B.Sc. (Hons) in Software Development

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

PUB WISER

By
Dara Lenaghan

April 26, 2024

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

This document presents the culmination of a comprehensive project aimed at developing "Pub-Wiser," a mobile application that revolutionizes the way users discover and interact with pubs. The project embodies a seamless blend of technology and user experience, harnessing the capabilities of Flutter and Firebase to offer real-time data, dynamic pricing updates, and interactive maps. While achieving its core objectives of providing reliable and up-to-date information, the app also demonstrates robustness in its design, highlighting the effectiveness of its MVVM architecture. The following pages will delve into the methodologies, design decisions, and technological integrations that have shaped "Pub-Wiser," reflecting on both its successes and the challenges faced during its development.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Source Code

The project source code is available within the following link: `https://github.com/DaraLenaghan-1/Pub-Wiser`

1.2 Project Screencast

Here is the link to the project screencast:
Project Screencast

1.3 Project Context

This study focuses on building a mobile app using Flutter and Dart, two tools provided by Google. Flutter allows developers to create apps for various platforms from one set of code, making the development process more efficient without lowering the quality of the app. Dart supports this by enabling the creation of strong, versatile mobile apps through its use of object-oriented programming. The app aims to solve a problem commonly faced by people who are new to a city or country, like students and tourists, who often have trouble finding places that meet their needs for price and quality. The development of this app is very relevant and could greatly help newcomers find what they're looking for more easily.

1.3.1 Classification of Mobile Applications

Mobile apps can be grouped into three types: native, web-based, and hybrid. Native apps are designed for one specific operating system like iOS or Android,

and they make full use of the device's features, leading to a smooth and high-quality user experience. However, creating separate versions for each system can be time-consuming and costly.

Web-based apps, on the other hand, use common web technologies such as HTML, CSS, and JavaScript. They require an internet connection to function and store most data online. While they can be accessed on any device with a web browser, they generally don't perform as well as native apps because they are slower and have limited access to device features.

Hybrid apps blend the features of native and web-based apps. They are built from a single codebase, making them more cost-effective and quicker to develop, and they perform better than web-based apps. Flutter, for example, helps create high-performing hybrid apps by compiling into native code, allowing them to work smoothly on different platforms.

1.4 Scope of the Project

The scope of the project encompasses the creation of a user-friendly mobile application tailored to the specific needs of newcomers to Galway's pub scene. The app will leverage technologies such as Flutter, Dart, Firebase, and Google Maps API to provide real-time information on drink prices and venue details. Features will include intuitive navigation, price comparison tools, and personalized recommendations based on user preferences. By focusing on a targeted user demographic and specific geographic location, the project aims to deliver a tailored solution that addresses a pressing need within the local community.

1.5 Objectives

The main goal of this dissertation is to analyse, design, and implement a mobile app called "Pub-Wiser," which is a pub discovery and pint comparison companion app, aimed at improving the experiences of students and tourists in new cities. The app aims to offer a user-friendly platform that helps users find pubs that match their preferences for ambiance, location, and, importantly, drink pricing.

Secondly explore and become proficient in modern software development technologies, specifically Flutter and Dart. These technologies are chosen for their robustness and their ability to enable development of cross-platform mobile applications from a single set of code. Throughout this project, an emphasis is placed on gaining a deep understanding of these technologies and applying them to solve real-world problems effectively.

1.5.1 Develop a User-Friendly Application

The goal here is to design a mobile app interface that is both easy to use and visually attractive. The app will help users effortlessly find pubs and bars in Galway. The design of the interface will focus on being clear, simple, and accessible to a wide range of users, ensuring that everyone can navigate and use the app easily.

1.5.2 Integrate Real-Time Data

The integration of real-time data from Firebase is crucial for providing users with accurate and up-to-date information. This objective entails establishing a seamless connection between the app and Firebase to retrieve drink prices, venue locations, opening hours, and other relevant details. The data should be consistently updated to reflect changes in real-time, ensuring users have access to the most relevant information.

1.5.3 Enhance User Experience

This objective aims to enhance the overall user experience by implementing features that add value and utility to the app. Interactive maps for easy navigation, and user reviews for social validation are examples of features designed to enhance user engagement and satisfaction. The app should strive to provide a compelling and enjoyable experience for users, encouraging them to return and explore further.

1.5.4 Integration with Google Maps and Firebase Firestore

An essential feature of the "Pub-Wiser" app is its integration with Google Maps and Firebase Firestore. The Google Maps API is used to provide interactive map features, allowing users to visually explore and compare pubs in their area or within specific search regions. This integration aims to enhance the user experience by enabling informed decisions based on location.

