item():
62
63      item_name = request.form['food']        # get the item name from the form
64      print(item_name)
65      try:
66          remove_food(item_name, session)     # remove the item from the cart
67          session.modified = True
68      except:
69          session['cart'] = []
70
71      return redirect(url_for("menu"))   # redirect to menu page
```

It is a POST-only view, since it only processes a form. It retrieves the food item to delete, and attempts to remove it. Once again, the try-except block is used to avoid any non-existent sessions producing errors. The item is removed, and the customer is redirected to the menu page.
The method `remove_food()` on line 66 is another cart-related procedure, so I have defined it in the cart.py file.

```
22  # removes an item from the cart
23  def remove_food(food_name, sesh):
24      for idx, item in enumerate(sesh['cart']):
25          if food_name == item[0]:
26              del sesh['cart'][idx]              # delete item from dictionary
27              return                             # stop
```

For example, if the red X button is clicked for the “Falafel platter (x2)” list item (shown previously in figure 3.x.), the customer is redirected to the menu and the sidebar is updated to show the following:



Creating the checkout page

Once the customer has chosen all the items for their order, they must input their details so that they can pick up the order at the restaurant. To gather these details, I shall create a checkout page which will consist of a form. The form will allow the following details to be inputted:

- Name and surname
- Contact number
- Takeaway or eating in

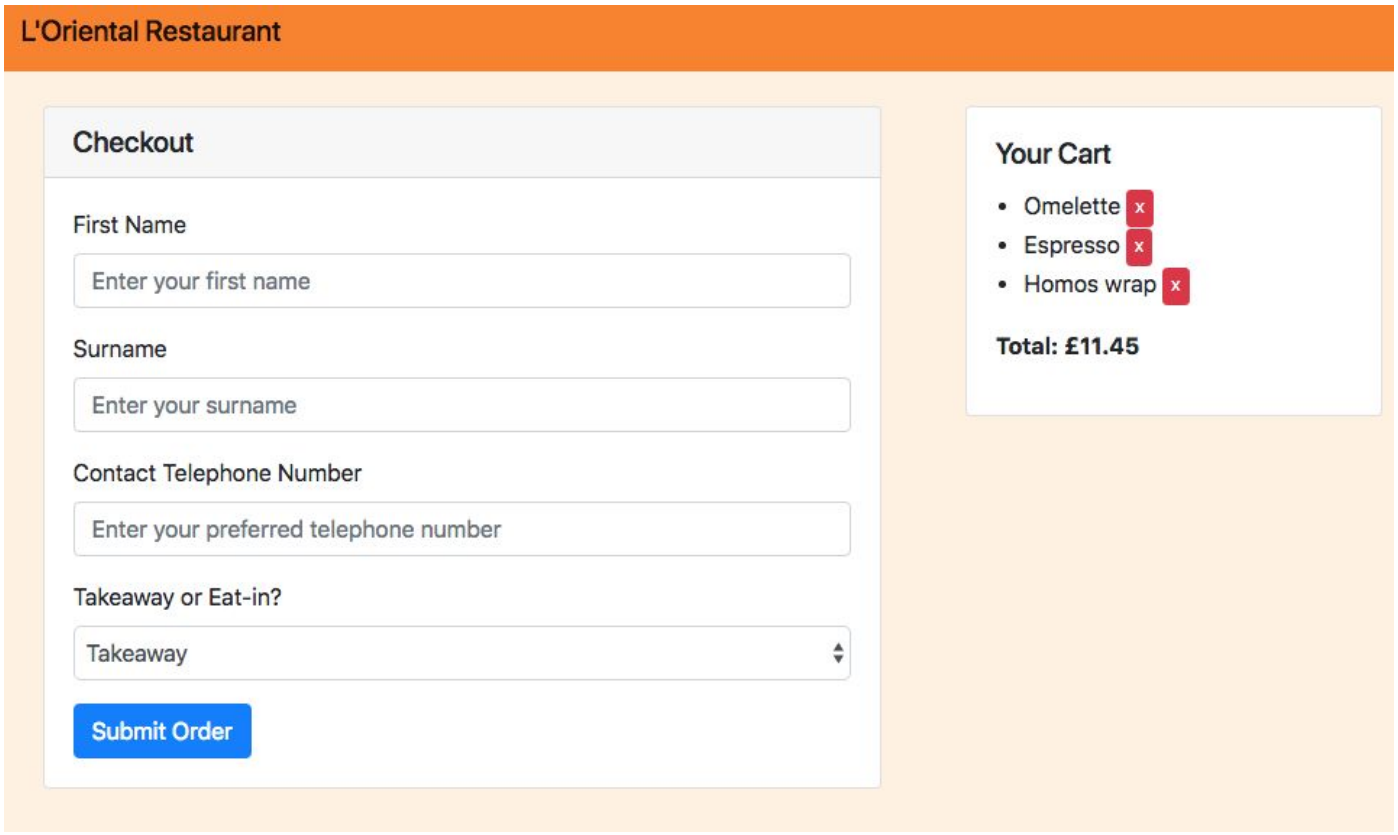
As all templates in the project, the checkout.html file will inherit from base.html using the `{% extends "base.html" %}`. The HTML code to create the form is shown below:

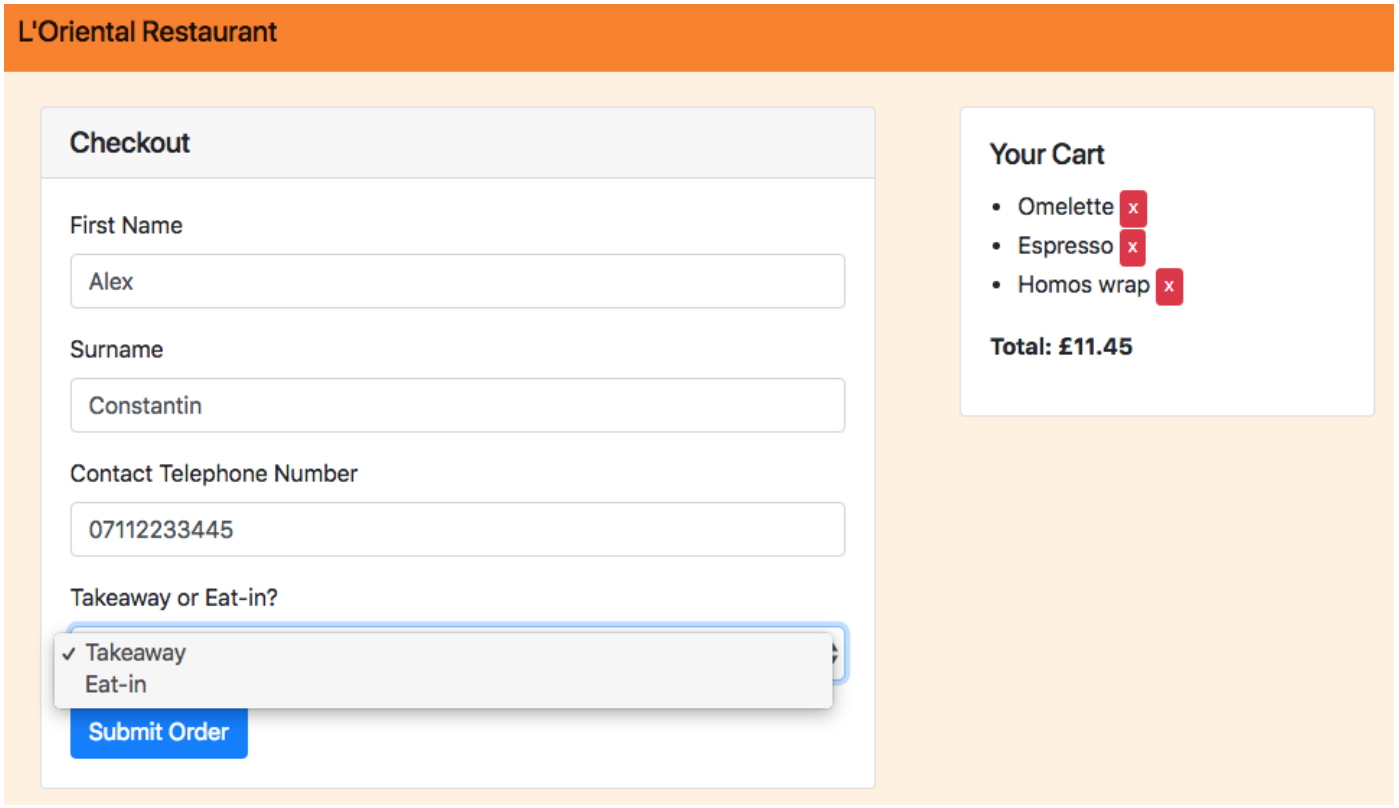
```
<div class="card-body">
    <form action="{{ url_for("process_checkout") }}" method="POST">
        <div class="form-group">
            <label for="name">First Name</label>
            <input type="text" class="form-control" name="firstName" placeholder="Enter your first name" required>
        </div>
        <div class="form-group">
            <label for="name">Surname</label>
            <input type="text" class="form-control" name="surname" placeholder="Enter your surname" required>
        </div>
        <div class="form-group">
            <label for="contactNumber">Contact Telephone Number</label>
            <input type="tel" class="form-control" name="contactNumber"
                   placeholder="Enter your preferred telephone number" required>
        </div>
        <div class="form-group">
            <label for="categorySelect">Takeaway or Eat-in?</label>
            <select class="form-control" name="category" id="categorySelect" required>
                <option>Takeaway</option>
                <option>Eat-in</option>
            </select>
        </div>
        <button type="submit" class="btn btn-primary">Submit Order</button>
    </form>
</div>
</div>
```

Inside a card element, a form is created which will be processed by the `process_checkout()` view. The form contains fields for the customer's first name, surname, contact number and a dropdown selection box for whether the customer would like the order as a takeaway or eat in. The required attribute at the end of each <input> tag ensures the customer types something into the input. This completes the first part of the validation. The rest of the validation algorithm will be fulfilled server-side.
The form can then be submitted by clicking the "Submit Order" button.

In addition to the form, the sidebar containing the cart status will also be displayed, in case the customer decides to remove an item. The code for this is exactly the same as in the menu template, however, without the checkout button and the links to the different parts of the menu.

All together, the checkout template produces the following output:





The view function for the checkout page is shown below.

```
74  # The checkout page of the website
75  @app.route('/checkout')
76  def checkout():
77      try:                            # try block to check for session existing
78          cart = session['cart']
79          # if the cart is empty, checkout is not allowed, so redirect to menu
80          if cart == []:
81              return redirect(url_for("menu"))
82
83          total = calculate_total(session)
84          return render_template("checkout.html", title="Checkout", total=total, cart=cart)
85      except:
86          return redirect(url_for("menu"))   # redirect to menu page
```

If the cart is empty, the checkout button does not appear on the sidebar of the menu page. However, this does not stop users from accessing the /checkout path. Therefore, another check is made to verify whether the cart is actually empty or not. This is done on line 80. If it is empty, the user is redirected back to the menu page.
If everything is valid, the checkout.html file is rendered with the total cost and the cart object as parameters to display the cart status on the sidebar. Otherwise, the user is redirected to the menu page.

Processing the order and the confirmation page
When the customer submits the form on the checkout by clicking on the Submit Order button, the form data must be validated, committed to a database and a confirmation of success should be returned to the user.
As defined in the template HTML code, the view function to process the form shall be called `process_checkout()`. The code for it is shown below:

```
89  # Handles the checkout once the form is submitted
90  @app.route('/process-checkout', methods=['POST'])
91  def process_checkout():
92      cart = session['cart']
93      if session['cart'] != []:    # if cart is not empty, proceed, otherwise, redirect back to menu
94          session['cart'] = []     # reset the cart
95
96          custFirstName = request.form['firstName']        # get all the customer's data from the form
97          custSurname = request.form['surname']
98          custTelephone = request.form['contactNumber']
99          category = request.form['category']
100
101         print("--- NEW ORDER ---")
102         print(custFirstName + " " + custSurname)
103         print("Telephone no.: " + custTelephone)
104         print("Type: " + category)
105
106         print(cart)
107
108         return render_template('confirmation.html', title='Order Confirmed') # show confirmation page
109     else:
110         return redirect(url_for("menu"))
```

As always, at the beginning of the view, the cart is checked. If it is empty, there is nothing to process, so the user is redirected to the menu page. Otherwise, the cart is reset and the details are retrieved from the form. For now, these details are simply printed on the server-side console. Then, a confirmation page called confirmation.html is rendered.

The HTML code for confirmation.html is very straightforward, since it simply shows a confirmation message, shown below.

```
<!-- Load base template from base.html -->
{% extends "base.html" %}

{% block content %}
    <div class="row">
        <div class="col">
            <div class="card" width="100%">
                <div class="card-body">
                    <h2 class="card-title">Your order has been confirmed</h2>
                    <p class="card-text">
                        Thank you for submitting an order. Please arrive shortly to the restaurant to collect your order.
                    </p>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

To test the functionality of the current website, I have added the following items to the cart:

- Chicken salad, quantity = 2
- Fresh lemonade, quantity = 1

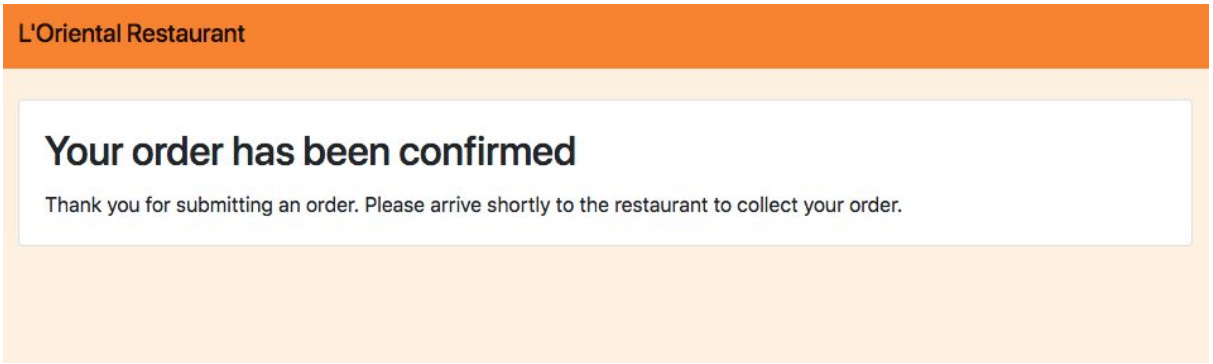
For the checkout form, I have added the following details:

- Alex
- Constantin
- 07712345678
- Takeaway

When clicking the checkout button, the following is shown on the server console:

```
10.240.0.187 - - [26/Mar/2018 09:33:42] "GET /checkout HTTP/1.1" 200 -
--- NEW ORDER ---
Alex Constantin
Telephone no.: 07712345678
Type: Takeaway
[('Chicken salad', '2', '6.00'), ('Fresh lemonade', '1', '1.95')]
10.240.1.220 - - [26/Mar/2018 09:34:37] "POST /process-checkout HTTP/1.1" 200 -
```

On the user side, the customer is redirected to the confirmation page, which looks like this:



Validating the form

To make sure the input data is correct, it must undergo a validation algorithm, based on the pseudocode defined in the Design section of the project.

The code to validate the data has been written as a function, similar to the pseudocode:

```
130 # validate form and return any errors
131 def validate_form(name, surname, telephone):
132
133     errors = []
134
135     if len(name) > 30 or len(name.replace(" ", "")) == 0:
136         errors.append("Invalid First Name")
137
138     if len(surname) > 35 or len(surname.replace(" ", "")) == 0:
139         errors.append("Invalid Surname")
140
141     if not re.match("^((\+44\s?\d{4}|\(?\d{5}\)?)\s?\d{6}|((\+44\s?|0)7\d{3}\s?\d{6})$", telephone):
142         errors.append("Invalid Telephone Number")
143
144     return errors
```

For the first name and surname, the length of the string is checked to be less than 30 and 35 respectively but also not 0. Any whitespace is removed using the `replace()` function, since whitespace counts as characters.
For the telephone number, it is validated using regular expressions, which have been explained in the Design section. It uses the built-in *regex* module, included using `import re`.
For the category, there are only two possible options which can be chosen using a dropdown box, so validation is not required for this field.

To provide feedback to the user, any errors are added to a list and returned back, so that they can be displayed on the checkout form. To do this, I have updated the bottom of the `process_checkout()` view function:

```
110         print("--- NEW ORDER ---")
111         print(custFirstName + " " + custSurname)
112         print("Telephone no.: " + custTelephone)
113         print("Type: " + category)
114         print(cart)
115
116         errors = validate_form(custFirstName, custSurname, custTelephone)
117         if not errors:
118             return render_template('confirmation.html', title='Order Confirmed') # show confirmation page
119         else:
120             for e in errors:
121                 flash(e)
122             session['cart'] = cart   # get current cart and reload it into session
123             return redirect(url_for("checkout"))
124
125     else:
126         return redirect(url_for("menu"))
```

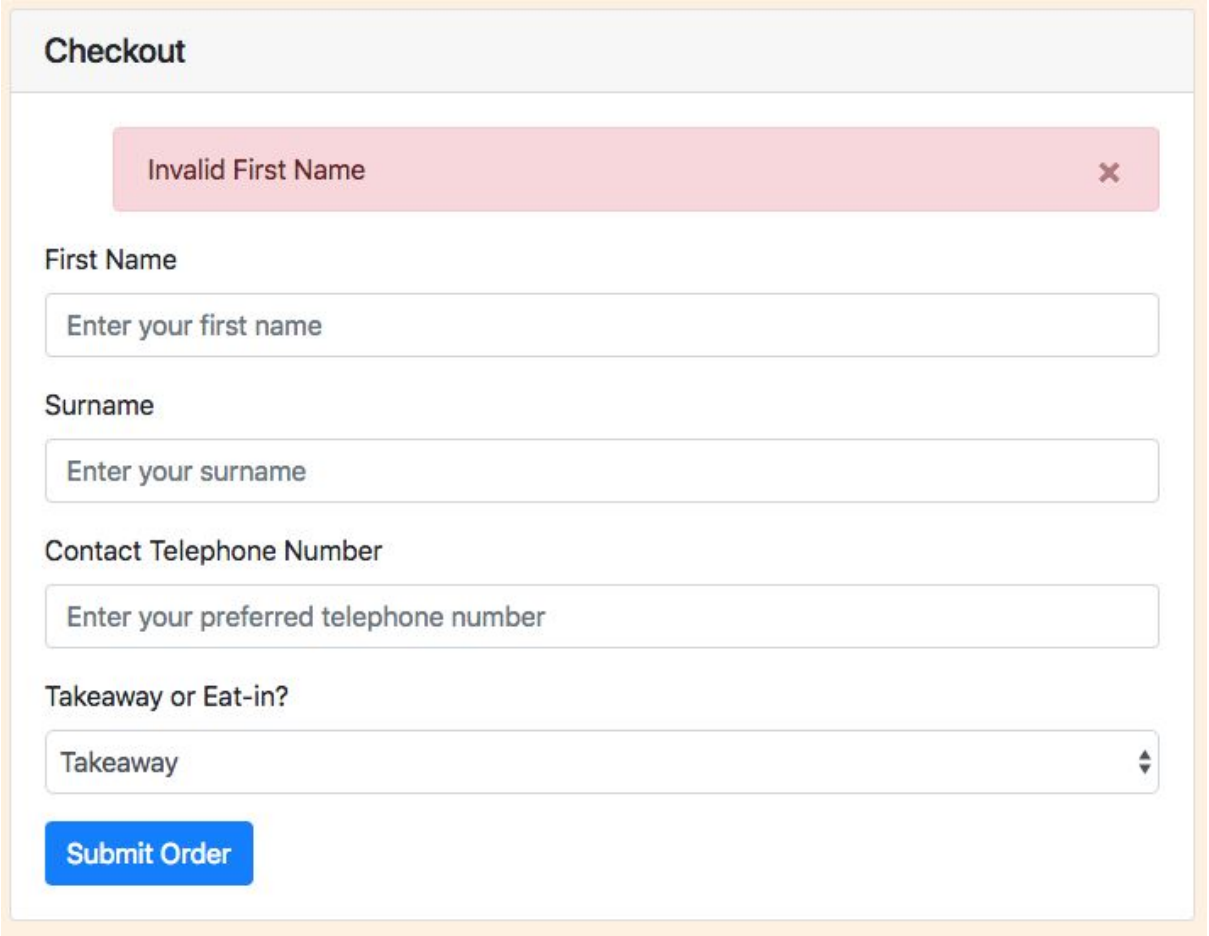
The list of errors from the `validate_form()` function is returned using the form data as parameters. If there are no errors (checked on line 117), the confirmation page is shown. If there are errors, each error is “flashed” back to the checkout page.
The flash function can be imported from the Flask library. The previous cart is also reloaded into the session, since it was set to be empty at the beginning of the view function.

Flask has a message “flashing” system by which any messages can be pushed into a queue and then displayed in any template, since it exists at a global scope. Therefore, the following template code can be added to checkout.html, just before the form:

```
{% with errors = get_flashed_messages() %}
  {% if errors %}
    <ul class=flashes>
    {% for e in errors %}
      <div class="alert alert-danger alert-dismissible fade show" role="alert">
          {{ e }}
          <button type="button" class="close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
          </button>
      </div>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

This simply loops through every error message that has been added to the “flash” queue and displays it in a red alert box, which can be dismissed by clicking a small “x” button. Once it has been displayed, it is also popped from the queue.

Now, purposely adding a very long name (length > 30) redirects the user back to the checkout page and shows the following error:



Test cases for the validation algorithm

Below is a table showing the test cases for valid, boundary and invalid data to be inputted in the checkout form and passed into the validation algorithm:

	First Name	Surname	Telephone Number
Valid	Alex John Natalie	Constantin Appleseed Smith	07112233445 +442073135800 02073135800
Boundary	A myverylongnameofexactly30chars	C myverylongsurnamewithexactly35chars	*(Either valid or invalid)*
Invalid	___ (whitespace) myverylongnamewithmorethan30chars	___ (whitespace) myverylongsurnamewithmorethan35chars	999 612345678 3453fsffmi435

After inputting all this data in their relevant field in the checkout form, the appropriate and expected response is produced, meaning the algorithm works correctly and the data is ready to be inserted into a database.

For all database related functionality, I have created a directory called /db, in order to maintain the a modular and clear structure.
To create the database file, the following script (setup.py) should be executed only once:

```
1  import sqlite3
2
3  conn = sqlite3.connect('database.db')  # creates the file for the database
4
5  # SQL queries
6  create_customers = "CREATE TABLE customers (id INTEGER PRIMARY KEY, firstName
7  create_orders = "CREATE TABLE orders (id INTEGER PRIMARY KEY, timestamp NUMERI
8
9
10 # create the customers and orders table in the database
11 conn.execute(create_customers)
12 conn.execute(create_orders)
13 conn.close()
```

The SQL queries (not shown in full in the screenshot above) to create the tables are the following:

create_customers:

```
CREATE TABLE customers (
      id INTEGER PRIMARY KEY,
      firstName TEXT,
      surname TEXT,
      telephone VARCHAR(15)
)
```

create_orders:

```
CREATE TABLE orders (
      id INTEGER PRIMARY KEY,
      timestamp NUMERIC,
      contents BLOB,
      category TEXT,
      customerID INTEGER,
      FOREIGN KEY(customerId) REFERENCES customer(id)
)
```

The Customers table must be created first due to customers_id foreign key in the Orders table, in order for a valid initialisation.

Upon running the Python script, a file called database.db is created in the main directory. The database is ready to be used.

Database operations (CRUD)
For all database operations such as selecting and creating data, the functions will be placed in a new file called operations.py, inside the /db/ directory.

Add an order to the database:

```
INSERT INTO orders (id, timestamp, contents, category, customerId)
VALUES (order_id, order_timestamp, order_contents, order_category,
customer_id)
```

The associated Python function to the query above is `add_order()`.

Add a customer to the database:

```
INSERT INTO customers (id, firstName, surname, telephone)
VALUES (customer_id, cust_firstName, cust_surname, order_category,
customer_id)
```

The associated Python function to the query above is `add_order()`.

Read an order from the database:

```
SELECT * FROM orders WHERE id=order_id
```

The associated Python function to the query above is `add_order()`.

Read a customer from the database:

```
SELECT * FROM customers WHERE id=customer_id
```

The associated Python function to the query above is `add_order()`.

Generating a unique ID for the Order and Customer classes

To insert a new order or customer into the database using the functions written in the operations.py file, I have implemented very basic Order and Customer classes, which simply populate attributes with data passed into the constructor.

In addition, a unique ID must also be created for the order ID and the customer ID. The following code does so, located in the order.py file inside the /orders/ directory:

```
1  import time
2
3  # generates a unique ID from the current timestamp and takes 5 digits
4  def generate_unique_id():
5      return str(int(time.time()*10000000))[10:15]
6
7  class Order:
8
9      def __init__(self, orderContents, orderCategory, custID, orderID="", orderTime=0):
10         if not orderID:
11             self.id = generate_unique_id()
12         else:
13             self.id = orderID
14         if orderTime == 0:
15             self.timestamp = int(time.time())
16         else:
17             self.timestamp = orderTime
18
19         self.contents = orderContents
20         self.category = orderCategory
21         self.customerId = custID
```

The function to generate an ID on line 4 simply works by taking the current unix timestamp and taking characters 10 to 15, producing an ID of 5 characters in length.

The constructor for the Order class takes in the contents, the category and the customer ID as required parameters. The order ID is generated using the previous function and the timestamp is simply obtained using the `time.time()` function.

The implementation for the Customer class is shown below and has a very simple constructor:

```
4  class Customer:
5
6      def __init__(self, custName, custSurname, contactNumber, custID=""):
7          if not custID:
8              # add 255 to avoid clash with order id
9              self.id = str(int(generate_unique_id()) + 255)
10         else:
11             self.id = custID
12
13         self.firstName = custName
14         self.surname = custSurname
15         self.telephone = contactNumber
```

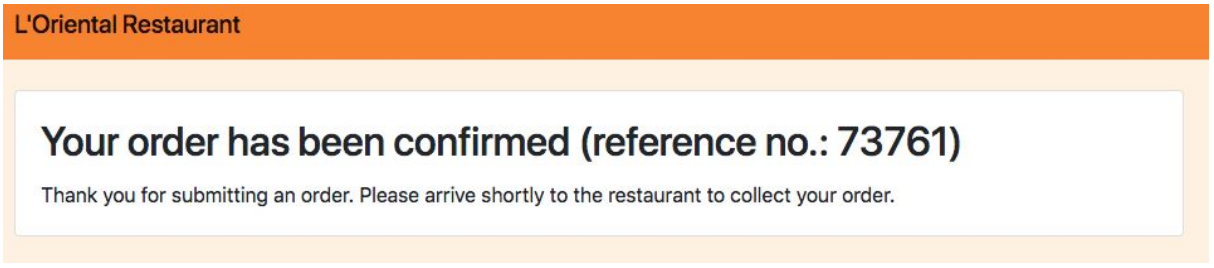
Adding new orders to the database after Checkout completion

Finally, to actually insert the new orders and customers into the database once the checkout form has been completed, the following code should be added to the `process_checkout()` view function:

```
91  # Handles the checkout once the form is submitted
92  @app.route('/process-checkout', methods=['POST'])
93  def process_checkout():
94
95      cart = session['cart']
96
97      if session['cart'] != []:    # if cart is not empty, proceed, otherwise, redirect back to menu
98          session['cart'] = []     # reset the cart
99          custFirstName = request.form['firstName']        # get all the customer's data from the form
100         custSurname = request.form['surname']
101         custTelephone = request.form['contactNumber']
102         category = request.form['category']
103
104
105         newCustomer = Customer(custFirstName, custSurname, custTelephone)    # create customer object from data
106         newOrder = Order(cart, category, newCustomer.id)                     # create order object
107
108         # commits the new order and customer to the database
109         add_order(newOrder)
110         add_customer(newCustomer)
111
112         errors = validate_form(custFirstName, custSurname, custTelephone)
113         if not errors:
114             return render_template('confirmation.html', title='Order Confirmed', order_id=newOrder.id)
115         else:
116             for e in errors:
117                 flash(e)
118             session['cart'] = cart   # get current cart and reload it into session
119             return redirect(url_for("checkout"))
120     else:
121         return redirect(url_for("menu"))
```

Lines 105 to 110 work by creating a new customer and order objects using the newly coded classes and then use the database operation functions add_order() and add_customer() to commit them to the database.

In addition, the order ID can be returned to the customer on the confirmation page, as some sort of reference number for when they arrive to the restaurant:



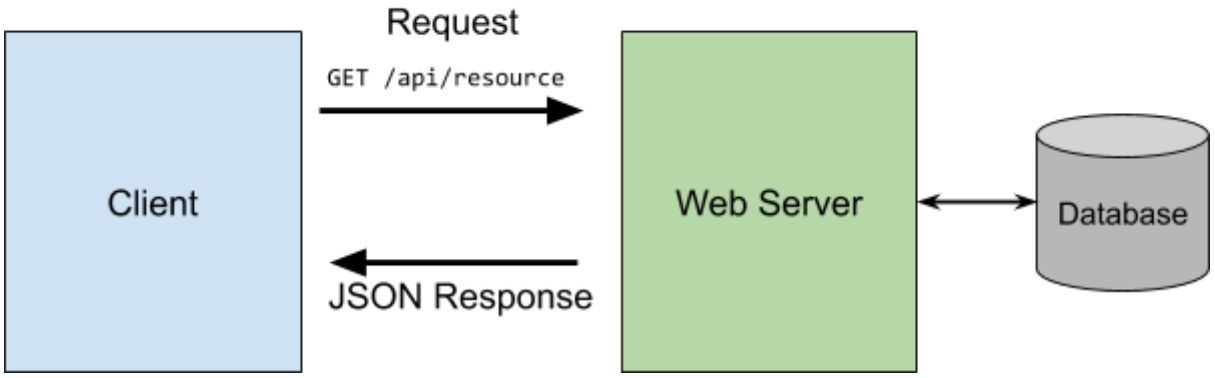
Sure enough, if the order ID is printed to the console, it matches the reference number, as expected:

