Developing a cross-platform mobile application for visiting restaurants in times of coronavirus

Pedro Henrique Belfort Fernandes

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

20/09/2020

Abstract

The COVID-19 pandemic provoked a series of disruptions. Social-distancing measures impacted consumer behaviour and business owners, which sought in technology an alternative way of mitigating the crisis. With a steady adoption of mobile technologies and online shopping over the last decade, smartphones may represent a valuable tool for adapting to this new context. The present paper describes the development of a cross-platform mobile application (Android and iOS) that reimagines the eating out experience given the consequences derived from the pandemic. The purpose of the product is to maximise convenience, democratise a high-level of service, and facilitate compliance with safety measures recommended by the World Health Organisation. By using Le Serveur, users can explore restaurant options, check-in at selected outlets, pick favourite dishes, place orders, and express check-out at any time. This work reviews the literature as well as popular Food & Drink mobile applications in the U.K. to guide the development of the system. Design and implementation details of the frontend, database, and backend middleware are discussed. Finally, the application is tested, and a user evaluation survey is undertaken to assess the efficacy and ease of use of the product.

Keywords: software, mobile, ios, android, flutter, cross-platform application, COVID-19, pandemic, social-distancing, lockdown, restaurant, e-commerce

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Pedro H. Belfort Fernandes Signature: 

Acknowledgements

I would like to express my immense gratitude to Dr Mireilla Bikanga Ada for her supervision and support throughout the development of this project. Dr Ada's genuine enthusiasm, insightful advice, and devoted guidance were paramount to the enhancement and completion of this work.

I further extend my acknowledgements to all friends and voluntaries, who helped me conceptualise and evaluate the digital restaurant product. I also convey my heartfelt thanks to my family, whose limitless stream of care, patience, and sacrifices enabled me to pursue my dreams.

Finally, I take the opportunity to dedicate this dissertation to the memory of my beloved grandfather, Geraldo Belfort. I am extremely honoured to be your grandson and have you as my source of inspiration, authenticity and integrity. I am sure COVID-19 was not the end for you. I am sure you are now at a better place. You have taught me to persevere and pursue excellence, for which I am eternally grateful.

Contents

Chapter 1: Introduction

In March 2020, the World Health Organization (WHO) promoted the COVID-19 outbreak to the status of pandemic [59]. As the virus quickly spread, nations across the globe began responding to the crisis in various forms. Until a vaccine could be fully developed and ubiquitously distributed, public health experts advised the adoption of rigorous social distancing measures to attenuate the virus transmission [83]. Even though highly effective in the short term [26], these isolation measures produced notable shortcomings for traditional businesses which were heavily reliant on the local community [73]. Restaurant owners, in particular, were urged to close their establishments to the general public, leaving them no choice but to reduce staff members and renegotiate rental agreements in order to cut fixed costs and survive the crisis.

The consequences derived from the pandemic have proven how strong a digital footprint can be to physical commerce. Some restaurants managed to maintain their operations active despite the lockdown by offering food delivery options through the internet [61]. Researchers evaluate that behavioural change enforced by policymakers are likely to produce subsequent long-term effects on consumers [74]. The mobility restrictions catalysed the adoption of digital technology by a broader portion of the population, which also got acquainted with the level of convenience provided by online shopping. In the absence of a vaccine, reduced person-to-person contact will endure in the post-lockdown world [12]. Hence, brick-and-mortar businesses are posed with the difficult challenge of adapting to this new reality. Customers will eventually go back to restaurants, but aside from superior level of service, they will also expect the "enactment of measures to ensure their safety" [61].

The present paper discusses the design and implementation of a mobile application (app) which pledged to reimagine the restaurant experience in the post-lockdown era. Developed for both Android and iOS operating systems, the app sought to automate the order taking and checkout processes carried out in brick-and-mortar food businesses. Due to the highly competitive nature of the online food industry, its design aimed to offer a similar level of user experience found in popular e-commerce apps available in the market. Businesses from all sizes are allowed to register in the platform. Hence, the expected outcome is to democratise high levels of service, reduce operational costs, and ensure adequate safety to customers.

The remainder of this dissertation is organised into five chapters. Chapter 2 reviews existing literature and gathers insight on mobile application adoption, the rise of m-commerce, challenges associated with restaurant management, and the probable long-term impacts of COVID-19 in consumer behaviour. It also compares three of the most well-known food & drink apps in the UK and describes how system requirements were formulated. Chapter 3 guides the reader through design decisions orbiting the system behaviour, foundational technologies, data modelling, system architecture, and the workflow methodology chosen to organise the project's development. Chapter 4 invites the reader into understanding some of the implementation's core aspects, whereas Chapter 5 covers the steps taken to test and evaluate the product. Finally, Chapter 6 summarises what was achieved and proposes further developments for the future.

Chapter 2: Analysis / Requirements

2.1 Background

2.1.1 Literature Review

Mobile technology adoption

The invention of smartphones transformed the way people perceived their telecommunications devices. Soon, the ability to engage in remote conversations or send text messages was deemed insufficient by consumers [56]. Nevertheless, it was only with the release of Google and Apple's mobile operating systems in 2008 — namely, Android and iOS — that the disruptive potential of these devices was unleashed. For the first time, developers were able to build self-contained software which addressed all sorts of issues for mobile users. This software could later be shipped as a digital product to a worldwide audience via the tech giants' digital marketplaces [17]. The vast assortment of apps promptly available in Google Play and Apple's App Store soon attracted an increasing volume of customers ranging all demographics. As a result, smartphones' target audience gradually shifted from corporate users to the mass market [67].

Comscore's 2019 Global State of Mobile report shows that users around the globe are spending record time online, mostly concentrated on mobile devices [19]. The U.S. alone witnessed a 43% increase in minutes spent online between June 2017 and June 2019; 77% of these minutes were spent on mobile devices. The U.K. market displayed the same figure, and 86% of U.K. mobile minutes were spent on mobile apps. Amongst the reasons which contributed to this continuous market adoption were the convenience of access and the ease of use offered by mobile applications [33]. In 2019, annual worldwide downloads reached the mark of 204 billion; consumers engaged an average of 3.7 hours per day and spent over $120 billion in purchases [3].

M-Commerce

Long before the rise of smartphones, the advent of the internet, combined with the diffusion of personal computers, encouraged businesses to expand their operations to the digital environment. Electronic commerce (e-commerce) led to the emergence of new business opportunities [20], which resulted in a profound change in how customers interacted with companies [37]. Every year consumers gradually feel more confident with online shopping. By 2020, worldwide e-commerce sales were expected to reach $4 trillion [89].

That new mobile computing paradigm is increasingly appealing to business environments due to its unique characteristics, principally the lack of geographic location constraints [25]. Researchers claim that mobile commerce (m-commerce) is a natural evolution to e-commerce [51].

M-commerce is often defined as "any monetary transactions related to purchases of goods or services through internet-enabled mobile phones or over the wireless telecommunication network." [86] In his study, Yang (2012) extended the Theory of Planned Model to identify positive traits influencing mobile shopping adoption [88]. Consumers are allured by the on-the-go nature of this channel, which enables them to enjoy an omnipresent shopping experience delivering localised and personalised content at the palm of their hands. "Fast data processing capability, user-friendly interface, and unlimited data usage plan" intrinsic to mobile devices also contribute to the adoption. Yang concluded that after completing the first purchase through the mobile channel, customers are more likely to become recurring visitors, either to purchase new items or to explore and get a new idea. Indeed, according to App Annie's 2020 State of Mobile report, the global shopping app demand grew 20% from 2018 to 2019, reaching over 5.4 billion downloads [3]. These mobile apps are particularly useful for driving research and consideration, ultimately facilitating fulfilment.

Managing a restaurant

All typical restaurant operations can be categorised in three areas: front of the house, back of the house, and the office [9]. The front of the house has the crucial responsibility of delivering customer satisfaction. It encompasses operations which are familiar to many restaurant visitors; those are, greeting guests, taking orders, serving food, removing used tableware, conducting payment, thanking guests, inviting comments, and encouraging returns. The back of the house can be compared to a factory. Its predominant task is to carry on food production. In their book, Barrows and Powers claim that quality and cost controls are parallel activities closely correlated to the restaurant's profitability. Lastly, the office is an area associated with administrative functions, such as coordination of the operations and accounting.

Managing a restaurant business usually translates to overcoming challenges daily. Restaurant owners often struggle with high labour costs and fierce competition, which in turn creates a positive attitude towards the adoption of creative solutions [35]. Despite being often labelled as latecomers to innovation, restaurants are increasingly embracing technology in several fronts [16] such as review and search, reservations, ordering/payment, loyalty, rewards programs, and human resources analytics. These researches also conducted a study where they introduced a tabletop technology system in 66 restaurants and analysed its impact on their performance. Their findings suggest that the new system has driven an increase in average sales per check by 1%, and reduced the staying duration by 10%. Hence, researches estimated that these figures represented approximately $2 million in additional sales or $ 1 million in increased profits per month — significant for an industry characterised by low margins [80].

Long-term impacts of the COVID-19 pandemic

In December 2019, a novel mutation of the coronavirus (SARS-CoV-2) originated from the province of Hubei, China, and rapidly diffused to an increasing number of nations around the globe [41]. Preliminary studies indicated that up till June 19, 2020, the acute respiratory disease caused by the virus, or COVID-19, manifested a 14% fatality rate in the U.K. and 5.3% worldwide [70]. On January 30, 2020, the World Health Organisation (WHO) agreed that the outbreak met the criteria for a Public Health Emergency of International Concern (PHEIC). Due to its escalation, on March 11, the organisation updated the outbreak classification to a pandemic [55]. As of August 28, 2020,

more than 24 million confirmed cases of COVID-19, including 827 thousand losses, have been reported globally to the WHO [87].

Authorities from all continents reacted to the crisis by enacting measures to restrict physical contact with the hope of mitigating the spread of the virus. Schools were closed, travel was restricted, and lockdowns were imposed [70]. These social distancing measures proved efficient in the combat of viruses transmitted by respiratory droplets [85]. However, they also scaled-down production, consumption, and the flow of goods, which disrupted supply chains and inaugurated a global recession [54]. Researches are concerned that the COVID-19 outbreak will impact long-term consumer behaviour. De Vos (2020) suggested that "travel demand might drop due to an increased amount of working from home, e-learning, and a reduced number of public activities and events" [21]. Customers might also be more willing to purchase goods via online channels (e.g. food, clothes) [75]. As consumers adapt to the social distancing measures for an extended period, they are more inclined to embracing novel technologies that facilitate learning, working, and consumption in a convenient manner. The adoption of digital technologies will likely modify existing habits [74].

2.1.2 Existing Products

As of July 2020, both Apple and Google's marketplaces were filled with apps that address customer needs related to food and drink. This section selects and analyses the highest-ranked apps in that category, shedding light on how companies successfully translated their value propositions into mobile applications.

UberEATS

In August 2014, the ride-hailing tech giant Uber decided to expand its range of services and inaugurate a new branch in the online food order industry. Baptised as UberFRESH, the company's new food delivery service was initially offered as an integrated feature of its popular ride-sharing app. However, due to its immense success, one year later, UberFRESH was detached from the Uber platform and rebranded to UberEATS [50]. Despite facing fierce competition, as of July 2020, UberEATS is considered the top Food & Drink app (Google Play and App Store) across many countries, including the UK [4].

The application offers an intuitive experience and clean graphical interface, in sync with Apple's Human Interface Guidelines [5] and Google's Material Design [30]. Its main functionalities are laid out in four bottom tabs. The first is the Home screen, which allows users to set up a delivery address and browse special offers, deals, recommendations, and popular restaurants nearby. These restaurants are represented as card items containing essential information such as the restaurant's name, price range, average preparation time, and rating. Tapping on these cards take the users to a dedicated screen for the store. There, they can browse the restaurant's menu and pick products to add to the cart. When satisfied with what is in the cart, users can finalise the order, make a payment, and track the driver's journey until the order arrives at the chosen delivery address.

The remaining tabs are Search, Orders, and Account. Search provides the user with the ability to find specific restaurants or meals by typing a term into the search bar. The Orders tab shows information about past or upcoming orders. Lastly, the Account tab holds user-related options such as payment methods, customer support, promotions, favourites, and application settings.

Just Eat

Only ranked behind UberEATS (UK), Just Eat is another prominent player in the Food & Drink category. Similarly to its competitor, the app provides an online channel where customers can browse and pay for food from local restaurant partners [43]. A significant difference between the two companies is that UberEATS uses its affiliated drivers to collect and deliver food, whereas Just Eat, on the other hand, relies on takeaways' delivery employees to carry on orders [11].

Upon entering the mobile app, users are presented once again with four bottom tabs. Only this time they have been labelled as "Restaurants", "For You", "Orders", and "Settings". In contrast to UberEATS, the Restaurants tab incorporates a search functionality in addition to a collection of recommendations, promotions, and cuisines. The search itself is reasonably different. Since the user is not allowed to set a delivery address at the beginning of the experience, the search bar expects postcodes or the user's current geolocation. Then, he/she is taken to a screen listing outlets which are close to the input location. Restaurant cards share nearly the same set of information seen in UberEATS, but with a few particularities. Just Eat includes the distance to the restaurant and excludes its price range as well as the ability to create a list of favourites.

Moreover, the app divides the menu browsing experience into two steps. First, the user views a list of categories and then enters into a list of products — no pictures provided in any of the list items. Aside from the request for a phone number and delivery address, the checkout process is identical to UberEATS. The "For You" tab shows a screen populated with personalised offers based on the user's order history, which can be viewed by tapping on the "Orders" tab. The last tab "Settings" encompasses customer service options and overall configurations.

Wetherspoon

The Wetherspoon app has a slightly different value proposition when compared to the previous apps. Instead of facilitating the delivery of takeaways, it was commissioned by J D Wetherspoons plc in 2017 to enable customers ordering menu items from within their pub chains without the need for human interaction [39]. It not only offers more convenience to those who wish to keep their tables but also contributes to enhancing the pub's operation efficiency, as it reduces agglomerations at the counter and automatically digitalises the order stack.

Although the app also takes advantage of tab-based navigation, the Customer Journey occurs distinctly. Inside the app, users are redirected to the Order tab, where they are instructed to choose a pub from a list of nearby outlets or a map view. An alternative way to pick a pub is to fetch for a specific outlet, which can be achieved via three different routes. One option is through the search icon in the top navigation bar. A secondary way is to tap on the "browse all pubs" button at the end of the screen. Lastly, users can perform the same action by accessing the Finder tab, which replicates the same set of functionalities seen in the Order tab. The reason for this duplicate is that the app expects users to choose a specific pub. After a pub is selected, the Order tab transforms itself to display categories of menu items such as pizza or drinks. From this step onwards, the flow is similar to the food-delivery apps. The user chooses a dish or drink, customises it, and adds it to the basket. When ready, the user selects a table number and places an order. Note that each time this is done, the user is also required to make a payment. The Home tab displays news and miscellaneous information provided by Whetherspoon. The More tab contains the user profile, order history, pubs visited, customer support, and application settings.

Comparison of the core features

The analyses above indicated that despite adopting different marketing strategies, these popular Food & Drink apps share similar core features. Regardless of the brand, users must be able to search for items, view them, and make purchases. Table A.1 in Appendix A elaborates on that notion, describing the identified taxonomy of the most critical pieces of functionality and underlining particular aspects of each implementation.

2.2 Aims and Objectives

This project builds on the premise that, in 2020, customers are acquainted with shopping on mobile devices, and that such technology can be a valuable tool in a world recovering from the side-effects of the coronavirus pandemic. Thus, the aim is to build a set of applications that enable brick-and-mortar restaurants to provide contact-free yet high-quality order and pay services. Customers should be able to find restaurants, visit them, and perform regular dining actions via the app; from browsing items in the menu to ordering and paying. To accomplish that, the following objectives must be met.

- Develop a multi-platform (Android and iOS) mobile application which serves as a digital representation of the restaurant to the end-user.
- Build a database to store familiar e-commerce entities such as products, stores, users, among others.
- Conceive a backend middleware application to connect the front-end mobile app with the database and perform safe data manipulation on the server-side.

2.3 Requirements

Requirements elicitation is a fundamental milestone in the planning phase of any software engineering project. Excelling on this process culminates in a clear map of the components which impact the success of the endeavour directly, paving the way for other design-related decisions. This section begins by introducing the reader to the techniques adopted to identify those requirements. Then, it presents an exhaustive list of requirements enhanced with the MoSCoW prioritisation framework [23].

2.3.1 Gathering Process

Preliminary literature review exposed the challenges faced by restaurants in adapting their daily operations to the post-lockdown world where social distancing is still a reality. Furthermore, the comparison among existing products elucidated standard functionalities that make it possible for customers to find and eventually purchase items. Hence, the first step towards generating a list of requirements was to identify actions taken by customers when shopping online for food. Figure 2.1 summarises the main phases pertained to the Customer Journey.