1.5.5 Ensure Reliability and Performance

The aim is to make sure the app works reliably and efficiently. To achieve this, extensive testing will be carried out to find and fix any bugs, errors, or performance issues. This includes unit testing (checking individual parts of the app), integration testing (ensuring all parts work together properly), and user acceptance testing (making sure the app meets user needs and works well on various devices and operating systems). These tests are crucial to ensure the app functions well and provides a good user experience.

1.5.6 Learning Outcomes

This project focuses on improving skills in mobile app development and broadening understanding of how to use Flutter and Dart along with Google's Firebase and Maps. The objective is to develop a practical application tailored to users. It serves as a comprehensive case study demonstrating how mobile technology, real-time data, and geolocation services can greatly enhance the user experience in the hospitality sector. This approach not only sharpens technical abilities but also provides key insights into how complex technologies can be integrated to effectively meet the needs of consumers.

1.6 Dissertation Structure

This dissertation is organised into six chapters, each focusing on distinct aspects of the project:

- **Chapter One: Introduction** - This chapter sets the stage by providing a clear context for the project. It outlines the objectives, justifies the relevance, and ensures the reader understands the project's scope and significance. Key elements include:
  - Explanation of the project's relevance and the need it addresses.
  - Clear articulation of the objectives and the metrics for evaluating success or failure.
  - An overview of each chapter and a link to the project's GitHub repository.
- **Chapter Two: Methodology** - This chapter details the approach taken to develop the project, including:
  - The blend of software development and research methodologies.
  - An agile, iterative approach encompassing planning, requirement analysis, and regular meetings.
  - The validation and testing processes used, highlighting tools like Junit.
  - The use of GitHub and other development tools throughout the project.
- **Chapter Three: Technology Review** - This chapter serves as the literature review, tightly integrated with the project's context and objectives. It includes:

  - Descriptions of each technology used at a conceptual level, like MongoDB, JSON, etc.
  - Discussion of standards and authoritative sources supporting the methodology.

- **Chapter Four: System Design** - Detailed explanation of the system architecture, informed by the technology review. Elements include:
  - Description of how components are integrated and interact.
  - Use of diagrams and screenshots to provide a clear visual representation of the architecture.

- **Chapter Five: System Evaluation** - Evaluation of the project against its initial objectives. This includes:
  - Testing for robustness and behaviour.
  - Analysis of system stability and performance benchmarks.
  - Discussion on limitations and opportunities revealed during the project.

- **Chapter Six: Conclusion** (- Summarises the dissertation and reflects on the project's objectives and findings. Highlights include:
  - Summary of the project's context and objectives.
  - Discussion of key findings from the system evaluation.
  - Identification of serendipitous insights and future research opportunities.

1.7 Summary

By achieving these objectives, the project aims to deliver a high-quality mobile application that meets the needs and expectations of users while contributing positively to the local community and economy.

Chapter 2

Methodology

The methodology section explains the approach used to meet the project's goals. It describes the strategies and processes followed during the development of the application. This part also sheds light on the reasons behind certain decisions and the techniques used to ensure the success of the project.

2.1 Project Management Approach

Given the dynamic nature of the "Pub-Wiser" project and its solitary development process, an Agile approach to project management was deemed most appropriate. Agile methodologies are well-suited for projects with evolving requirements and a need for rapid adaptation. Although SCRUM is typically employed in team environments, a modified version was adapted to suit the individual execution of this project.

2.1.1 Implementation of Agile Methodologies

The project was organised into bi-weekly sprints, with each sprint focused on delivering a predetermined set of features or improvements. This bi-weekly cycle facilitated systematic progress while retaining the flexibility to adjust the workflow as needed. Formal SCRUM meetings were avoided in favor of self-reviews at the end of each sprint, with any incomplete tasks being carried forward to the next sprint, with any incomplete tasks being carried forward to the next sprint. This iterative process promoted continuous development and integration of feedback, minimising the need for formal meetings and thus reducing overhead.