```
10.240.0.87 - - [14/Apr/2018 16:44:17] "GET /checkout HTTP/1.1" 200 -
New order ID: 73761
10.240.1.0 - - [14/Apr/2018 16:44:19] "POST /process-checkout HTTP/1.1" 200 -
```

Building the GET API and serialising the data for transmission

In order to transfer the order data over the Internet so that it is received by the kitchen client, I shall be using a *RESTful* approach. In other words, I shall be creating a REST API. This architecture follows the client-server architecture: a client makes a request to a specific path of the API and the web server returns the relevant data in the form of JSON (JavaScript Object Notation), which is a standardised language through which data is commonly transferred.

This architecture is shown in the diagram below:



In this case, the kitchen client will act as the client, which requests data from the API on the web server, which is connected directly to the relational database.

The following paths will be available on the API:

`/orders/<order_id>`
This will return the JSON data for a specific order, given by the `order_id` parameter.

`/orders/all/<limit>`
This will return the JSON data for the first X number of orders in the database (given by the `limit` parameter), with the most recently submitted ones appearing first.

`/customers/<customer_id>`
This will return the JSON data for a specific customer, given by the `customer_id` parameter.

`/customers/all/<limit>`
This will return the JSON data for the first X number of customers in the database (given by the `limit` parameter).

It is important to limit the number of orders returned to the client since the database may contain many orders, making the transmission less efficient. Also, the older orders will be irrelevant to the kitchen client program when updating the list to show the most recent orders. Therefore, the limit parameter will be important.

To create these paths/routes, within the app.py script, I have added a section dedicated to the API paths. These are implemented just like ordinary views, except no HTML data will be rendered.

The starter code for now is shown below. These views simply return text, specified by the return statements:

```
148 ################ REST API CODE ####################
149
150 # return json data for order given its id
151 @app.route('/orders/<order_id>')
152 def get_order(order_id):
153     return 'Order ' + order_id
154
155
156 # return json data for all orders up to a certain number
157 @app.route('/orders/all/<limit>')
158 def get_orders(limit):
159     return 'The first ' + limit + ' orders'
160
161
162 # return json data for customer given its id
163 @app.route('/customers/<customer_id>')
164 def get_customer(customer_id):
165     return 'Customer ' + customer_id
166
167
168 # return json data for all customers up to a certain number
169 @app.route('/customers/all/<limit>')
170 def get_customers(limit):
171     return 'The first ' + limit + ' customers'
```

Accessing `/orders/31047` returns the following:

Order 31047

Accessing `/orders/all/5` returns the following:

The first 5 orders

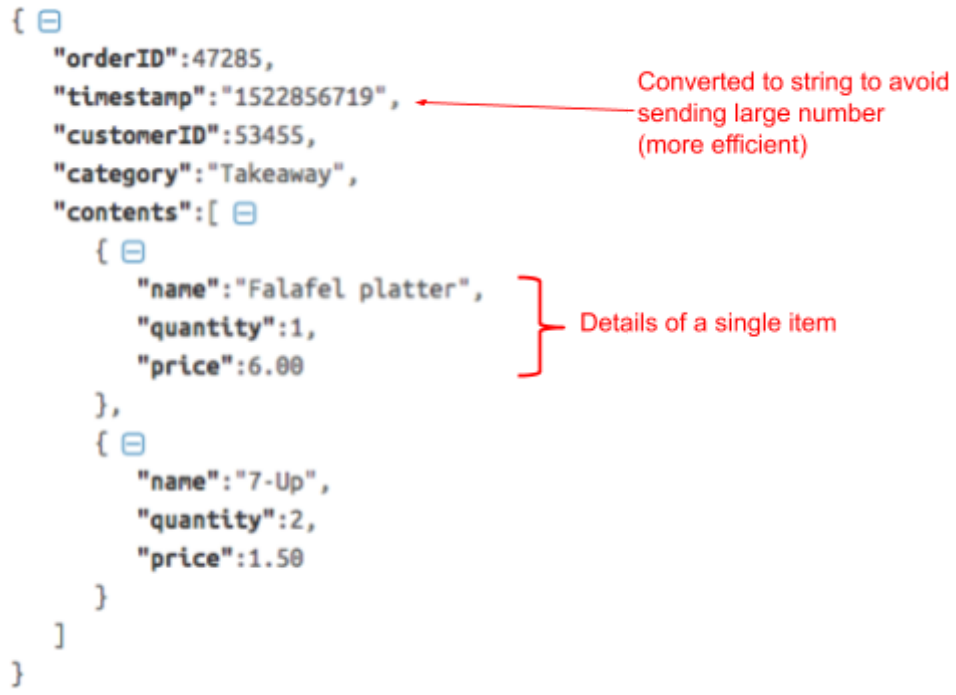
Accessing `/customers/43345` returns the following:

Customer 43345

Accessing `/customers/all/3` returns the following:

The first 3 customers

The objective of this section of the development is to produce a JSON version of an order so that it can be sent over the network. The end goal is to produce an output of the following structure:



The JSON above shows the full details of a single order, ready to be transmitted over the network. This, for example, would be the response when requesting the resource located at `/orders/47285`.

To convert data into JSON, it must first be converted into a dictionary data structure. The function to serialise a single order is shown below, implemented in a new file, serialisation.py in the /db directory.

```
30 def serialise_order(order):
31     so = {
32         "orderID": order[0],
33         "timestamp": str(order[1]),
34         "customerID": order[4],
35         "category": order[3],
36         "contents": []
37     }
38
39     contents = ast.literal_eval(order[2])   # parse text blob into list
40
41     for item in contents:
42         so["contents"].append( {"name": item[0], "quantity": item[1], "price": item[2]} )
43
44     return so
```

It works by creating a dictionary with all the required key-value pairings. For the actual items (i.e.: the content attribute), they are parsed from the text blob stored in the database and then appended to the list in the contents key in the dictionary.

Now, since the function serialises a single order, the `/orders/<order_id>` path can be updated with the following code:

```
179 # return json data for order given its id
180 @app.route('/orders/<order_id>')
181 def get_order(order_id):
182     try:
183         order = get_order_by_id(order_id)[0]
184         data = serialise_order(order)
185         return jsonify(data)
186     except:
187         return jsonify({"data": "Invalid request/error occured"})
```

As always, the code is encased by a try-except block to handle any unexpected errors smoothly. The order is obtained from the database using the `get_order_by_id()` function written previously. The order is passed into the `serialise_order()` function and the dictionary is returned. Finally, the dictionary is converted into JSON using the `jsonify()` function provided by Flask.

The code for the `/orders/all/<limit>` path is very similar. Rather than returning a single order, the `serialise_order()` function is applied to all the orders.

```
179 # return json data for all orders
180 @app.route('/orders/all/<limit>')
181 def get_orders(limit):
182     try:
183         limit = int(limit)                    # convert limit to numeric value
184         orders = get_all_orders(limit)
185         data = []
186         for o in orders:
187             data.append(serialise_order(o))
188         return jsonify(data)
189     except:
190         return jsonify({"data": "Invalid request/error occured"})
```

Firstly, the limit parameter is converted into an integer. This is where any value which isn't an integer returns an error. Otherwise, the orders first X numbers given by the limit parameter are selected from the database using the he `get_all_orders()` function. Then, every order is serialised and appended to an empty list. Finally, the list containing all the orders is converted into JSON and returned.

The API paths for orders are complete. Now, customer data must also be sent, so very similar functions will be used. To serialise a single customer, the following code is used:

```
20 def serialise_customer(customer):
21     sc = {
22         "customerID": customer[0],
23         "firstName": customer[1],
24         "surname": customer[2],
25         "telephone": customer[3]
26     }
27 
28     return sc
```

For customers, it is extremely simple; a dictionary is created based on the attributes and returned back.

The code for the API paths are exactly the same as the code for the orders, shown below:

`/customers/<customer_id>` path:

```
177 # return json data for customer given its id
178 @app.route('/customers/<customer_id>')
179 def get_customer(customer_id):
180     try:
181         customer = get_customer_by_id(customer_id)[0]
182         data = serialise_customer(customer)
183         return jsonify(data)
184     except:
185         return jsonify({"data": "Invalid request/error occured"})
```

`/customers/all/<limit>` path:

```
188 # return json data for all customers up to a certain number
189 @app.route('/customers/all/<limit>')
190 def get_customers(limit):
191     try:
192         limit = int(limit)                    # convert limit to numeric value
193         customers = get_all_customers(limit)
194         data = []
195         for c in customers:
196             data.append(serialise_customer(c))
197         return jsonify(data)
198     except:
199         return jsonify({"data": "Invalid request/error occured"})
```

Test cases for the API

The API for GET requests is now complete. Next are some example valid and invalid test cases for each path:

`/orders/<order_id>`

Valid case (path: /orders/98622)


```
{
  "category": "Takeaway",
  "contents": [
    {
      "name": "Skewer and chips",
      "price": "6.00",
      "quantity": "2"
    },
    {
      "name": "Almaza Lebanese beer",
      "price": "3.75",
      "quantity": "1"
    }
  ],
  "customerID": 98820,
  "orderID": 98622,
  "timestamp": "1519986599"
}
```


Invalid case: (path: /orders/12345)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: order ID does not exist (not in database)

Invalid case: (path: /orders/skj213)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: order ID does not exist (incorrect format)

`/customers/<customer_id>`

Valid case (path: /customers/36944)


```
{
  "customerID": 36944,
  "firstName": "Bob",
  "surname": "QWERTY",
  "telephone": "07712345678"
}
```


Invalid case: (path: /customers/12345)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: order ID does not exist (not in database)

Invalid case: (path: /customers/skj213)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: order ID does not exist (incorrect format)

/orders/all/<limit>

Valid case (path: /orders/all/3)


```
[
  {
    "category": "Takeaway",
    "contents": [
      {
        "name": "Falafel platter",
        "price": "6.00",
        "quantity": "1"
      },
      {
        "name": "7-Up",
        "price": "1.50",
        "quantity": "2"
      }
    ],
    "customerID": 5209,
    "orderID": 5011,
    "timestamp": "1522857385"
  },
  {
    "category": "Takeaway",
    "contents": [
      {
        "name": "Skewer and chips",
        "price": "6.00",
        "quantity": "2"
      },
      {
        "name": "Roast chicken wrap",
        "price": "4.00",
        "quantity": "1"
      },
      {
        "name": "Cafe Latte",
        "price": "1.60",
        "quantity": "1"
      },
      {
        "name": "Diet Coke",
        "price": "1.50",
        "quantity": "1"
      }
    ],
    "customerID": 36944,
    "orderID": 36746,
    "timestamp": "1520501211"
  }
]
```


Invalid case: (path: /orders/all/test)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: limit is not numeric

Invalid case: (path: /orders/all/)

Not Found

(404 error)

Reason: not a valid path, must have a limit

For limit = 0, an empty array is returned.

`/customers/all/<limit>`

Valid case (path: /customers/all/3)


```
[
  {
    "customerID": 5209,
    "firstName": "Alex",
    "surname": "Cons",
    "telephone": "07737933943"
  },
  {
    "customerID": 36944,
    "firstName": "Bob",
    "surname": "QWERTY",
    "telephone": "07712345678"
  },
  {
    "customerID": 77776,
    "firstName": "Dat",
    "surname": "Boi",
    "telephone": "0294958946"
  }
]
```


Invalid case: (path: /customers/all/test)


```
{
  "data": "Invalid request/error occured"
}
```


Reason: limit is not numeric

Invalid case: (path: /customers/all/)

(404 error)
Reason: not a valid path, must have a limit

Conclusion for the first web application prototype

That concludes the first prototype of the web application and API. The web application now has the majority of the originally planned functionality. Any improvements and extensions to the web API will be made in following iterations of the development cycle.

Now, the focus of development is to start building the first prototype of the kitchen client program and integrating it with the current prototype of the web API.

Part II: Developing the kitchen client program

Choosing and justifying the GUI framework

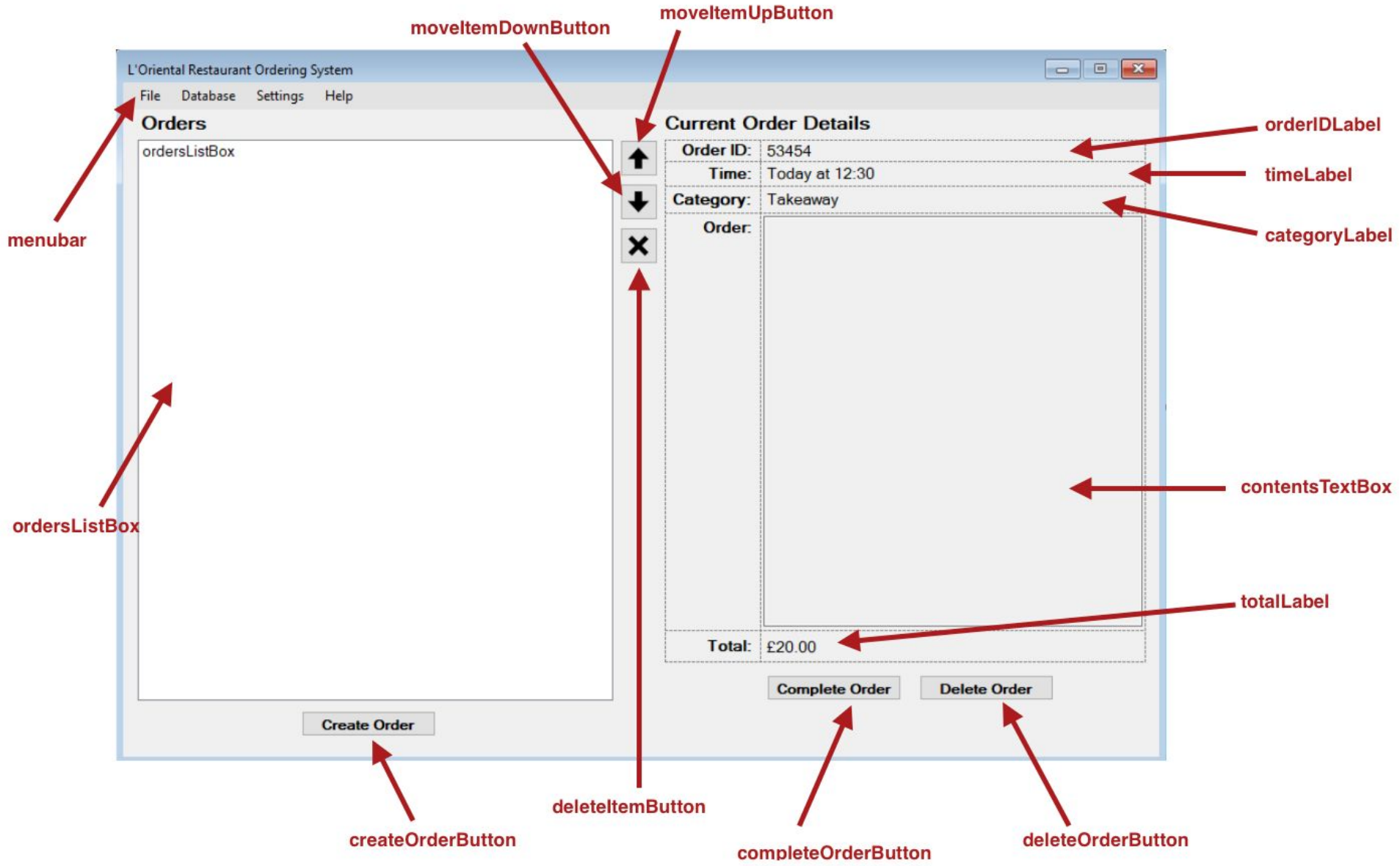
The technology framework I have decided to use to develop the kitchen client desktop program is Windows Forms, part of the .NET library developed by Microsoft.
This framework is designed to produce graphically rich and high-performing Windows applications that are easy to deploy. This perfectly suits the requirements identified by the stakeholder in the Analysis, since the client must run on a Windows computer, hence the reason I have selected Windows Forms.

The IDE I shall be using is Visual Studio, which has very good support for creating and compiling *WinForms* applications, which are written in the C# programming language.

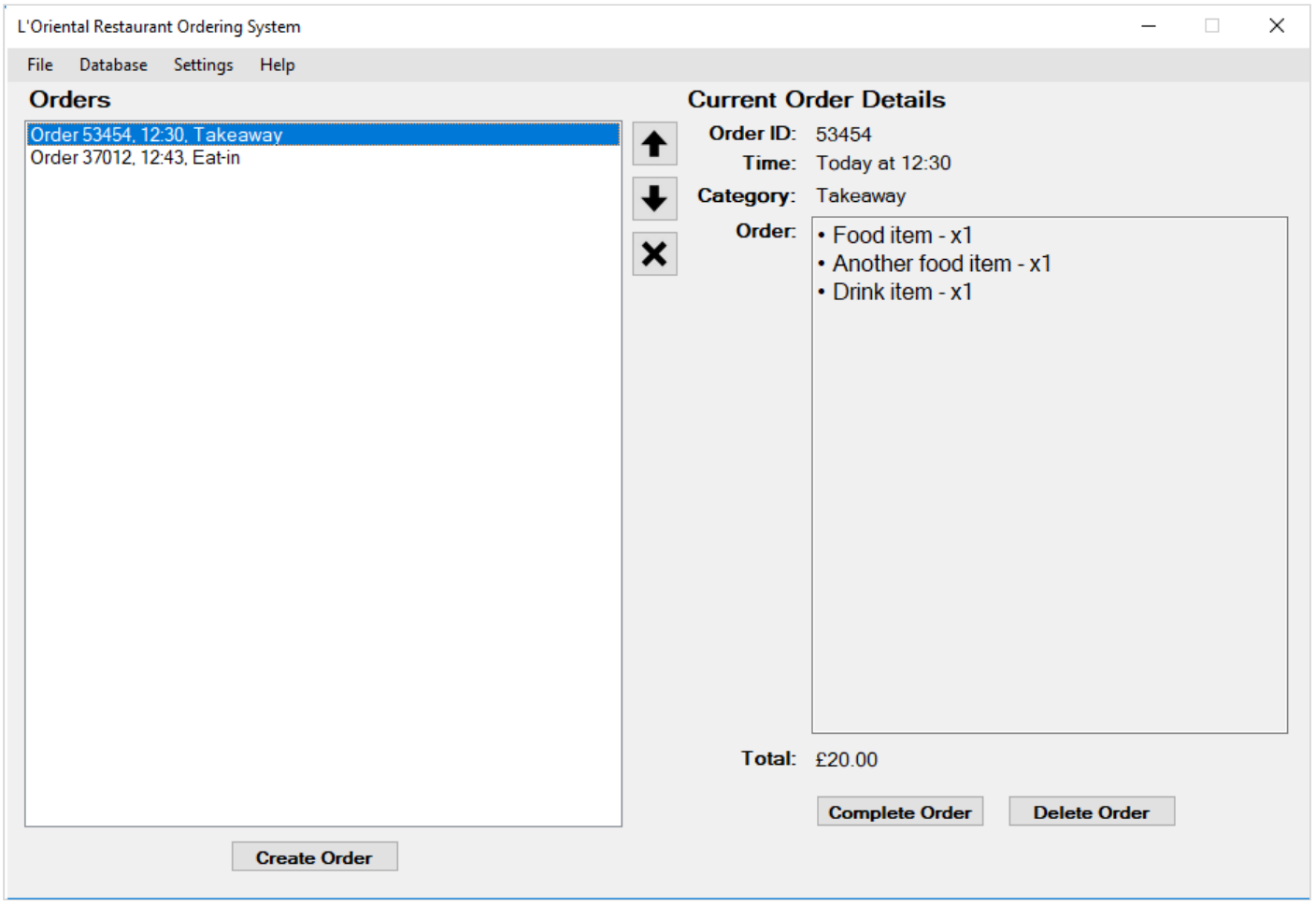
Designing the Main Window

To implement the mockups of each window/page sketched in the Design section, I shall be using the visual toolbox editor provided by the Visual Studio IDE, which allows components to be dragged and dropped onto the window, as well as changing some of their attributes, for example, the font size of a label.

As the described in the Design section, the main window will display a list of current orders and the details of a selected order. It is likely to be the most viewed/used screen in the program, so it must be implemented to suit the stakeholder's graphical needs: to be clear and easy to use and to understand.


moveItemDownButton
moveItemUpButton
L'Oriental Restaurant Ordering System
File
Database
Settings
Help
Orders
ordersListBox
Create Order
Current Order Details
Order ID:
53454
Time:
Today at 12:30
Category:
Takeaway
Order:
Total:
£20.00
Complete Order
Delete Order
orderIDLabel
timeLabel
categoryLabel
menubar
ordersListBox
contentsTextBox
totalLabel
createOrderButton
deleteItemButton
completeOrderButton
deleteOrderButton

Compiling and running the program at this point with some placeholder data produces the following application:



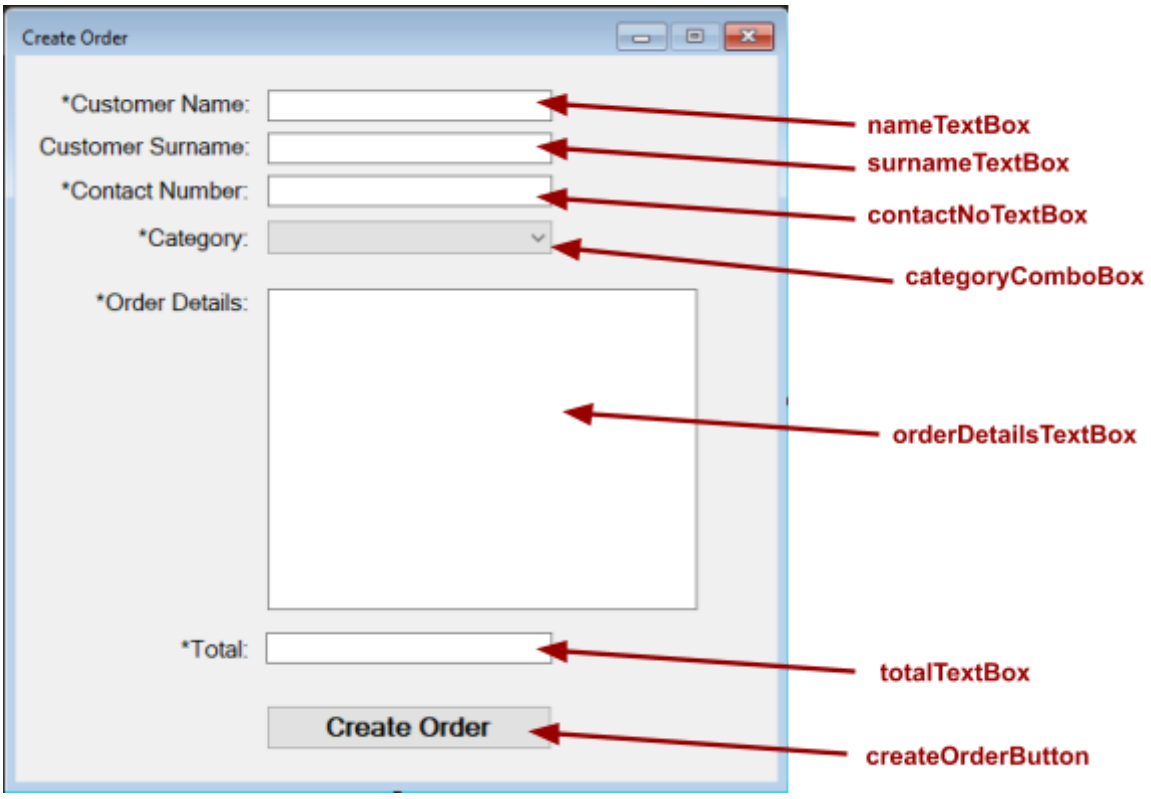
All the components are rendered as expected, including the Windows-themed native look and feel.

Of course, as of now, there is no functionality behind the interface since the code for the buttons and interactive has not been implemented yet.

Designing the Create Order Window

The next mockup to be implemented is the window for creating a new order. This window has a very simple design, since it is only a form.
In this case, the form is slightly more extensive compared to the form on the checkout page on the web application, with the user required to type in the order contents as well as the customer's details. In some cases, the customer's surname will not be known, so it is not a required field in the form.

Below is the layout produced in the visual editor, annotated with the component's names.

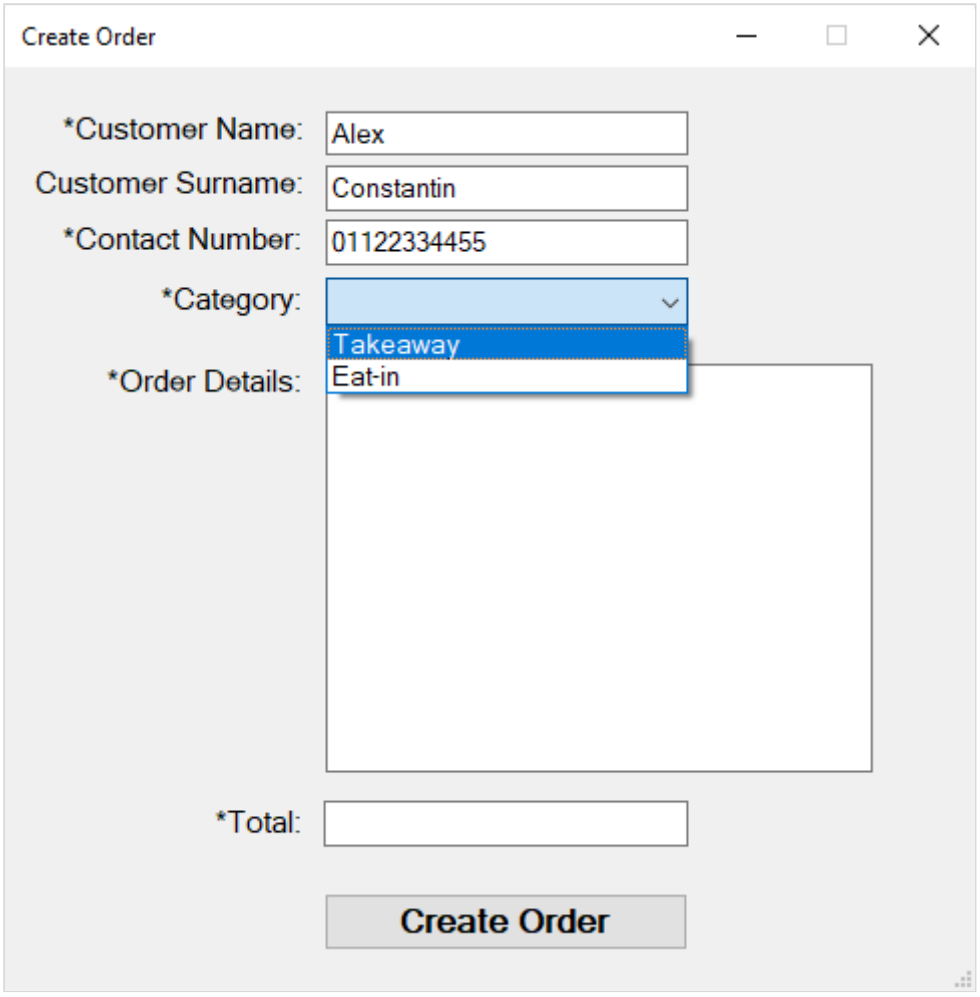


The categoryComboBox component has two possible values: "Takeaway" or "Eat-in", just like on the website. The rest of the components are TextBox objects, which allows the user to type in data. This data can then be retrieved in the code, when implemented later.

The createOrderButton will be in charge of:

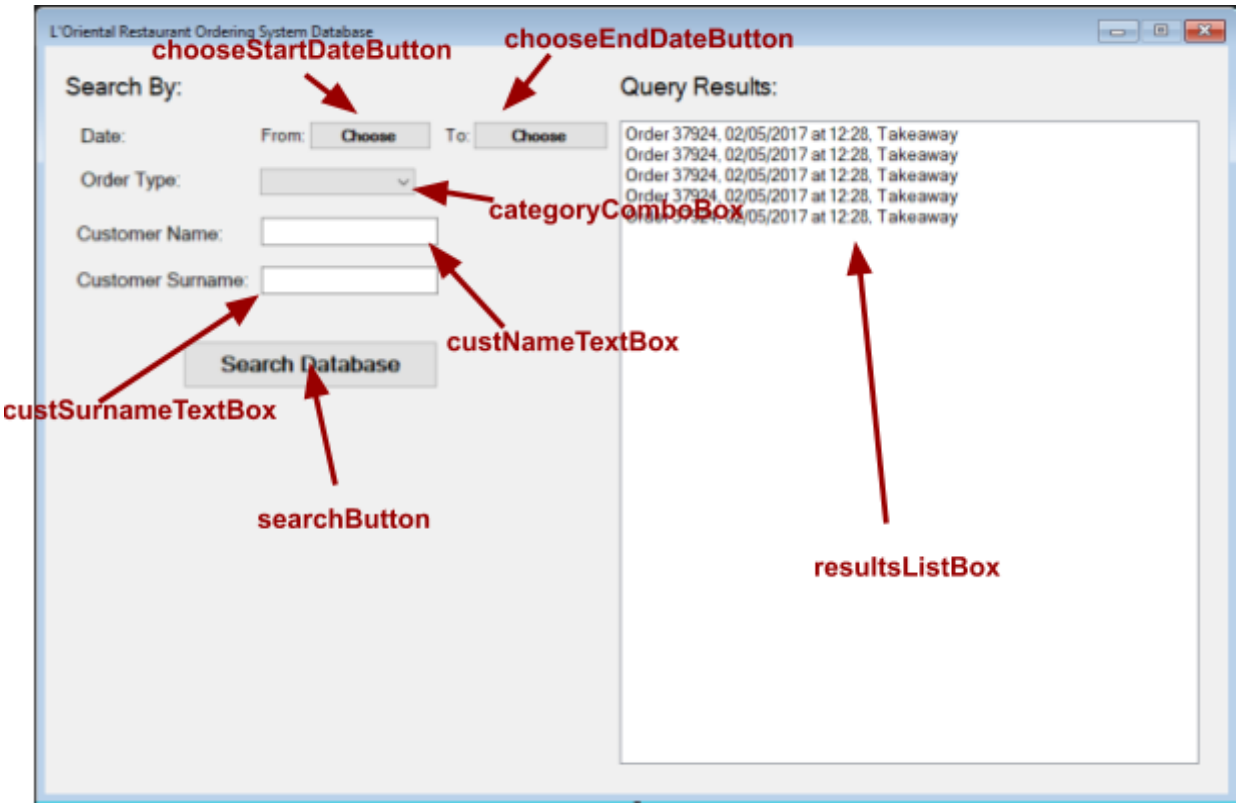
1. Validating the form data
    a. Returning any errors if necessary
2. Creating an Order object
3. Adding it to the ordersListBox on the main window
4. Sending it to the database via an API call

The font size for the input text in the text boxes has been increased to 10px, since I found that the default size (8px) was too small.
When compiling this individual window, the following output is produced:

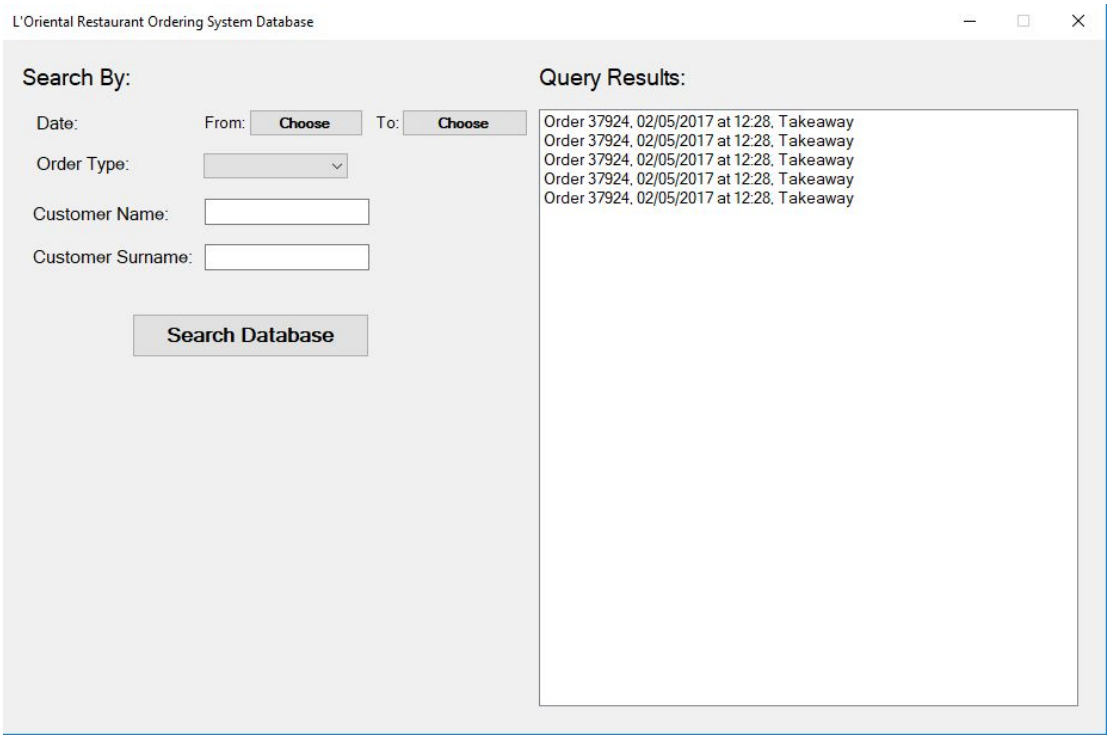

Create Order
*Customer Name:
Alex
Customer Surname:
Constantin
*Contact Number:
01122334455
*Category:
Takeaway
Eat-in
*Order Details:
*Total:
Create Order

Designing the Database Search Window

The next mockup is the window for querying the database. The form on the left allows the user to essentially build an SQL query, which is then sent to the server and the results are returned and displayed on the list box shown on the right.
Since past orders are likely to be from other days, in this case, the list box will also show the date rather than just the time.



Running this individual window produces the following output:



Implementing the Order class

Since C# is a heavily object-oriented language, it makes sense to create classes to represent an order and a customer. It also makes the code easier to read and provides a more robust structure.

The Order class will consist of encapsulated attributes, *getter* methods, a constructor and other methods. It will be implemented in the file Order.cs.