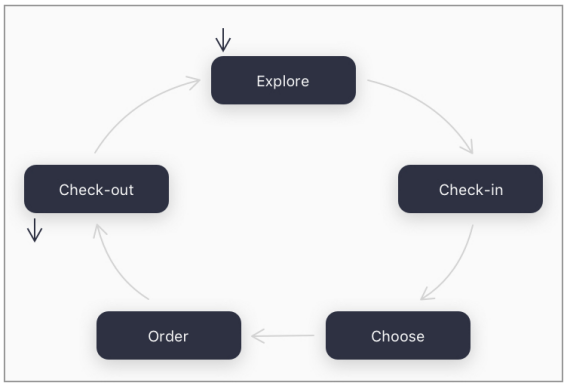


Figure 2.1: The continuous Customer Journey cycle. Users explore restaurants, check-in, browse items from menus, place orders, and check-out once they are satisfied.

Since the project targets customers who are visiting restaurants in-person, Wetherspoon's value proposition proved to be the most relevant source of inspiration. The customer's journey commences at the exploration phase, where restaurants are discovered. Then, the arrival at the preferred outlet is indicated in the check-in phase. Once inside the restaurant, users browse the menu for dishes and add products to the tab. When ready, they must be able to place orders, which are automatically transferred to the kitchen. Finally, users must have the ability to check-out the experience and pay for the bill at any time.

The association of the Customer Journey with the taxonomy observed in popular m-commerce Food & Drink apps resulted in the first set of requirements. The next step taken was devising a high-fidelity prototype using Adobe XD [63] to reduce the level of abstraction. This prototype comprised 28 interactive screens which mimicked a functional application. Ten potential users from distinct demographics were invited to evaluate the prototype. They were briefed about the Customer Journey, and given simple tasks to accomplish (e.g. find a restaurant, check-in, add a dish to the tab, try to make a payment). Comments were left on each screen via the platform's "Comments" tab. The feedback was later gathered and used to refine the list of requirements, resulting on its final version. See Appendix B for a condensed view of the high-fidelity prototype and the comments tab.

2.3.2 MoSCow Analysis

The final set of requirements was formulated by applying the MoSCoW prioritisation methodology [23]. This technique's name is an acronym which stands for "Must have", "Should have", "Could have", and "Will not have", which are frequently used in software engineering projects to create four clusters with distinct hierarchical values that indicate urgency in their development [34]. Table 2.1 summarises the "Must have" and "Should have" functional requirements. For the full set of requirements, go to Appendix C.

MoSCoW Priorisation			
[Mo] **must have**: critical to the project			
Title	**Description**	**Phase**	**Estimated Effort (h)**
QR-Code scanning	Scan QR-Code to check-in a local restaurant	Explore	12
Restaurant visualisation	Visual representation of a restaurant and its characteristics	Explore	10
Product visualisation	Visual representation of a dish and its characteristics	Explore; Choose	10
Restaurant check-in	Set a restaurant as the home content	Check-in	10
Menu visualisation	List of products offered by a given restaurant	Choose	6
Tab visualisation	Visual representation of the restaurant tab -- open/closed items	Order	12
Adding products	Addition of products to the tab	Order	4
Editing tab products	Edition of products contained in the tab	Order	4
Ordering	Request sent to the kitchen to prepare the products in the open tab	Order	8
Check-out	Process of closing the tab and making payments	Check-out	32
Credit-card payment	Registration and selection of credit-card as payment method	Check-out	16
[S] **should have**: significant value but not vital			
Title	**Description**	**Phase**	**Estimated Effort (h)**
Navigation by category	Find restaurants by the type of cuisine	Explore	10
User signup	Creation of a user account to store basic information	Account	10
User login/logout	Association of a user with a given active session	Account	10
Password recovery	Password reset in case a user forgets the password	Account	8
Map navigation	Discovery of restaurants through a map	Explore	16
Search	Discovery of restaurants through a search query	Explore	12
Filters	Narrowing-down restaurant results by setting filter options	Explore	10
About/more	Screen to access terms and conditions ass well as version information	Miscellaneous	6

Table 2.1: Functional requirements expressed via the MoSCoW priorisation technique. Each requirement has been associated with a pertinent Customer Journey phase. The full table can be found in Appendix C.

2.3.3 Non-functional Requirements

The specifications below describe critical characteristics that must be inerrant to the system's operation. Failing to meet those requirements may result in not satisfying customer needs.

1. **Platform requirements**; due to the equal distribution of iOS and Android users in the UK [53], it is important that the mobile application is made available for both operating systems.
2. **The system must remain stable even at scale**; the user experience must not be affected by a high throughput of service requests. All application tiers must be able to manage their operations concurrently.
3. **The user interface must adapt to distinct screen sizes**; The plethora of available device models makes it impossible to assume the user's exact screen size. The application must take advantage of layout constraint techniques to remain responsive to different scenarios.
4. **Sensitive data and operations must be secured**; The RESTful API must ensure that senstive resources are properly hidden from the general public. Data such as account passwords must hashed before stored in the database.

Chapter 3: Design

3.1 System Appearance and Behaviour

The user interface (UI) and experience (UX) extend Google's Material Design principles [30]. Inspired by elements commonly observed in the physical world such as ink and paper, this set of UI/UX guidelines ensures consistent, high-quality digital experiences across apps available in multiple platforms. The system combines typography, space, grids, colour, and imagery to denote meaning and create a sense of hierarchy. Motion is used to draw attention, provide feedback, and create continuity between contexts. Material encourages the adoption of reusable UI building blocks called Components. These elements must respect a unified Theme and attend to interface needs, including display, navigation, actions, inputs, and critical events communication. The overall system behaviour was designed around eleven interaction flows that connected features and led to the completeness of specific Customer Journey milestones. Appendix D highlights these flows using the high-fidelity prototype [63].

The experience begins with the Onboarding flow. As the name suggests, it introduces users to what the application has to offer. Newcomers are presented with a set of three paginated slides which briefly describe in two-lined sentences the app's value proposition. Users can choose to proceed to an authentication flow or skip and access app features as a guest. The authentication flow encompasses the creation of new user accounts, log in with credentials, and the retrieval of lost passwords. By identifying themselves, users unlock the ability to set preferred payment methods, bookmark restaurants as favourites, and consult a list of last visited outlets. Since guest sessions are permitted by design, features pertaining to this flow can be accessed from multiple contexts (e.g. from an unauthenticated Profile Screen).

After skipping or logging in, users are redirected to the app's main set of screens, which is laid out in a bottom navigation bar to enable parallel interaction with distinct flows. At this point, users are officially considered in the Explore stage. Every flow associated with this Customer Journey cluster has the goal of facilitating the decision-making process and culminating in an outlet's check-in. There are three flows through which this can be achieved. The first and quickest is by scanning a QR-code that is physically displayed inside a restaurant. In that particular case, it is assumed that users are already located at their chosen outlet, and they are only looking to integrate their experience with the app. Alternatively, users can choose to browse restaurant options by using an embedded map view on the Explore screen or by scrolling through a list of recommendations on the Home screen (Outlet Discovery flow). Further information about each restaurant, including its menu catalogue, can be accessed by tapping on Outlet cards and navigating back and forth between the Outlet, Product, and Cuisine screens. When finally ready to make a decision, users are supposed to interact with a slider button to indicate they want to check-in the outlet.

That action inaugurates the Check-in stage, which is relatively short. Users are assigned with a table number, and the Home screen content is transformed to reflect the chosen outlet's menu catalogue. Now in the Choose phase, users have access to the Menu Exploration flow. Similarly to the Outlet

Discovery, users can tap on cards to view further information such as dish description, unit price, pictures, and ingredients. Note that this time the content is built around dish options, and instead of check-in, users can add products to their baskets. The flow terminates when users place all their desired items in the basket. At this moment, users commence the Order stage. There is only one flow here: users may edit cart item quantities and send orders to the kitchen. Since most restaurants only ask for payment at the end of the service, the app allows for dispatching multiple orders without touching the wallet. The order statuses can be tracked on the Summary tab, which contains a slider button that can be used to trigger the check-out process.

The Payment flow handles the Check-out stage. First, the user selects its desired payment method and then chooses a certain tip amount to be added to the final bill. The bill is calculated by summing up all cart items ordered by the user plus the tip. When ready, users can tap on a button to finalize the payment operation. Upon performing a successful payment, the user is redirected to the Home screen, which has its contents reset to the initial state. The Customer Journey has officially come to an end and can be reinitiated at any moment.

3.2 Foundation Technologies

As previously discussed in Chapter 2, the project consists of three complementary applications: a frontend mobile app to display data and capture user interaction, a database to store relevant entities, and a backend middleware to bring these two together safely. Due to the short time frame (two months) and extensive project scope, choosing an appropriate technology stack was vital to succeed with the endeavour. The next subsections will compare and contrast the framework options for each application and justify the final design decision.

3.2.1 Frontend

It can be inferred from the functional requirements that the mobile app will make use of several custom UI components and that its final version is supposed to offer an experience comparable to competitors such as UberEATS or JustEat. Furthermore, non-functional requirements emphasise the importance of having the application available on both Android and iOS operating systems (OS). With that said, the first technical decision lies on whether to invest time in developing the application natively for each platform or to adopt cross-platform tools to accelerate the process.

Developing mobile apps natively has its pros and cons. On the one hand, it guarantees that new OS capabilities introduced by Google or Apple can be incorporated as soon as they are released (e.g. sign in with Apple/Google). Native apps are also more likely to respect each platform's particular characteristics, less prone to facing problems during the release process, and they usually offer a more integrated development environment [82]. Nevertheless, despite resulting in a similar final product, engineering apps for Android differs a great deal from iOS. Android developers are accustomed to programming languages such as Java or Kotlin, whereas iOS embraces Swift or Objective-C [6]. Besides, Apple makes available a range of proprietary tools and frameworks such as Xcode and Cocoa-Touch that are fundamentally different from Google's Android Studio or Android SDK [27]. In other words, creating apps natively often means learning about two uncorrelated worlds and writing the same pieces of functionality twice. This workflow is incredibly costly as it

requires experts from both technologies or twice as much time. Challenges also arise when trying to maintain consistency between the two platforms as the app grows.

On the other hand, cross-platform technologies have evolved significantly since their first emergence in 2010 with Titanium [44]. As of 2020, popular solutions such as Windows' Xamarin, Facebook's React-Native, and Google's Flutter can be used to build apps with native-like experiences and performance [72]. Since the project does not rely on hardware intensive resources (e.g. memory management or GPS) and both speed and quality can be achieved through cross-platform development, this strategy was champion.

The next step was to choose which tool to use. Priority was given to the three most popular cross-platform mobile frameworks, according to Stackoverflow's 2019 Most Popular Technologies survey [77]. The set of assessment criteria below was created to support in the stack comparison and ultimate selection. The comparison table E.1 can be found in Appendix E.

- **License**. The tool must be free for commercial use and open-source.
- **Programming language**. The tool must support an object-oriented, statically typed programming language with a robust foundation package and easy to learn.
- **Reactive programming**. The tool must offer a convenient implementation of the Observer pattern to propagate changes across the app as they happen.
- **Code reuse**. The tool must allow for developing Android and iOS reusing the same code base.
- **User Interface components**. The tool must offer convenient, ready-to-use UI components to create rich layouts.
- **Development speed**. The tool must offer advanced debugging capabilities such as the use of breakpoints, layout inspector, and hot reloading.
- **Performance**. The tool must be able to build apps that run smoothly at 60 frames per second.

Flutter scored the highest amongst all candidates. This technology is both a Software Development Kit (SDK) and a UI framework with a rich catalogue of ready-to-use widgets that can be further customised via composition. Unlike other popular alternatives like React-Native, Flutter does not rely on an interpreter or a Javascript Bridge to communicate with the native API. Instead, it uses the modern object-oriented programming language called Dart to compile code ahead of time into the native platforms. Additionally, the Flutter team decided to avoid the use of original equipment manufacturer (OEM) and to draw every pixel on the screen from scratch using the Skia 2D engine. This set of strategies make Flutter an absolute champion in terms of performance, as its powerful engine is capable of maintaining 60 frames per second even across complex animations or transitions. Its hot reloading functionality also stands out as it preserves the app's state — meaning that any modifications to the interface during development instantaneously reflect on the interface, without the need of triggering a new build process. Flutter is so promising that in just one year after its first stable release it was voted as one of the most beloved frameworks in the Stackoverflow's 2019 survey. For this project, Flutter's performant engine and seamless integration with Material theming will enable the development of a multi-platform mobile app in record time that conveys brand identity and offers a remarkable user experience.

3.2.2 Database

The primary consideration that must take place when assessing the best database fit for a project is the overall characteristics of the dataset in question. If the data has an intrinsic tabular structure such as an accounting document, then a relational model might be adequate. On the contrary, if the data is too complicated to be expressed as one coherent structure (e.g. engineering parts or geospatial data), then a non-relational or NoSQL database may represent a better option [42]. By carefully observing the system behaviour described in the previous section, it is reasonable to assume that this project requires a model typically seen in e-commerce use-cases. For instance, users browse through lists of Outlets which belong to certain types of Cuisines. Each Outlet has its own menu, which is composed of different Products. These Products can be added to a Cart, which in turn belongs to a specific user Account. Clearly, the application demands several entities that hold strong relationships amongst themselves; users are also required to make payments at a certain point of time. Since the database schema is unlikely to change in the future, and consistency is crucial for this type of application, the relational model was preferred.

Analogous to the frontend, there are numerous relational database solutions available in the market. Stackoverflow's 2019 survey has a dedicated database section which reveals that MySQL, PostgreSQL, and Microsoft SQL are the most commonly used in the industry. PostgreSQL is also ranked first as the most loved and wanted relational database on the list. Microsoft does not stand as an option because it requires a license for commercial use after the application has gained some scale [58]. Between MySQL and PostgreSQL, performance is equivalent even when considering future growth [38]. PostgreSQL is also an object-relational database (supports extra features such as table inheritance and function overloading). Its API adheres closely to SQL standards, and it is famous for protecting data integrity at the transaction level [66]. For all these reasons, PostgreSQL was deemed the best fit.

3.2.3 Backend

The backend middleware is an application that runs on the server-side and acts as a bridge between the frontend and the database. It continuously listens to incoming HTTP requests dispatched by clients and processes them based on the provided pair of URI and HTTP verb (route). This processing triggers code which applies business logic by modifying the request object and querying the database application. Upon conclusion, the middleware outputs an HTTP response back to its clients. Technically, the primary purpose of this project's backend is to provide a secure way for clients to communicate with the database. Additionally, non-functional requirements explicitly state that services must function efficiently, even in high demand scenarios, and sensitive data must be well-protected. All of these requirements can be met by building a representational data transfer (REST) web service API. Similarly to the Frontend decision-making methodology, a set of criteria was elaborated to guide on the middleware technology selection. These are laid out below. A comparison table contrasting the mentioned alternatives can be consulted in Appendix E.

- **Programming language**. The tool must support a dynamic programming language that works well with JSON format and is shipped with helper methods to facilitate data manipulation.
- **Community support**. A vibrant and active community must widely support the tool.
- **Flexibility**. The tool must be light-weight and leave architecture decisions to the developer.

- **Development speed**. The tool must count with a diverse ecosystem of third-party libraries and offer advanced debugging capabilities.
- **Scalability and performance**. When hosted in a remote server, the tool must be able to respond quickly even with a high throughput of HTTP requests.

Node.js stands out as the most flexible, community-engaged, and faster development technology. It is often categorized as a JavaScript runtime environment with some notorious architectural advantages. Unlike traditional web servers which spawn a new thread of execution for every incoming process, Node runs a single-threaded event loop registered with the system to handle all new connections. Each of them triggers a JavaScript callback function that handles HTTP requests asynchronously. The platform rarely performs any direct input or output (I/O) operations, which means no process is ever blocked. Thus, programmers using Node.js do not need to worry about deadlocks. This event-driven environment is built on the top of Google's V8 engine (virtual machine with a built-in interpreter, compilers, and optimizers), which makes it even more performant. Due to its single-thread nature, Node.js is not recommended for CPU-intensive operations [60].

To create RESTful APIs, Node.js is frequently paired with one of its most acclaimed frameworks: Express.js. This library offers a myriad of HTTP utility methods and middleware to accelerate the development workflow [78]. Despite being advertised as a minimalistic framework, it bundles various middlewares within its core that may not always be relevant. For this reason, the team behind Express developed another web framework named Koa.js, which aims to expose a lighter and more expressive API to build robust web applications and APIs. Koa offers better error-handling and allows developers to attach extensions tailored to their needs [49]. Therefore, Node.js and Koa.js have been selected as the foundation of the backend middleware stack.

3.3 Data Model

As discussed previously, the current project brief significantly resembles an e-commerce use-case. Hence, several elements and interactions were assessed and translated into entities and relationships. This process occurred in two steps: first, the principal entities and the way they relate to each other were identified by drawing a condensed, Chen notation entity-relationship diagram (ERD); second, these elements were enhanced with attributes in a Barker notation ERD [8]. These two diagrams can be consulted in Appendix F.

3.4 System Architecture

The client application must be mobile in order to support multiple users from various geolocations. Additionally, to offer the same high-quality experience witnessed in competitors such as UberEATS and JustEat, the best approach is to develop a mobile application that complies to native code instead of a web or hybrid solution [18]. Since the system relies on third-party content to function adequately, and that content must be consistent across all clients, a web service is required. This service must concentrate core business logic so it can be easily modified and redeployed fast without the need of receiving marketplace authority approval. Finally, since the application must be able to scale in case of high demand and sensitive user data is stored, a dedicated database application

becomes particularly relevant. As anticipated in Chapter 2, it was deemed logical to design the system in a 3-tier architecture where a frontend mobile client sends HTTP requests to a backend middleware, which in turn applies business logic and manipulates data stored in a relational database management system. The diagram below summarizes the top-level system architecture.