2.1.2 Supervisory Meetings

Throughout the "Pub-Wiser" project, regular weekly meetings were held with the project supervisor to ensure continuous oversight and guidance. These meetings played a crucial role in the project's development for several reasons:

Feedback and Insights: The project supervisor provided valuable feedback on the project's progress, offering insights and suggestions that significantly shaped the development direction. Review of Architectural Decisions: Each meeting included a review of the application's architecture, ensuring that any technical decisions aligned with best practices and project objectives. Adaptation and Improvements: The feedback obtained during these meetings allowed for the agile adaptation of the project plans and objectives, accommodating new insights or shifts in project direction. Motivation and Support: Regular interactions with the supervisor provided motivational support and helped maintain momentum throughout the development process, ensuring consistent progress and attention to detail. These meetings were integral to the project management approach, reinforcing the Agile methodology's emphasis on communication and iterative review. They helped ensure that the project remained on track and aligned with the initial goals, while also adapting to any necessary changes prompted by ongoing development insights.

2.2 Requirements

An agile approach to writing requirements is to be taken. Wireframes are to be created to help visualise the requirements. The requirements are to be kept at a high level, and need only to be described in detail at the beginning of a sprint. Requirements for the project were to be written using informal user stories. The User stories do not need to follow the format "as a user .. ". User stories are to be added to the different sprints according to where they will be developed.

2.3 Design

An agile approach is to be taken to designing the application. The emphasis will be on writing working software rather than producing documentation or using a methodology such as UML which can be very time consuming and not necessary for the size and scope of this project. As per the agile approach, the design of the application will be carried out in the sprint where the requirement is developed.

2.4 Development

Android Studio and/or VS Code is to be used for development. The Android Studio emulator is to be used to test the application. Android Studio and Visual Studio Code enable rapid development and are well suited to developing in Flutter and Dart. It is possible to deploy apps onto an android device, flutter will build and deploy the app making it fast and easy to visualise the changes made to the mobile app.

2.5 Testing

A manual approach to testing is to be used. A spreadsheet will be used for the test scripts and test results. Automated testing is not a goal of this project but will be investigated. The approach to testing in Flutter is documented here [?].

2.6 Version Control and Build

GitHub [?] is to be used for code version management. GitHub is a free online platform designed to host software development projects and facilitate version control through Git. It offers developers the ability to collaborate on code, monitor changes, and manage code repositories. With GitHub, users can create, share, and contribute to both open-source projects and private projects. The platform comes equipped with a range of features such as bug tracking, project management, code reviews, and documentation, enabling users to manage their development workflows more efficiently. In this project, GitHub will be used for software code management. When committing changes, descriptive and concise commit messages will be used that explain what has been changed and why.

2.7 Deployment

The following options will be considered for deployment of a flutter app:

1. Deploy using a web server: You can host your app on a web server and share the URL with others. To deploy a Flutter app to the web, you can use the flutter build web command to generate a set of static files that can be hosted on a web server. You can then upload these files to your web server using tools like FTP, SFTP, or SCP.

2. Deploy using Firebase Hosting: Firebase Hosting provides a simple way to host and deploy web apps, including Flutter apps. To deploy a Flutter app

to Firebase Hosting, you can use the flutter build web command to generate a set of static files, then follow the Firebase Hosting setup process to deploy the app.

3. Deploy using a mobile device: You can deploy your Flutter app directly to a mobile device using tools like Android Studio or Xcode. To do this, you'll need to connect your device to your computer and enable USB debugging. Then, you can use the flutter run command to deploy the app to your device.

4. Deploy using app distribution platforms: There are various app distribution platforms, such as TestFlight for iOS and Google Play Console for Android, that allow you to distribute pre-release versions of your app to specific users or groups without publishing it to the app store.

2.8 Summary

This chapter described the methodologies to be used for research, life cycle, requirements, design, testing, development, code management, and deployment.

Chapter 3

Technology Review

This section of the dissertation focuses on the Flutter development framework and its associated technologies, exploring their roles, benefits, and how they support the overarching goals of the "Pub-Wiser" app. By examining Flutter in the context of cross-platform development and its integration with backend services like Firebase and APIs such as Google Places, this review aims to substantiate the technological choices made during the app's development. The review will also touch upon the importance of these technologies in meeting user expectations for real-time interactions and seamless user experiences across various devices.

3.1 Flutter Development Framework

Flutter is an open-source User Interface (UI) software development kit developed by Google. It is specifically engineered for creating natively compiled applications across multiple platforms—mobile, web, and desktop—from a single codebase. Dart, the programming language used by Flutter, is designed to facilitate rapid application development and is optimized for performance across platforms. As an object-oriented language, Dart offers a syntax that is familiar to developers with backgrounds in Java or C#. A key strength of Flutter is its ability to deliver high-performance applications that also maintain aesthetic fidelity across different operating systems. This capability significantly streamlines the development process, reducing the time to market for applications that need to operate seamlessly across various platforms [**?**, **?**].