```
10  public class Order
11  {
12      // Class attributes (encapsulated)
13      private string id;
14      private string customerID;
15      private DateTime timestamp;
16      private string category;
17      private List<Tuple<string, int, float>> contents;
18      private float total;
19
20      // Class constructor from JSON input
21      public Order(dynamic orderData)
22      {
23          id = orderData.orderID;
24          customerID = orderData.customerID;
25          category = orderData.category;
26          contents = LoadContents(orderData.contents);
27          total = CalculateTotal();
28
29          // Create a DateTime object from the timestamp
30          DateTime dateTime = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc);
31          timestamp = dateTime.AddSeconds((int)orderData.timestamp).ToLocalTime();
32      }
```

Above shows the class attributes and the class constructor, which is in charge of populating the attributes with the JSON data passed into it.
For the timestamp, it converts the unix timestamp provided by the JSON data and creates a DateTime object, which produces a human readable date and time in the DD/MM/YYYY hh:mm:ss format.
For the total price, since it was not included in the payload, it is recalculated in another method shown later, only after the contents have been loaded into a list of tuples, which is the exact structure used in the Python web application.

To load the contents into a list of tuples in the form `(foodName, quantity, price)`, a for loop is used to iterate through all the items passed in as a parameter. For each item, a tuple is created with the item's name, quantity and price and added to the list, which is returned.

```
60  // Parses the contents from a dynamic data type into a list of tuples
61  private List<Tuple<string, int, float>> LoadContents(dynamic items)
62  {
63      // create an empty list of tuples (to store each item)
64      List<Tuple<string, int, float>> parsedContents = new List<Tuple<string, int, float>>();
65
66      // loop through all items and add it the the list of tuples with appropriate data types
67      foreach (var item in items)
68      {
69          // create the tuple
70          parsedContents.Add(new Tuple<string, int, float>(item.name.ToString(), (int)item.quantity, (float)item.price));
71      }
72
73      return parsedContents;
74  }
```

To calculate the total price of the order, the same algorithm is used as in the web server:

```
112 // Calculates the total cost of the order
113 private float CalculateTotal()
114 {
115     float sum = 0;
116     foreach (var food in contents)
117     {
118         sum = sum + food.Item2 * food.Item3; // multiplies price of item by its quantity
119     }
120     return sum;
121 }
```

A for loop is used to add to the sum variable, which is calculated by multiplying the quantity (represented by `food.Item2`) by the price (represented by `food.Item3`).

Finally, in order to access the attributes, public methods must be used, since the attributes are encapsulated. These are called *getter functions*, because they simply return the attributes.

```
51  // Getter methods for class attributes
52  public string GetID() { return id; }
53  public string GetCustomerID() { return customerID; }
54  public DateTime GetTimestamp() { return timestamp; }
55  public List<Tuple<string, int, float>> GetContents() { return contents; }
56  public string GetCategory() { return category; }
57  public float GetTotal() { return total; }
```

These are typically written in a single line, since they are so simple and straightforward.

Implementing the Customer class

Just like each order has its own object, each customer will also have its object. The file shall be called Customer.cs and the code for the Customer class is shown below:

```
 9  public class Customer
10  {
11      // Class attributes (encapsulated)
12      private string id;
13      private string firstName;
14      private string surname;
15      private string telephone;
16
17      // Class constructor from JSON data
18      public Customer(dynamic customerData)
19      {
20          id = customerData.customerID;
21          firstName = customerData.firstName;
22          surname = customerData.surname;
23          telephone = customerData.telephone;
24      }
```

The attributes defined above are encapsulated, just like in the Order class. Its constructor works in the same way: the attributes are assigned to the JSON attributes passed in as a parameter.

The *getter* functions are shown below:

```
36  // Get class attributes using methods for encapsulation
37  public string GetID() { return id; }
38  public string GetFirstName() { return firstName; }
39  public string GetSurname() { return surname; }
40  public string GetTelephone() { return telephone; }
```

With the Order and Customer class ready to be used, the next step is to actually retrieve the JSON data from the API, and populate the client program with real data.

Fetching the JSON data from the web API

Before adding functionality to all the buttons and other components in the graphical user interface designed so far, I have decided to first code the network part of the program, in order to integrate the program with the web application.

To download the JSON data from the API on the web server, rather than serialising the data into JSON, it now must be deserialised from JSON into an object.
For all API related code, I have created an Orders.cs and Customers.cs file in the Kitchen_Client namespace. All code will be written inside a class, since C# is an object-oriented programming language.

The code to get the JSON data from the `/orders/all/<limit>` path is shown below:

```
10  namespace Kitchen_Client
11  {
12      class Orders
13      {
14          private const string ORDERS_API_BASE_URL = "http://loriental-restaurant-thealexcons.c9users.io/orders/";
15
16          // Calls the API endpoint at /orders/all/<limit>
17          private static dynamic GetOrders(int limit)
18          {
19              // Download Json data from the response by the API
20              var jsonData = new WebClient().DownloadString(ORDERS_API_BASE_URL + "all/" + limit.ToString());
21
22              // Parse Json data into a "dynamic" data type variable
23              dynamic orders = JsonConvert.DeserializeObject(jsonData);
24              return orders;
25          }
26
27      }
```

A base URL on which the website is running on is defined as a constant string on line 14. Then, the `GetOrders()` function is defined, taking in the number of posts to limit and returning a list of orders as a dynamic data type. This means a data type is assigned at runtime, since its data type is unknown.
It works by downloading the JSON string and then deserialising it using a built-in library provided by the .NET framework, called JsonConvert.

However, as you can see, it is a private method meaning it cannot be accessed outside the class (encapsulation). This is because this will act as a base function to other methods, such as `GetTodaysOrders()`, implemented next.

To test whether it's working correctly, I printed out the JSON data returned by the function with `Console.WriteLine(Orders.GetOrders(1))` and I got the following response, which was in fact what I was expecting:


```
[
  {
    "category": "Eat-in",
    "contents": [
      {
        "name": "Skewer and chips",
        "price": 6.0,
        "quantity": 1
      },
      {
        "name": "Banana smoothie",
        "price": 3.5,
        "quantity": 1
      }
    ],
    "customerID": 66318,
    "orderID": 66065,
    "timestamp": "1523381417"
  }
]
```

```
28  // Get today's orders
29  public static List<Order> GetTodaysOrders()
30  {
31      dynamic ordersData = GetOrders(30); // get the 30 latest orders
32      List<Order> orders = new List<Order>();
33
34      foreach (dynamic orderData in ordersData)
35      {
36          // Convert the timestamp into a Datetime object
37          DateTime dateTime = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc);
38          DateTime timestamp = dateTime.AddSeconds((int)orderData.timestamp);
39
40          // Check if the timestamp is today
41          if (timestamp.Date == DateTime.Today)
42          {
43              Order newOrder = new Order(orderData);
44              orders.Add(newOrder);
45          }
46      }
47      return orders;
48  }
```

As inferred by the self-explanatory identifier, this method returns a list of Order instances from the current day. On line 31, it uses the `GetOrders()` function to fetch the latest 30 orders, which is a safe assumption of number of orders in a day.
Then, a for loop on line 34 is used to iterate through all the orders in the deserialised JSON data returned and the order timestamp given by timestamp attribute in the JSON is converted into a DateTime object. Once in this format, an if statement can be used to check whether the order timestamp is today. If so, it instantiates a new Order object using the JSON data and it is added to the orders list, which is later returned on line 47.

In the case where only the latest order is needed, for example, when polling the API for any new orders, I also wrote a `GetLatestOrder()` method, which simply got the first element from the list returned by the `GetOrders()` function:

```
50  // Gets the first order
51  public static Order GetLatestOrder()
52  {
53      dynamic orderData = GetOrders(1)[0]; // get the first order returned from the API
54      Order order = new Order(orderData);
55      return order;
56  }
```

Just like the `GetTodaysOrders()` method, it is set to be a public method since this will be used from outside the class.

I also wrote a method to retrieve a specific order if its ID is known, using the API endpoint located at `/orders/<order_id>`. The code is very similar to the `GetOrders()` function, since it requires to call a different resource from the API:

```
58      // Calls the API endpoint at /orders/<order_id>
59      public static Order GetOrderByID(string orderID)
60      {
61          // Download Json data from the response by the API
62          var jsonData = new WebClient().DownloadString(ORDERS_API_BASE_URL + orderID);
63
64          // Parse Json data into a "dynamic" data type variable
65          dynamic orderData = JsonConvert.DeserializeObject(jsonData);
66          Order order = new Order(orderData);
67          return order;
68      }
```

To retrieve customer data, very similar code is used. To maintain a consistent project structure, I have created a Customers.cs file in the Kitchen_Client namespace, as mentioned previously. This file contains two methods: a private “base” method, `GetCustomers()` and a public method, `GetCustomerByID()`.

```
11  class Customers
12  {
13      private const string CUSTOMERS_API_BASE_URL = "http://loriental-restaurant-thealexcons.c9users.io/customers/";
14
15      // Calls the API endpoint at /customers/all/<limit>
16      private static dynamic GetCustomers(int limit)
17      {
18          // Download Json data from the response by the API
19          var jsonData = new WebClient().DownloadString(CUSTOMERS_API_BASE_URL + "all/" + limit.ToString());
20
21          // Parse Json data into a "dynamic" data type variable
22          dynamic customers = JsonConvert.DeserializeObject(jsonData);
23          return customers;
24      }
25
26      // Calls the API endpoint at /customers/<customer_id>
27      public static Customer GetCustomerByID(string customerID)
28      {
29          // Download Json data from the response by the API
30          var jsonData = new WebClient().DownloadString(CUSTOMERS_API_BASE_URL + customerID);
31
32          // Parse Json data into a "dynamic" data type variable
33          dynamic customerData = JsonConvert.DeserializeObject(jsonData);
34          Customer customer = new Customer(customerData);
35          return customer;
36      }
37  }
```

Everything is exactly the same, however the base URL has been changed to match the customers part of the API and the JSON data is deserialised into a Customer object.

Remedial action to improve JSON parsing:

When converting the JSON data into an Order object, I had trouble converting the numerical quantities from strings into integers or floats. To fix this, I went back to the serialisation.py file on the web server and converted the types into float and int there. This was much easier to fix in Python, rather than in C#.

```
3  def serialise_order(order):
4      so = {
5          "orderID": order[0],
6          "timestamp": str(order[1]),
7          "customerID": order[4],
8          "category": order[3],
9          "contents": []
10     }
11
12     contents = ast.literal_eval(order[2])   # parse text blob into list
13
14     for item in contents:
15         so["contents"].append( {"name": item[0], "quantity": int(item[1]), "price": float(item[2])} )
16
17     return so
```

This tweak results in the following change in the JSON data:

Before:


```
{
  "category": "Takeaway",
  "contents": [
    {
      "name": "Skewer and chips",
      "price": "6.00",
      "quantity": "2"
    },
    {
      "name": "Almaza Lebanese beer",
      "price": "3.75",
      "quantity": "1"
    }
  ],
  "customerID": 98820,
  "orderID": 98622,
  "timestamp": "1519986599"
}
```


After:


```
{
  "category": "Takeaway",
  "contents": [
    {
      "name": "Skewer and chips",
      "price":  6.0  ,
      "quantity":  2
    },
    {
      "name": "Almaza Lebanese beer",
      "price":  3.75 ,
      "quantity":  1
    }
  ],
  "customerID": 98820,
  "orderID": 98622,
  "timestamp": "1519986599"
}
```

In order to add an order to the list box component, the following function is used:

```
33  public void AddOrderToListBox(Order order)
34  {
35      string orderName = "Order " + order.GetID() + ", " + order.GetTimestamp().ToShortTimeString() + ", " + order.GetCategory();
36      ordersListBox.Items.Insert(0, orderName); // add to front, since orders are in reverse
37  }
```

Since the list box can only display a string of text for each row in the table, each order is given a “name”, which consists of its unique ID, the time of submission and the category. This string is created on line 35, and then it is inserted to the orders list box component on line 36.

This means orders will look like this in the table:

The list box is a great way of displaying the orders because not only it does it display them row by row, it also allows the user to select/highlight an order using the mouse. To take advantage of this great feature, whenever the user selects a different order in the list, the details panel should display the specific details of the selected order. This can be achieved with the following pair of functions:

```
88  // Load the selected order's details onto the panel
89  private void ViewSelectedOrderDetails()
90  {
91      // Check if an item has been selected. If not, stop
92      if (ordersListBox.SelectedItem == null || ordersListBox.SelectedIndex < 0)
93          return;
94
95      currentOrderTable.Enabled = true;
96
97      // Get the order id from the name on the item display
98      string currentItemName = ordersListBox.SelectedItem.ToString();
99      string currentOrderID = currentItemName.Substring(6, 5);
100
101     // Search the current orders for the relevant order object using the order id
102     foreach (Order order in CurrentOrdersList)
103     {
104         if (currentOrderID == order.GetID())
105         {
106             // Fill out the details panel with the order's attributes
107             orderIDLabel.Text = currentOrderID;
108             timeLabel.Text = "Today at " + order.GetTimestamp().ToShortTimeString();  // shows only hh:mm format
109             categoryLabel.Text = order.GetCategory();
110             totalLabel.Text = "£" + order.GetTotal().ToString("0.00"); // show 2 decimal places
111             contentsTextBox.Text = order.GetDisplayContents();
112             break;
113         }
114     }
115 }
```

This function uses the ID in the order name string to find the relevant order object, using a for loop. Once it has been found, the label components have their text attribute set to the order’s attributes, as shown in lines 107 to 111. To display the contents, a new method for the Order class called `GetDisplayContents()` has been implemented, shown next.

Moving orders around using the arrow buttons

To satisfy the requirements of making the program easy to use, some visual arrow buttons have been implemented to move orders around in the list.

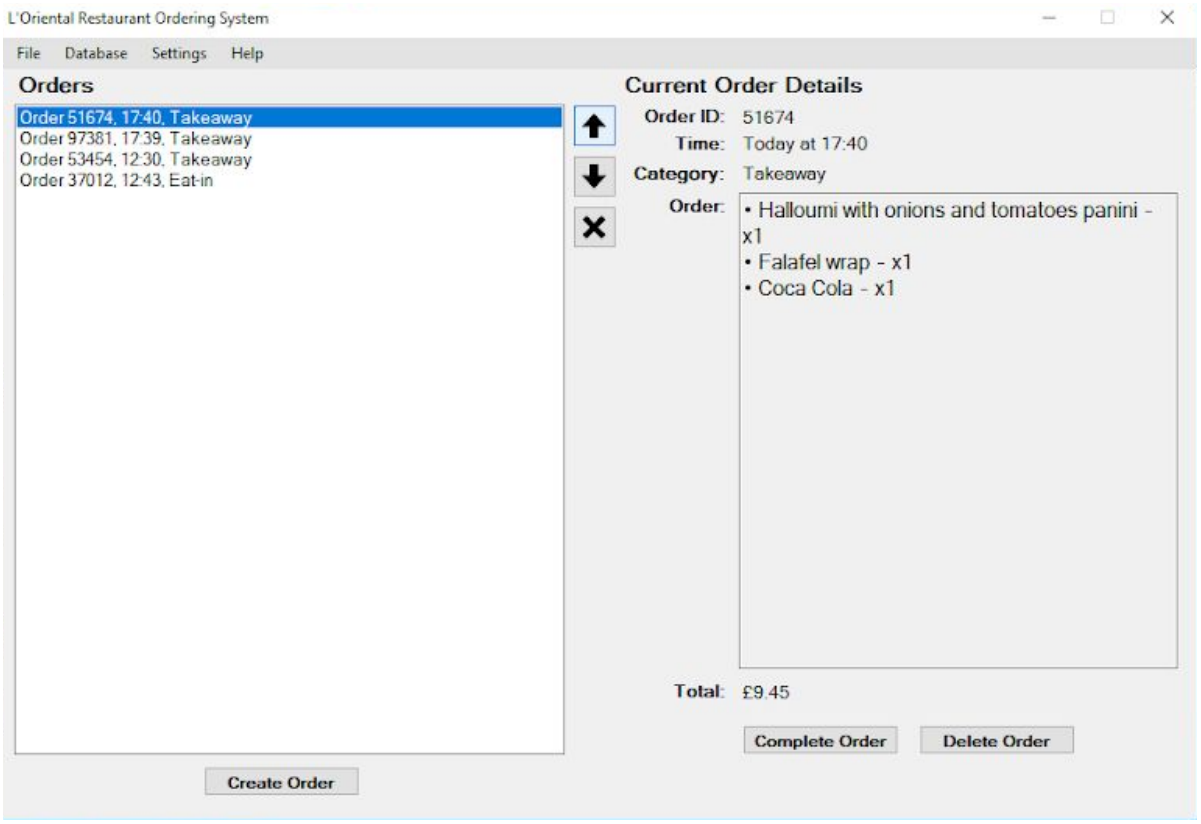
```
49    // Move the items up or down in the list
50    private void MoveItem(int direction)
51    {
52        // Check if an item has been selected. If not, stop
53        if (ordersListBox.SelectedItem == null || ordersListBox.SelectedIndex < 0)
54            return;
55
56        // Calculate new index using direction (-1 for up, +1 for down)
57        int newIndex = ordersListBox.SelectedIndex + direction;
58
59        // Checking indices of the listbox. If out of range, stop
60        if (newIndex < 0 || newIndex >= ordersListBox.Items.Count)
61            return;
62
63        // get the selected element
64        object currentItem = ordersListBox.SelectedItem;
65
66        // remove element from current position
67        ordersListBox.Items.Remove(currentItem);
68        // Insert it in new position
69        ordersListBox.Items.Insert(newIndex, currentItem);
70        // Reselect new index
71        ordersListBox.SetSelected(newIndex, true);
72    }
```

First, it checks whether an item has actually been selected. If not, the function stops. Then, the new index is calculated by adding -1 for moving up or +1 for moving down. However, at this point, the new index may be out of range, so a check is made to see if an order at the top is moved up or an order at the bottom is moved down.
If everything works out, the current item is selected, removed from its current position, inserted into its new position using the newly calculated index and it re-selects the item.

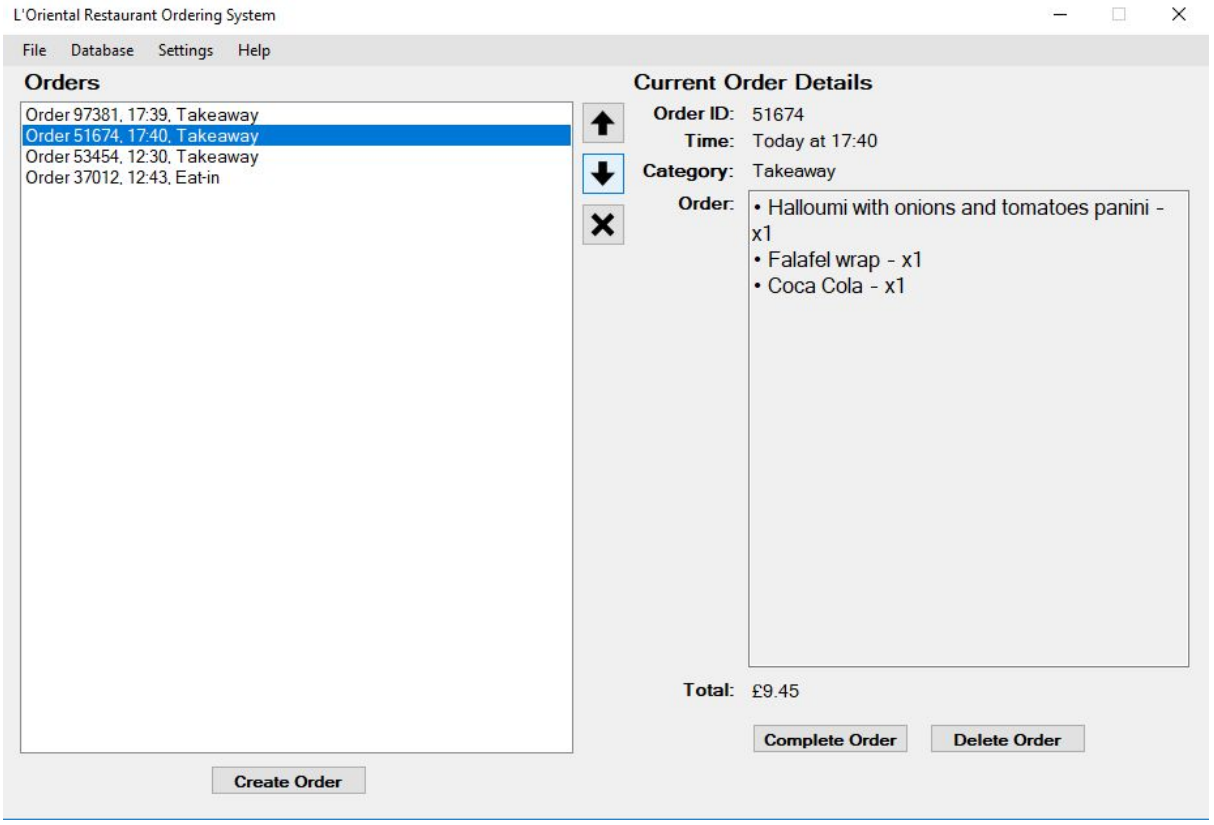
Once again, for the program to respond to an event, in this case, a mouse click on the button, event handlers must be used to execute the desired response.
The event handlers for when the arrow buttons are clicked are shown below:

```
74    private void moveItemUpButton_Click(object sender, EventArgs e)
75    {
76        MoveItem(-1);
77    }
78
79    private void moveItemDownButton_Click(object sender, EventArgs e)
80    {
81        MoveItem(1);
82    }
```

If the first order is selected and the up button is clicked, nothing happens, as expected:



If the first order is selected and the down button is clicked, the order moves down into second position, as expected.



Deleting an order

Both delete buttons are bound to the same event handler, shown below:

```
153  // Deletes the selected item from the ordersListBox
154  private void deleteItemButton_Click(object sender, EventArgs e)
155  {
156      // Check if an item has been selected. If so, delete it from the list
157      if (ordersListBox.SelectedItem != null || ordersListBox.SelectedIndex >= 0)
158      {
159          // Create a warning message box before deleting
160          const string question = "Are you sure that you would like to delete this order?";
161          const string title = "Deleting Order";
162          var decision = MessageBox.Show(question, title, MessageBoxButtons.YesNo,
163                                         MessageBoxIcon.Warning, MessageBoxDefaultButton.Button2);
164
165          // If user chooses Yes, then delete
166          if (decision == DialogResult.Yes)
167          {
168              string orderName = ordersListBox.Items[ordersListBox.SelectedIndex].ToString();
169              ordersListBox.Items.RemoveAt(ordersListBox.SelectedIndex); // remove from display list box
170              DeleteOrder(orderName);    // remove from actual list
171              ClearOrderDetails();
172          }
173      }
174  }
```

Firstly, it checks if an order has been selected. Otherwise, clicking the button has no effect. If so, a warning message box is shown to the user, to confirm the deletion. If the user clicks the "Yes" button, the selected order is retrieved, removed from the list box and the current orders list variable. Finally, the details are cleared from the screen with `ClearOrderDetails()`, implemented next.

To remove order from the actual `CurrentOrdersList` variable, the ID is taken from the order name and the order is searched for in the list. When found, it is removed. This code is shown below, in the `DeleteOrder()` function

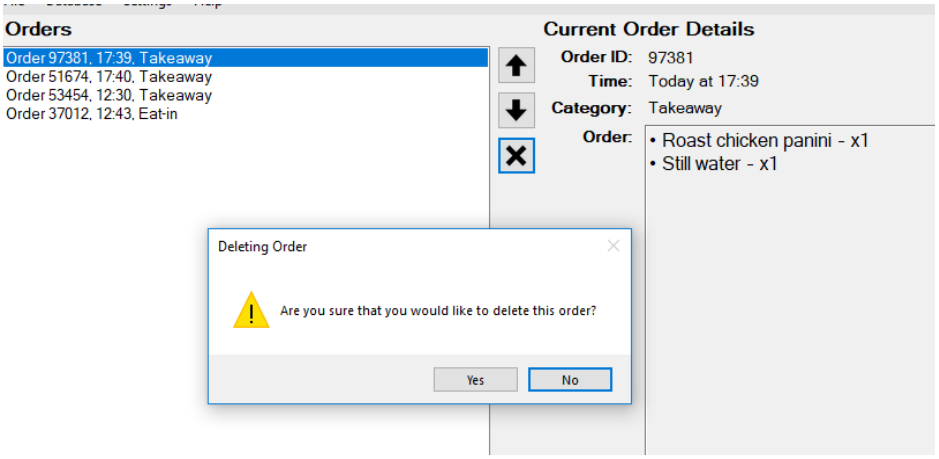
```
119  // Remove an order from the list
120  private void DeleteOrder(string displayName)
121  {
122
123      // Get the order id from the name on the item display
124      string currentOrderID = displayName.Substring(6, 5);
125
126      // Search the current orders for the relevant order object using the order id
127      foreach (Order order in CurrentOrdersList)
128      {
129          if (currentOrderID == order.GetID())
130          {
131              CurrentOrdersList.Remove(order); // remove order if it matches the ID
132              break;
133          }
134      }
135  }
```

The deletion from the `CurrentOrdersList` can be confirmed by printing out the length of the list before and after the `DeleteOrder()` call. As expected, the length decreases by one, so it works correctly.

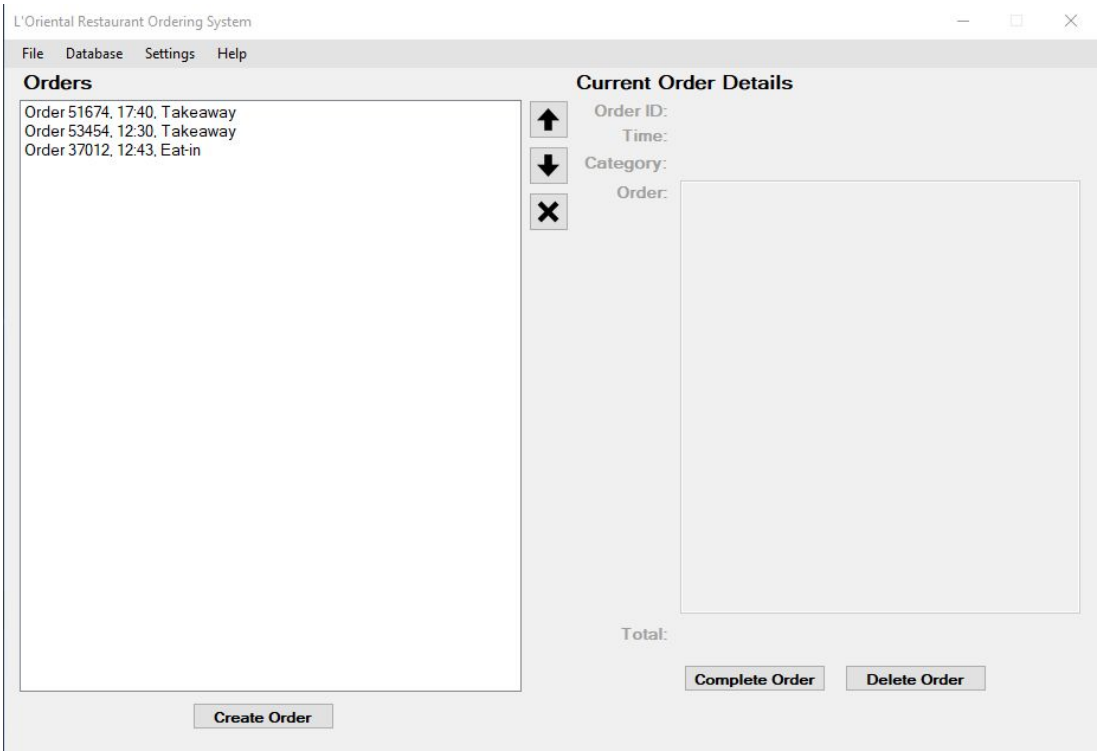
Once an order has been deleted, since it was the last one to be selected, its details are still being shown on the screen. Therefore, a function can be use to clear the order details from the screen, by setting the text of each label to an empty string, as shown next:

```
158     // Clear all the labels from the order details section when an item is deleted
159     private void ClearOrderDetails()
160     {
161         orderIDLabel.Text = "";
162         timeLabel.Text = "";
163         categoryLabel.Text = "";
164         totalLabel.Text = "";
165         contentsTextBox.Text = "";
166         currentOrderTable.Enabled = false;
167     }
```

When click either of the delete buttons, the following warning prompt appears:



If the user clicks “No”, the prompt closes and nothing happens. If they press “Yes”, the order is removed and its details are cleared:

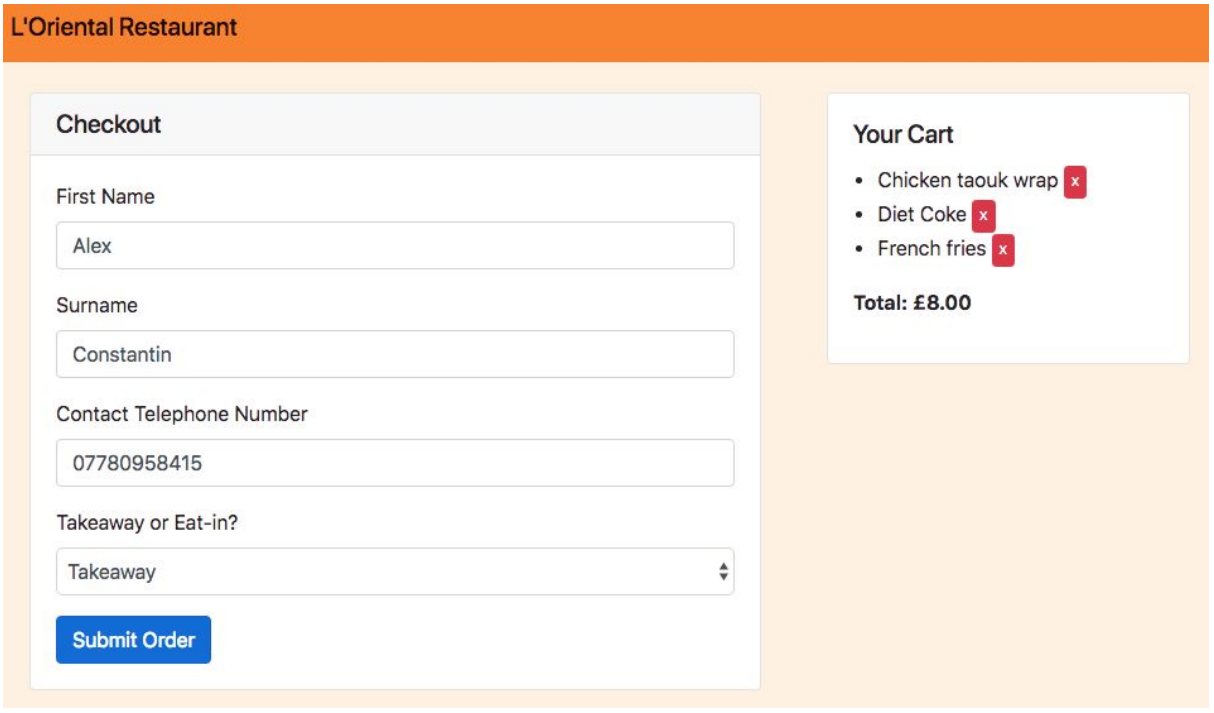


Integration test between the web application and client program

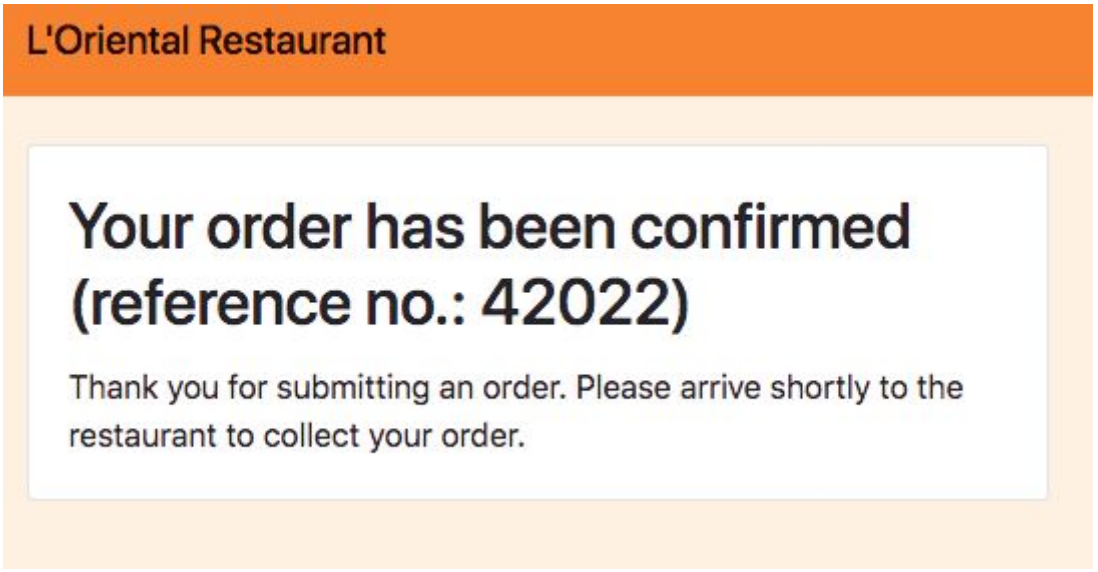
To test the functionality of the system as a whole at this point, I will carry out some integration testing to make sure the web application and client program are working correctly together. This will be a good indicator of progress. The testing algorithm will consist of two steps:

1. Add a new order via the website
2. Run the client program
3. Check if the new order has been added

Step 1:
Select the items and fill out the checkout form, with the following valid test data:



Confirmation page with the reference number, which is the order ID:



Building the Create New Order form

The window for manually creating a new order consists of two parts: validating the form data (and returning any errors as feedback to the user) and creating the new order as an object when the “submit” button is clicked.

The validation algorithm takes up the complexity of this form, since every input field must be validated. Therefore, it will be slightly more complex than the validation algorithm used in the web application for the checkout page. Also, the contents must be parsed to create the list of tuples, so it must be typed in a standard format.

Firstly, to return errors as feedback, an errors array has been defined with a size of 5, since there are five required fields:

```
17      // a list holding any errors to give the user feedback on the completion of the form
18      private string[] errors = new string[5];
```

In order to validate the form data, the button must be clicked. Therefore, an event handler must be written. It is shown below:

```
131     private void createOrderButton_Click(object sender, EventArgs e)
132     {
133         // If the form is valid, return the order. Otherwise, return errors
134         if (validateForm())
135         {
136             Order order = new Order(nameTextBox.Text, surnameTextBox.Text, contactNoTextBox.Text,
137                                     categoryComboBox.SelectedItem.ToString(), contents,
138                                     float.Parse(totalTextBox.Text));
139             NewOrder = order;
140             DialogResult = DialogResult.OK; // confirm that the form has been succesfully completed
141             Close();
142         }
143         else
144         {
145             string errorString = "";
146             for (int i = 0; i <= 4; i++)
147             {
148                 errorString += "\n• " + errors[i];
149             }
150             MessageBox.Show("Some data you have entered is invalid: " + errorString, "Form invalid",
151                             MessageBoxButtons.OK, MessageBoxIcon.Warning);
152             // Reset the errors array
153             for (int i = 0; i <= 4; i++)
154             {
155                 errors[i] = "";
156             }
157         }
158     }
```

Firstly, it checks whether the form is valid or not using the `ValidateForm()` function. This is written next, and it will return a boolean value. If true, an order object is instantiated using the form data, and it is returned back to the MainWindow. Otherwise, a message box is shown containing all the errors gathered from the `ValidateForm()` function, and the errors list are reset back to empty strings.

When creating the new order object on line 122, it uses a different constructor, because it takes in different parameters. Hence, an overloaded constructor must be implemented for the Order class, which is shown under the remedial action section next.

Finally, to actually show the window, the following event handler must be implemented in MainWindow.cs for when the “Create Order” button is clicked:

```
144     // Open the create order form window
145     private void createOrderButton_Click(object sender, EventArgs e)
146     {
147         CreateOrderWindow createOrderWindow = new CreateOrderWindow();
148         var result = createOrderWindow.ShowDialog();
149 
150         // if the form was completed, add order to the list box
151         if (result == DialogResult.OK)
152         {
153             // Add new order to the list box and the actual list
154             Order newOrder = createOrderWindow.NewOrder;
155             CurrentOrdersList.Add(newOrder);
156             AddNewOrderToListBox(newOrder);
157         }
158     }
```

It opens the window, and if the form returns a status of OK (i.e.: valid), the order is returned and added to the `CurrentOrdersList` and the listbox.

Remedial action/improvement for the Order and Customer class constructor

Justification: a second constructor is needed because there are two ways of creating orders (from JSON or from the form) and they’re inputs are gathered differently, so a second overloaded constructor is required:

```
38  // Class constructor from manual input
39  public Order(string custFirstName, string custSurname, string custTelephone, string ordCategory, string ordContents, float ordTotal)
40  {
41      id = ((int)((DateTime.UtcNow.Subtract(new DateTime(1970, 1, 1))).TotalSeconds)).ToString().Substring(5,5);
42      timestamp = DateTime.UtcNow.ToLocalTime();
43      category = ordCategory;
44      total = ordTotal;
45      contents = ParseManualContents(ordContents);
46      customer = new Customer(custFirstName, custSurname, custTelephone);
47  }
```

It sets all of its attributes to the parameters passed in from the form fields, except for the ID, which is generated the same way as in the web application, and the timestamp, which is obtained using the DateTime class. Also, a customer is embedded into it, rather than the customer ID only. This makes it easier to access the customer linked to an order.

The constructor above also requires the creation of a customer object, so another constructor is also needed for the Customer class, shown below:

```
26      // Class constructor from manual input
27      public Customer(string custFirstName, string custSurname, string custTelephone)
28      {
29          id = (Convert.ToInt32(GenerateID()) + 255).ToString();
30          firstName = custFirstName;
31          surname = custSurname;
32          telephone = custTelephone;
33      }
```

Now that the contents have been validated and parsed, the `ParseManualContents()` function can be used within the validation algorithm, whose code is split into sections:

Validating the customer name and telephone number:

```
30  // Validates the form and returns true/false
31  private bool validateForm()
32  {
33      // use a boolean to flag whether a field is invalid
34      bool valid = true;
35
36      // get the values from the text boxes
37      string customerName = nameTextBox.Text;
38      string customerSurname = surnameTextBox.Text;
39      string contactNumber = contactNoTextBox.Text;
40
41      // validate the customer first name (surname not required)
42      if (customerName == "")
43      {
44          errors[0] = "You must at least provide the customer's name.";
45          valid = false;
46      }
47
48      // validate the phone number using regular expressions to match a UK number format (mobile or landline)
49      Regex phoneNumberRegex = new Regex(@"^(((\+44\s?\d{4}|\(?\d{5}\)?)\s?\d{6})|((\+44\s?|0)7\d{3}\s?\d{6})$");
50      Match match = phoneNumberRegex.Match(contactNumber);
51      if (!match.Success)
52      {
53          errors[1] = "The phone number you have entered is not valid.";
54          valid = false;
55      }
```

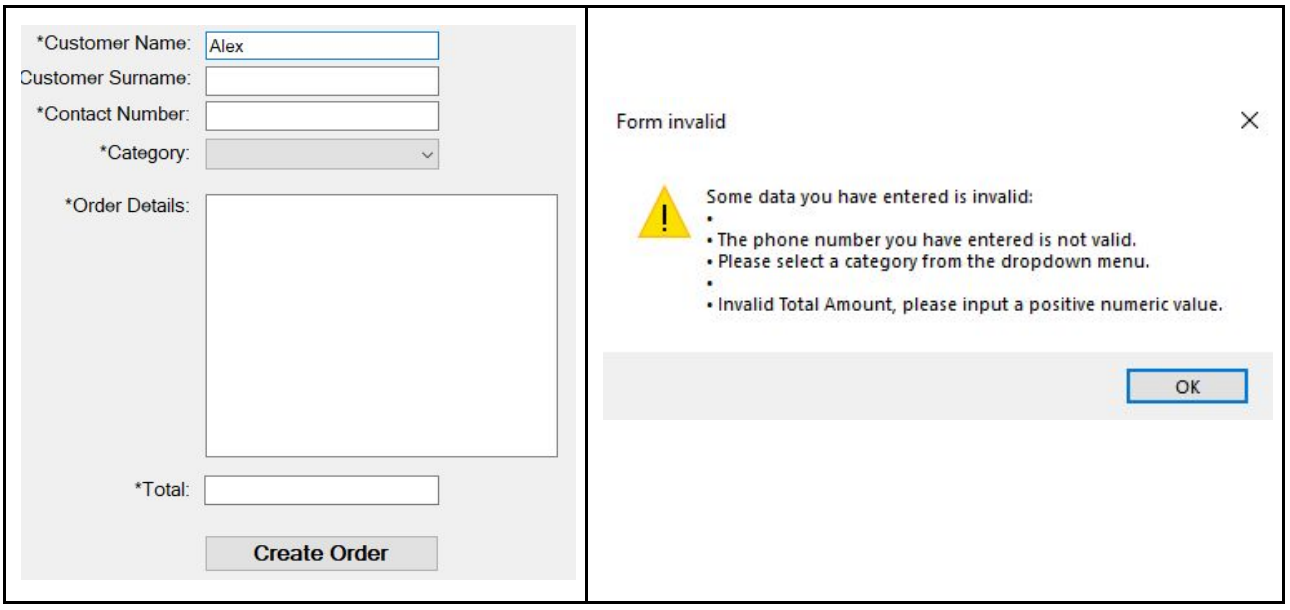
First, a boolean variable called valid is used to track whether a field is invalid or not. It will be used at the end of the function to determine if the entire form is valid. Next, the text values are obtained from the text boxes. On line 42, the customer name is validated by ensuring it is not left empty. Otherwise, an error message is added to the errors list and valid is flagged to false. On line 49, the telephone number is validated using regular expressions, just like on the web application.

Validating the customer name and telephone number:

```
57      // validate the category entry using a try-catch statement (if not selected, an error will be thrown)
58      try
59      {
60          string category = categoryComboBox.SelectedItem.ToString();
61      }
62      catch
63      {
64          errors[2] = "Please select a category from the dropdown menu.";
65          valid = false;
66      }
67
68      // try to parse the contents
69      contents = ParseManualContents(orderDetailsTextBox.Text);
70      if (contents == null)
71      {
72          errors[3] = "The order details are not in the correct format.";
73          valid = false;
74      }
```

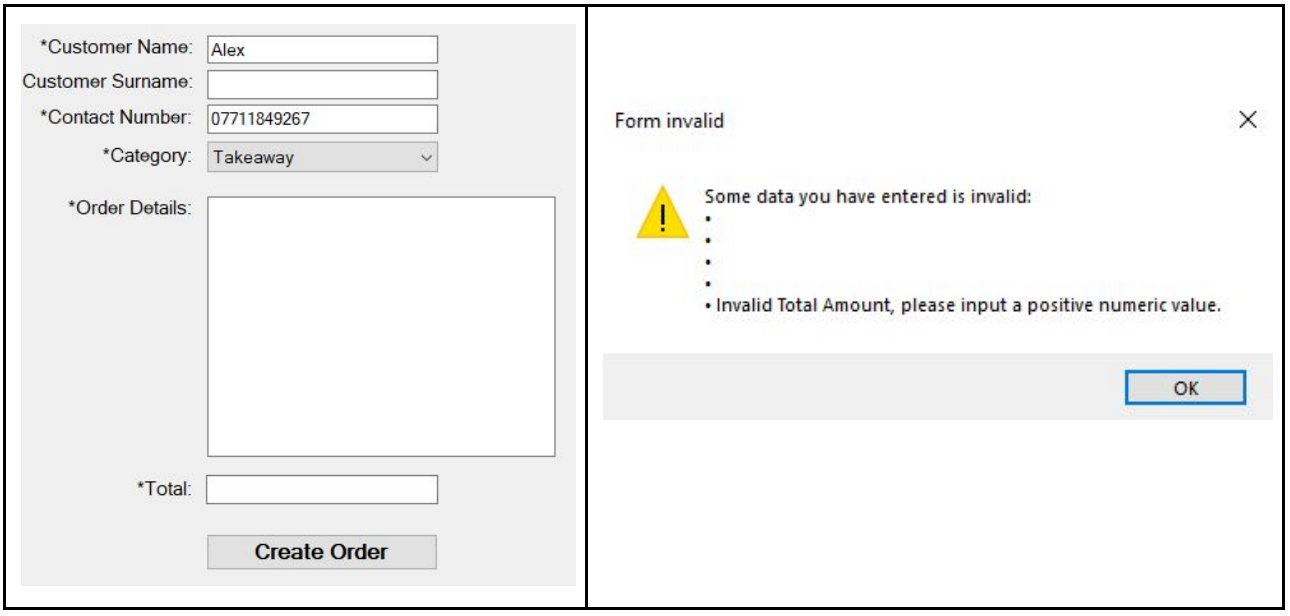
The category is validated by checking whether the user has selected an item. If not, an exception is raised since it would hold null. For the contents, the ParseManualContents() function is used (written previously) to get the list of tuples. If an error occurs, the error message is added.

Incomplete required data example 1:



Working as expected? Yes, it shows the relevant warning message

Incomplete required data example 2:



Working as expected? Yes, it shows the relevant warning message

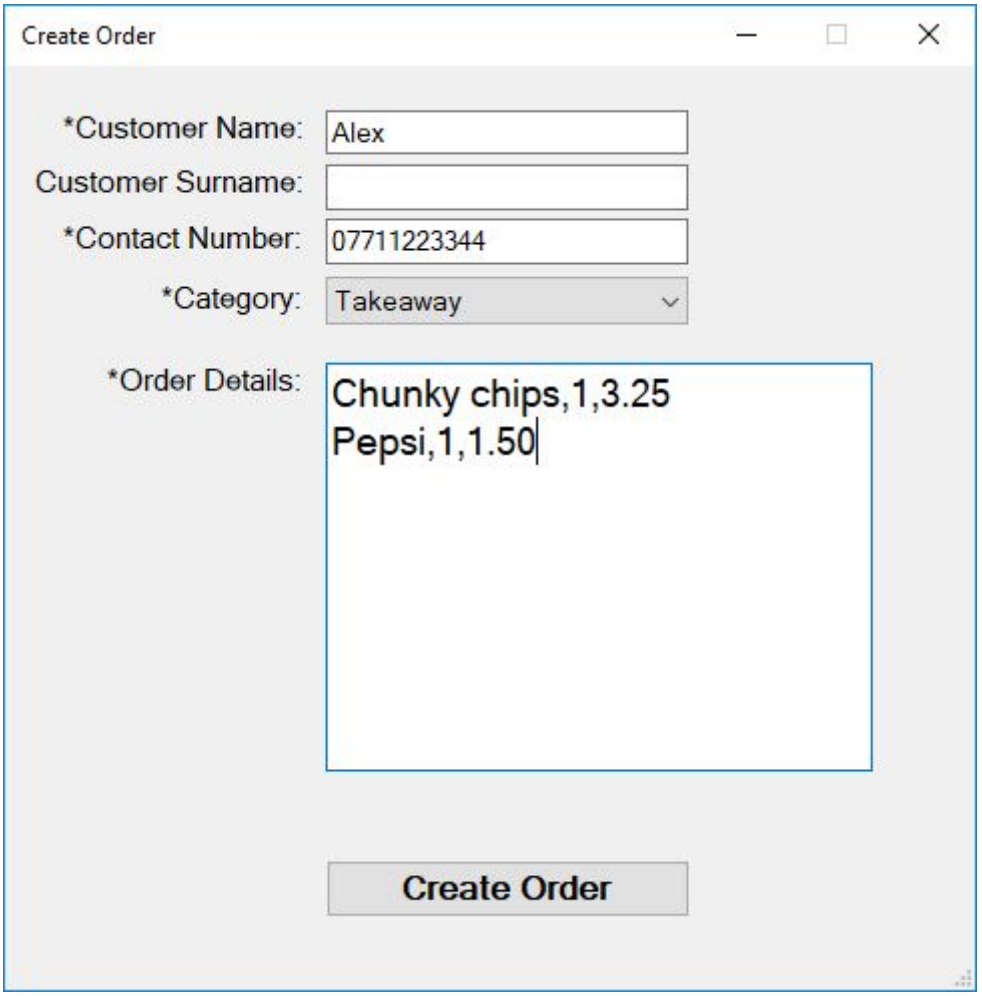
Improving the form by automatically calculating the total price

Since the contents can be parsed successfully, there is no need for the user to manually input the total price, since this can be automated, saving time and hassle.

This can be done simply by calling the `CalculateTotal()` method that has already been implemented in the Order class. Hence, the total is not needed as parameter to the overloaded constructor, so it looks like this:

```
37  // Class constructor from manual input
38  public Order(string custFirstName, string custSurname, string custTelephone, string ordCategory, List<Tuple<string, int, float>> ordContents)
39  {
40      id = ((int)((DateTime.UtcNow.Subtract(new DateTime(1970, 1, 1))).TotalSeconds)).ToString().Substring(5,5);
41      timestamp = DateTime.UtcNow.ToLocalTime();
42      category = ordCategory;
43      contents = ordContents;
44      total = CalculateTotal();
45      customer = new Customer(custFirstName, custSurname, custTelephone);
46      completed = false;
47  }
```

Therefore, the try-catch block in the validation algorithm can be removed and the text box in the form can be removed, so the new form window looks like this:



Polling the API for live updates using multi-threading

To continuously poll the server in the background without disrupting the functionality of the user interface, a separate thread must be used. The function to be execute in the thread is shown below:

```
208     // poll the API server in the background
209     private void PollServerThread()
210     {
211         // continuous loop
212         while (true)
213         {
214             Thread.Sleep(Orders.REFRESH_RATE); // pause for the refresh rate
215             bool exists = false;
216 
217             Order latestOrder = Orders.GetLatestOrder();  // get latest order from API
218 
219             // check if the latest order is already in the current orders list
220             foreach (var order in CurrentOrdersList)
221             {
222                 if (order.GetID() == latestOrder.GetID())
223                 {
224                     exists = true;
225                     Console.WriteLine();
226                     break;
227                 }
228 
229             }
230 
231             // if the latest order has not been added yet, add it
232             if (exists == false)
233             {
234                 // Check if the timestamp is today
235                 if (latestOrder.GetTimestamp().Date == DateTime.Today)
236                 {
237                     if (ordersListBox.InvokeRequired)
238                     {
239                         AddNewOrderCallback c = new AddNewOrderCallback(AddNewOrderToListBox);
240                         Invoke(c, new object[] { latestOrder });
241                         CurrentOrdersList.Add(latestOrder);
242                     }
243                     else
244                     {
245                         AddNewOrderToListBox(latestOrder);
246                         CurrentOrdersList.Add(latestOrder);
247                     }
248                 }
249             }
250 
251         }
252     }
```

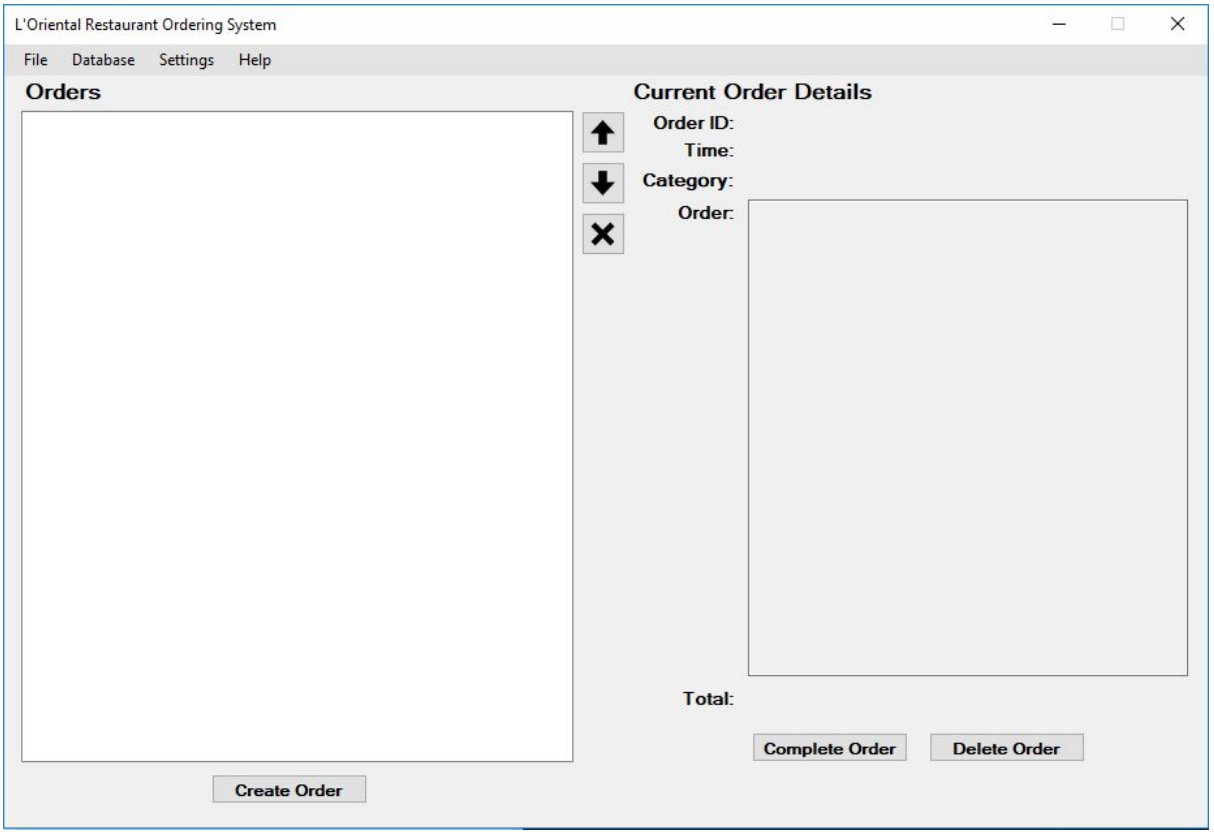
Since the API must be continuously polled, the function is encased by an infinite while loop. Then, the execution of the thread is halted for the amount of time given by the refresh rate. Then, a call is made to the API using the `GetLatestOrder()` method from the Orders.cs file to retrieve the latest order. Next, a for loop is used to check whether or not the order is already in the list of current orders.

If the order does not exist, then a check is made to see if the order is from the current day. If so, the order is finally added to the `CurrentOrdersList` and the listbox. The `InvokeRequired` attribute and the `AddNewOrderCallback()` method is used to safely access variables that exist in other threads. Without these extra bits of code, the data may be corrupted/overwritten unexpectedly.

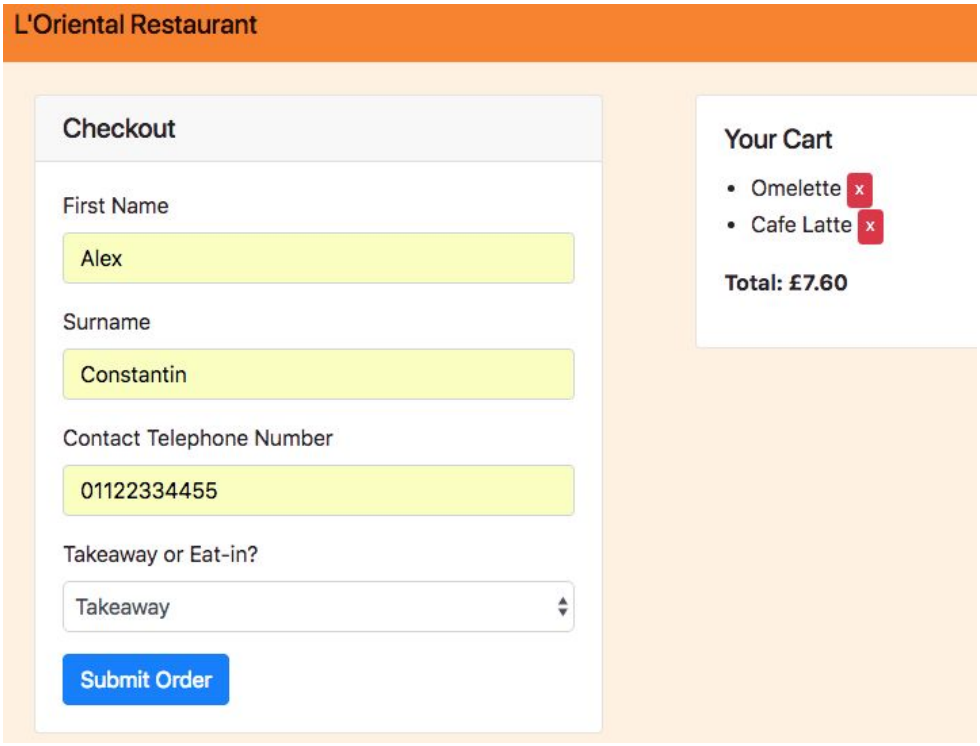
To start the execution of the thread, a new thread object is created and started in the constructor of the MainWindow.cs file, which is the start of the whole program, shown below:

```
34    // start background thread for polling the server
35    Thread backgroundThread = new Thread(new ThreadStart(PollServerThread));
36    backgroundThread.IsBackground = true;
37    backgroundThread.Start();
```

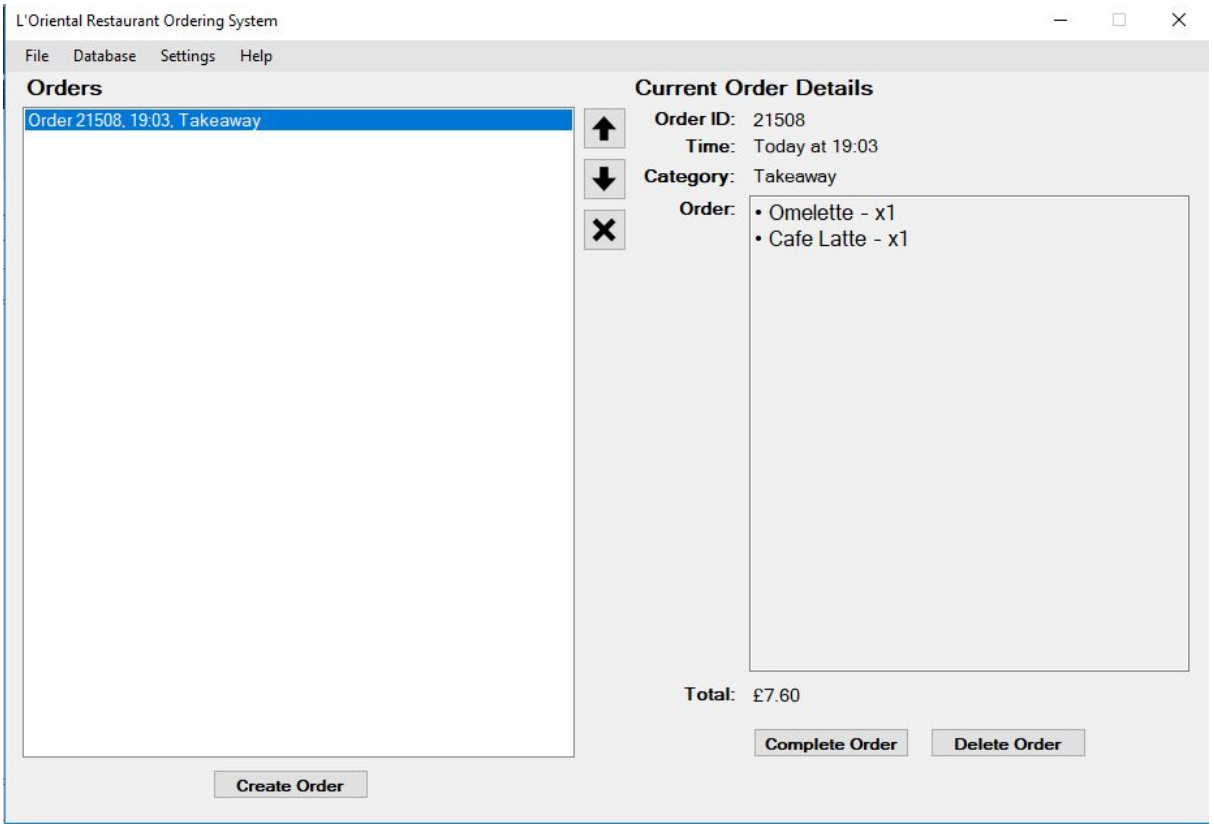
An empty main screen looks like so:



Adding a new order via the website:



After a few seconds, the order appears on screen successfully!



Generating an SQL statement from the Database Search Window

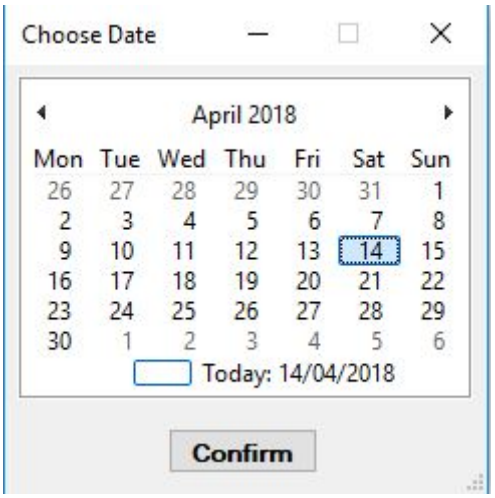
In order to search for past orders using the Database Search Window, an SQL query must be constructed based on the inputted form data. Then, this query can be sent to the API, where the database will be queried and the results will be returned.

The first step is two gather the dates selected from the calendar prompts. This can be done with the following event handlers for both buttons:

```
126     // get the start date from the calendar control
127     private void chooseStartDateButton_Click(object sender, EventArgs e)
128     {
129         CalendarPopUpWindow calendar = new CalendarPopUpWindow(null);
130         var result = calendar.ShowDialog();
131 
132         if (result == DialogResult.OK)
133         {
134             // get the start date
135             StartDate = DateTime.ParseExact(calendar.Date, "dd/MM/yyyy", null);
136             chooseStartDateButton.Text = calendar.Date;
137         }
138     }
139 
140     // get the end date from the calendar control
141     private void chooseEndDateButton_Click(object sender, EventArgs e)
142     {
143         // pass in the start date to limit the range
144         CalendarPopUpWindow calendar = new CalendarPopUpWindow(StartDate.ToShortDateString());
145         var result = calendar.ShowDialog();
146 
147         if (result == DialogResult.OK)
148         {
149             // get the end date
150             EndDate = DateTime.ParseExact(calendar.Date, "dd/MM/yyyy", null);
151             chooseEndDateButton.Text = EndDate.ToShortDateString();
152         }
153     }
```

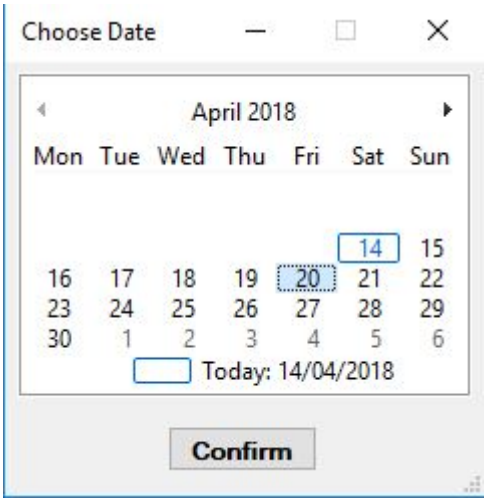
The first button event handler gets the date from the calendar prompt, and sets the button's text to the date selected.
The second button requires the start date to be passed in as a parameter so that the start date is the minimum selected date.

The calendar pop up window looks like so:



If the 14th of April 2018 is selected after clicking the Confirm button, it is added to the text value of the button:

Now, if an end date is chosen, the minimum possible date selected is created:



And finally, the date range is shown:

Once the date is selected, the SQL query can now be built, using the `ComposeQuery()` function below:

```
26  // Build the SQL query from the data inputted
27  private string ComposeQuery()
28  {
29      string[] conditions = new string[2];
30      string query = "SELECT * FROM Orders WHERE";
31
32      string startTimestamp = ((Int32)(StartDate.Subtract(new DateTime(1970, 1, 1))).TotalSeconds).ToString();
33      string endTimestamp = ((Int32)(EndDate.Subtract(new DateTime(1970, 1, 1))).TotalSeconds).ToString();
34
35      // date condition
36      if (chooseStartDateButton.Text != "Choose")
37      {
38          if (chooseEndDateButton.Text != "Choose")
39              conditions[0] = " (timestamp BETWEEN " + startTimestamp + " AND " + endTimestamp + ") AND";
40          else
41              conditions[0] = " (timestamp > " + startTimestamp + ") AND";
42      }
43      else
44      {
45          if (chooseEndDateButton.Text != "Choose")
46              conditions[0] = " (timestamp < " + endTimestamp + ") AND";
47          else
48              conditions[0] = "";
49      }
50
51      // category condition
52      if (OrderCategory != "")
53          conditions[1] = " (category = '" + OrderCategory + "')";
54      else
55          conditions[1] = "";
```

Firstly, an array of strings called conditions is initialised. This will hold all the conditions for the WHERE statement in the query.
Then, the start and end dates are converted into unix timestamps, so that comparison operators can be used, because the timestamp stored has a numeric data type.
From lines 36 to 49, multiple if statements are used to cover all possible date selections:

- Start date and End date
- Start date but no End date
- End date but no Start date
- No Start date or end date

The BETWEEN SQL operator is used for when both dates are selected. When only one date is selected, comparison operators are used.
On line 52, the condition for the order category is built. If no category is selected, the condition is an empty string.

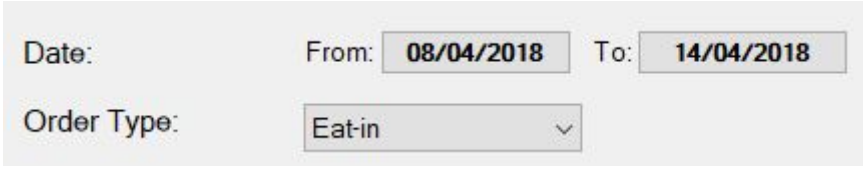
Finally, a for loop is used to build the entire SQL query by appending the WHERE conditions onto the initial query string, defined on line 30.

```
58      // build the query by adding all the where conditions
59      foreach (string clause in conditions)
60      {
61          if (clause != "")
62              query += clause;
63      }
64      query += ";";   // add a semicolon
65      return query;
66  }
```

White box testing for all possible query cases

The justification for this testing step is to ensure all queries are correctly constructed, because database operations are usually safety-critical. Also, since there are a limited amount of cases, white box testing will be appropriate.

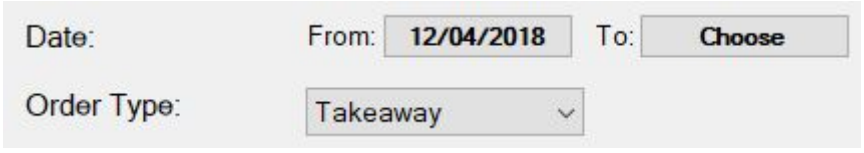
Start date, end date and category:




```
Output:
SELECT * FROM Orders WHERE (timestamp BETWEEN 1523145600 AND 1523664000) AND (category = 'Eat-in');
```

Working as expected: Yes

Start date, no end date and category:




```
Output:
SELECT * FROM Orders WHERE (timestamp > 1523491200) AND (category = 'Takeaway');
```

Working as expected: Yes

Conclusion for the first client program prototype

That concludes the first prototype of the kitchen client program. Most of the functionality has now been implemented, including the ability to create orders via manual input and displaying orders on screen

To improve the current prototype, the web API must be developed further in order to include more network and database features, such as committing new orders to the database or making custom queries.

Part III: Extending the web API and database operations

Sending new orders to the database via the API

In order to commit new orders to the database which are created using the form on the kitchen client program, the new order object must be converted into JSON for transmission over the network. A new method in the Orders.cs file has been defined to do this:

```
72  // Calls the API endpoint at /orders/create/
73  public static bool CreateOrder(Order newOrder, Customer newCustomer)
74  {
75      bool success = false;
76
77      // setup request and headers
78      var request = (HttpWebRequest)WebRequest.Create(ORDERS_API_BASE_URL + "create/");
79      request.ContentType = "application/json";
80      request.Method = "POST";
81
82      // write json data to request
83      using (var streamWriter = new StreamWriter(request.GetRequestStream()))
84      {
85          string unixTimestamp = ((Int32)(newOrder.GetTimestamp().Subtract(new DateTime(1970, 1, 1))).TotalSeconds).ToString();
86          string jsonContents = JsonConvert.SerializeObject(newOrder.GetContents());
87
88          string json = JsonConvert.SerializeObject (new
89                          {
90                              firstName = newCustomer.GetFirstName(),
91                              surname = newCustomer.GetSurname(),
92                              telephone = newCustomer.GetTelephone(),
93                              customerID = newCustomer.GetID(),
94                              orderID = newOrder.GetID(),
95                              timestamp = unixTimestamp,
96                              category = newOrder.GetCategory(),
97                              contents = jsonContents
98          });
99
100         streamWriter.Write(json);
101         streamWriter.Flush();
102         streamWriter.Close();
103     }
```

The first part of the function sets up a JSON POST request to the `/orders/create/` endpoint on the API, implemented later. This is done in lines 77 to 79. Then, the object is serialised into a JSON string, which is written to the request. Each attribute in the string maps to the attributes of an order and its related customer, to make the transmission all in one and preserve the database integrity.

In the second part of the function, shown below, the response is obtained to make sure the server has successfully added the order and customer to the database:

```
102         // get response to confirm it was successfully created
103         var response = (HttpWebResponse) request.GetResponse();
104         using (var streamReader = new StreamReader(response.GetResponseStream()))
105         {
106             dynamic responseData = JsonConvert.DeserializeObject(streamReader.ReadToEnd());
107             Console.WriteLine(responseData);
108             if (responseData.data == "Success")
109             {
110                 success = true;
111             }
112         }
113
114         return success;
115     }
```

Now, we must create the endpoint on the web server. The path will be located at `/orders/create/` as mentioned before.

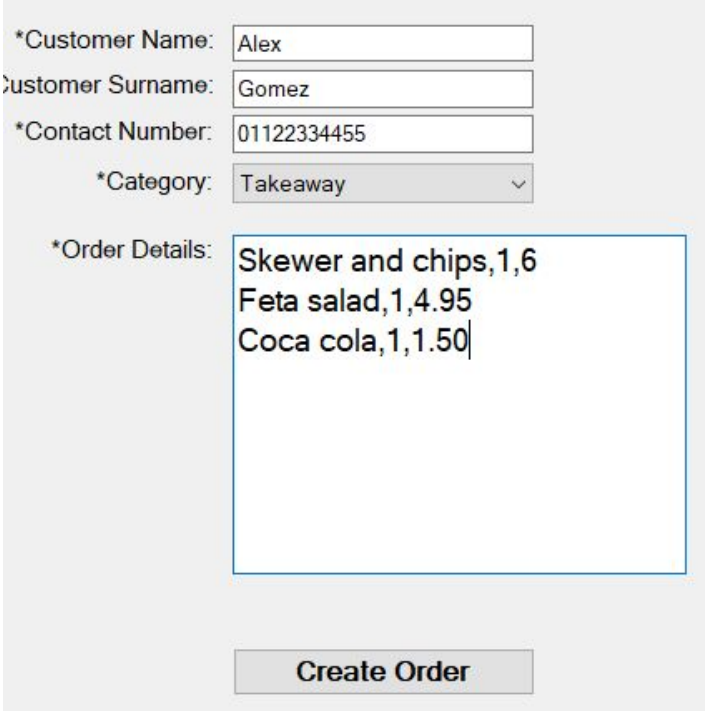
```
206 @app.route('/orders/create/', methods=['POST'])
207 def create_order_and_customer():
208     try:
209         data = request.get_json()
210 
211         # create customer object from data
212         newCustomer = Customer(data["firstName"], data["surname"],
213                                data["telephone"], data["customerID"])
214 
215 
216         # create order object from data
217         newOrder = Order(data["contents"], data["category"], data["customerID"],
218                          data["orderID"], data["timestamp"])
219 
220         add_order(newOrder)
221         add_customer(newCustomer)
222 
223         return jsonify({"data": "Success"})
224     except:
225         return jsonify({"data": "Invalid request/error occured"})
```

In this case, the JSON data is received from the kitchen client and it is used to create a new customer and order object. Finally, it is added to the database with add_order() and add_customer() operations, which are implemented below, in the operations.py file:

```
3  def add_order(orderObject):
4      conn = sqlite3.connect("database.db")
5      cur = conn.cursor()
6      cur.execute("INSERT INTO orders (id, timestamp, contents, category, customerId) VALUES (?,?,?,?,?)",
7                  (orderObject.id, orderObject.timestamp, str(orderObject.contents), orderObject.category, orderObject.customerId))
8      conn.commit()
9      conn.close()
10 
11 
12 def add_customer(custObject):
13     conn = sqlite3.connect("database.db")
14     cur = conn.cursor()
15     cur.execute("INSERT INTO customers (id, firstName, surname, telephone) VALUES (?,?,?,?)",
16                 (custObject.id, custObject.firstName, custObject.surname, custObject.telephone))
17     conn.commit()
18     conn.close()
```

Both operations are self-explanatory: they execute the SQL statements shown above with the order and customer attributes as parameters. Then, they are committed to make sure changes are saved.

To test this, I have created a new order and the JSON data on the right has been generated:




```
{
   "telephone":"01122334455",
   "customerID":"47723",
   "firstName":"Alex",
   "contents":[
      {
         "Item1":"Skewer and chips",
         "Item2":1,
         "Item3":6.0
      },
      {
         "Item1":"Feta salad",
         "Item2":1,
         "Item3":4.95
      },
      {
         "Item1":"Coca cola",
         "Item2":1,
         "Item3":1.5
      }
   ],
   "orderID":"47468",
   "category":"Takeaway",
   "timestamp":"1523651068",
   "surname":"Gomez"
}
```


The JSON is successfully received by the server and the order and customer objects are added to the database.

Also, if an API call is sent to get the latest order, it will appear as JSON, meaning the entire system is working correctly:


```
{
  "category": "Takeaway",
  "contents": [
    {
      "name": "Skewer and chips",
      "price": 6.0,
      "quantity": 1
    },
    {
      "name": "Feta salad",
      "price": 4.95,
      "quantity": 1
    },
    {
      "name": "Coca cola",
      "price": 1.5,
      "quantity": 1
    }
  ],
  "customerID": 49698,
  "orderID": 49443,
  "timestamp": "1523653043"
},
```

Deleting orders from the database via the API

Another extension to the API is to be able to delete orders from the database. Currently, whenever the “Delete Order” button is pressed on the client program, the order is only deleted from the listbox and the current orders list. To permanently remove an order, another method has been added to the Orders.cs file:

```
121  // Deletes an order by posting the ID to the /orders/delete/ endpoint on the API
122  public static bool DeleteOrderFromServer(Order order)
123  {
124      bool success = false;
125
126      // setup request and headers
127      var request = (HttpWebRequest)WebRequest.Create(ORDERS_API_BASE_URL + "delete/");
128      request.ContentType = "application/json";
129      request.Method = "POST";
130
131      // write json data to request
132      using (var streamWriter = new StreamWriter(request.GetRequestStream()))
133      {
134          string json = JsonConvert.SerializeObject(new
135          {
136              id = order.GetID()
137          });
138
139          streamWriter.Write(json);
140          streamWriter.Flush();
141          streamWriter.Close();
142      }
143
144      // get response to confirm it was successfully deleted
145      var response = (HttpWebResponse)request.GetResponse();
146      using (var streamReader = new StreamReader(response.GetResponseStream()))
147      {
148          dynamic responseData = JsonConvert.DeserializeObject(streamReader.ReadToEnd());
149          Console.WriteLine(responseData);
150          if (responseData.data == "Success")
151          {
152              success = true;
153          }
154      }
155
156      return success;
157  }
```

The first part of the function sets up a JSON POST request to the `/orders/delete/` endpoint on the API, implemented later. This is done in lines 127 to 129. Then, a JSON object containing only the order ID is written to the request and sent.
Finally, a response is obtained and it checks whether the response data contains “Success”, to indicate that the order was successfully deleted.

The JSON data sent is simply:

```
{
    "id":49443
}
```

Now, we must create the endpoint on the web server. The path will be located at `/orders/delete/` as mentioned before.

```
194 # delete order based on the order id
195 @app.route('/orders/delete/', methods=['POST'])
196 def delete_order():
197     try:
198         data = request.get_json()   # get the json data
199         delete_order_by_id(data["id"])      # call the database operation
200         return jsonify({"data": "Success"})
201     except:
202         return jsonify({"data": "Invalid request/error occured"})
```

At this endpoint, the JSON data is received and the order ID is passed as a parameter into the `delete_order_by_id()` database operation function, which is shown below:

```
56 def delete_order_by_id(order_id):
57     conn = sqlite3.connect("database.db")
58     cur = conn.cursor()
59     cur.execute('DELETE FROM orders WHERE id=(?)', (order_id,))
60     conn.commit()
61     conn.close()
```

The function is very straightforward: it executes an SQL DELETE command using the order ID as the parameter in the WHERE clause. The changes are then committed.

Querying the database for past orders via the API

In order to send a custom query to the database using the Database Search Window, another API endpoint must be created. It shall be found at `/orders/query/`. Once again, it is a POST-only endpoint. The definition for the endpoint is found below:

```
233 # query the database using a custom query
234 @app.route('/orders/query/', methods=['POST'])
235 def query_database():
236     try:
237         data = request.get_json()
238         orders = custom_query(data["query"])  # get orders
239         response = []
240         for o in orders:
241             response.append(serialise_order(o))  # serialise orders
242         return jsonify(response)
243     except:
244         return jsonify({"data": "Invalid request/error occured"})
```

It gets the JSON data from the request, and uses the query attribute as a parameter passed into the custom_query() function, which is the database operation function, implemented next. Then, the returned orders are serialised and converted into JSON for the response.

The database operation is defined in the operations.py file:

```
63 def custom_query(sql):
64     conn = sqlite3.connect("database.db")
65     cur = conn.cursor()
66     orders = []
67     for order in cur.execute(sql):    # execute the custom sql query
68         orders.append(tuple(order))   # add orders to list
69     conn.close()
70     return orders
```

Rather than executing a defined SQL query, it simply executes the statement it received through the JSON, which was built in the kitchen client program. Any orders matching the query are returned.

For example, POSTing the following query to /orders/query/ will return the following orders:

```
"SELECT * FROM Orders WHERE (timestamp BETWEEN 1523404800
AND 1523577600) AND (category = 'Eat-in');"
```

Hence, the API endpoint is working correctly.