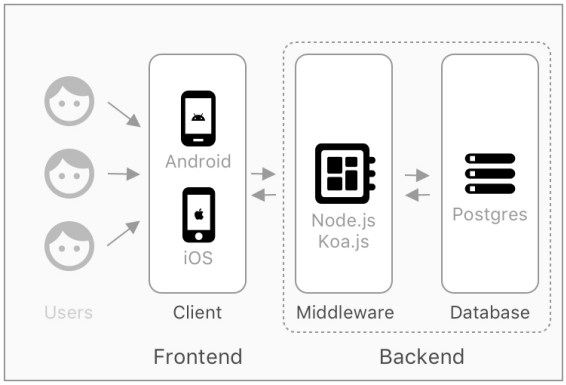


Figure 3.1: High-level system architecture diagram. 3-tiered system interconnecting a mobile frontend application, a backend middleware, and a relational database management system.

The frontend architecture was designed as a set of complementary modules assigned with particular roles. Although interconnected, these modules encapsulate code addressing specific sets of information (concerns) and expose a well-defined interface to enable communication with other modules. This separation of concerns yields advantages such as more agile code maintenance, reusability and ownership. The application also incorporated two modern design concepts: reactive programming and the repository pattern. The first refers to extensive use of the Observer design pattern, where Objects subscribe to events and produce side-effects once they are notified about updates. The second is an abstraction of the frontend data access layer. This paradigm introduces a layer between the business logic and the data sources, establishing a single source of truth for accessing data.

Hence, the mobile app was divided into four layers: presentation, business logic, data access, and data source. The build block widgets (or components) offered by Flutter compose the app's graphical user interface (UI). When a user interacts with a UI element, an event is dispatched to the business logic layer, which adopts the modern Business Logic Component (BLoC) design pattern. This pattern enforces the separation of business logic from the UI, making it more testable and maintainable. It maps the incoming event to an output state. If necessary, it requests data to the repository (data access layer). The repository verifies if the requested data has already been cached in the local data source. If not, it fetches data through the remote data source, which will send an HTTP request asynchronously to the backend middleware. When data has arrived, the flow of information goes on the opposite direction, providing data to the BLoC, which in turn emits a new state to the presentation layer. The UI reacts to this new state by rebuilding its elements. The result is a predictable flow of interaction, where each part of the code has a well-defined purpose. The diagram below summarizes this notion.

A similar rationale was adopted for the backend. It was designed as a multi-layered application with a Controller, Business Logic, and Data Access layer. The router controller is responsible for interpreting incoming HTTP requests and redirecting them to relevant services, which contain business logic that modifies the request body and request data to the Data Access layer. Next, this layer interfaces with the database management system to retrieve entities and forward them to the calling service. Once data has arrived retrieved, the flow goes backwards, being consumed by a

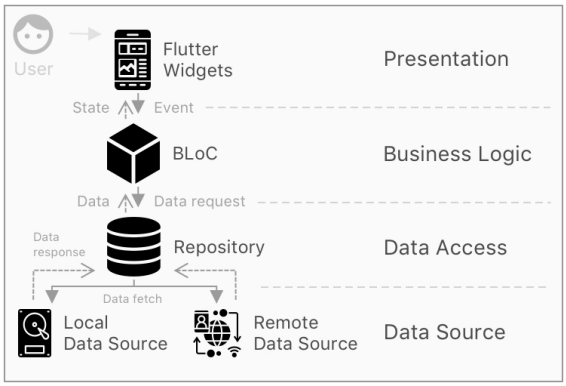


Figure 3.2: Mobile frontend architecture.Presentation, Business Logic, Data Access, and Data Source layers. The architecture leverages the concepts of Reactive Programming, BLoC state management architecture [13], and Repository.

service, which outputs an adequate response body that is sent back to the client via the controller. The interaction between the client application and the backend middleware was established via a representational state transfer application programming interface (RESTful API) [57].

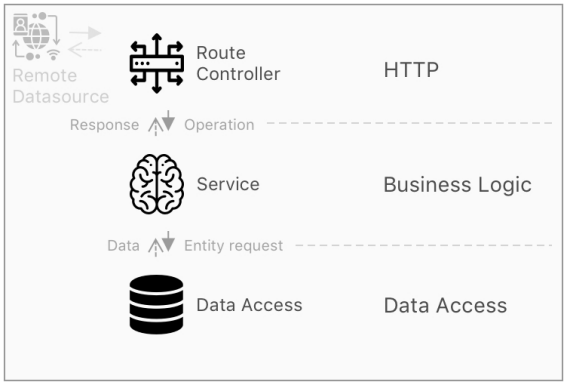


Figure 3.3: Backend middleware architecture. HTTP (Controller), Business Logic, and Data Access layers.

3.5 Workflow Methodology

Considering the project's short 2-month timeframe, its broad scope, and the heavy use of new technologies, agility seemed like an essential trait for a successful workflow. Therefore, all applications from this project were developed in line with the Agile manifesto principles of "Individuals and Interactions Over Processes and Tools", "Working Software Over Comprehensive Documentation", "Customer Collaboration Over Contract Negotiation", and "Responding to Change Over Following a Plan" [10]. In practice, an extension of Scrum [7] with the use of Kanban boards [1] was adopted as the project management methodology.

Essentially, several user stories were generated by revisiting the project requirements. These were broken down into technical tasks and prioritised in the backlog to guarantee that by the end of each 2-week Sprint real value would be added to the client. The completion of two or more Sprints led to the achievement of milestones. By the end of each week, a sprint review and planning was carried out with the assigned supervisor. These were short (30 minutes long) and served as checkpoints to guarantee that enough value was being added. Appendix G contains the Gantt chart used to track the project's progress and the Kanban board used to organise development.

Chapter 4: Implementation

This project was conceived over the course of five Sprints, each of which comprised two weeks. The first three were dedicated to laying down a solid foundation and achieving the first minimum viable product (MVP) milestone — that is, the "must-have" features from Chapter 2. By the end of that phase, users were allowed to go through a full Customer Journey cycle. The two remaining Sprints were used to increment the product with convenience features that enhanced the overall experience; for instance, user authentication. The next subsections discuss implementation details and challenges faced along the way.

4.1 Database

The PostgreSQL [66] database and schema were created by issuing several standard SQL commands via pgAdmin's query tool [64]. In total, 26 tables representing entities and relationships were added [1] according to the entity-relationship diagram from Chapter 3. Each command also defined constraints such as the primary key (auto-generated integer identifier), foreign keys, nullable restrictions, and update/deletion triggers. The code snippet below illustrates the SQL commands issued to create the outlets' table.

```sql
-- Create outlets table
CREATE TABLE outlets (
    id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    cuisine_id INT NOT NULL,
    title CITEXT NOT NULL,
    rating NUMERIC(2,1) DEFAULT 0.0,
    price_level INT NOT NULL,
    location_id INT NOT NULL,
    CONSTRAINT fk_cuisine
      FOREIGN KEY(cuisine_id)
      REFERENCES cuisines(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_location
      FOREIGN KEY(location_id)
      REFERENCES locations(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT price_level_limit
        CHECK (price_level < 6.0),
        CHECK (price_level > 0.0),
    CONSTRAINT rating_limit
        CHECK (rating < 6.0),
        CHECK (rating >= 0.0)
);
```

SQL

Figure 4.1: SQL commands issued to create the outlet table in the PostgreSQL database.

Two extension modules were included to complement PostgreSQL standard library: pgcrypto and citext. The first added cryptographic functions which were used to hash account passwords [65]. The second served to facilitate case insensitive comparison between text attributes [84].

[1] Consult the Github repository for the full list of SQL commands: `https://shorturl.at/fosHN`

4.2 Backend Middleware

Regardless of the feature in question, they all involved listening for client inputs and querying the database accordingly. Thus, the same concept was frequently reapplied. A JavaScript file named *www.js* was created and set as the application's entry point. This file read from environment variables (*env.yaml*) and started an HTTP server enhanced in *server.js* with the nine middlewares listed below.

- **errorHandler**: caught any errors and populated the context object provided by Koa with the appropriate HTTP status code and error message.
- **compress**: imported from koa-compress add-on dependency [45]. This library acted on reducing the response payload size.
- **respond**: imported from the koa-respond add-on dependency [48]. This library acted on injecting a number of utility methods into Koa's context object for generating responses.
- **cors**: imported from @koa/cors add-on dependency [47]. This library allowed cross-origin recourse sharing (CORS).
- **bodyParser**: imported from koa-bodyparser add-on dependency [46]. This library facilitated parsing HTTP POST body from formats such as JSON or XML.
- **scopePerRequest**: together with the awilix-koa dependency, [40] this middleware implemented the concept of dependency injection, where objects such as services were provided to other parts of the application if previously registered [57]. This implementation of the inversion of control pattern contributed to achieving a better separation of concerns, which consequently increased code reuse.
- **registerContext**: added request-specific data (e.g. user identification) to the scope.
- **loadControllers**: registered controllers (routes) with the support of awilix-koa [40].
- **notFoundHandler**: added a default feedback message in case of HTTP status code 404.

Bristol (logger) [71] and palin (formatter) [36] dependencies were combined to create a custom logger object that improved the overall debugging experience in Node.

Besides the server configuration, a connection with the database had to be established. To accomplish that, both pg and pg-hstore packages were used. The first is a non-blocking PostgreSQL client for Node [14], and the second provided an abstraction for serialization and deserialization of JSON data to the hstore format [15]. The project also incorporated two tools that were crucial to productivity: Sequelize [68] and sequelize-auto [69]. Sequelize acted as a promise-based object-relational-mapping (ORM) for Node. sequelize-auto allowed the generation of Sequelize models automatically based on the pre-defined database schema.

All database-related files were placed under the database directory. After declaring some custom configurations on *db-options.json*, the following script was written and ran to generate Sequelize entity models automatically:

To teach Sequelize about the entity relationships, an additional file named *associations.js* was created under the same directory. Code snippet 4.3 shows how one-to-many association were programmed.

```bash
node ./node_modules/sequelize-auto/bin/sequelize-auto -o
"./src/database/models" -d digital_restaurant -h localhost
-u postgres -p 5432 -x postgres -e postgres -C ut -l esm
-f ut -c "src/database/config/db-options" -s public
```

Bash

Figure 4.2: Bash script created to generate Sequelize models based on the PostgreSQL schema automatically.

```javascript
// Cuisine relationship
Images.hasMany(Cuisines, {
    foreignKey: 'imageId',
});
Cuisines.belongsTo(Images, {
    foreignKey: 'imageId',
    as: 'image'
});
```

JavaScript

Figure 4.3: Javascript code used to associate Cuisine models to Image models. One-to-many relationship.

All database configurations were declared in the *index.js* file under the models directory. These configurations involved initializing the Sequelize object with the proper database credentials (retrieved from environment variables), loading the models, and making associations known to the object. The database was then registered as an injectable dependency inside *container.js*.

The three remaining pillars of the backend application were the routes, services, and store directories. These were populated as the project progressed through new features, but the 3-layered architecture blueprint was put in place since the beginning to guarantee well-defined responsibilities.

- **Routes**: the API controller abstraction. This part of the application was the entry point for clients. Each new endpoint had to be declared in here. For instance, the *cuisine-api* created its controller by mapping API methods to route paths.

```javascript
/**
 * Cuisine API controller.
 * This file contains functions which map HTTP calls to CuisineService methods.
 */
const api = cuisineService => ({
    findCuisines: async ctx => ctx.ok(await cuisineService.find(ctx.query)),
    getCuisine: async ctx => ctx.ok(await cuisineService.get(ctx.params.id)),
});

export default createController(api)
    .prefix('/cuisines')
    .get('', 'findCuisines')
    .get('/:id', 'getCuisine');
```

Figure 4.4: Javascript code that maps HTTP calls to service methods.

- **Services**: the business logic abstraction. Methods included here manipulated request objects and accessed relevant database data via stores – Figure 4.5.
- **Stores**: data access abstraction. Here the application communicated directly with the database via the Sequelize model object (e.g. Cuisine) – Figure 4.6.

```javascript
/**
 * Fetches all cuisines according to the given parameter.
 * @param params optional parameters to filter the result.
 * @returns {Promise<{result: Cuisines[]}>}
 */
async find(params) {
    const result = await this.cuisineStore.find();
    return {result};
}
```

JavaScript

Figure 4.5: Javascript code that prepares the response object.

```javascript
/**
 * Retrieves all cuisine entities from the database.
 * @returns {Promise<Cuisines[]>}
 */
async find() {
    logger.debug('Retrieving cuisines...')
    return await Cuisine.findAll(options);
},
```

JavaScript

Figure 4.6: Javascript code that accesses the database with the help of Sequelizer ORM.

In summary, a predictable data flow was put in place. The client sent a request to a specific route declared in the controller, which mapped the request object to an appropriate service. Finally, the service performed business logic and retrieved data through the stores.

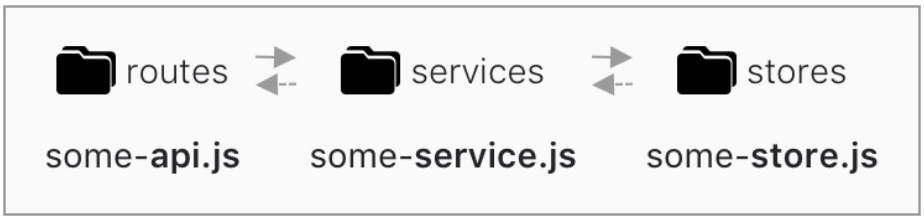


Figure 4.7: Layered data flow in the backend middleware. The client reaches a route controller which propagates the request to a service. The service shapes data retrieved from a store and returns it to the client in an HTTP's response body.

4.3 Frontend

Flutter favours composition over inheritance [28], which means that rather than extending from parent classes and overriding methods, UI elements are built by combining simpler UI objects. In practice, this means that the graphical user interface is written in a declarative style, where new objects are instantiated and passed around as parameters to other objects' constructors. This forms a tree structure that represents more sophisticated pieces of UI. In general, these building blocs are called by Flutter as Widgets.

The first file created was main.dart. This entry-point contained the app's main method and its top-most widget, which defined core aspects of a material design app [31]. To this concise but essential stateless widget were given the app's theme object and initial route.

```dart
/// The App's entry-point. Runs the [AppContainer] widget defined below.
void main() {
  Bloc.observer = AppBlocObserver();
  final router = AppRouter();
  runApp(AppContainer(
    router: router,
  ));
}

/// A [StatelessWidget] that provides app-wide configurations.
class AppContainer extends StatelessWidget {
  static const isFirstLaunchKey = 'is-first-launch';
  final AppRouter router;

  const AppContainer({Key key, this.router}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: LUTheme.of(context), // App styles
      onGenerateRoute: router.onGenerateRoute, // Route registration
      home: OnboardingScreen(router: router), // Illustrative initial route
    );
  }
```

Figure 4.8: Dart code that configures the Flutter application.

The *LUTheme* class was created to centralise material design theme customisations. For instance, in line with the original design plan, the primary and accent colours were respectively set as navy blue and orange. This implementation made it trivial to perform any style modifications in the future — even if those modifications meant a complete re-branding. Besides, the app was ready to accommodate a future dark mode.

Another fundamental concept for any mobile app is navigation, which allows users to explore different portions of the app. Flutter calls these portions "routes", and they have to be pre-declared in order for a global *Navigator* API to manage their transactions. For this project, navigation was slightly more complicated than the usual cases. Not only the routes had to be declared, but also a tab-based navigation system had to be implemented. This was achieved with the classes under the navigation directory.

As observed in the code snippet above, the app's initial route was set as the *OnboardingScreen*. This screen eventually takes users to the app's primary content, which is formed by five tabs placed in a bottom bar. Behind the hood, these tabs are customised buttons that switch the order of widgets contained in a 5-level stack. The stack is composed by *NavigatorContainer*s, that receive available routes through the *onGenerateRoute* parameter. Each time a user presses a new tab, the app sends the appropriate *Navigator* [32] to the top of the stack, making it visible to the user.

The communication with the backend was accomplished via the networking system, which is formed by the interactions of services, repositories, and models. In essence, data is requested via a repository class, which encapsulates calls to remote or local data sources (Figure 4.9).

The remote data source (e.g. *CuisineApiClient*) performs an asynchronous operation that ultimately sends out an HTTP request. One example is the *getCuisines* method, where a GET request is sent out to the server. Since the majority of these requests follow the same structure, the *BaseApiClient* class was created to minimize code repetition. This class took advantage of the http dependency to create and dispatch the request (Figure 4.10.

In the example above, the *cuisineApiClient* was eventually notified of a server response. In that case, it immediately redirected the retrieved data above to the calling *CuisineRepository*, which in

```dart
/// This class is the only data source for the Cuisine entity.
/// It should be seen as the single source of truth for fetching
/// or storing data.
class CuisineRepository {
  final _cuisineApiClient = CuisineApiClient();

  /// Retrieves all stored Cuisine entities
  Future<List<Cuisine>> fetchAllCuisines() async {
    final body = await _cuisineApiClient.getCuisines();
    return body.results;
  }

  /// Retrieves a single Cuisine entity by the given identifier
  Future<Cuisine> fetchCuisine(int id) async {
    // (...)
  }
}
```

Figure 4.9: Single source of through for manipulating the Cuisine model.

```dart
/// Abstracts GET http request boilerplate.
  @protected
  Future<String> getRequest(String path, [Object params]) async {
    try {
      final uri = Uri.http(_BASE_URL, path, params);
      debugPrint('GET request to ${_BASE_URL + path} with params $params');
      final response = await _client.get(uri);
      return _filterResponseBody(response);
    } catch (e) {
      debugPrint(e);
      throw FetchDataException('GET request failed');
    }
  }
```

Figure 4.10: Dart code abstraction that enables GET request with ease.

turn created an isolate [29]. Inside this isolate, JSON deserialisation was delegated to the relevant model class. Once deserialised, the domain model was then returned as the output of the repository method — in this case, *getCuisines* returning a collection of parsed *Cuisines*.

To accelerate development, JSON deserialization counted with the support of the json_annotation package, which auto-generated the *fromJson* method based on properties declared in the data model class (Figure 4.11).

Features from the backlog were implemented in a similar pipeline. First, material widgets provided by the Flutter SDK were combined to create customised versions that expressed the app's design. Then, these widgets were placed under the components directory and mixed and matched to create screens. As explained previously, screens were assigned with an identifier and registered as routes so that their content could be accessible from various parts of the app.

Tab-bar screens were considered gateways to other features. For instance, the *HomeScreen* displayed material Cards that allowed users to view restaurants and their menus. Additionally, this same screen contained a *LUIconButton* that could be tapped on to navigate to the *ScannerScreen*. In contrast, this other screen had one clear goal: reading QR-codes. To do so, it imported a forked version of the qr_code_scanner [76] library. Upon identifying a QR-code, this modal screen un-

```dart
/// A representation of the Cuisine entity.
@JsonSerializable()
class Cuisine {
  final int id;
  final String title;
  final Image image;

  Cuisine(this.id, this.title, this.image);

  static Cuisine fromJson(Map<String, dynamic> json) => _$CuisineFromJson(json);

  Map<String, dynamic> toJson() => _$CuisineToJson(this);
}
```

Figure 4.11: Dart code representing the Cuisine model.

wrapped the encoded value and returned it to its parent route; in the scenario, the *HomeScreen*.

As postulated in Chapter 3, business logic had to be detached from the presentation layer. This was achieved by creating intermediary classes named "blocs". There, the Business Logic Component pattern was implemented with the help of the bloc and flutter_bloc dependencies [2] and consumed by screens to put in place the adequate behaviour.

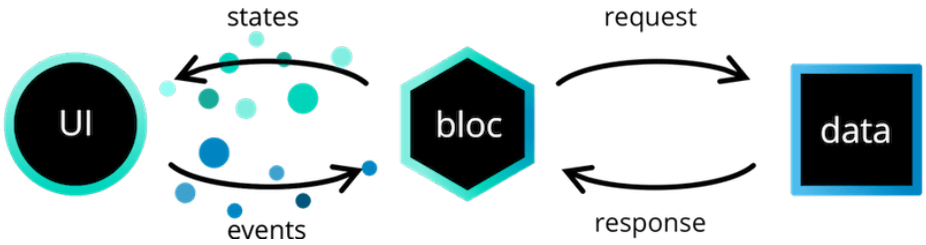


Figure 4.12: The bloc architecture [2]

The *HomeScreen* offers a good representation of how features were implemented. Its presentation layer consists of a stateful widget composed by two major UI fragments: header and content. These fragments accommodate a series of custom widgets that result in the screen graphical representation. However, most of content depends on data stored in the backend. Therefore, as soon as the *HomeScreen* is instantiated in the *AppRouter*, it also emits a data request event to the *HomeBloc*. Then, the bloc maps this event to a state that is generated by accessing data via the appropriate repository. Finally, this state is sent back to the screen, which rebuilds its widget tree with the updated state.

```dart
// Inside the HomeScreen, outlets and cuisines are requested
@override
  void initState() {
    // (...)
    _homeBloc = _homeBloc = BlocProvider.of<HomeBloc>(context)
      ..add(CuisinesRequested())
    // (...)
  }
```

Figure 4.13: Dart code initiating the *HomeBloc* and adding a *CuisineRequested* event to the stream.

```dart
// The HomeBloc maps events to states
@override
  Stream<HomeState> mapEventToState(
    HomeEvent event,
  ) async* {
    switch (event.runtimeType) {
      case CuisinesRequested:
        yield* _mapCuisinesRequestedToState();
        break;
     // (...)
    }
  }

// The state is built by retrieving data from the repository
Stream<HomeState> _mapCuisinesRequestedToState() async* {
    yield CuisineLoadInProgress();
    try {
      final List<Cuisine> cuisines = await cuisineRepository.fetchAllCuisines();
      yield CuisineLoadSuccess(cuisines: cuisines);
    } catch (error) {
      yield CuisineLoadFailure();
    }
  }
```

Figure 4.14: Dart code illustrating how the *HomeBloc* maps events into state.

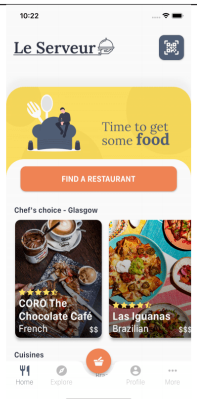
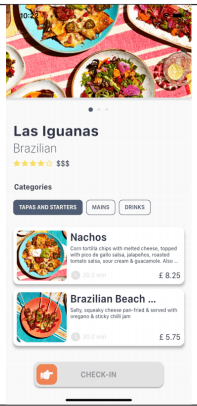
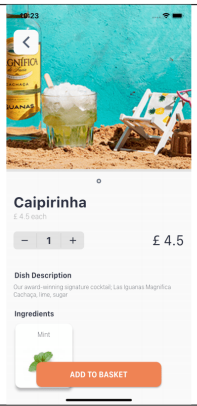
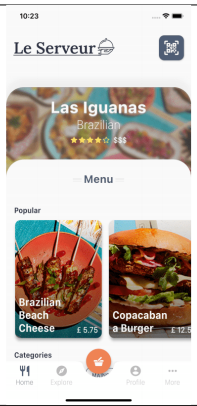
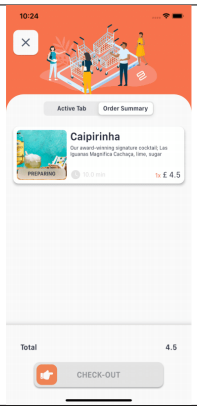
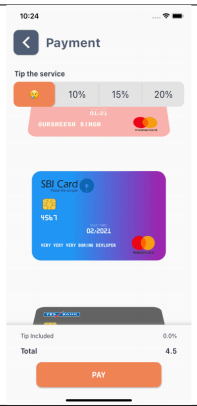
<table>
<tr><th colspan="3">Implemented Screens Preview</th></tr>
<tr><th>Checked-out Home Screen</th><th>Outlet Screen</th><th>Product Screen</th></tr>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
<tr><th>Checked-in Home Screen</th><th>Cart Screen - Order Summary</th><th>Payment Screen</th></tr>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
</table>

Table 4.1: Screenshots of 6 implemented screens.

For more details about the Le Serveur and its implementation, please consult the **User Guide** document available in the compressed folder or access `https://shorturl.at/zCLR4`. A **demonstration video** is also available at `https://youtu.be/ozpR2CIe0yM`.

Chapter 5: Evaluation

5.1 Software Testing

Software testing is critical to preserve the quality of an application as it scales and absorbs new features. For this project, two test strategies were adopted: manual and automated testing. Manual testing was carried out by defining several black-box functional test cases, which covered the app's behaviour from an end user's perspective. The main goal of this strategy was to ensure a usable application, where a full Customer Journey cycle could be completed. Automated testing took the form of 33 unit tests targeting the frontend business logic components. These tests covered all mappings from events to states, which contributes to preserve original business logic and minimize defects caused by future modifications to the code. Due to the time constraints and the scope of this research project, it was not possible to achieve full unit test coverage.

5.1.1 Automated Tests

The architectural decision of encapsulating business logic in dedicated objects facilitated the process of unit testing. Three popular libraries were used to assist in the process:

- **test**: provides standard methods for writing unit tests in Dart.
- **mockito**: emulates the response of repositories or web services.
- **bloc_test**: emulates the emissions of bloc states.
- **flutter_test**: exposes constructs by which tests can be configured.

The code snippets exhibited in Figures 5.1 and 5.2 illustrate two test cases covering the *OutletsRequested* event. There are two possible outcomes. In case of a successful *fetchAllOutletProducts* operation, the bloc must emit a loading state followed by a success state, which also contains the category map fetched by the outlet repository. On the other hand, if an error occurs in the middle of the process, the bloc is expected to emit a loading state followed by an error state, which will be handled by the presentation layer in due time. The reader can issue the command below at the project's directory to execute all unit test cases.

```
$ flutter test
00:10 +33: All tests passed!
```

Figure 5.3: Flutter test command followed by its result.

```dart
blocTest<OutletBloc, OutletState>(
      'should fetch outlet products when requested',
      build: () {
        when(outletRepository.fetchAllOutletProducts(1))
            .thenAnswer((_) => Future.value(_mockProductList));
        return outletBloc;
      },
      act: (bloc) async {
        final event = OutletProductsRequested(outletId: 1);
        bloc.add(event);
      },
      expect: <OutletState>[
        OutletState.loading(),
        OutletState.success({_mockCategory: _mockProductList}, null),
      ],
);
```

Figure 5.1: This code tests the *OutletProductsRequested* event mapping in *OutletBloc*

```dart
blocTest<OutletBloc, OutletState>(
      'should emit error when fetchAllOutlets operation fails',
      build: () {
        when(outletRepository.fetchAllOutletProducts(1))
            .thenAnswer((_) => throw (FetchDataException('')));
        return outletBloc;
      },
      act: (bloc) async {
        final event = OutletProductsRequested(outletId: 1);
        bloc.add(event);
      },
      expect: <OutletState>[
        OutletState.loading(),
        OutletState.error(),
      ],
    );
```

Figure 5.2: This code tests error schenario of the *OutletProductsRequested* event in *OutletBloc*

5.1.2 Manual Tests

Manual test cases were created to validate app flows and guarantee that users were able to check-in restaurants, browse menus, place orders, and check-out adequately. The methodology adopted to accomplish that goes by the name of Black Box Testing. It consists of testing software functionalities without any knowledge of its implementation details. This form of testing focuses on the actions taken by an imaginary user whose expectations derive solely from software requirements and specifications [22]. Tables 5.1 and I.1 (Appendix I) summarise the tests conducted by the describing test cases, expected results, actual results and final outcome.

Black-box Testing Table				
#	Test Description	Expected Result	Actual Result	Outcome
1	User launches the app for the first time	Onboarding flow is displayed	Onboarding flow is displayed	Pass
2	User launches the app more than once	Main content is displayed	Main content is displayed	Pass
3	User presses the skip button at the Onboarding screen	Main content is displayed	Main content is displayed	Pass
4	User presses the login button at the Onboarding screen	Authentication flow is displayed	Authentication flow is displayed	Pass
5	[Home] User taps on the top right button with a QR-Code icon	Scanner screen is displayed	Scanner screen is displayed	Pass
6	[Scanner] User scans a Le Serveur's QR-Code	Check-in alert dialog is displayed	Check-in alert dialog is displayed	Pass
7	[Scanner] User scans an known QR-Code	Error dialog is displayed	Error dialog is displayed	Pass
8	[Scanner] User presses on the check-in button in the alert dialog	Scanner screen is poped and Home screen content changes to reflect the checked-in restaurant	Scanner screen is poped and Home screen content changes to reflect the checked-in restaurant	Pass

9	[Home] User enters the checked-out Home screen	Featured outlets, cuisines, and nearby outlets are loaded from the backend	Featured outlets, cuisines, and nearby outlets are loaded from the backend	**Pass**
10	[Home] User taps on a Chef's choice Outlet Card	Outlet Screen is displayed	Outlet Screen is displayed	**Pass**
11	[Home] User taps on a Cuisine Card	Cuisine Screen is displayed	Cuisine Screen is displayed	**Pass**
12	[Home] User taps on a nearby restaurant Outlet Card	Outlet Screen is displayed	Outlet Screen is displayed	**Pass**
13	[Outlet] User scrolls across the restaurant's image galery at the top	Available images are switched	Available images are switched	**Pass**
14	[Outlet] User taps on distinct categories (e.g. Drinks)	Products associated with the selected category are displayed (e.g. Caipirinha)	Products associated with the selected category are displayed (e.g. Caipirinha)	**Pass**
15	[Outlet] User swipes the check-in slider	Alert dialog is presented	Alert dialog is presented	**Pass**
16	[Outlet] User taps on the check-in button when the alert dialog appears	User is redirected to the checked-in Home screen	User is redirected to the checked-in Home screen	**Pass**
17	[Outlet] User taps on the cancel button when the alert dialog appears	Alert dialog is hidden	Alert dialog is hidden	**Pass**
18	User taps on the back button at the top app bar	Current screen is poped from the navigation stack	Current screen is poped from the navigation stack	**Pass**
19	[Cuisine] User enters the Cuisine screen	A list of outlets belonging to the selected cuisine is displayed	A list of outlets belonging to the selected cuisine is displayed	**Pass**
20	[Cuisine] User taps on an Outlet Card	Outlet Screen is displayed	Outlet Screen is displayed	**Pass**
21	[Home] User enters the checked-in Home screen	The checked-in restaurant is loaded from the backend; popular dishes and the menu are displayed	The checked-in restaurant is loaded from the backend; popular dishes and the menu are displayed	**Pass**
22	[Home] User taps on a popular product	Product Screen is displayed	Product Screen is displayed	**Pass**
23	[Home] User taps on a product category (e.g. Cakes)	Products associated with the selected category are displayed (e.g. Brownie)	Products associated with the selected category are displayed (e.g. Brownie)	**Pass**
24	[Product] User taps on the stepper button	Product quantity is modified and total price is recalculated	Product quantity is modified and total price is recalculated	**Pass**
25	[Product] User taps on an Ingridient Card	No side-effects happen	No side-effects happen	**Pass**
26	[Product] User taps on the "Add to cart/tab/basket" button	The selected product is added to the cart screen and the user is redirected to the Home screen	The selected product is added to the cart screen and the user is redirected to the Home screen	**Pass**
27	[Product] User navigates back to a product that has been added to the card	User sees the same product quantity which was added to the cart	User sees the same product quantity which was added to the cart	**Pass**
28	[Product] User tries to reduce the product quantity to zero	The quantity is prevented from reaching zero	The quantity is prevented from reaching zero	**Pass**

Table 5.1: Test table – part one. Each row defines a test case that covers a particular aspect of the app's behaviour.

5.2 Requirements Status

Chapter 2 presented a list of functional requirements which were devised for a production-ready version of the application. As development progressed, it became apparent that due to the time constraints and all complexities involved, not all features would make it to the submission version. For each of them, GUI elements and state management had to be implemented in the frontend, whereas database entities had to be created, manipulated, and exposed in the backend. Thus, priority was given to features deemed as "Must have" in the MoSCoW table. A final overview of which requirements were met can be consulted in Appendix H.

5.3 Evaluation

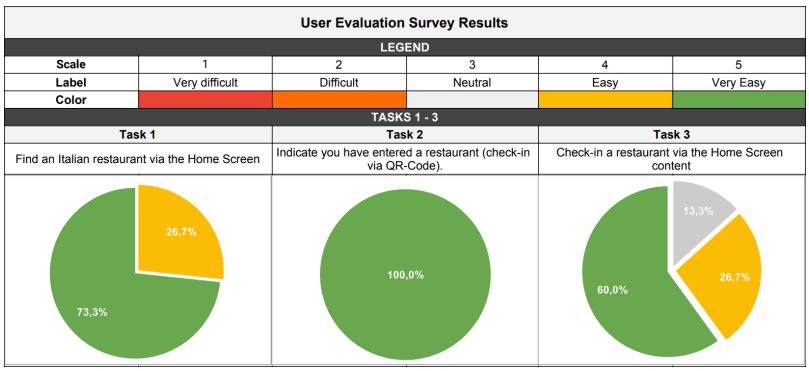
5.3.1 Strategy

In order to evaluate the system, fifteen potential users from varying demographics were invited individually into Zoom [90] video meeting at a pre-scheduled appointment time. This medium was chosen due to restrictions imposed by Apple and Google's distribution policies, which prevents developers from releasing apps mostly populated with fictional data. When participants joined, they were presented with both the Android and iOS applications running locally in the author's machine. In sequence, they were asked to take part in a user evaluation survey containing eleven tasks spread across nine sections which were designed to evaluate the app's ease of use and efficacy.

Each section introduced a specific application context. Its questions tasked users with simple missions that stimulated the exploration of app features and were placed in a particular order to guide users through all Customer Journey stages (explore, check-in, choose, order, and check-out). Participants were asked to communicate which steps they would take to accomplish each mission out loud, and the author immediately reproduced these actions in real-time. For each task, there were two possible outcomes: success or failure. In case of failure, the author revealed in which ways a particular task could be completed. Upon its completion, participants were asked to describe their actions in bullet points and rate how easy it would be to perform them if they could use a copy of the app. The system used for the rating was a 5-point Likert scale [52] with options ranging from "Very difficult" to "Very easy". A "Future Work" section was placed at the end of the survey form. There, participants could optionally choose to answer two open-ended questions to suggest improvements or request new features for the app. To consult the survey in full, please visit the survey folder submitted along the project's code [1].