```
[
    {
        "category": "Eat-in",
        "contents": [
            {
                "name": "Roast chicken panini",
                "price": 4,
                "quantity": 1
            },
            {
                "name": "Cafe Latte",
                "price": 1.6,
                "quantity": 1
            }
        ],
        "customerID": 77745,
        "orderID": 77491,
        "timestamp": "1523485053"
    },
    {
        "category": "Eat-in",
        "contents": [
            {
                "name": "Veggi platter",
                "price": 6,
                "quantity": 1
            }
        ],
        "customerID": 82544,
        "orderID": 82290,
        "timestamp": "1523572548"
    }
]
```

In order to POST the JSON data, another function must be coded in the Orders.cs file. It is shown below:

```
159  // Calls the API endpoint at /orders/query/
160  public static List<Order> QueryDatabase(string sqlQuery)
161  {
162      List<Order> orders = new List<Order>();
163
164      // setup request and headers
165      var request = (HttpWebRequest)WebRequest.Create(ORDERS_API_BASE_URL + "query/");
166      request.ContentType = "application/json";
167      request.Method = "POST";
168
169      // write json data to request
170      using (var streamWriter = new StreamWriter(request.GetRequestStream()))
171      {
172          string json = JsonConvert.SerializeObject(new{ query = sqlQuery });
173
174          streamWriter.Write(json);
175          streamWriter.Flush();
176          streamWriter.Close();
177      }
178
179      // get the orders in the response
180      dynamic response = (HttpWebResponse)request.GetResponse();
181      using (var streamReader = new StreamReader(response.GetResponseStream()))
182      {
183          dynamic responseData = JsonConvert.DeserializeObject(streamReader.ReadToEnd());
184          foreach (dynamic orderData in responseData)
185          {
186              Order newOrder = new Order(orderData);
187              orders.Add(newOrder);
188          }
189      }
190
191      return orders;
192  }
```

It sets up a JSON POST request to the /orders/query/ endpoint, where the “query” attribute contains the SQL statement passed in as a parameter (this will be used later). The JSON is written and the request is sent.

In lines 180 to 191, the JSON response is decoded and deserialised into Order objects, which will be returned to the results list box in the Database Search Window.

Displaying the specific details of a past order

As a continuation to the Database Search Window, the user should be able to view the order's specific details, rather than just the information shown on the list box. To do this, a new window has been created, called PastOrderDetails. The code for the class is shown here:

```
11      private Order order;
12
13      public PastOrderDetails(Order specificOrder)
14      {
15          InitializeComponent();
16          order = specificOrder;
17          DisplayOrderDetails();
18      }
19
20      // Load the details on screen
21      private void DisplayOrderDetails()
22      {
23          orderIDLabel.Text = order.GetID();
24          timeLabel.Text = order.GetTimestamp().ToString();
25          categoryLabel.Text = order.GetCategory();
26          totalLabel.Text = "£" + order.GetTotal().ToString("0.00"); // show 2 decimal places
27          contentsTextBox.Text = order.GetDisplayContents();
28          customerNameLabel.Text = order.GetCustomer().GetFullName();
29          telephoneLabel.Text = order.GetCustomer().GetTelephone();
30      }
```

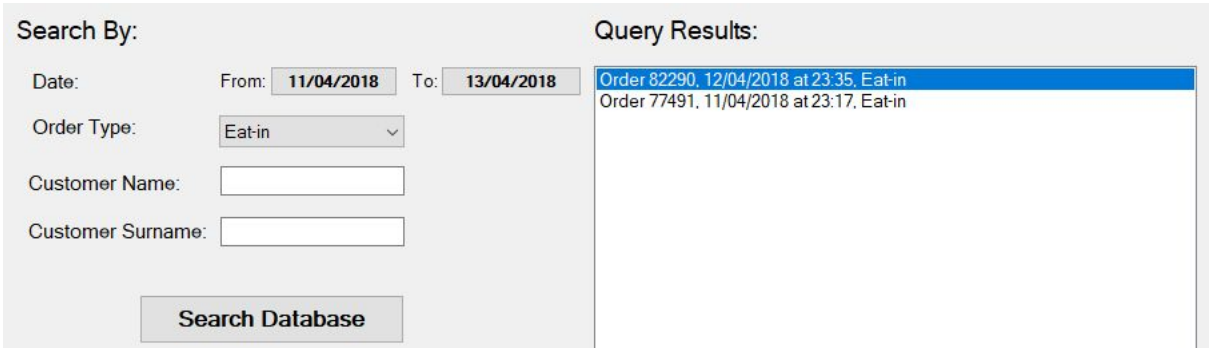
In the constructor, the order attribute is initialised to the order object passed in as a parameter, and the details are displayed. This is done using the DisplayOrderDetails() method, which sets the text value of the labels to the selected order's attributes:

To show this new window, an event handler is used for when the list box is clicked:

```
108     // Open a window containing the details for an order from the list when clicked
109     private void resultsListBox_DoubleClick(object sender, EventArgs e)
110     {
111         ListBox lb = sender as ListBox;
112         if (lb != null)
113         {
114             try
115             {
116                 // Get the order id from the name on the item display
117                 string selectedOrderID = lb.SelectedItem.ToString().Substring(6, 5);
118
119                 // Search the current orders for the relevant order object using the order id
120                 foreach (Order order in ResultsOrdersList)
121                 {
122                     if (selectedOrderID == order.GetID())
123                     {
124                         // open a window containg the specific details for the selected order
125                         PastOrderDetails pastOrderWindow = new PastOrderDetails(order);
126                         pastOrderWindow.ShowDialog();
127                         break;
128                     }
129                 }
130
131             }
132             catch {}
133
134         }
135         lb.ClearSelected();
136     }
```

The event handler works by getting the selected order's ID and then searching the results list for the relevant order object. When found, the Past Order Window is opened with the selected order object passed in as a parameter.

To test this new window, when clicking the highlighted order below:



The following window pops up with the specific order details:

It is working as expected.

Conclusion for the second system prototype

That concludes the extension to the web API, which is now complete. As for the kitchen client program, its first prototype has also been improved upon, however there are a few more features yet to implement, which will arrive in the third and final prototype.

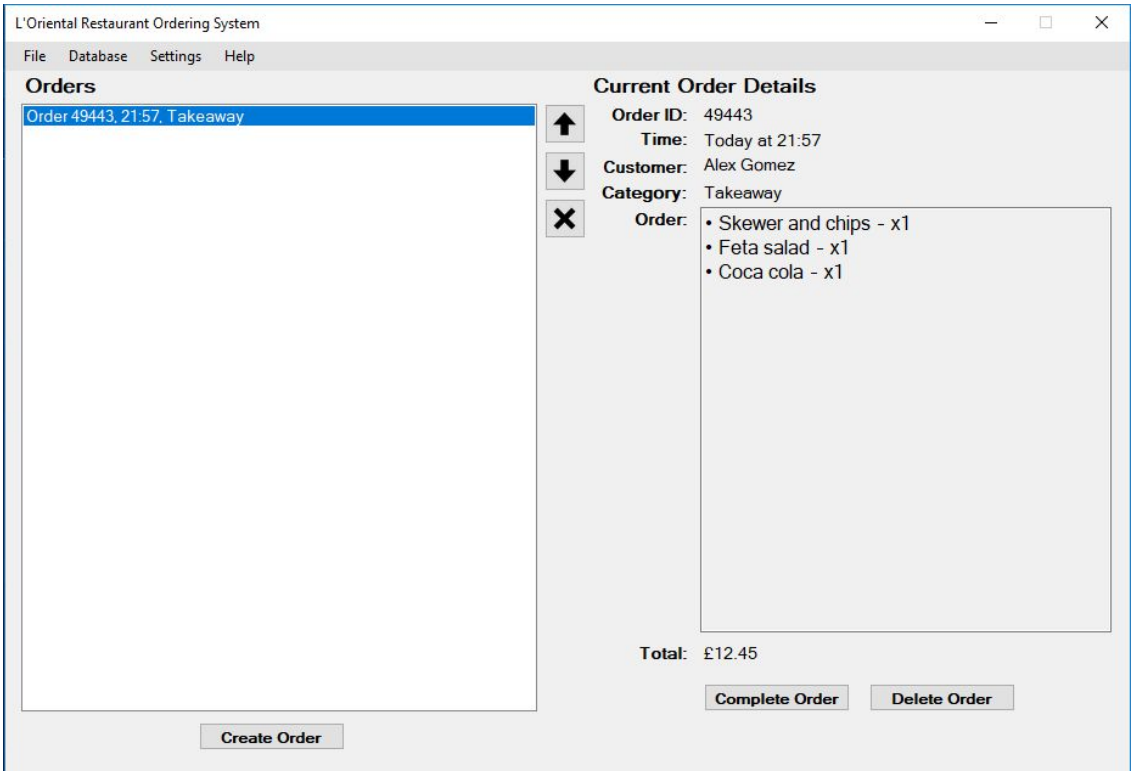
Part IV: Finalising the kitchen client program

Displaying the customer details on the main window

In order to provide more details to user about the order, it is probably a good idea to include the customer's name in the details panel. To do this, only a single line should be added to the `ViewSelectedOrderDetails()` function in MainWindow.cs:

```
97  private void ViewSelectedOrderDetails()
98  {
99      // Check if an item has been selected. If not, stop
100     if (ordersListBox.SelectedItem == null || ordersListBox.SelectedIndex < 0)
101         return;
102
103     currentOrderTable.Enabled = true;
104
105     // Get the order id from the name on the item display
106     string currentItemName = ordersListBox.SelectedItem.ToString();
107     string currentOrderID = currentItemName.Substring(6, 5);
108
109     // Search the current orders for the relevant order object using the order id
110     foreach (Order order in CurrentOrdersList)
111     {
112         if (currentOrderID == order.GetID())
113         {
114             // Fill out the details panel with the order's attributes
115             orderIDLabel.Text = currentOrderID;
116             timeLabel.Text = "Today at " + order.GetTimestamp().ToShortTimeString();  // shows only hh:mm format
117             categoryLabel.Text = order.GetCategory();
118             customerNameLabel.Text = order.GetCustomer().GetFullName();
119             totalLabel.Text = "£" + order.GetTotal().ToString("0.00"); // show 2 decimal places
120             contentsTextBox.Text = order.GetDisplayContents();
121             break;
122         }
123     }
124 }
```

The new window looks like so:



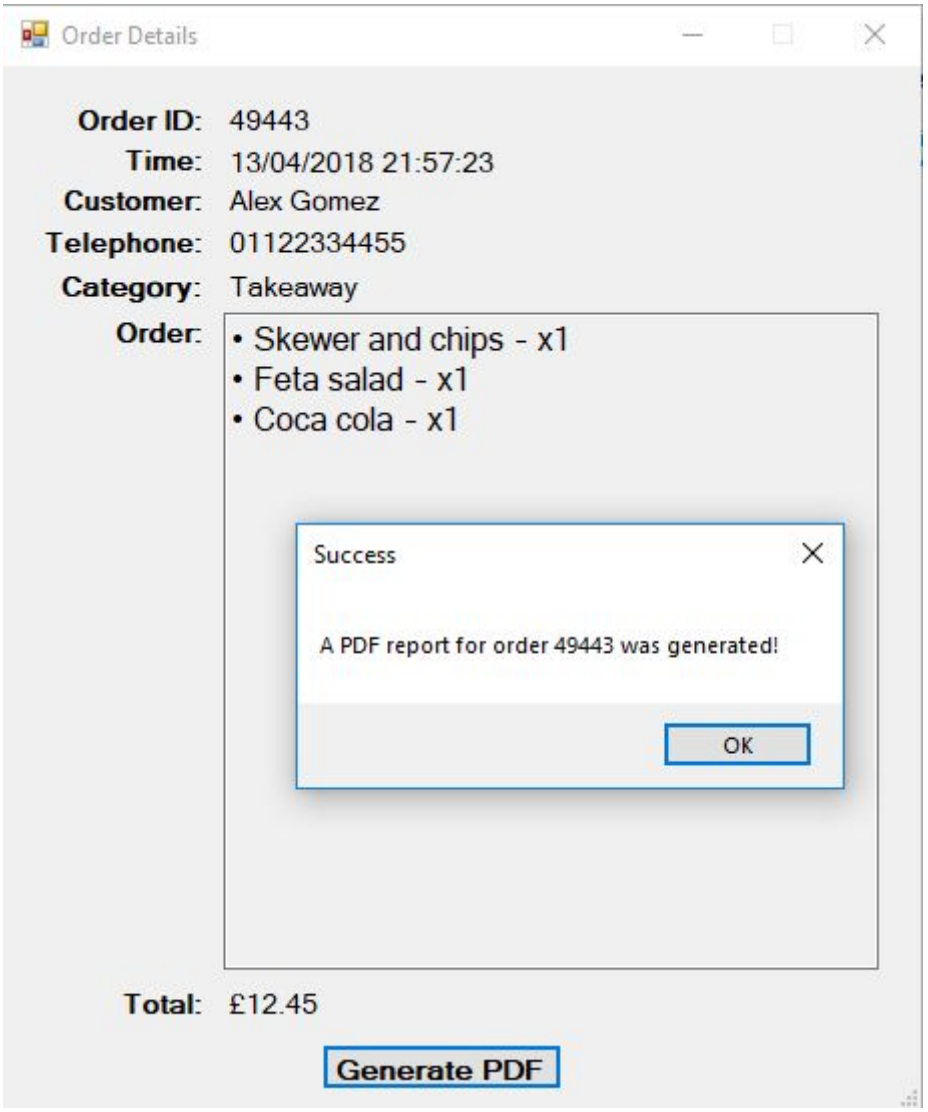
Generating a PDF of an order's details

In the case a PDF report of an order is desired to avoid querying the database repeatedly, the following function generates a PDF. A new button called "Generate PDF" has been added to the PastOrderDetails window and its event handler is shown below:

```
32  // Generate a PDF with the order's details
33  private void generatePDFButton_Click(object sender, EventArgs e)
34  {
35      Document doc = new Document(PageSize.A4.Rotate());
36      try
37      {
38          PdfWriter.GetInstance(doc, new FileStream("pdfs/order"+order.GetID()+".pdf", FileMode.Create));
39
40          BaseFont font = BaseFont.CreateFont(BaseFont.HELVETICA_BOLD, BaseFont.CP1250, false);
41          BaseFont font2 = BaseFont.CreateFont(BaseFont.HELVETICA, BaseFont.CP1250, false);
42
43          Font title = new Font(font, 20);
44          Font labels = new Font(font2, 14);
45
46          doc.Open();
47          doc.Add(new Chunk("Details for order number: " + order.GetID(), title));
48
49          doc.Add(new Paragraph("Customer Name: " + order.GetCustomer().GetFullName(), labels));
50          doc.Add(new Paragraph("Telephone: " + order.GetCustomer().GetTelephone(), labels));
51          doc.Add(new Paragraph("Date and time: " + order.GetTimestamp(), labels));
52          doc.Add(new Paragraph("Category: " + order.GetCategory(), labels));
53          doc.Add(new Paragraph("Order: ", labels));
54          doc.Add(new Paragraph(order.GetDisplayContents(), labels));
55          doc.Add(new Paragraph("Total: GBP" + order.GetTotal().ToString("0.00"), labels));
56
57          MessageBox.Show("A PDF report for order " + order.GetID() + " was generated!", "Success",
58                          MessageBoxButtons.OK, MessageBoxIcon.None);
59
60      }
61      catch
62      {
63          MessageBox.Show("An error occurred and the PDF could not be created.", "Error",
64                          MessageBoxButtons.OK, MessageBoxIcon.Error);
65      }
66      doc.Close();
67  }
```

Firstly, it creates a new empty PDF file in a relative /pdfs/ directory. Then, a couple of fonts are created and the data is written to the PDF in lines 49 to 55. A message box is also displayed to confirm the creation of the PDF. If an error occurs, a message box will also appear notifying of the error. Finally, the document is closed to avoid data corruption.

When the “Generate PDF” button is clicked, the following message box appears:



The actual PDF generated looks like so:

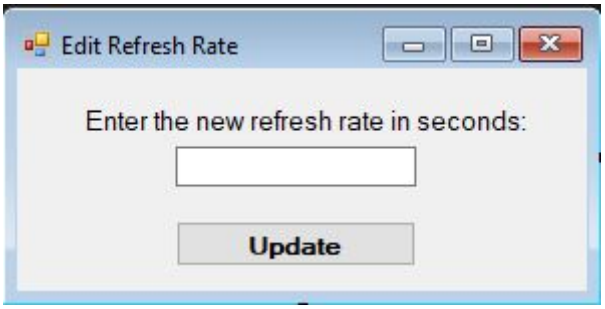
Modifying the polling refresh rate

In the case that the user would like to edit the polling refresh rate for whatever reason, for example, to reduce bandwidth usage, it would be a helpful feature to implement.

Currently, the refresh rate is set to 10 seconds (10 000 ms), defined in Orders.cs:

```
14      public static int REFRESH_RATE = 10*1000;
```

The prompt window to modify the refresh rate is shown below. It consists of an input text box and a submit button.



The code for the prompt consists of the event handler for when the button is clicked:

```
22  private void submitButton_Click(object sender, EventArgs e)
23  {
24      try
25      {
26          RefreshRate = Convert.ToUInt16(refreshRateTextBox.Text);
27          Console.WriteLine(RefreshRate);
28          DialogResult = DialogResult.OK; // confirm that the form has been succesfully completed
29          Close();
30      }
31      catch
32      {
33          MessageBox.Show("The value you entered for the refresh rate is invalid", "Invalid value",
34                          MessageBoxButtons.OK, MessageBoxIcon.Warning);
35      }
36
37  }
```

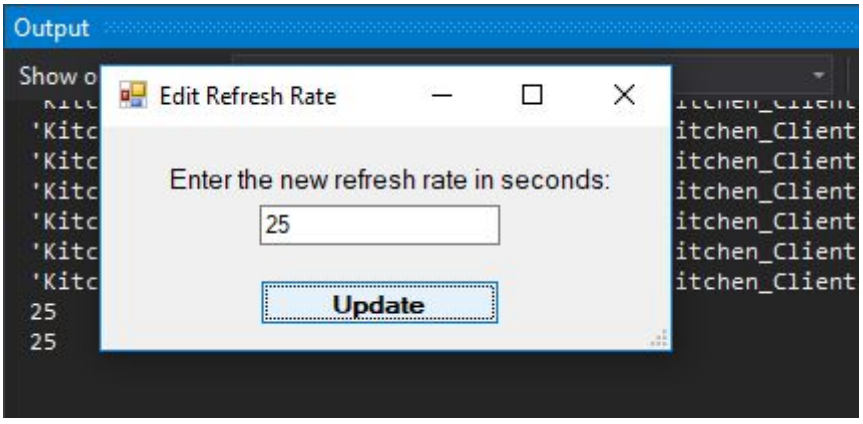
It attempts to convert the value inputted into an unsigned integer. If this fails, for example if a negative value is added or a string is given, a message box showing a warning will be displayed, since the code is encased by a try-catch block.

To actually open this prompt, an event handler must be written for the “Edit Refresh Rate” menu item in the menu bar. The code is very straightforward and shown below:

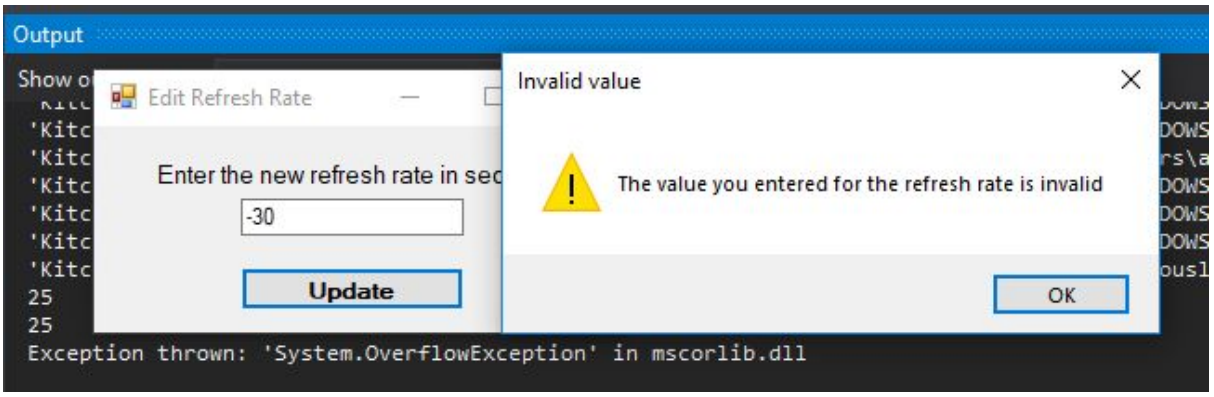
```
254     // Allows the user to edit the refresh rate
255     private void editRefreshRateToolStripMenuItem_Click(object sender, EventArgs e)
256     {
257         Prompt prompt = new Prompt();
258         var result = prompt.ShowDialog();
259 
260         // if the form was completed, add order to the list box
261         if (result == DialogResult.OK)
262         {
263             // Get the new refresh rate and update it
264             int newRefreshRate = prompt.RefreshRate;
265             Orders.REFRESH_RATE = newRefreshRate * 1000;
266             Console.WriteLine(Orders.REFRESH_RATE);
267         }
268     }
```

Testing the refresh rate prompt

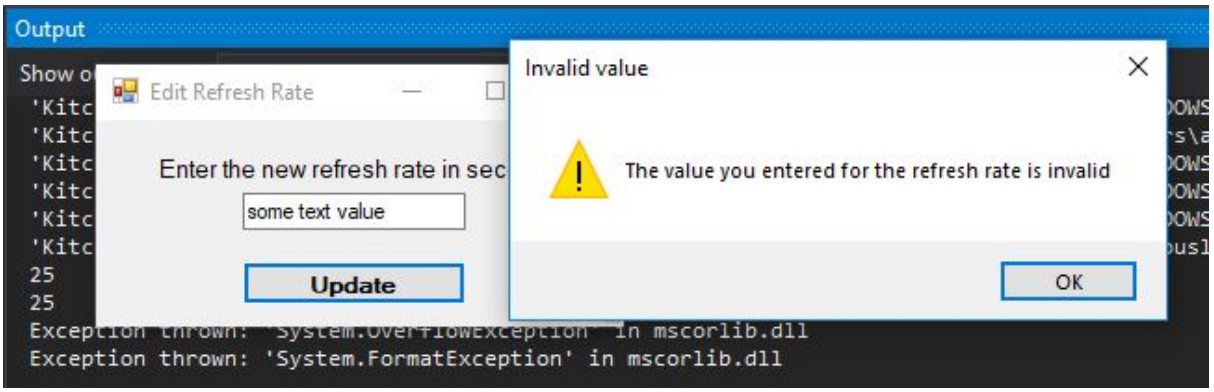
Valid data:



Invalid data (negative integer):



Invalid data (string value):



Remedial action for setting boundaries for extreme values:

The refresh rate must be between 10 seconds and 3 minutes, otherwise it is too fast and memory-consuming or too slow and inefficient.

Hence, the new improved event handle for the button click is below:

```
22  private void submitButton_Click(object sender, EventArgs e)
23  {
24      try
25      {
26          RefreshRate = Convert.ToUInt16(refreshRateTextBox.Text);
27          if (RefreshRate < 10 || RefreshRate > 180)
28              throw new Exception();
29
30          DialogResult = DialogResult.OK; // confirm that the form has been succesfully completed
31          Close();
32      }
33      catch
34      {
35          MessageBox.Show("The value you entered for the refresh rate is invalid", "Invalid value",
36                          MessageBoxButtons.OK, MessageBoxIcon.Warning);
37      }
38
39  }
```

The only change to this code is an extra if statement which checks that the new value is not less than 10 or greater than 180. Otherwise, an exception is thrown and the catch block is triggered, displaying the warning message box.

Here is some new test data, which confirms it is working properly:

30 => valid	-23 => invalid
10 => valid (boundary)	3 => invalid
180 => valid (boundary)	250 => invalid

Conclusion for the final prototype

That concludes the development of the solution. The final prototype of the kitchen client is now finished, with improved features and some which needed remedial action. In the next section, the solution shall be evaluated with reference to the Analysis and Design sections.

EVALUATION

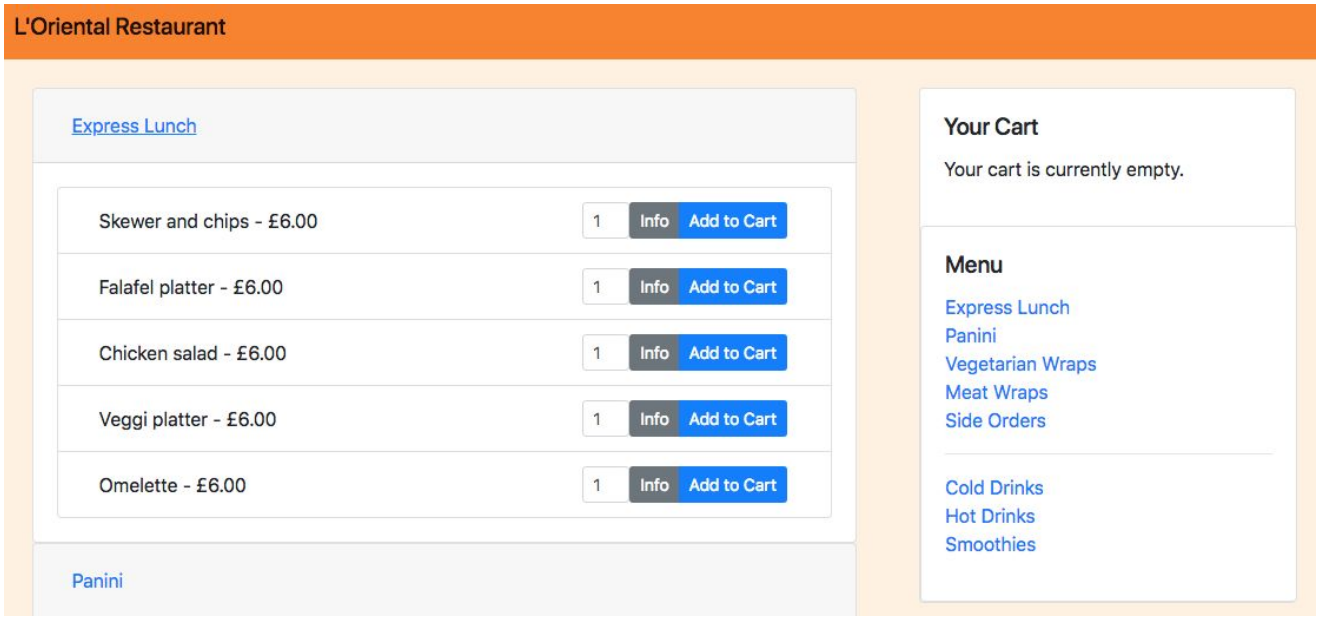
4.1. Testing to inform evaluation

Post-development testing for functionality

To evaluate the solution, further testing will be carried out to ensure the functionality of the system is correct and working as expected, annotated accordingly.

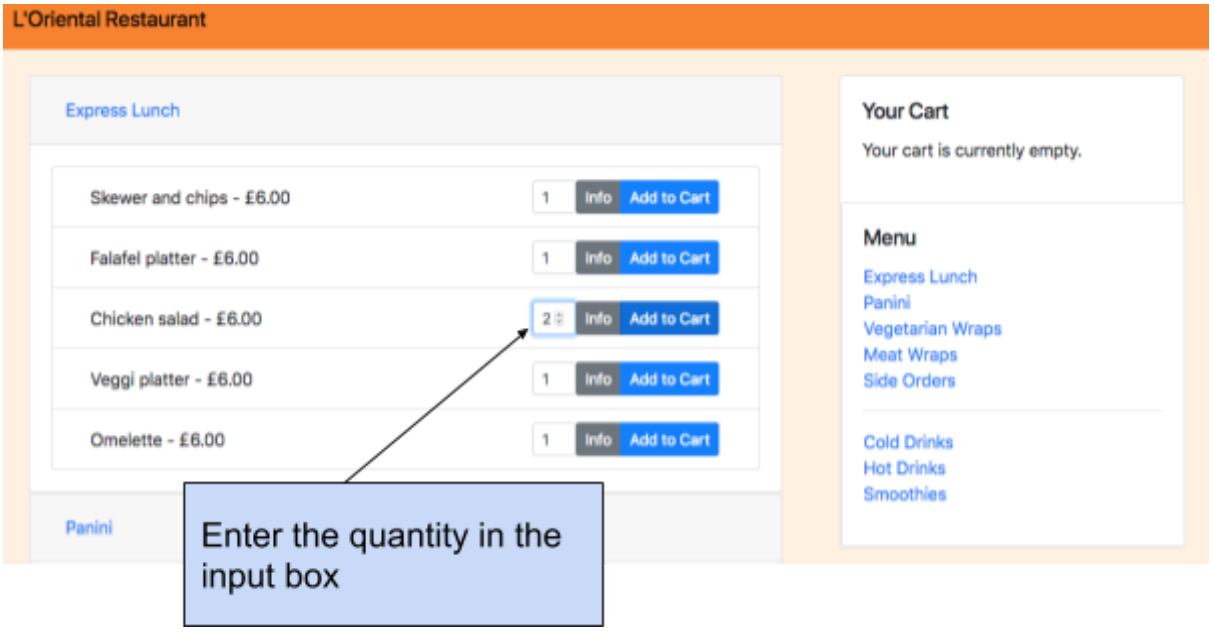
Testing the website's menu page functionality:

When entering the website for the first time:

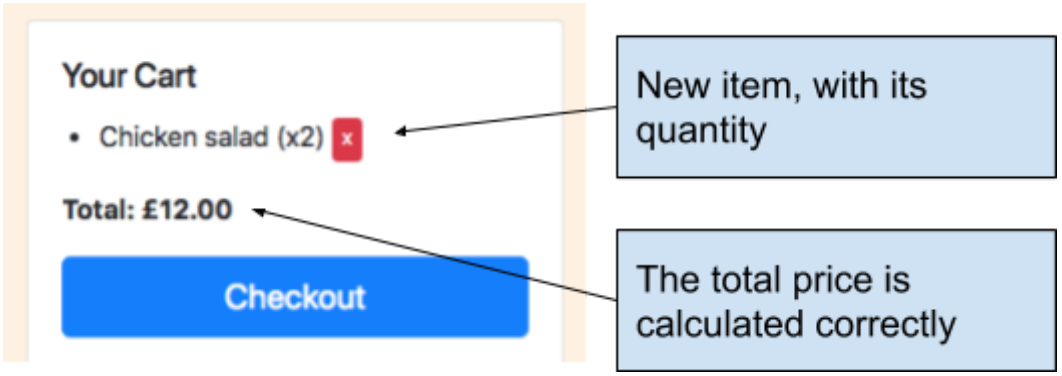


As expected, the menu items are rendered correctly under their corresponding tabs and the cart is empty.

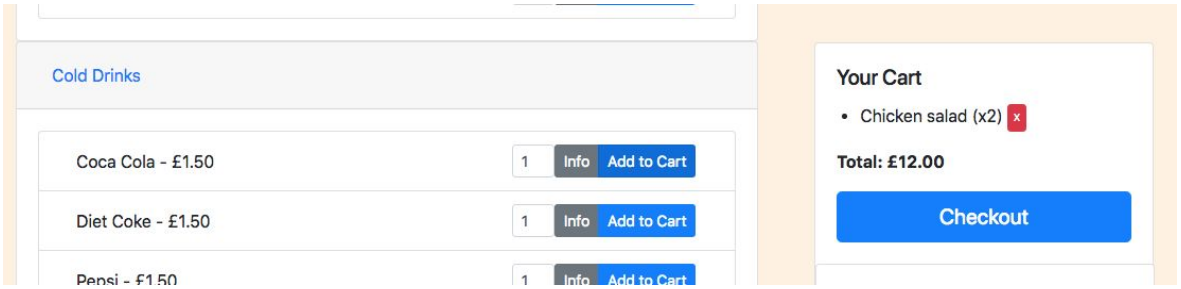
When adding an arbitrary item to the cart...



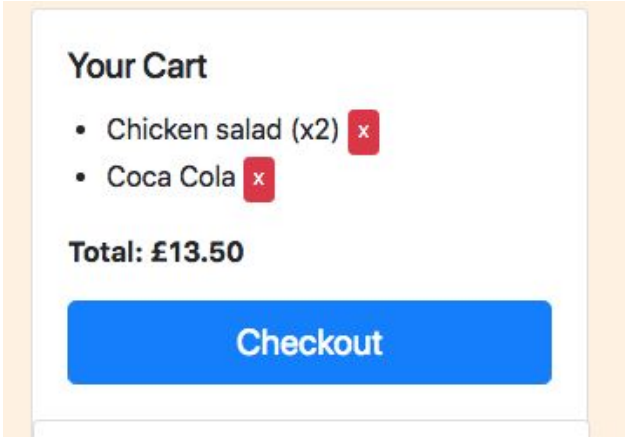
... the cart is updated accordingly with the new item and its quantity, if it's greater than one:



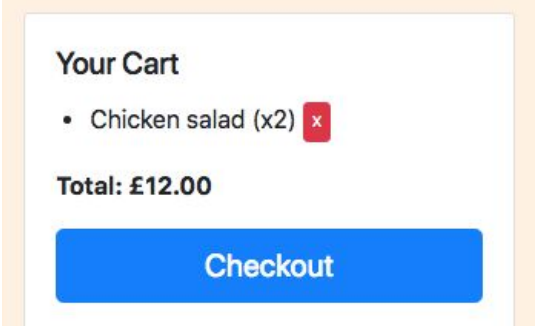
If another item is added...



... the item is appended to the cart status display, without the quantity because it is only one and the total price is updated:

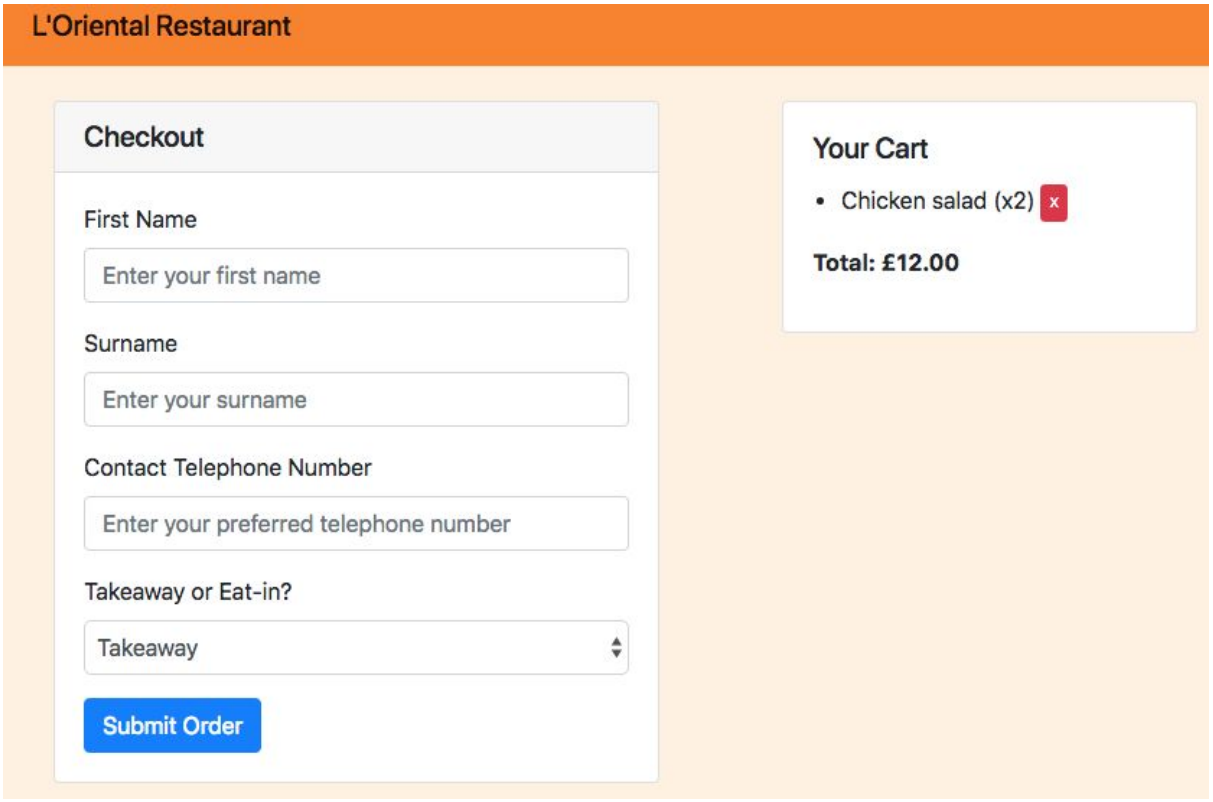


If an item is deleted, by clicking the red "x" button next to it:

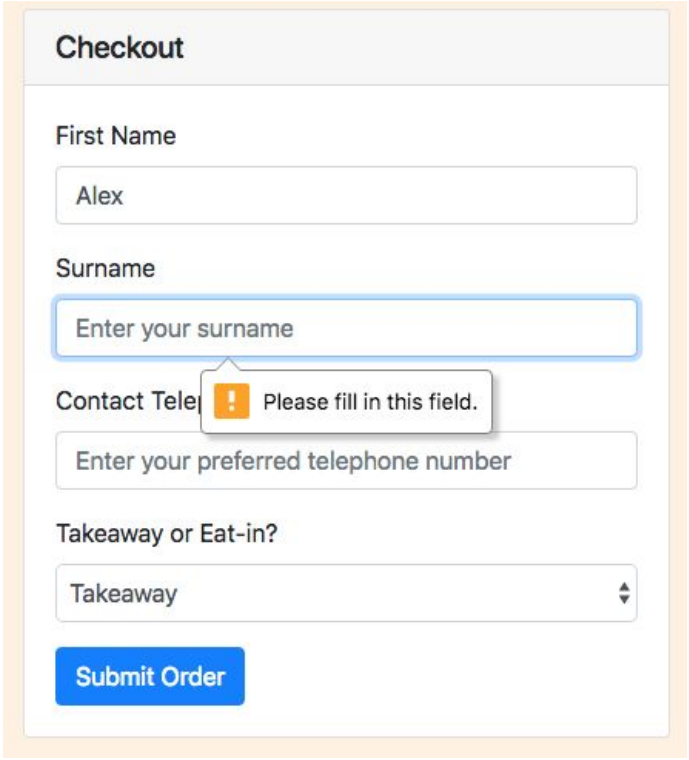


Testing the website's checkout page functionality:

When the "Checkout" button is clicked on the menu page, the checkout page is loaded successfully, with an empty form and the cart status with the total price:

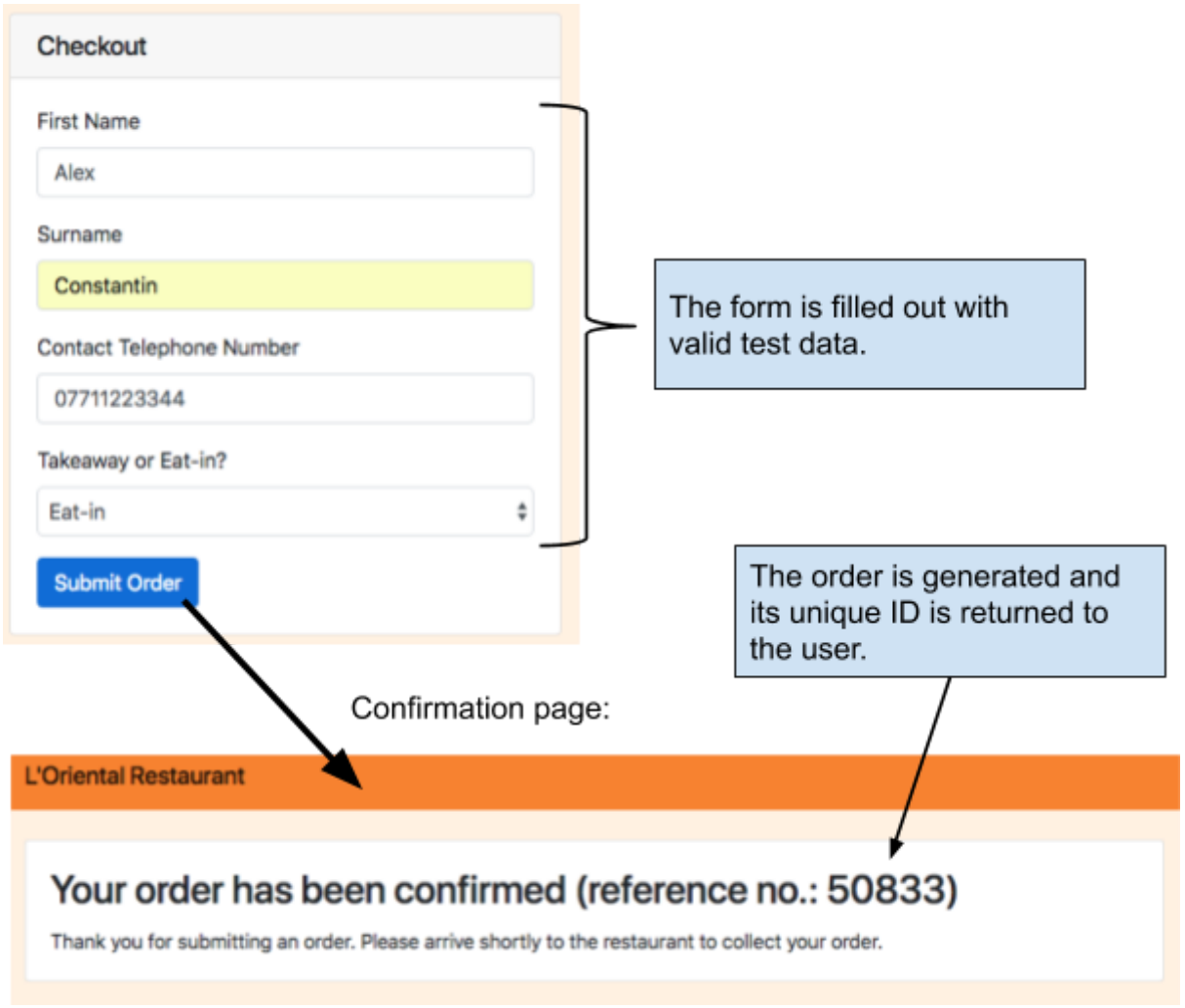


If the a field is left blank, the HTML validation tag will notify the user accordingly:



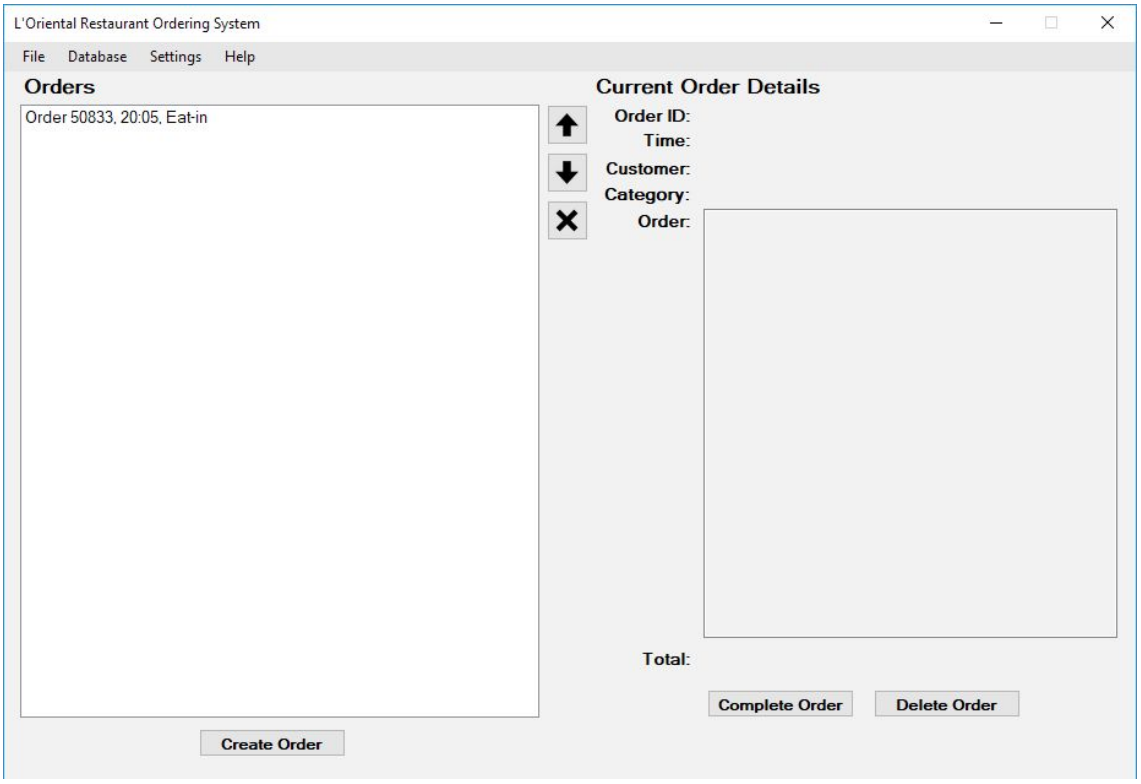
In this case, the first name field was filled in and then the submit button was pressed. Therefore, the next uncompleted field was the source of the warning message.

Once the form has been completed and the submit order is pressed, the confirmation page is loaded, with the order's ID number for reference:

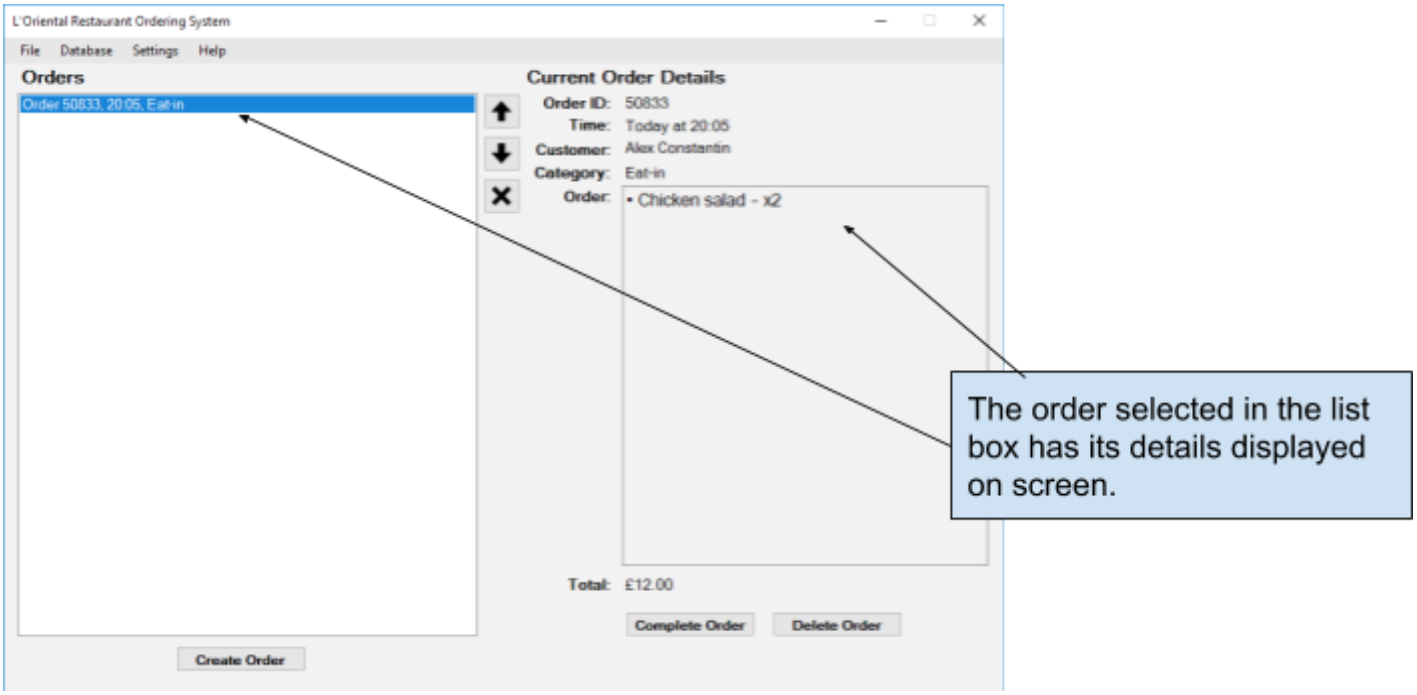


Testing the kitchen program's main window functionality:

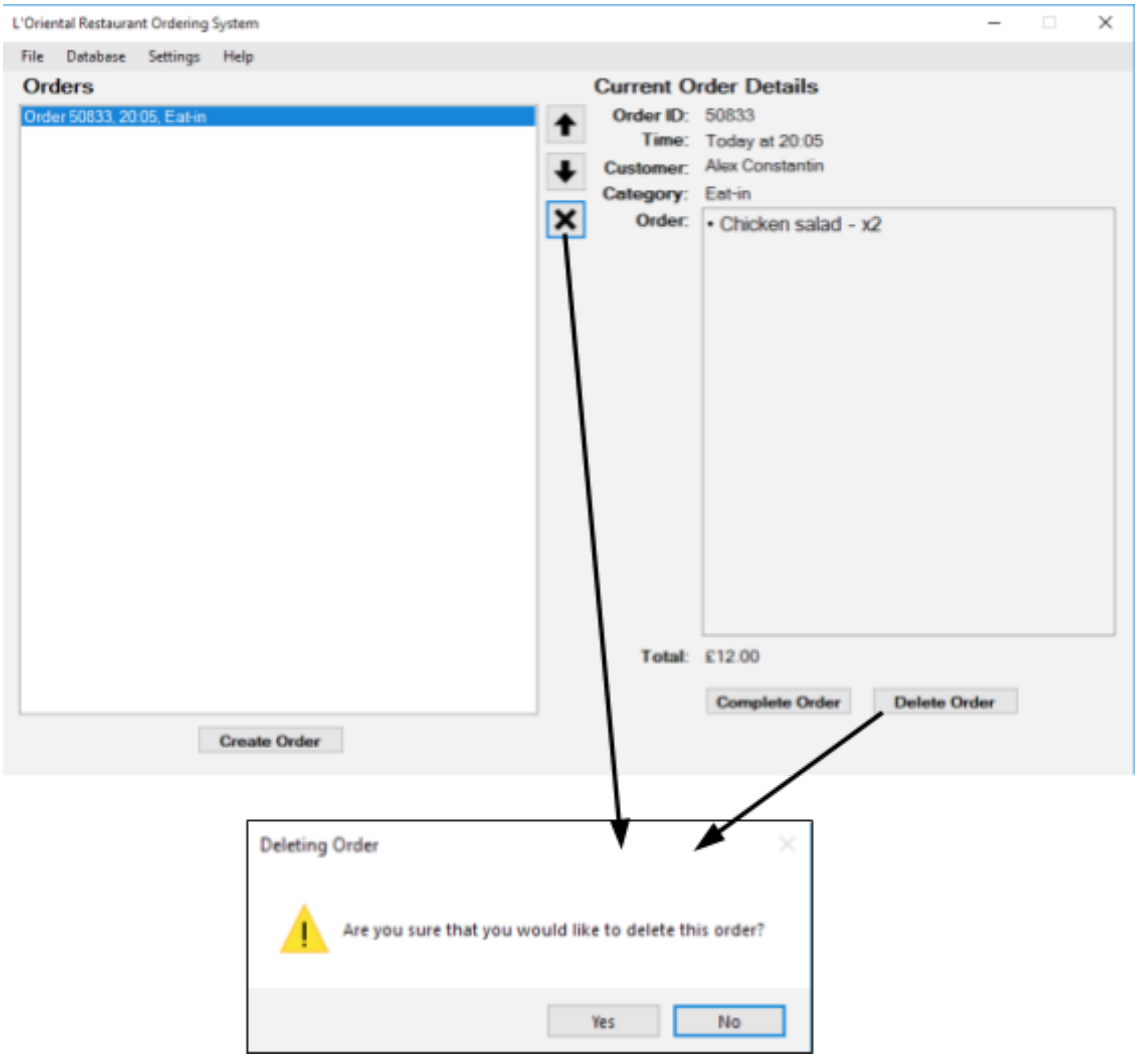
On the other side of the system, assuming the order above is the first one of the day, the main window would look like so:



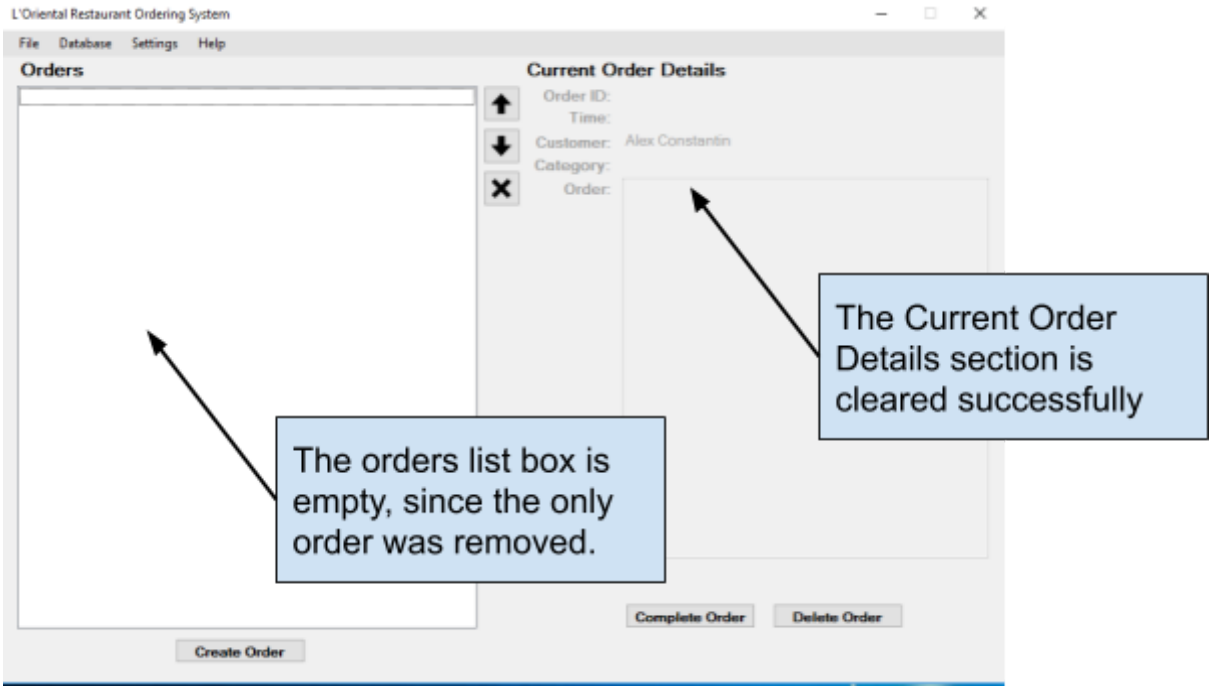
The new order has successfully been retrieved and it is shown on the current orders list box. If it is selected, its details will be shown on the right hand side panel:



If the one of the delete buttons are clicked, a warning message is shown:

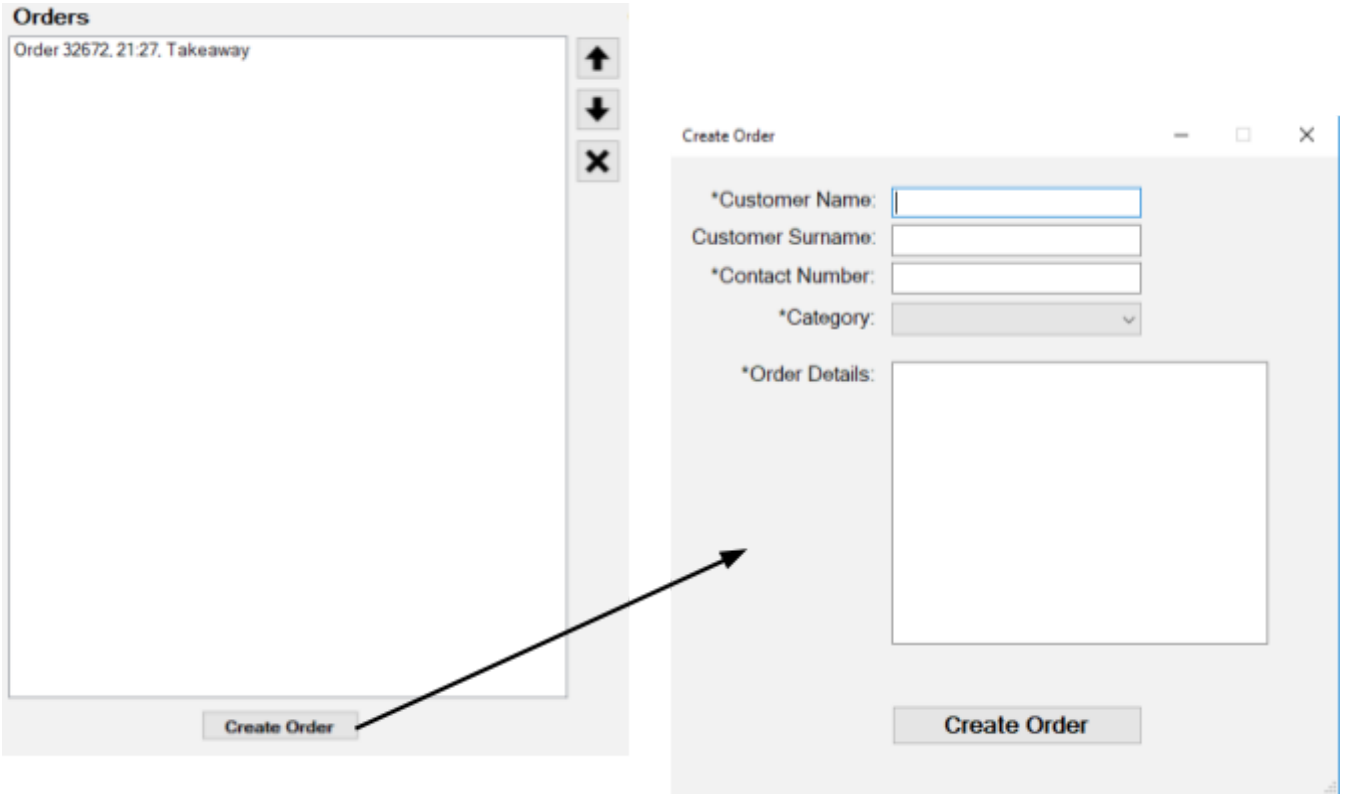


Clicking on the Yes button deletes the order and clears the details panel:

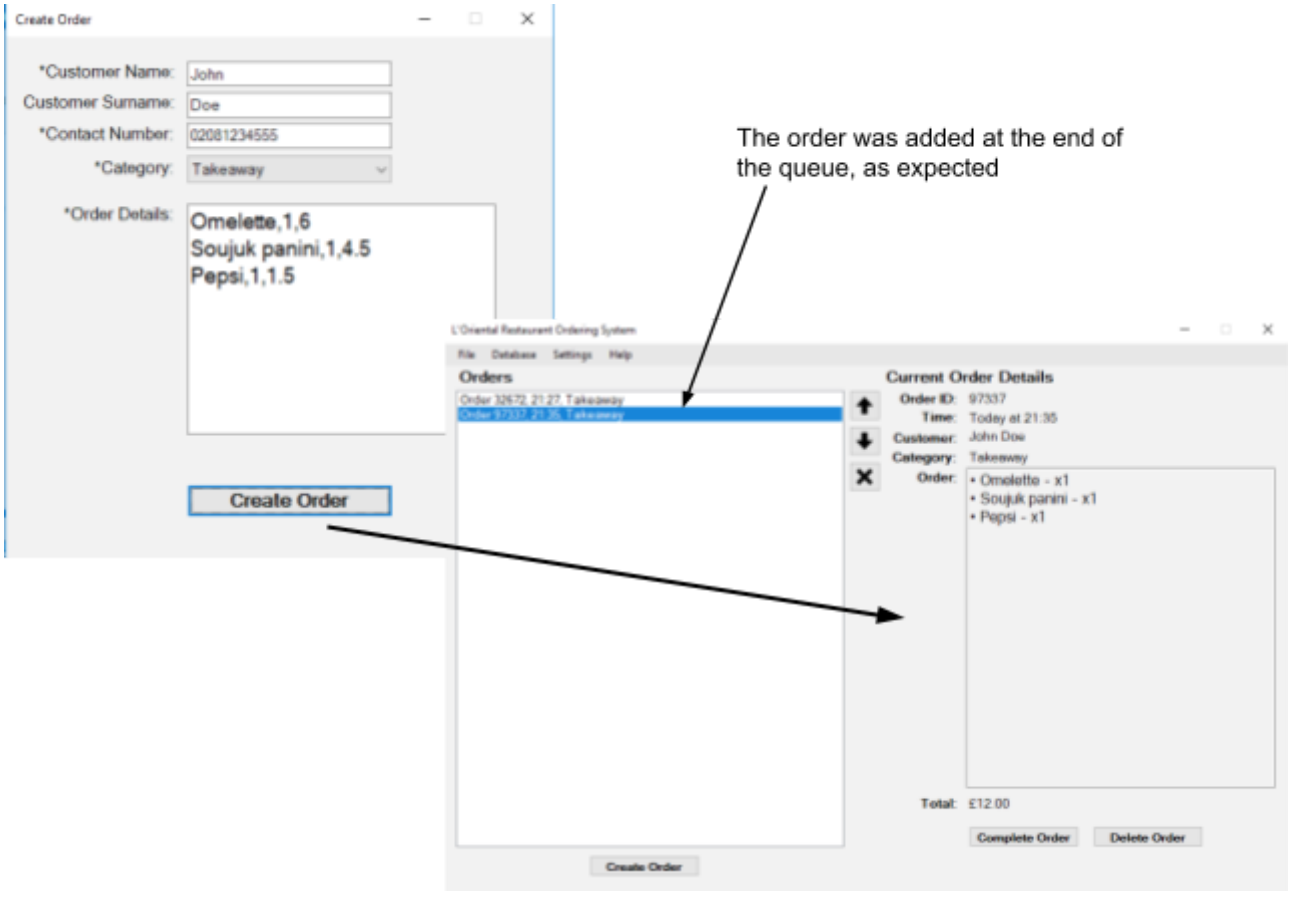


Testing the kitchen program's create new order window functionality:

On the main window, if the "Create Order" button is clicked, the window to create a new order appears:

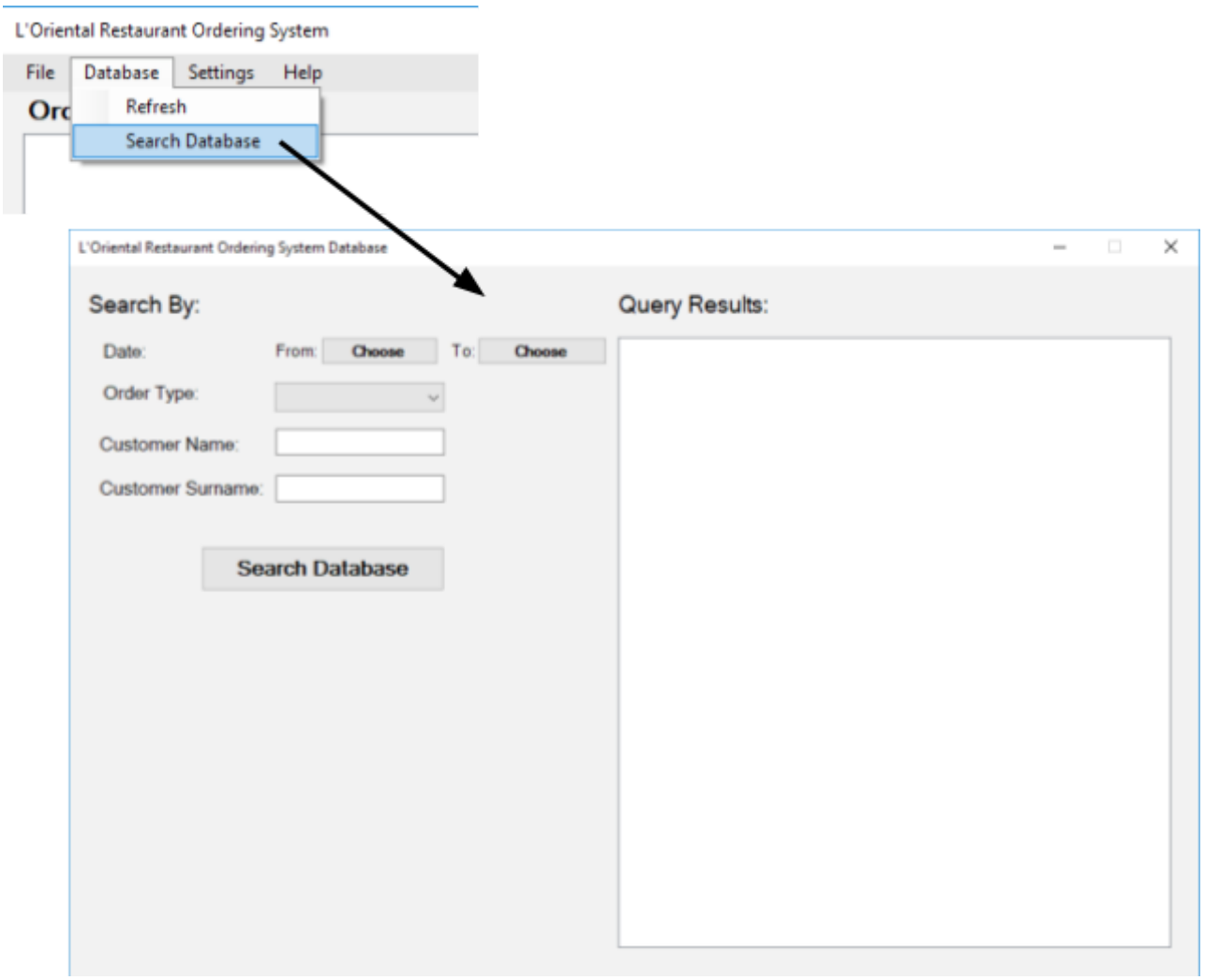


Once the form is filled out and submitted, the order is created and added to the list:

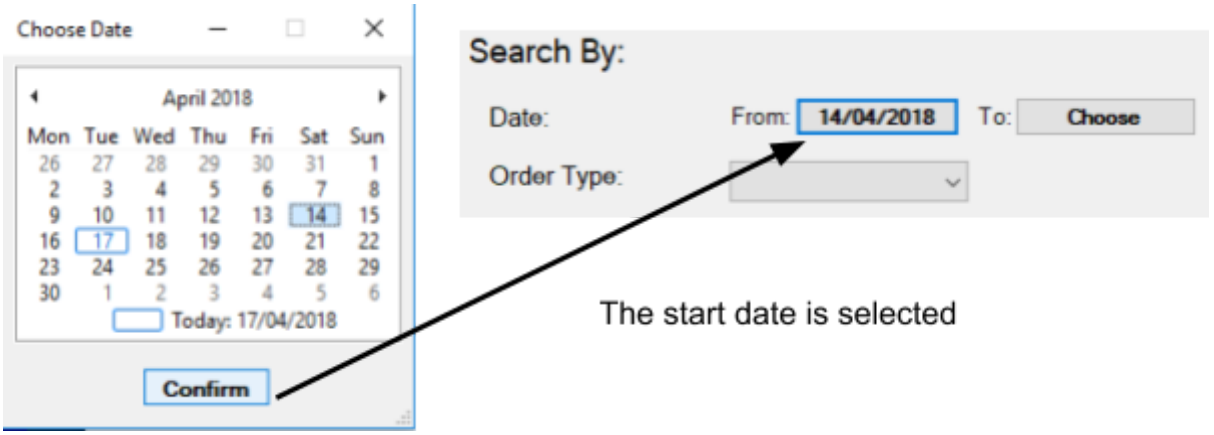


Testing the kitchen program's database search window functionality:

On the main window, if the "Search Database" button is clicked on the menu bar, the Database Search Window will appear:

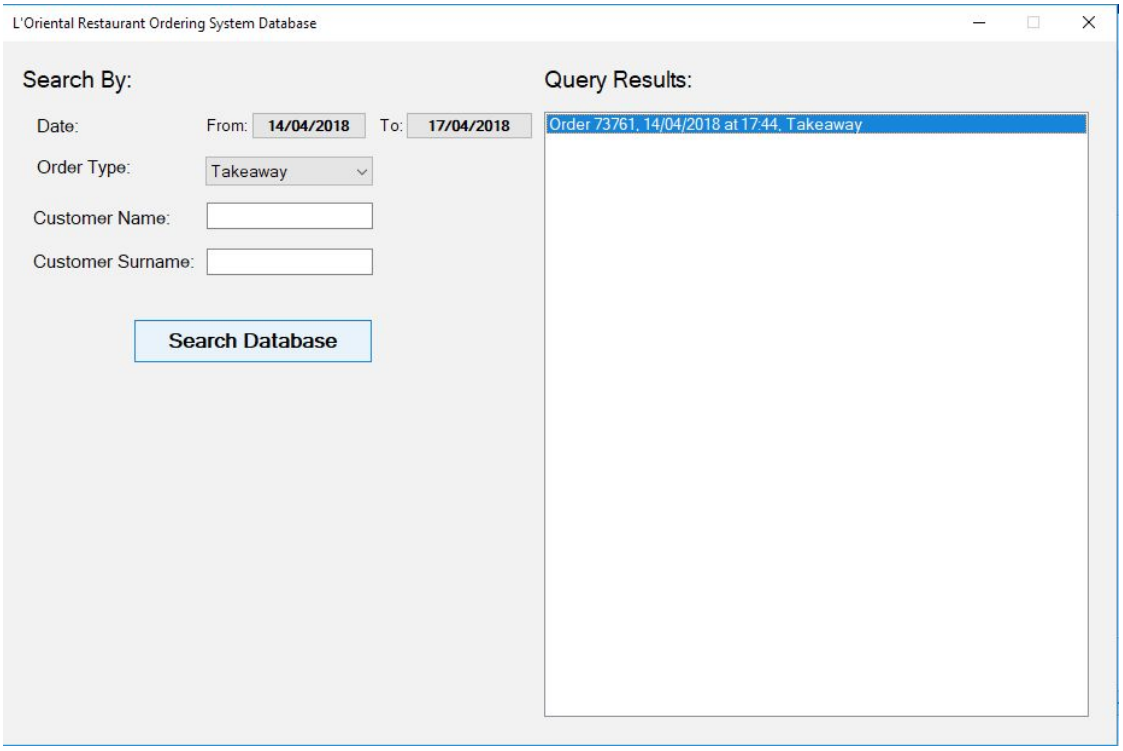


If the "Choose" button is clicked for the dates, the calendar selection window will appear:

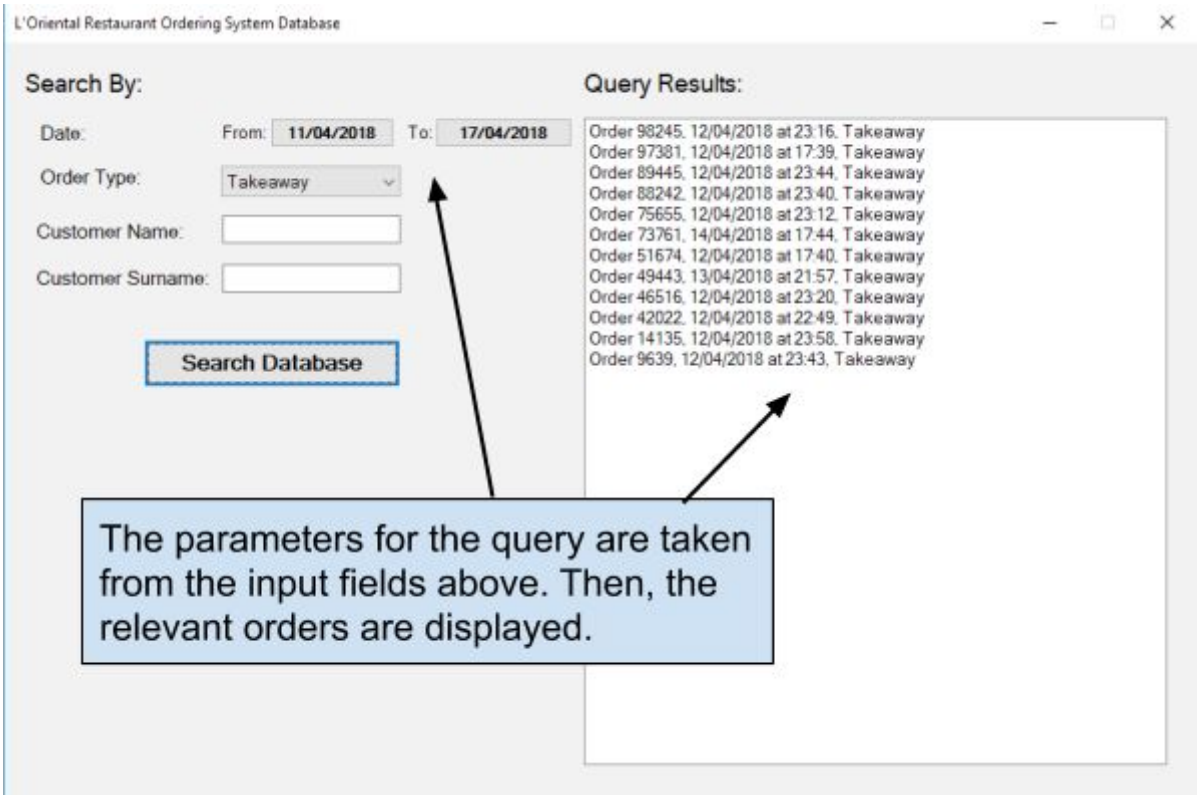


An end date can also be selected, along with the order category:

When the “Search Database” button is clicked, the relevant orders are retrieved:

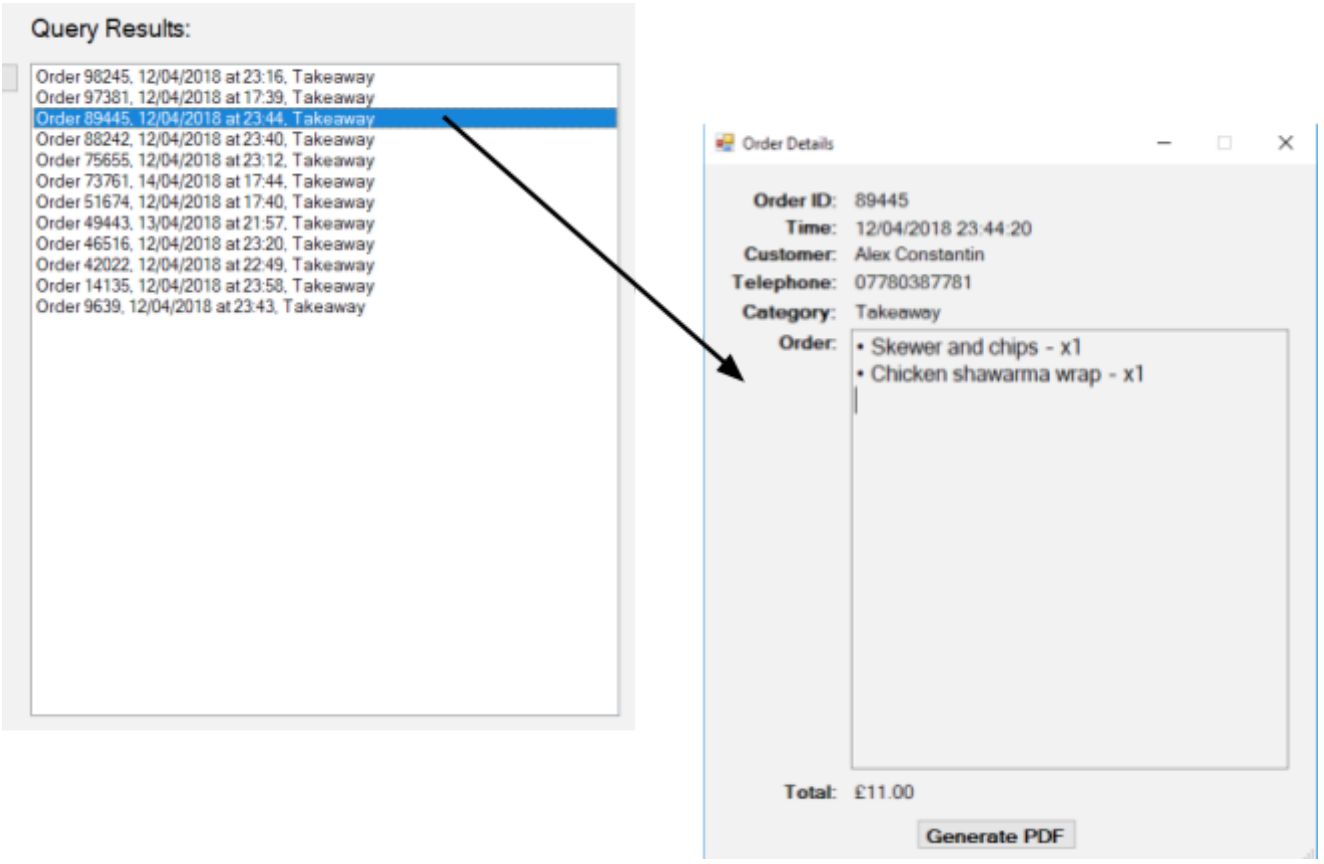


If the user changes one of the dates and searches, the new list will appear:

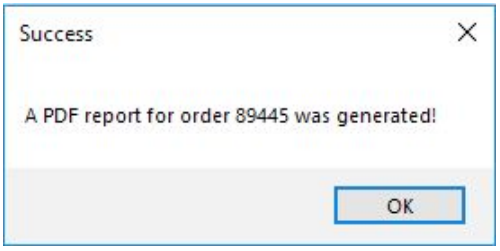


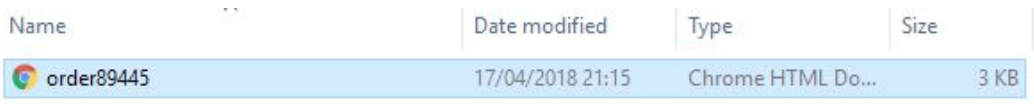
Testing the kitchen program's past order details window functionality:

If an order is double clicked from the query results on the Database Search window, the window for with the order details appears:



All the correct details are shown for the selected order. If the "Generate PDF" button is clicked, the following file is created in the /pdfs/ folder, after the success message:





Details for order number: 89445

Customer Name: Alex Constantin

Telephone: 07780387781

Date and time: 12/04/2018 23:44:20

Category: Takeaway

Order:

• Skewer and chips x1

• Chicken shawarma wrap x1

Total: GBP11.00

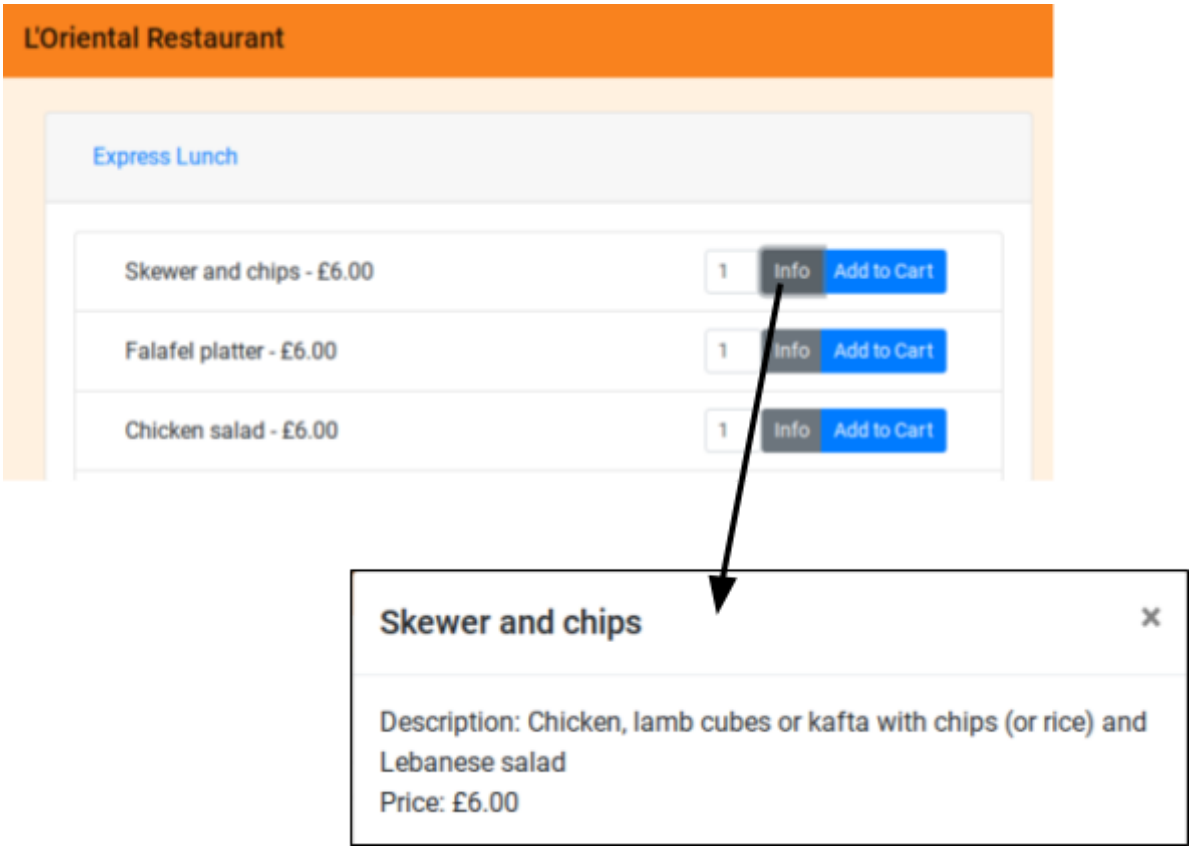
Category	Valid	Takeaway	Yes	

Usability testing with feedback

To evaluate the usability of the solution, the usability features of the system will be tested by the stakeholders: a possible customer will be using the website and the main stakeholder (the kitchen staff) will be using the kitchen client program.

Testing the website's usability features (customer's point of view):

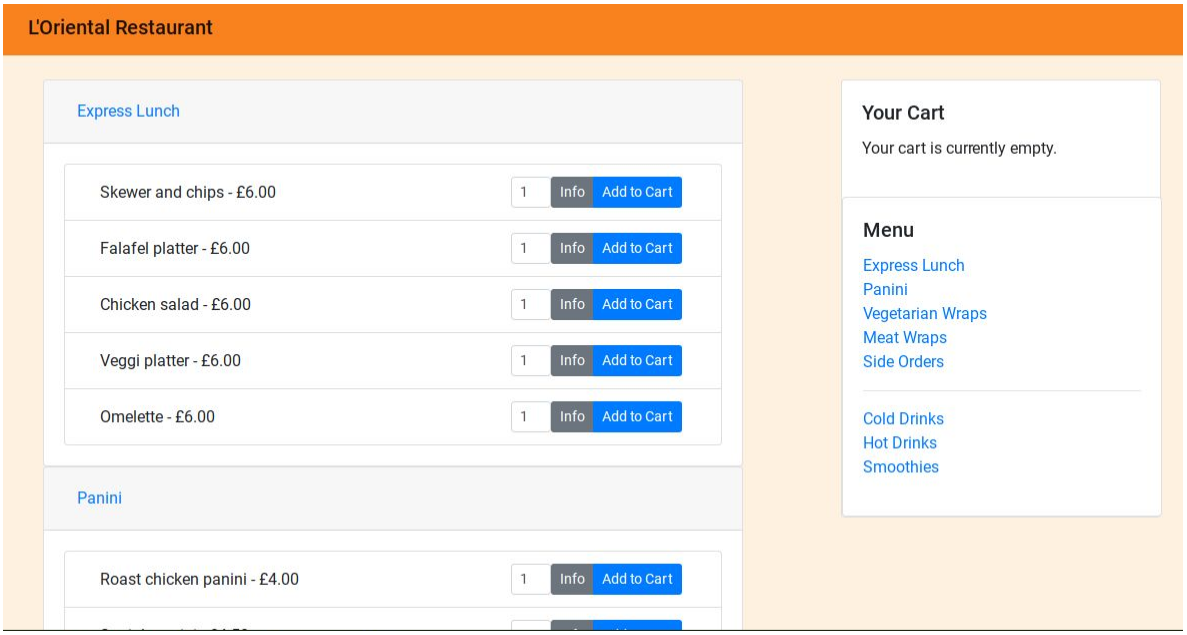
1. When clicking the "Info" button for an item, a popup appears with its description:



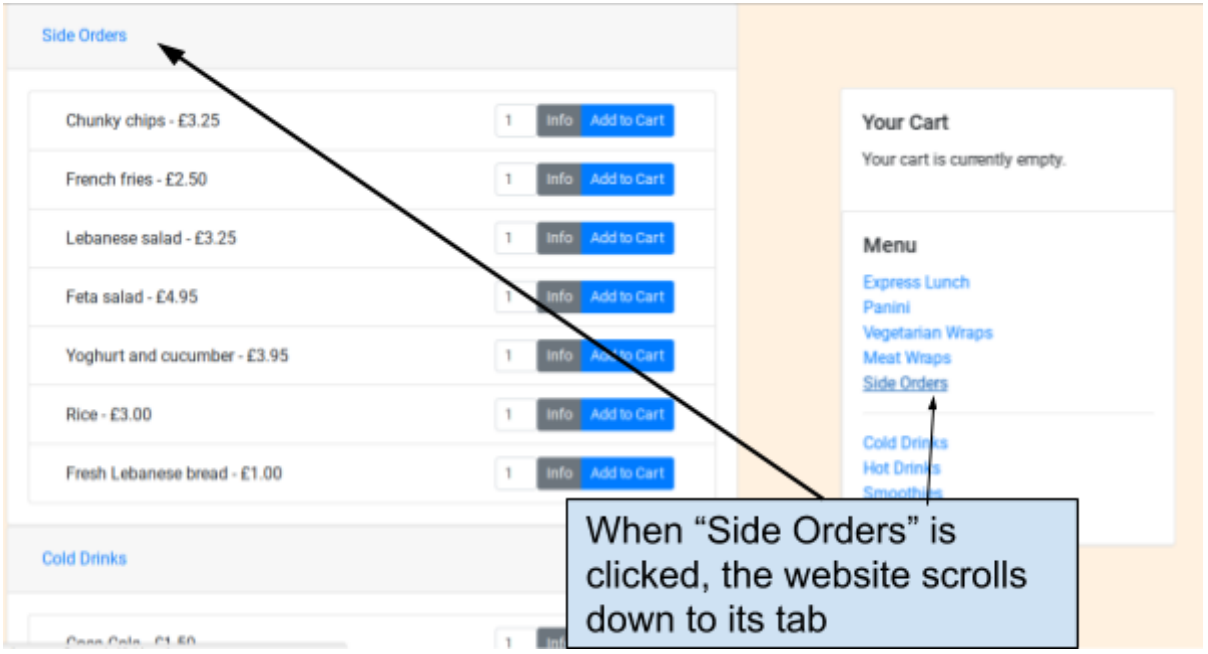
Example customer feedback: *"The info button works just as required. It is very simple and makes it easy to view a more detailed description of the food."*

Example customer feedback: "*It's another very simple yet effective feature which improves the user experience, and makes browsing through the menu easy and customisable.*"

3. The links to different parts of the menu on the sidebar can be tested:



If the "Side Orders" link is clicked, the page will move to the Side Orders menu tab:



Example customer feedback: "*The menu allows users to easily browse the menu just as if they were using a real paper-copy one. It works fine.*"

Stakeholder feedback: "*The arrow buttons make it extremely easy to move orders around, allowing the cooks to prioritise certain orders. The visual arrows also make it easy to understand what each button does."*

4.2. Evaluating the success of the solution

Use the test evidence from the development and post development process to evaluate the solution against the success criteria from the analysis.

Success criteria	Criteria met?	Test Evidence
Store order and customer data in a database	Yes	This has been confirmed by development testing, during the development of the API, because the API successfully showed new order and customer data from the database, meaning the database was storing data as expected.
Have a simple and easy to understand user interface	Yes	This has been confirmed directly by the users (both customers and kitchen staff) through feedback in the post-development usability features testing.
Customers should submit their own orders	Yes	This has been confirmed from development and post-development testing phases, when the validation algorithm for the checkout form on the website was thoroughly tested using black box methods.
Staff should submit their own orders manually	Yes	This has been confirmed from development and post-development testing phases, when the validation algorithm for the Create New Order form on the kitchen client was thoroughly tested using black box and white box methods.
Orders should be categorised	Yes	This has been confirmed from all testing phases, since the orders can be categorised by "Eat-in" or "Takeaway" because it's a class attribute.
Orders should be moved around in the queue	Yes	This has been confirmed from the post-development usability testing when the arrow buttons where tested.
Daily revenue should be calculated automatically	No	This criteria has not been met.

Addressing unmet or partially met criteria in further development

In further development, daily revenue can be calculated automatically for the current day or a past day by querying the database for the relevant orders and then iterating through them so