5.3.2 Results

Responses from 15 participants were collected in total. Out of these, 60% were male, and 40% were female. Ages ranged from 22 to 54, with an average situated at 29 years old. In average, participants claimed that they usually went to restaurants three times per week — before the COVID-19 outbreak. 100% of the tasks were completed successfully. 60% of the participants suggested improvements and 70% requested at least one additional feature. The figure below summarises the survey results.



[1] Survey results can also be accessed online through this link: `https://shorturl.at/gxHLS`

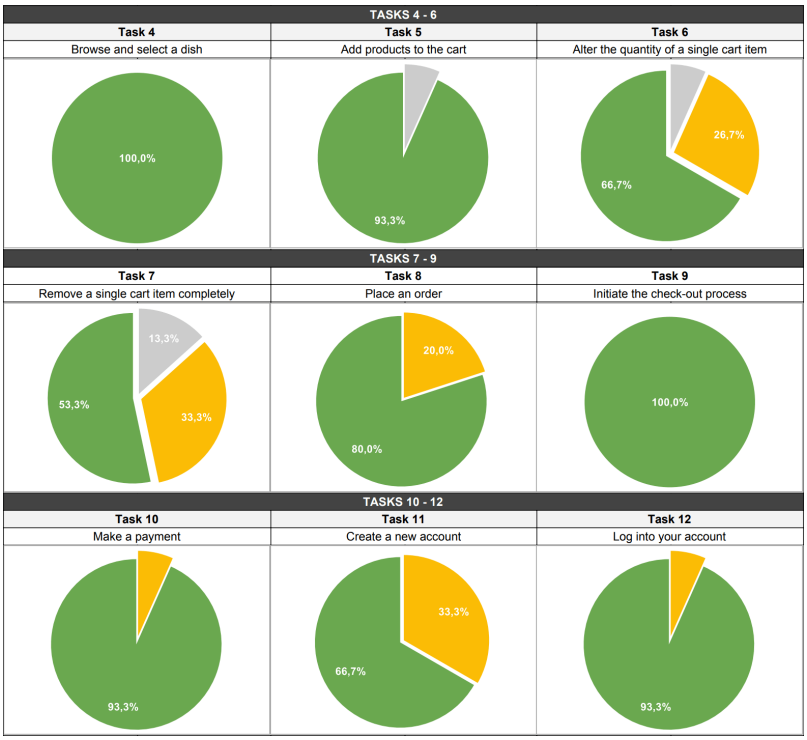


Figure 5.4: A summary of the results gathered from 15 participants who took part in the user evaluation survey.

The results suggest that users had no significant issues when performing tasks related to "must-have" features. Hence, it is highly probable that future users will be capable of going on a full Customer Journey cycle without impediments. The convergence of the step-by-step descriptions indicates that participants felt the app intuitive to use. However, before future releases, particular attention should be given to the removal of cart items. Nearly half of the participants did not consider this operation "very easy" to perform. A feasible improvement would be to implement a swipe to delete gesture or to include a cancel button next to the item's stepper.

In the last section, nine participants provided feedback for improvements and suggestions for additional features. Generally, users complained about the tab bar item colours when unselected, claiming that the contrast was so low that it was difficult to read. The "tab" icon was also criticised. At least three people took a slightly longer to understand that they could navigate to the cart screen through this button. Another suggestion was to redirect users to the menu after they have added a product to their carts. Finally, one participant reported that he did not recognise which credit cart was selected at first in the payment screen. He suggested to include a coloured border to highlight the selection. Some of these suggestions were implemented prior to submission.

In terms of additional features, users requested the ability to save previous orders, place orders before arrival at outlets, personalised recommendations, credit card scanning, directions to restaurants, restaurant reservations, "request help from attendant", payment with cash, promotions, rewards program, user reviews, table selection, and food delivery. An interesting observation is that some of these features (e.g. credit card scanning) were already included in the product's prototype, but did not make to the submission version due to its low development priority. The main takeaway extracted from the user evaluation survey is that the app was deemed useful by its testers and that there is enormous growth potential, notably in the unprecedented context imposed by the COVID-19 pandemic.

Chapter 6: Conclusion

The purpose of this research project was to devise a cross-platform mobile application that reimagined the eating out experience in view of the disruptions caused by the COVID-19 pandemic. The resulting product sought to maximise convenience and, at the same time, facilitate compliance with safety measures imposed by governments around the world. By adopting ”Le Serveur”, users were able to explore restaurant options, check-in at selected outlets, pick their favourite dishes from food menus, place orders, and express check-out at their preferred time. The present report guided the reader through all steps undertaken to research, conceptualise, implement, and evaluate the systems designed to enable the product.

There is a plethora of possibilities for future work. First, some of the improvements suggested in the user evaluation survey could be easily incorporated. Namely, the swipe to delete in the cart screen and the highlighted credit card border to indicate selection in the payment screen. Some of the additional features requested in the survey can find the application in minimal time. For instance, directions to selected restaurants, ”request help from attendants”, and user reviews. Nevertheless, the most important step towards the future would be to persuade real restaurant owners to join the platform. After that, the applications can be deployed in Apple and Google’s digital marketplaces. Future work should consider concluding the implementation of the behaviours planned in Chapter 3 — specially the map exploration and search functionalities. Some interesting projects that could derive from this product would be a client application for restaurant owners to register their outlets and menu options. To increase competitiveness, this app’s backend could be integrated to restaurants’ point of sale systems to offer a fully automated solution.

In summary, despite a few incomplete features, this research project has successfully achieved the goals set in Chapter 2. Within two months, a cross-platform mobile application, a backend middleware, and a relational database were devised to allow restaurant owners to provide contact-free yet high-quality order and pay services. These applications adopted modern technologies and followed industry best practises for software engineering. Feedback extracted from the user evaluation survey confirmed the efficacy of the endeavour, which also carries enormous commercial potential that could be further explored in different circumstances. To conclude, this project aspired to offer an alternative approach to ensure the safety of customers in times of pandemic and democratise high-levels of service in all kinds of restaurants.

Bibliography

[1] Agile Alliance. *Glossary; Kanban*. 2018. URL: https://www.agilealliance.org/glossary/kanban/ (visited on 08/04/2020).

[2] Felix Angelov. *Bloc library*. 2019. URL: https://pub.dev/packages/bloc (visited on 08/31/2020).

[3] App Annie. *State of Mobile 2020*. Tech. rep. App Annie, 2020, pp. 1–49. URL: https://www.appannie.com/en/go/state-of-mobile-2020/.

[4] App Annie. *Uber Eats App Rank History*. 2020. URL: https://www.appannie.com/en/apps/ios/app/ubereats-food-delivery-faster/ (visited on 07/04/2020).

[5] Apple. *Human Interface Guidelines*. 2020. URL: https://developer.apple.com/design/human-interface-guidelines/ (visited on 07/04/2020).

[6] Apple. *iOS Development Program*. 2020. URL: https://developer.apple.com/ios/ (visited on 07/20/2020).

[7] Atlassian. *Modern practices and where it's headed*. 2020. URL: https://www.atlassian.com/software-development (visited on 08/03/2020).

[8] Richard Barker. *CASE Method: Entity Relationship Modelling*. Reading, MA: Addison-Wesley Professional, 1990. ISBN: 0201416964.

[9] Clayton Barrows and Tom Powers. *Management in the hospitality industry*. Ed. by Inc John Wiley & Sons. 9th ed. New Jersey: John Wiley & Sons, Inc., Hoboken, New Jersey. ISBN: 978-0-471-78277-3.

[10] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/ (visited on 08/03/2020).

[11] Natasha Bernal. *Uber Eats has lost its biggest advantage in the food delivery war*. 2020. URL: https://www.wired.co.uk/article/uber-eats-mcdonalds (visited on 07/05/2020).

[12] Per Block et al. "Social network-based distancing strategies to flatten the COVID-19 curve in a post-lockdown world". In: *Nature Human Behaviour* 4.6 (June 2020), pp. 588–596. ISSN: 2397-3374. DOI: 10.1038/s41562-020-0898-6. arXiv: 2004.07052. URL: http://dx.doi.org/10.1038/s41562-020-0898-6%20http://www.nature.com/articles/s41562-020-0898-6.

[13] Didier Boelens. *Reactive Programming - Streams - BLoC*. 2018. URL: https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/ (visited on 08/01/2020).

[14] Brian Carlson. *node-postgres*. 2019. URL: https://github.com/brianc/node-postgres/tree/master/packages/pg (visited on 08/31/2020).

[15] Seth Carney. *pg-hstore*. 2014. URL: https://github.com/scarney81/pg-hstore (visited on 08/31/2020).

[16] CB Insights. *The Future of Dining: 89+ Startups Reinventing The Restaurant In One Infographic*. Tech. rep. CBInsights, 2017. URL: https://www.cbinsights.com/blog/restaurant-tech-market-map-company-list/.

[17] Chiao Chen Chang. "Exploring mobile application customer loyalty: The moderating effect of use contexts". In: *Telecommunications Policy* 39.8 (2015), pp. 678–690. ISSN: 03085961. DOI: 10.1016/j.telpol.2015.07.008. URL: http://dx.doi.org/10.1016/j.telpol.2015.07.008.

[18] Clearbridge Mobile. *Mobile App Vs. Mobile Website: A UX Comparison – Which Is The Better Option?* 2020. URL: https://clearbridgemobile.com/mobile-app-vs-mobile-website-which-is-the-better-option/ (visited on 08/01/2020).

[19] Comscore. *Global State of Mobile 2019*. Tech. rep. 1. Comscore, Inc., 2019, pp. 1–53. URL: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2019/Global-State-of-Mobile.

[20] Yue (Nancy) Dai et al. "Risk assessment in e-commerce: How sellers' photos, reputation scores, and the stake of a transaction influence buyers' purchase behavior and information processing". In: *Computers in Human Behavior* 84.March 2006 (July 2018), pp. 342–351. ISSN: 07475632. DOI: 10.1016/j.chb.2018.02.038. URL: https://doi.org/10.1016/j.chb.2018.02.038%20https://linkinghub.elsevier.com/retrieve/pii/S0747563218300967.

[21] Jonas De Vos. "The effect of COVID-19 and subsequent social distancing on travel behavior". In: *Transportation Research Interdisciplinary Perspectives* 5 (May 2020), p. 100121. ISSN: 25901982. DOI: 10.1016/j.trip.2020.100121. URL: https://doi.org/10.1016/j.trip.2020.100121%20https://linkinghub.elsevier.com/retrieve/pii/S2590198220300324.

[22] Rupesh Dev, Antti Jääskeläinen, and Mika Katara. "Model-Based GUI Testing". In: *Advances in Computers*. 2012, pp. 65–122. DOI: 10.1016/B978-0-12-396526-4.00002-3. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780123965264000023.

[23] Dynamic Systems Development Method Consortium. *DSDM Agile Project Framework*. 2014. URL: https://www.agilebusiness.org/page/ProjectFramework%7B%5C_%7D10%7B%5C_%7DMoSCoWPrioritisation (visited on 08/05/2020).

[24] Hady ElHady. *Flutter vs. React Native vs. Xamarin*. 2019. URL: https://blog.logrocket.com/flutter-vs-react-native-vs-xamarin (visited on 07/20/2020).

[25] Khaled M.S. Faqih and Mohammed-Issa Riad Mousa Jaradat. "Assessing the moderating effect of gender differences and individualism-collectivism at individual-level on the adoption of mobile commerce technology: TAM3 perspective". In: *Journal of Retailing and Consumer Services* 22 (Jan. 2015), pp. 37–52. ISSN: 09696989. DOI: 10.1016/j.jretconser.2014.09.006. URL: http://dx.doi.org/10.1016/j.jretconser.2014.09.006%20https://linkinghub.elsevier.com/retrieve/pii/S0969698914001398.

[26] Shuresh Ghimire et al. "Sampling and degradation of biodegradable plastic and paper mulches in field after tillage incorporation". In: *Science of The Total Environment* 703 (Feb. 2020), p. 135577. ISSN: 00489697. DOI: 10.1016/j.scitotenv.2019.135577. URL: https://doi.org/10.1016/j.scitotenv.2019.135577%20https://linkinghub.elsevier.com/retrieve/pii/S004896971935572X.

[27] Google. *Android Devevelopment Program*. 2020. URL: https://developer.android.com/ (visited on 07/20/2020).

[28] Google. *Flutter architectural overview*. 2020. URL: https://flutter.dev/docs/resources/architectural-overview%7B%5C#%7Dcomposition (visited on 08/31/2020).

[29] Google. *Isolate class*. 2020. URL: https://api.flutter.dev/flutter/dart-isolate/Isolate-class.html (visited on 08/31/2020).

[30] Google. *Material Design System*. 2020. URL: https://material.io/design (visited on 07/04/2020).

[31] Google. *MaterialApp class*. 2020. URL: https://api.flutter.dev/flutter/material/MaterialApp-class.html (visited on 08/31/2020).

[32] Google. *Navigator class*. 2020. URL: https://api.flutter.dev/flutter/widgets/Navigator-class.html (visited on 08/31/2020).

[33] Sebastian Gurtner, Ronny Reinhardt, and Katja Soyez. "Designing mobile business applications for different age groups". In: *Technological Forecasting and Social Change* 88 (Oct. 2014), pp. 177–188. ISSN: 00401625. DOI: 10.1016/j.techfore.2014.06.020. URL: http://dx.doi.org/10.1016/j.techfore.2014.06.020%20https://linkinghub.elsevier.com/retrieve/pii/S0040162514002200.

[34] Sarah Hatton. "Choosing the Right Prioritisation Method". In: *19th Australian Conference on Software Engineering (aswec 2008)*. IEEE, Mar. 2008, pp. 517–526. ISBN: 978-0-7695-3100-7. DOI: 10.1109/ASWEC.2008.4483241. URL: http://ieeexplore.ieee.org/document/4483241/.

[35] Cindy Yoonjoung Heo. "New performance indicators for restaurant revenue management: ProPASH and ProPASM". In: *International Journal of Hospitality Management* 61 (Feb. 2017), pp. 1–3. ISSN: 02784319. DOI: 10.1016/j.ijhm.2016.10.005. URL: http://dx.doi.org/10.1016/j.ijhm.2016.10.005%20https://linkinghub.elsevier.com/retrieve/pii/S0278431916303140.

[36] Mark Herhold. *Palin*. 2018. URL: https://github.com/MarkHerhold/palin (visited on 08/31/2020).

[37] Ángel Herrero Crespo and Ignacio Rodríguez del Bosque. "The effect of innovativeness on the adoption of B2C e-commerce: A model based on the Theory of Planned Behaviour". In: *Computers in Human Behavior* 24.6 (Sept. 2008), pp. 2830–2847. ISSN: 07475632. DOI: 10.1016/j.chb.2008.04.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0747563208000903.

[38] Krasimir Hristozov. *MySQL vs PostgreSQL – Choose the Right Database for Your Project*. 2019. URL: https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres%7B%5C#%7D:%7B~%7D:text=Postgres%20is%20an%20object-relational,more%20closely%20to%20SQL%20standards (visited on 07/26/2020).

[39] JD Wetherspoon. *Wetherspoon app*. 2019. URL: https://www.jdwetherspoon.com/pubs/order-and-pay-app (visited on 07/04/2020).

[40] Jeff Hansen. *awilix-koa*. 2017. URL: https://github.com/jeffijoe/awilix-koa (visited on 08/08/2020).

[41] Fang Jiang et al. "Review of the Clinical Characteristics of Coronavirus Disease 2019 (COVID-19)". In: *Journal of General Internal Medicine* 35.5 (May 2020), pp. 1545–1549. ISSN: 0884-8734. DOI: 10.1007/s11606-020-05762-w. URL: http://link.springer.com/10.1007/s11606-020-05762-w.

[42] Jnan Dash. *RDBMS vs. NoSQL: How do you pick?* 2013. URL: https://www.zdnet.com/ (visited on 07/26/2020).

[43] Just Eat. *Just Eat - About us*. 2020. URL: https://www.just-eat.com/about-us (visited on 07/04/2020).

[44] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* 8.2 (June 2017), pp. 163–190. ISSN: 20904479. DOI: 10.1016/j.asej.2015.08.004. URL: http://dx.doi.org/10.1016/j.asej.2015.08.004%20https://linkinghub.elsevier.com/retrieve/pii/S2090447915001276.

[45] Koa Compress. *Koa Compress*. 2019. URL: https://github.com/koajs/compress (visited on 08/31/2020).

[46] Koa-bodyparser. *koa-bodyparser*. 2019. URL: https://github.com/koajs/bodyparser (visited on 08/31/2020).

[47] Koa-cors. *koa-cors*. 2019. URL: https://github.com/koajs/cors (visited on 08/31/2020).

[48] Koa-respond. *koa-respond*. 2017. URL: https://github.com/jeffijoe/koa-respond (visited on 08/31/2020).

[49] Koa.js. *Koa.js*. 2020. URL: https://koajs.com/%7B%5C#%7D (visited on 07/28/2020).

[50] Maya Kosoff. *How Uber's latest update could pose a major threat to GrubHub*. 2015. URL: https://www.businessinsider.com/ubers-app-update-heavily-emphasizes-food-delivery-poses-threat-to-grubhub-2015-8 (visited on 07/04/2020).

[51] Ting-Peng Liang et al. "Adoption of mobile technology in business: a fit-viability model". In: *Industrial Management & Data Systems* 107.8 (Oct. 2007), pp. 1154–1169. ISSN: 0263-5577. DOI: 10.1108/02635570710822796. URL: https://www.emerald.com/insight/content/doi/10.1108/02635570710822796/full/html.

[52] R Likert. "A technique for the measurement of attitudes". In: *Archives of Psychology* (1932).

[53] Shanhong Liu. *Market share of mobile operating systems in the United Kingdom (UK) from 2010 to 2019*. 2020. URL: https://www.statista.com/statistics/487373/market-share-mobile-operating-systems-uk/ (visited on 07/09/2020).

[54] Yipeng Liu, Jong Min Lee, and Celia Lee. "The challenges and opportunities of a global health crisis: the management and business implications of COVID-19 from an Asian perspective". In: *Asian Business & Management* 19.3 (July 2020), pp. 277–297. ISSN: 1472-4782. DOI: 10.1057/s41291-020-00119-x. URL: https://doi.org/10.1057/s41291-020-00119-x%20http://link.springer.com/10.1057/s41291-020-00119-x.

[55] Edward Livingston, Karen Bucher, and Andrew Rekito. "Coronavirus Disease 2019 and Influenza 2019-2020". In: *JAMA* 323.12 (Mar. 2020), p. 1122. ISSN: 2575-3126. DOI: 10.1001/jama.2020.2633. URL: https://www.who.int/emergencies/diseases/novel-coronavirus-2019%20https://jamanetwork.com/journals/jama/fullarticle/2762386.

[56] Kelty Logan. "Attitudes towards in-app advertising: a uses and gratifications perspective". In: *International Journal of Mobile Communications* 15.1 (2017), p. 26. ISSN: 1470-949X. DOI: 10.1504/IJMC.2017.080575. URL: http://www.inderscience.com/link.php?id=80575.

[57] Rc Martin. *Design principles and design patterns*. 2000. URL: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles%7B%5C_%7Dand%7B%5C_%7DPatterns.pdf (visited on 08/01/2020).

[58] Microsoft. *Microsoft SQL Server*. 2020. URL: https://www.microsoft.com/en-gb/sql-server/sql-server-downloads (visited on 07/26/2020).

[59] John Middleton et al. "ASPHER statement on the novel coronavirus disease (COVID-19) outbreak emergency". In: *International Journal of Public Health* 65.3 (Apr. 2020), pp. 237–238. ISSN: 1661-8556. DOI: 10.1007/s00038-020-01362-x. URL: https://doi.org/10.1007/s00038-020-01362-x%20http://link.springer.com/10.1007/s00038-020-01362-x.

[60] OpenJS Foundation. *About Node.js*. 2020. URL: https://nodejs.org/en/about/ (visited on 07/28/2020).

[61] Eleonora Pantano et al. "Competing during a pandemic? Retailers' ups and downs during the COVID-19 outbreak". In: *Journal of Business Research* 116.05 (Aug. 2020), pp. 209–213. ISSN: 01482963. DOI: 10.1016/j.jbusres.2020.05.036. URL: https://doi.org/10.1016/j.jbusres.2020.05.036%20https://linkinghub.elsevier.com/retrieve/pii/S0148296320303209.

[62] Jaydeep Patadiya. *Choosing The Best Cross Platform Mobile App Development Framework*. 2020. URL: https://radixweb.com/blog/cross-platform-app-development-frameworks (visited on 07/25/2020).

[63] Pedro Belfort. *Adobe XD prototype*. 2020. URL: https://xd.adobe.com/view/96ae6c3c-4155-48aa-6d06-a77ccec9d0d6-e09f/screen/96a875a4-fc41-4609-8d74-0ab448e84abf/ (visited on 09/01/2020).

[64] PgAdmin. *pgAdmin*. 2020. URL: https://www.pgadmin.org/ (visited on 08/31/2020).

[65] PostgreSQL Global Development Group. *pgcrypto*. 2020. URL: https://www.postgresql.org/docs/12/pgcrypto.html (visited on 08/31/2020).

[66] PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2020. URL: https://www.postgresql.org/ (visited on 07/26/2020).

[67] Muhammad Sarwar and Tariq Rahim Soomro. "Impact of Smartphone ' s on Soci ety". In: *European Journal of Scientific Research* 98.2 (2013), pp. 216–226.

[68] Sequelize.org. *Sequelize*. 2017. URL: https://sequelize.org/master/ (visited on 08/31/2020).

[69] Sequelize.org. *Sequelize-Auto*. 2013. URL: https://github.com/sequelize/sequelize-auto (visited on 08/31/2020).

[70] Shahbaz A. Shams, Abid Haleem, and Mohd Javaid. "Analyzing COVID-19 pandemic for unequal distribution of tests, identified cases, deaths, and fatality rates in the top 18 countries". In: *Diabetes & Metabolic Syndrome: Clinical Research & Reviews* 14.5 (Sept. 2020), pp. 953–961. ISSN: 18714021. DOI: 10.1016/j.dsx.2020.06.051. URL: https://doi.org/10.1016/j.dsx.2020.06.051%20https://linkinghub.elsevier.com/retrieve/pii/S1871402120302204.

[71] Tom Shawver. *Bristol*. 2018. URL: https://github.com/TomFrost/Bristol (visited on 08/31/2020).

[72] Amyra Sheldon. *11 Popular Cross-Platform Tools for App Development in 2020*. 2020. URL: https://hackernoon.com/9-popular-cross-platform-tools-for-app-development-in-2019-53765004761b (visited on 07/20/2020).

[73] Jagdish Sheth. "Business of business is more than business: Managing during the Covid crisis". In: *Industrial Marketing Management* 88.May (July 2020), pp. 261–264. ISSN: 00198501. DOI: 10.1016/j.indmarman.2020.05.028. URL: https://doi.org/10.1016/j.indmarman.2020.05.028%20https://linkinghub.elsevier.com/retrieve/pii/S0019850120303977.

[74] Jagdish Sheth. "Impact of Covid-19 on consumer behavior: Will the old habits return or die?" In: *Journal of Business Research* 117 (Sept. 2020), pp. 280–283. ISSN: 01482963. DOI: 10.1016/j.jbusres.2020.05.059. URL: https://doi.org/10.1016/j.jbusres.2020.05.059%20https://linkinghub.elsevier.com/retrieve/pii/S0148296320303647.

[75] Kunbo Shi et al. "Does e-shopping replace shopping trips? Empirical evidence from Chengdu, China". In: *Transportation Research Part A: Policy and Practice* 122.01 (Apr. 2019), pp. 21–33. ISSN: 09658564. DOI: 10.1016/j.tra.2019.01.027. URL: https://doi.org/10.1016/j.tra.2019.01.027%20https://linkinghub.elsevier.com/retrieve/pii/S0965856418305160.

[76] Dominik Spicher. *QR Code Scanner*. 2019. URL: https://pub.dev/packages/qr%7B%5C_%7Dcode%7B%5C_%7Dscanner (visited on 08/31/2020).

[77] Stack Overflow. *Stack Overflow: Developer Survey Results 2019*. Tech. rep. Stack Overflow, 2019. URL: https://insights.stackoverflow.com/survey/2019%7B%5C#%7Dmost-popular-technologies.

[78] StrongLoop. *Express*. 2020. URL: https://expressjs.com/ (visited on 07/28/2020).

[79] Intagleo Systems. *The Most Popular Backend Frameworks for Web Development in 2020*. 2020. URL: https://www.intagleo.com/blog/most-popular-backend-frameworks-for-web-development-in-2019/ (visited on 07/28/2020).

[80] Tom Tan and Serguei Netessine. "At Your Service on the Table: Impact of Tabletop Technology on Restaurant Performance". In: *SSRN Electronic Journal* July 2020 (2017). ISSN: 1556-5068. DOI: 10.2139/ssrn.3037012. URL: https://www.ssrn.com/abstract=3037012.

[81] Sushil Kumar Tripathi. *Top 7 Backend Web Development Frameworks 2019*. 2019. URL: https://www.kelltontech.com/kellton-tech-blog/top-7-backend-web-development-frameworks-2019 (visited on 07/28/2020).

[82] Helen Vakhnenko. *What You Need to Know About Native and Cross-Platform Apps*. 2019. URL: https://agilie.com/en/blog/what-you-need-to-know-about-native-and-cross-platform-apps (visited on 07/20/2020).

[83] Robert West et al. “Applying principles of behaviour change to reduce SARS-CoV-2 transmission”. In: *Nature Human Behaviour* 4.5 (May 2020), pp. 451–459. ISSN: 2397-3374. DOI: 10.1038/s41562-020-0887-9. URL: http://dx.doi.org/10.1038/s41562-020-0887-9%20http://www.nature.com/articles/s41562-020-0887-9.

[84] David E. Wheeler. *citext*. 2020. URL: https://www.postgresql.org/docs/12/citext.html (visited on 08/31/2020).

[85] A. Wilder-Smith and D. O. Freedman. “Isolation, quarantine, social distancing and community containment: pivotal role for old-style public health measures in the novel coronavirus (2019-nCoV) outbreak”. In: *Journal of Travel Medicine* 27.2 (Mar. 2020), pp. 1–4. ISSN: 1708-8305. DOI: 10.1093/jtm/taaa020. URL: https://academic.oup.com/jtm/article/doi/10.1093/jtm/taaa020/5735321.

[86] Wong, C. H., Lee, H. S., Lim, Y. H., Chua, B. H., Tan, G. W. H. “Predicting the consumers’ intention to adopt mobile-shopping: an emerging market perspective”. In: *International Journal of Network and Mobile Technologies* 3.3 (2012), pp. 24–39. ISSN: 2229-9114.

[87] World Health Organization (WHO). *WHO coronavirus disease (COVID-19) dashboard*. Aug. 2020. URL: https://covid19.who.int/.

[88] Kiseol Yang. “Consumer technology traits in determining mobile shopping adoption: An application of the extended theory of planned behavior”. In: *Journal of Retailing and Consumer Services* 19.5 (Sept. 2012), pp. 484–491. ISSN: 09696989. DOI: 10.1016/j.jretconser.2012.06.003. URL: http://dx.doi.org/10.1016/j.jretconser.2012.06.003%20https://linkinghub.elsevier.com/retrieve/pii/S0969698912000665.

[89] Xuemei Zhang et al. “Evolving strategies of e-commerce and express delivery enterprises with public supervision”. In: *Research in Transportation Economics* 80.December 2019 (May 2020), p. 100810. ISSN: 07398859. DOI: 10.1016/j.retrec.2019.100810. URL: https://doi.org/10.1016/j.retrec.2019.100810%20https://linkinghub.elsevier.com/retrieve/pii/S0739885919303373.

[90] Inc Zoom Video Communications. *Zoom Video Communications*. URL: https://zoom.us/ (visited on 08/31/2020).

Appendix A: Existing Apps Taxonomy

Feature	UberEATS	Just Eat	Wetherspoon
Navigation	Tab-based. Home; Search; Orders; Account.	Tab-based. Restaurants; For You; Orders; Settings.	Tab-based. Home; Order; Finder; More.
Restaurant visualisation	Card-based. Displays the restaurant's name, picture, price range, cuisine, estimated delivery time, and rating. Tapping on the card takes the user to a new screen with more details about the restaurant, including its menu.	Card-based. Displays the restaurant's name, picture, cuisine, estimated delivery time, rating, and distance to the current location. Tapping on the card takes the user to a new screen with more details about the restaurant, including its menu.	List-based. Each list item displays the pub's name, short address, and distance to the current location. Tapping on the item activates a modal view that asks the user if he/she wants to check-in the selected pub. If positive, the user is taken to a new screen with a list of product categories.
Product visualisation	List item displaying the product's name, description, picture, and price. Tapping on the item takes the user to a screen dedicated to product personalisation. When ready, the user can choose the quantity and add it to the cart.	List item displaying the product's name, description, picture, and price. Tapping on the item activates a modal view where the user can customise the product and add to order.	List item displaying the product's name, age restrictions, description, price, and calories. Tapping on an item activates a modal view asking the user to set the table. Next, he/she is redirected to a screen where the product can be personalised and added to the basket.
Menu visualisation	List of product items. It prevents multiple levels of nested navigation by keeping the items organised into sections which can be quickly accessed by tapping on carousel items displayed at the top of the screen.	List of category items containing their names and description. Each item takes the user to a new screen with a list of products.	Multiple screen levels displaying categories and sub-categories. The flow ends with a list of product items.
Restaurant check-in	Not required. Moreover, the user is not allowed to order from multiple restaurants at the same time.	Not required. Moreover, the user is not allowed to order from multiple restaurants at the same time.	Required. The user must set a restaurant to browse its menu options. Just one restaurant can be checked-in at a time.
Search	Customers can use a dedicated tab to search for specific restaurants, categories, or cuisines. They can also conduct searches once inside the restaurant detail screen.	Customers can use a search bar on the top of the Restaurants tab to look for restaurants near a postcode or their current location. Search for dishes can be conducted inside the restaurant detail screen.	Customers can search for pubs/hotels in the Order and Finder tabs (map view or search bar). Once checked-in a restaurant, they can look for specific menu items.
Filter	A filter button in the Home screen allows users to filter for specific dining options (e.g. delivery, pickup), price range, max delivery fee, dietary restrictions.	Absent.	Absent.
Favourites	Users can add restaurants to a Favourites list which is accessible through the Profile tab.	Absent.	Users can add products to a temporary Favourites list which is accessible through the top navigation bar while checked-in a restaurant.
Cart	Visible when users add products to the cart. After that, the cart button becomes visible at the bottom of every screen. The cart itself displays the delivery address, the restaurant's name, order details, and payment method. Users can tap on a "next" button to proceed with the order.	Visible in the Restaurants tab when users add products to the cart. The cart button assumes many forms (e.g. floating action button or a regular container displaying cart details). It disappears if users switch tabs or enter different restaurants. The cart itself gives the option for users to choose between delivery or in-person collection. It also displays order details and a button to proceed with payment.	Aside from the Finder tab, the cart is visible and accessible from all screens via the top navigation bar. Despite the cart icon, the app refers to the cart screen as "My Basket". There, users can visualise order details and proceed to payment.
Check-out	The check-out process initiates when the user taps on the "next" button inside a populated cart. Next, he/she is given the option to leave a tip to the delivery partner. To conclude the order, the user must tap on a "place order" button at the bottom of the screen. Then, payment is processed, and the order status can be monitored via the Orders tab.	The check-out process initiates when the user taps on the "Go to payment" button inside a populated cart. After that, he/she is taken to a screen asking for a phone number and delivery address. This screen also provides options to set delivery time and to leave notes for the restaurant. The user can choose to pay with apple pay, cash, or credit card. Payment is processed upon confirmation.	The check-out process initiates when the user taps on the "proceed to payment" button inside a populated cart. Next, a modal view overlays the cart screen, and the user is given options to pay with Apple Pay, PayPal, or credit card. Each new order also requires a new payment.

Table A.1: A taxonomy comparison of the three most popular Food & Drink apps from App Store and Google Play Store in the U.K. – UberEATS, Just Eat, and Wetherspoon. Descriptions recorded in July 2020.

Appendix B: High-fidelity prototype

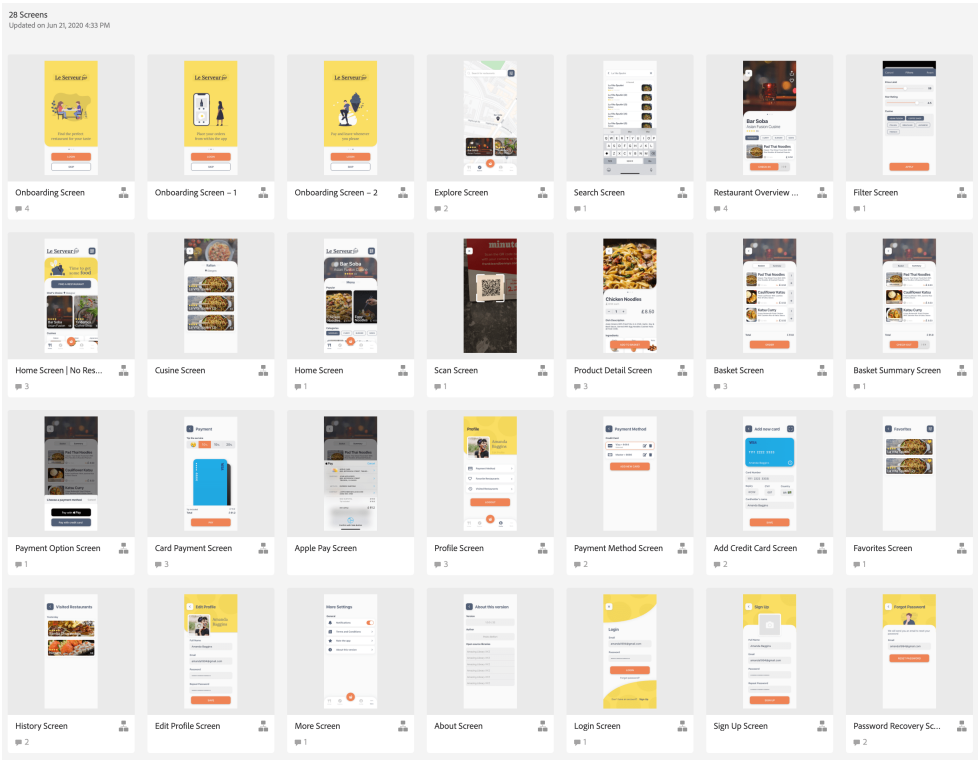


Figure B.1: Mosaic of all screens created as part of the high-fidelity prototype. The prototype can be accessed in full through the following URL: `https://shorturl.at/huBO0`. Password: Digital2020.

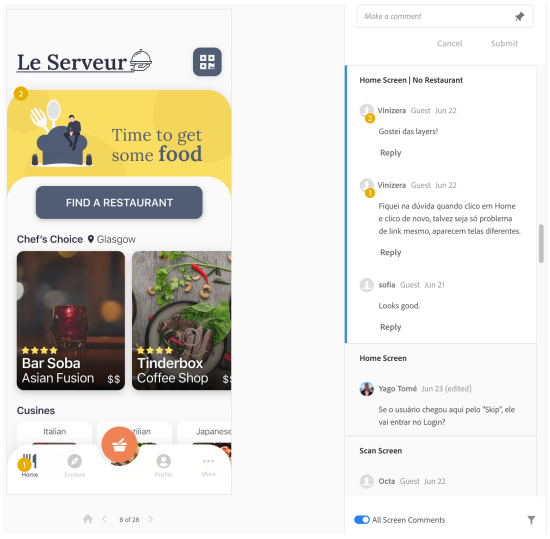


Figure B.2: Screenshot of Adobe XD comments tab, where users provided feedback on screens for the prototype.

Appendix C: Functional requirements

MoSCoW Priorisation			
[Mo] **must have**: critical to the project			
Title	**Description**	**Phase**	**Estimated Effort (h)**
QR-Code scanning	Scan QR-Code to check-in a local restaurant	Explore	12
Restaurant visualisation	Visual representation of a restaurant and its characteristics	Explore	10
Product visualisation	Visual representation of a dish and its characteristics	Explore; Choose	10
Restaurant check-in	Set a restaurant as the home content	Check-in	10
Menu visualisation	List of products offered by a given restaurant	Choose	6
Tab visualisation	Visual representation of the restaurant tab -- open/closed items	Order	12
Adding products	Addition of products to the tab	Order	4
Editing tab products	Edition of products contained in the tab	Order	4
Ordering	Request sent to the kitchen to prepare the products in the open tab	Order	8
Check-out	Process of closing the tab and making payments	Check-out	32
Credit-card payment	Registration and selection of credit-card as payment method	Check-out	16
[S] **should have**: significant value but not vital			
Title	**Description**	**Phase**	**Estimated Effort (h)**
Navigation by category	Find restaurants by the type of cuisine	Explore	10
User signup	Creation of a user account to store basic information	Account	10
User login/logout	Association of a user with a given active session	Account	10
Password recovery	Password reset in case a user forgets the password	Account	8
Map navigation	Discovery of restaurants through a map	Explore	16
Search	Discovery of restaurants through a search query	Explore	12
Filters	Narrowing-down restaurant results by setting filter options	Explore	10
About/more	Screen to access terms and conditions ass well as version information	Miscellaneous	6
[Co] **could have**: disarible, but small impact if left out			
Title	**Description**	**Phase**	**Estimated Effort (h)**
Onboarding	Sequence of screens showcasing the app's purpose	Explore	10
Order status	Indication if the current order status (prepared or delivered)	Order	4
Tipping	Selection of a certain amount to tip the restaurant	Check-out	4
Apple-pay	Payment via apple-pay	Check-out	10
Profile edition	Ability to edit profile information such as email and password	Account	6
Add favourite restaurants	Ability to mark restaurants as favourites	Explore	8
Last visited restaurants	Ability to visualize restaurants which were checked-in	Explore	8
Photo uploading	Ability to upload a photo to the user profile	Account	6
[W] **will not have**: not a priority for the specific time frame			
Title	**Description**	**Phase**	**Estimated Effort (h)**
Push notification	Notifications to alert the user about order status or promotions	Explore; Choose	12
Analytics	Monitoring of user behaviour to support future app enhancements	Miscellaneous	10
Card scanning	Ability to conveniently scan a credit card and auto-fill the form	Check-out	10
Booking	Ability to make restaurant reservations	Check-in	14

Table C.1: Functional requirements expressed via the MoSCoW priorisation technique. Each requirement has been associated with a pertinent Customer Journey phase.

Appendix D: Application design flows

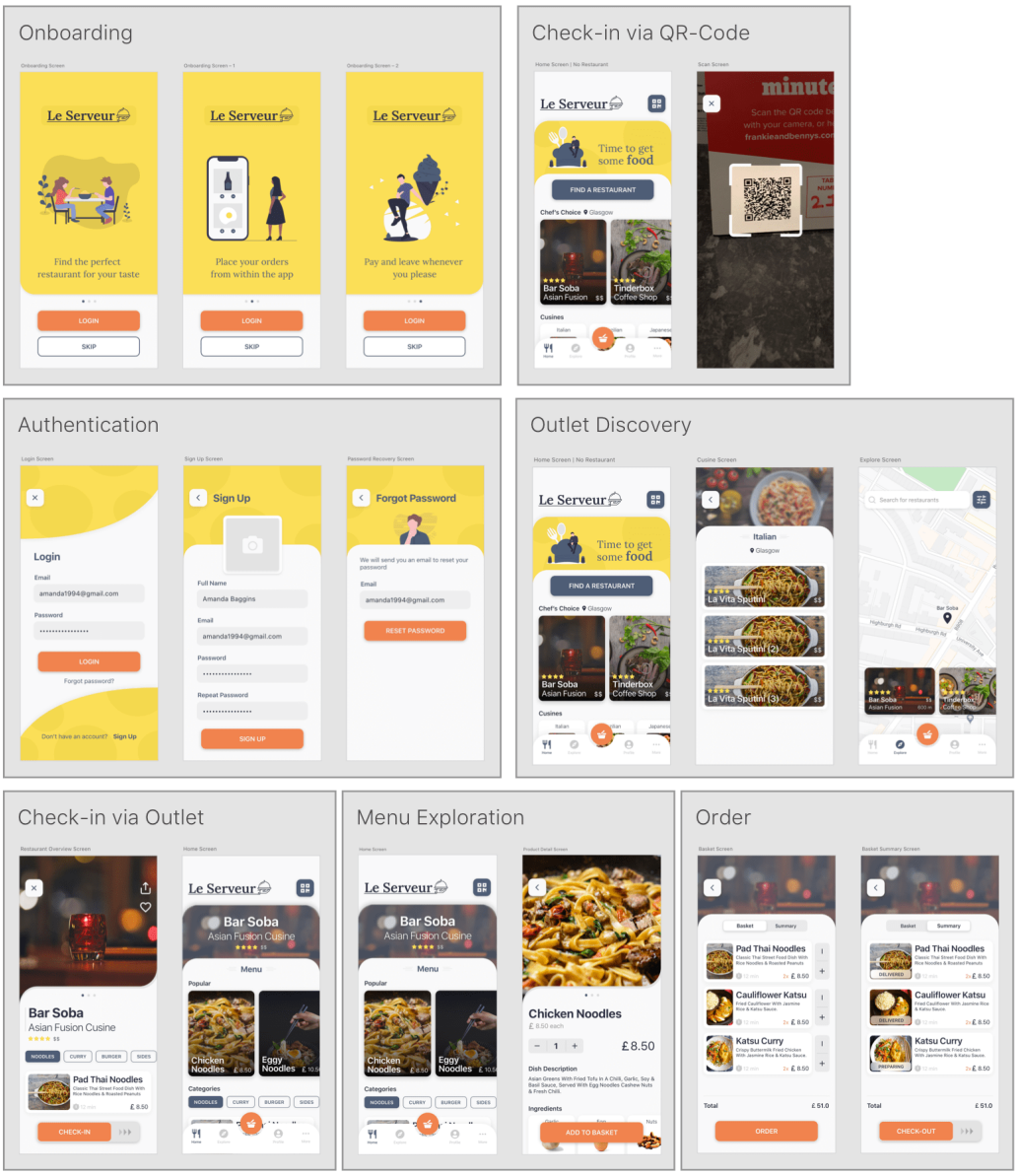


Figure D.1: Application flows designed for the App – part one. High-fidelity prototype URL: `https://shorturl.at/huBO0`. Password: Digital2020.

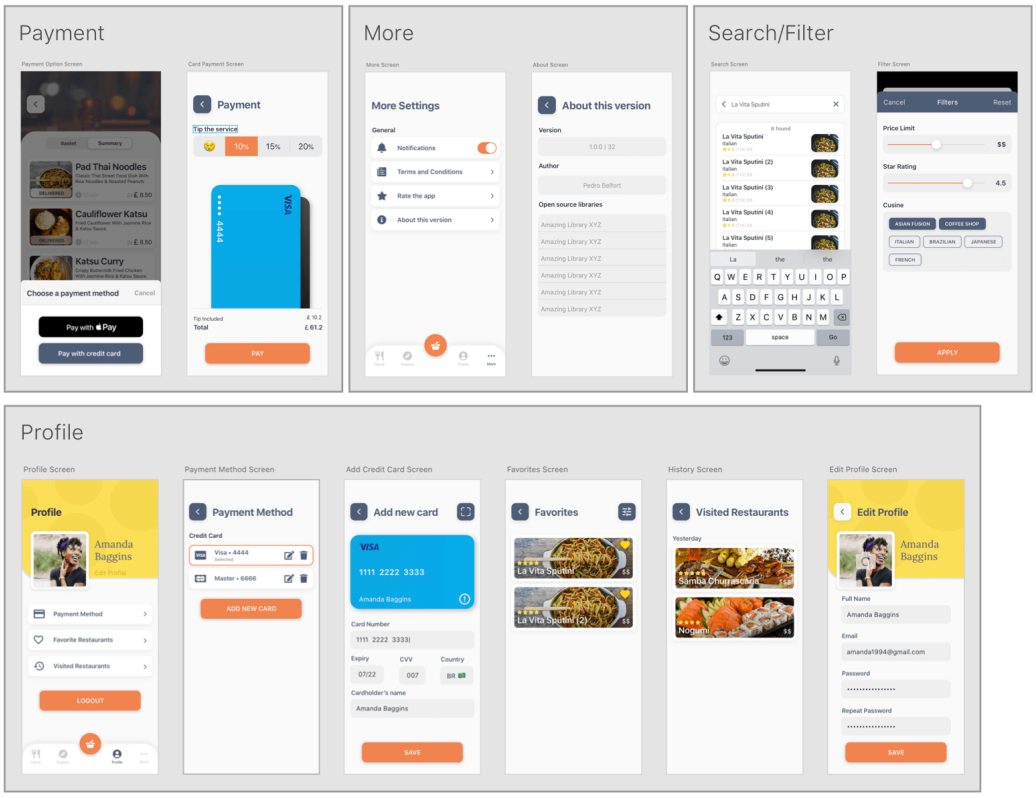


Figure D.2: Application flows designed for the App – part two. High-fidelity prototype URL: `https://shorturl.at/huBO0`. Password: Digital2020.

Appendix E: Technologies comparison

Frontend Framework Comparison				
Criteria	**Xamarin**	**React-Native**	**Flutter**	**Champion**
License	Open-source, but paid as it scales	Open-source	Open-source	**Flutter and React Native**
SCORE	7	10	10	
Programming language	C# or .NET	Javascript; Typescript	Dart	**Flutter**
SCORE	8	6	9	
Reactive programming	Only through 3rd party library such as ReactiveUI	Built-in	Bult-in	**Flutter and React Native**
SCORE	6	10	10	
Code reuse	High	High	Very high	**Flutter**
SCORE	8	8	10	
UI components	Poor. Rely on vendors such as Syncfusion, DevExpress, UX Divers, GrapeCity, Telerik, and others	Limited. Rich UI apps must recur to several community libraries	Built-in Widget system	**Flutter**
SCORE	5	7	10	
Development speed	Good integration with Visual Studio but problems with hot reloading	Requires a lot of setup and use of extensions. Hot reload does not work well on complex apps	Good intergation with Android Studio and Visual Studio Code. Hot reloading is consistent and reliable	**Flutter**
SCORE	8	7	9	
Performance	Close to native, but with limitations for complex UIs	Can be close to native, but it is easy to fall on API misuse	Closer to native. 60 FPS.	**Flutter**
SCORE	7	7	10	

Table E.1: Comparison of the most popular cross-platform mobile technologies[77]. Online community discussions were used to draw the results [24] [62].

.

Backend Framework Comparison						
Criteria	**Django**	**Ruby on Rails**	**Laravel**	**Spring Boot**	**Node.js + Express.js**	**Champion**
Programming language	Python	Ruby	PHP	Java	JavaScript	**Django**
SCORE	10	8	7	5	9	
Community support	51.3k stars and 2,017 contributors on Github	46.2k stars and 4,268 contributors on Github	60.8k stars and 579 contributors on Github	38.8k stars and 495 contributors on Github	72.3k stars and 2,826 contributors on Github	**Node.js**
SCORE	8	8	9	6	10	
Flexibility	Enforces the MVT architecthure. RESTful APIs can be built with the DRF framework	Enforces the MVC architecture	Imposes almost no restrictions, but encourages the MVC pattern	Auto-configures the application based on the MVC pattern	Does not enforce any patterns; no default configurations	**Node.js**
SCORE	7	7	9	8	10	
Development speed	Very high	High	High	Regular	Very high	**Django and Node.js**
SCORE	10	9	9	7	10	
Scalability & Performance	Easy horizontal scalling with access to many tools to help on the process	Many reports of issues when scaling horizontally. Optimization relies havily on the application configuration	Easy horizontal scalling with access to many tools to help on the process	Highly scalable with many tools available to help on the process	Non-blocking, event-driven I/O with single thread paradigm. Easy to scale horizontally.	**Spring Boot**
SCORE	8	6	9	10	9	

Table E.2: High-level comparison of the most popular backend web development frameworks [77]. Online community discussions were used to draw the results [79] [81] .

Appendix F: Data model diagrams

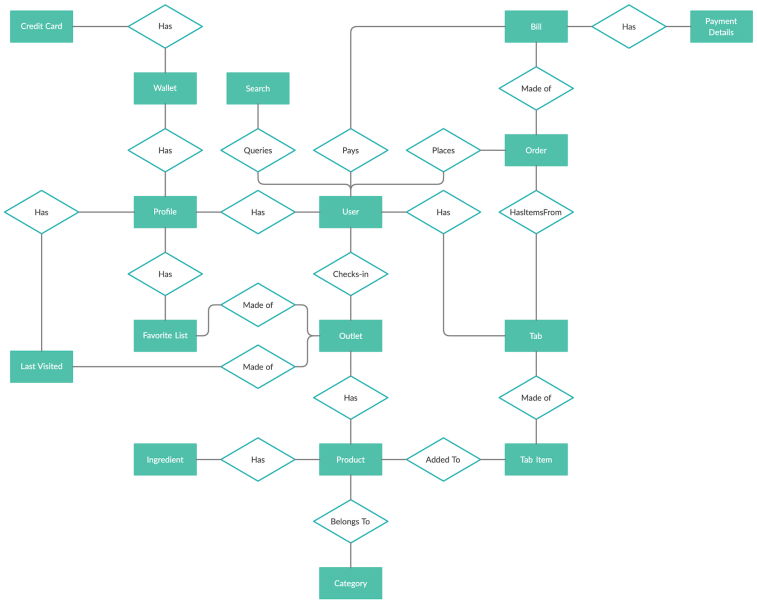


Figure F.1: First entity-relationship diagram — condensed Chen's notation. This diagram represents at a high-level the elements and interactions pertained to the project's domain.

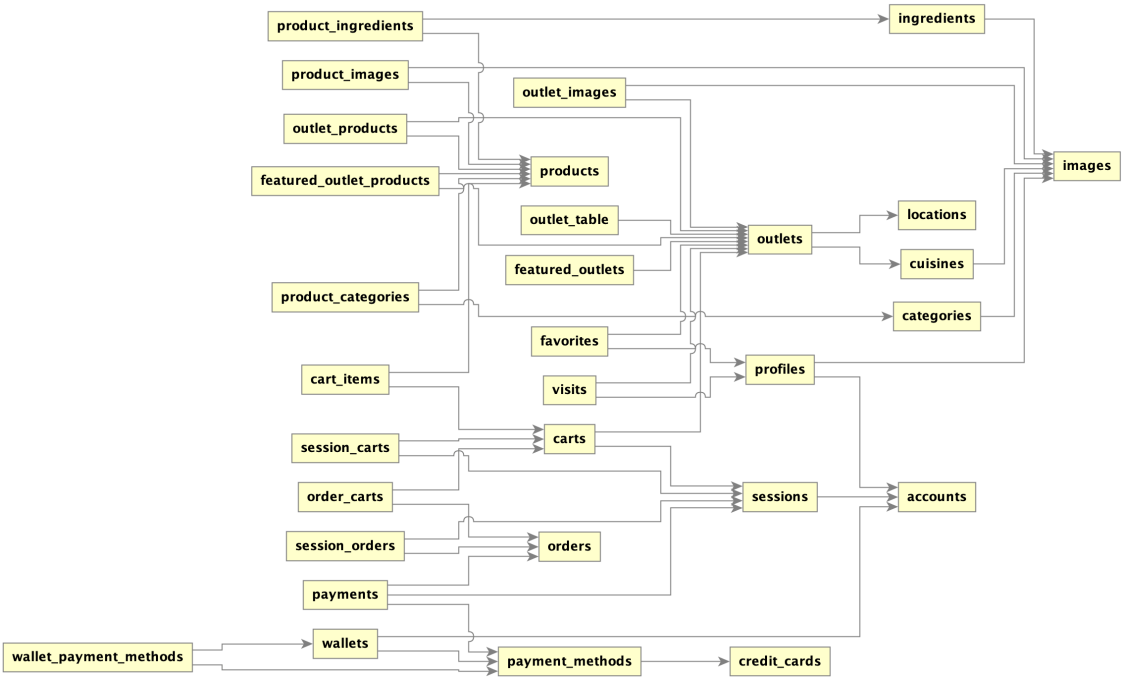


Figure F.2: Collapsed entity-relationship diagram — Simplified Barker's notation.

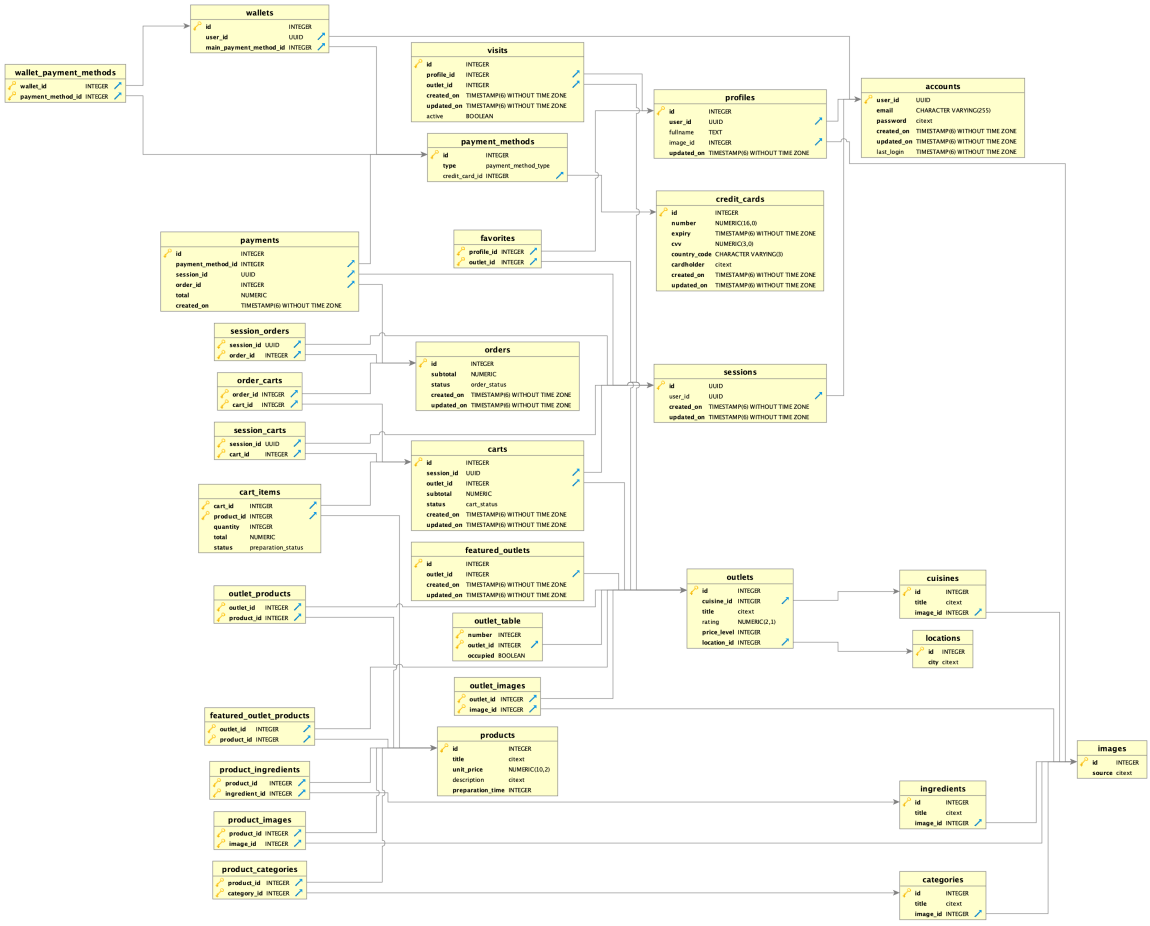
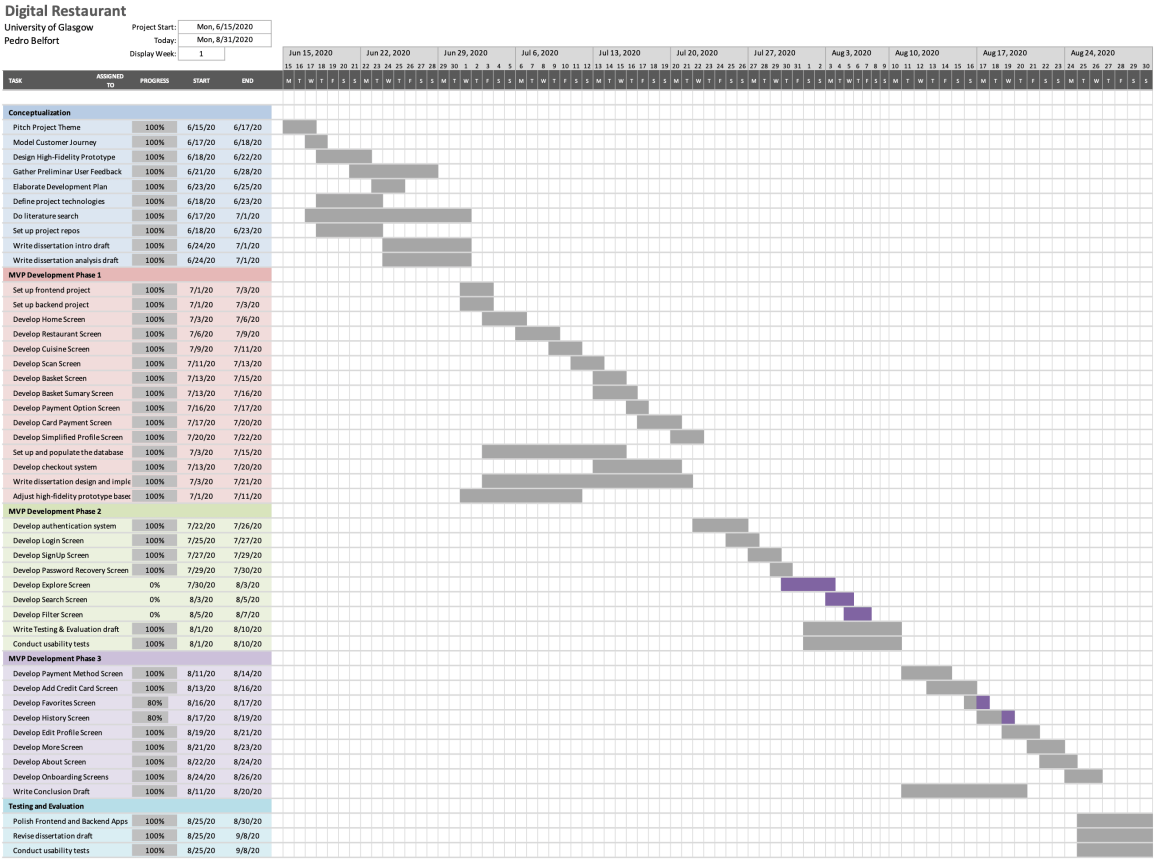


Figure F.3: Expanded entity-relationship diagram — Simplified Barker's notation.

Appendix G: Project management tools



Digital Restaurant

University of Glasgow
Pedro Belfort

Project Start: Mon, 6/15/2020
Today: Mon, 8/31/2020
Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
Conceptualization				
Pitch Project Theme		100%	6/15/20	6/17/20
Model Customer Journey		100%	6/17/20	6/18/20
Design High-Fidelity Prototype		100%	6/18/20	6/22/20
Gather Preliminar User Feedback		100%	6/21/20	6/28/20
Elaborate Development Plan		100%	6/23/20	6/25/20
Define project technologies		100%	6/18/20	6/23/20
Do literature search		100%	6/17/20	7/1/20
Set up project repos		100%	6/18/20	6/23/20
Write dissertation intro draft		100%	6/24/20	7/1/20
Write dissertation analysis draft		100%	6/24/20	7/1/20
MVP Development Phase 1				
Set up frontend project		100%	7/1/20	7/3/20
Set up backend project		100%	7/1/20	7/3/20
Develop Home Screen		100%	7/3/20	7/6/20
Develop Restaurant Screen		100%	7/6/20	7/9/20
Develop Cuisine Screen		100%	7/9/20	7/11/20
Develop Scan Screen		100%	7/11/20	7/13/20
Develop Basket Screen		100%	7/13/20	7/15/20
Develop Basket Sumary Screen		100%	7/13/20	7/16/20
Develop Payment Option Screen		100%	7/16/20	7/17/20
Develop Card Payment Screen		100%	7/17/20	7/20/20
Develop Simplified Profile Screen		100%	7/20/20	7/22/20
Set up and populate the database		100%	7/3/20	7/15/20
Develop checkout system		100%	7/13/20	7/20/20
Write dissertation design and imple		100%	7/3/20	7/21/20
Adjust high-fidelity prototype basec		100%	7/1/20	7/11/20
MVP Development Phase 2				
Develop authentication system		100%	7/22/20	7/26/20
Develop Login Screen		100%	7/25/20	7/27/20
Develop SignUp Screen		100%	7/27/20	7/29/20
Develop Password Recovery Screen		100%	7/29/20	7/30/20
Develop Explore Screen		0%	7/30/20	8/3/20
Develop Search Screen		0%	8/3/20	8/5/20
Develop Filter Screen		0%	8/5/20	8/7/20
Write Testing & Evaluation draft		100%	8/1/20	8/10/20
Conduct usability tests		100%	8/1/20	8/10/20
MVP Development Phase 3				
Develop Payment Method Screen		100%	8/11/20	8/14/20
Develop Add Credit Card Screen		100%	8/13/20	8/16/20
Develop Favorites Screen		80%	8/16/20	8/17/20
Develop History Screen		80%	8/17/20	8/19/20
Develop Edit Profile Screen		100%	8/19/20	8/21/20
Develop More Screen		100%	8/21/20	8/23/20
Develop About Screen		100%	8/22/20	8/24/20
Develop Onboarding Screens		100%	8/24/20	8/26/20
Write Conclusion Draft		100%	8/11/20	8/20/20
Testing and Evaluation				
Polish Frontend and Backend Apps		100%	8/25/20	8/30/20
Revise dissertation draft		100%	8/25/20	9/8/20
Conduct usability tests		100%	8/25/20	9/8/20

Figure G.1: Gantt chart showcasing the project workflow and milestones.

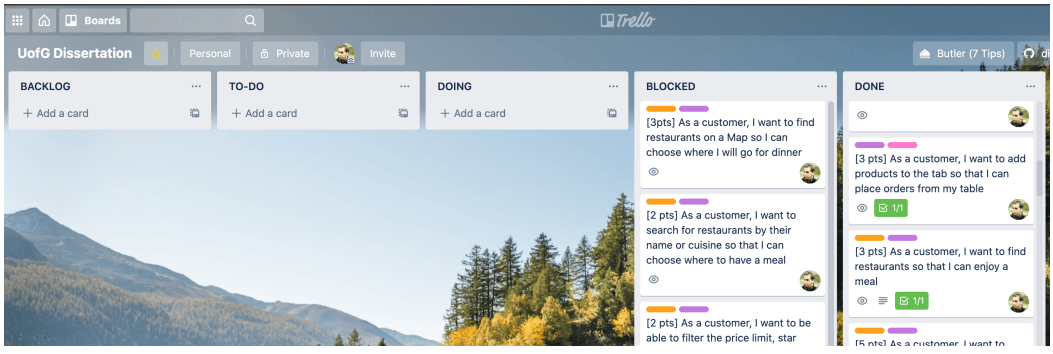


Figure G.2: Kanban board (Trello). User stories were created and broken down into smaller tasks.

Appendix H: Requirements Status

Requirements Status					
	Title	**Phase**	**Frontend Status**	**Backend Status**	**Outcome**
Must-have	QR-Code scanning	Explore	Completed	Does not apply	Complete
	Restaurant visualisation	Explore	Completed	Completed	Complete
	Product visualisation	Explore; Choose	Completed	Completed	Complete
	Restaurant check-in	Check-in	Completed	Completed	Complete
	Menu visualisation	Choose	Completed	Completed	Complete
	Tab visualisation	Order	Completed	Completed	Complete
	Adding products	Order	Completed	Completed	Complete
	Editing tab products	Order	Completed	Completed	Complete
	Ordering	Order	Completed	Completed	Complete
	Check-out	Check-out	Completed	Completed	Complete
	Credit-card payment	Check-out	Completed	Completed	Complete
Should-have	Navigation by category	Explore	Completed	Completed	Complete
	User signup	Account	Incomplete - Must connect UI the with backend	Completed	Complete
	User login/logout	Account	Incomplete - Must connect UI the with backend	Completed	Complete
	Password recovery	Account	Undone	Undone	Undone
	Map navigation	Explore	Undone	Undone	Undone
	Search	Explore	Undone	Undone	Undone
	Filters	Explore	Undone	Undone	Undone
	About/more	Miscellaneous	Completed	Does not apply	Complete
Could-have	Onboarding	Explore	Completed	Does not apply	Complete
	Order status	Order	Does not apply	Completed	Complete
	Tipping	Check-out	Completed	Incomplete - Needs to be taken into account	Incomplete
	Apple-pay	Check-out	Undone	Undone	Undone
	Profile edition	Account	Incomplete - Needs backend	Incomplete - Needs API endpoint	Undone
	Add favourite restaurants	Explore	Undone	Undone	Undone
	Last visited restaurants	Explore	Undone	Undone	Undone
	Photo uploading	Account	Undone	Undone	Undone

Table H.1: An overview of each requirement's status at the research project's submission date.

Appendix I: Black-box test table (part two)

29	User taps on the orange floating action button docked at the middle of the bottom app bar	The cart screen is displayed	The cart screen is displayed	**Pass**
30	[Cart] User enters the cart screen	The cart screen is displayed with a modal presentation, covering the bottom app bar; cart products are displayed in a list	The cart screen is displayed with a modal presentation, covering the bottom app bar; cart products are displayed in a list	**Pass**
31	[Cart] User presses the plus sign in a Cart Item card's stepper	The product quantity is increased and the total price is recalculated	The product quantity is increased and the total price is recalculated	**Pass**
32	[Cart] User presses the minus sign in a Cart Item card's stepper	The product quantity is decreased and, if the quantity was one, the cart item is removed	The product quantity is decreased and, if the quantity was one, the cart item is removed	**Pass**
33	[Cart] User taps on the "Order" button	An order is placed and the "Order Summary" segment control is activated	An order is placed and the "Order Summary" segment control is activated	**Pass**
34	[Cart] User swipes the "Check-out" slider	The payment method bottom sheet is displayed	The payment method bottom sheet is displayed	**Pass**
35	[Cart] User taps on the "Active Tab" segment control immidiately after placing an order	Empty placeholder message is displayed	Empty placeholder message is displayed	**Pass**
36	[Cart] User taps on the "Pay with credit card" button at the payment method bottom sheet	The payment screen is displayed	The payment screen is displayed	**Pass**
37	[Payment] User selects a tip amount	Selected tip amount is highlighted in orange	Selected tip amount is highlighted in orange	**Pass**
38	[Payment] User scrolls the card carousel	A new card is placed at the center	A new card is placed at the center	**Pass**
39	[Payment] User taps on the "pay" button	User is redirected to the checked-out Home screen	User is redirected to the checked-out Home screen	**Pass**
40	User taps on the "profile" bottom bar tab	Login screen is displayed	Login screen is displayed	**Pass**
41	[Login] User taps on the Sign Up button	Sign Up screen is displayed	Sign Up screen is displayed	**Pass**
42	User taps on the "more" bottom bar tab	More screen is displayed	More screen is displayed	**Pass**
43	[More] User taps on the "about this version" list tile	App version, author, and opern source libraries are listed	App version, author, and opern source libraries are listed	**Pass**
44	User taps on the "explore" bottom tab bar	Explore screen content is displayed	"Work in progress" label is displayed	**Fail**
45	[Login] User fills in the form with correct credentials and taps on the Login button	User is authenticated and profile screen is displayed	Login button is disabled; no side-effects take place	**Fail**
46	[SingnUp] User fills in the form and taps on the "sign up" button	New user account is created and user is logged in; Profile screen is displayed	Mock Profile screen is displayed	**Fail**

Table I.1: Black-box test table – part two. Each row defines a test case that covers a particular aspect of the app's behaviour.