Architectural Overview: Flutter's architecture is distinguished by several key components that contribute to its robust performance and flexibility:

- **Dart Platform:** Dart's just-in-time (JIT) compilation facilitates a rapid and iterative development cycle with the capability for hot reloads, allowing

developers to instantly see the results of their changes without a full restart [?, ?].

- **Flutter Engine:** Written primarily in C++, the engine is at the core of Flutter's performance. It supports the rendering process, manages the Dart runtime, and provides a bridge to the host operating system via platform-specific SDKs [?].

- **Widgets:** At the heart of Flutter's design are widgets, which are the basic building blocks of a Flutter app's interface. Each widget is an immutable declaration of part of the user interface, with a variety of widgets available that mimic native components [?].

- **Design-Specific Widgets:** Flutter provides widgets that adhere to the material design (introduced by Google) and Cupertino (Apple's iOS design norms) aesthetics, ensuring that applications not only feel native to each platform but also adhere to the design standards and guidelines of each ecosystem [?].

- **Embedder:** The Embedder provides a platform-specific shell for hosting the Flutter application. It connects the app with the operating system's native APIs, handling crucial functionalities like input, accessibility, and graphics through platform channels [?].

- **Runner:** The Runner serves as the execution environment, initializing the Flutter engine and managing the application lifecycle. It processes key lifecycle events from the operating system, ensuring smooth integration and operation of the app within native settings [?].

3.2 Flutter Widgets

Widgets in Flutter are fundamental to the construction of the application's user interface, acting as immutable declarations of UI components. Each widget, once defined, cannot be altered; any change in the interface necessitates the creation of a new widget configuration rather than an amendment to an existing one [?].

In the Flutter development environment, widgets function as versatile and reusable building blocks. This enables developers to assemble complex UIs from a series of simpler, standardized elements. Such a compositional approach not only simplifies the interface design process but also enhances adaptability across different screen sizes and device orientations, ensuring that applications remain responsive and maintain the fidelity of the native platform's user experience [?].

The immutability of widgets serves a pivotal role in optimizing the performance of Flutter applications. It underpins a reactive rendering cycle where only the widgets that have undergone changes are rebuilt. This selective re-rendering significantly boosts the efficiency of the application, minimizing resource consumption and enhancing responsiveness. This design principle is particularly beneficial for applications like "Pub-Wiser," which require a dynamic and uninterrupted flow of information and interaction. By preventing unnecessary updates and delays, it ensures that users enjoy a smooth and engaging experience, crucial for maintaining user satisfaction and engagement [**?**].

3.2.1 Stateful and Stateless Widgets in Flutter

In Flutter, widgets are categorized into two main types: Stateless and Stateful, distinguished by their ability to maintain state.

- **Stateless Widgets:** These widgets are immutable, which means once they are created, their properties cannot be changed. All values are final and only depend on the information provided at the time of their creation. Stateless widgets are perfect for static UI elements that do not require any interactive or changing features, such as labels, icons, and decorative graphics. They are efficient for simple displays where the content does not need to adjust based on user interaction or other factors [**?**].

- **Stateful Widgets:** Contrary to Stateless Widgets, Stateful Widgets are designed to be dynamic and can maintain state that changes over time. This capability is crucial for UI components that need to respond to user interactions or other events that might trigger a change in the UI. Examples of such components include text fields, checkboxes, sliders, and other interactive forms. Stateful Widgets allow the app to be interactive and responsive by updating the UI in response to events, ensuring that the user interface reflects the most current states of the application [**?**]. This fundamental distinction between Stateless and Stateful Widgets is a cornerstone in Flutter development, as it directly influences how an app's interface responds to user interactions and manages updates efficiently, ultimately affecting the user experience and performance [**?**].

3.2.2 Widget Variations

Widgets in Flutter act as the foundational bricks of the application's user interface, each tailored for specific roles on the screen:

- **List View Widget:** Used to display a list of widgets that are scrollable. Highly practical tool for building lists of items such as contacts, messages, or other similar items. This widget can be customized, allowing you to adjust its properties to create the desired look and feel for your list [**?**].
- **Scaffold Widget:** The Scaffold widget is the most important widget in Flutter and is essential for creating the app's layout. It provides a basic structure for other widgets to be added, serving as a foundational framework for building complex user interfaces. The Scaffold widget in Flutter comes with several pre-built elements, including an App Bar, Floating Action Button, Drawer, and Bottom Navigation Bar, which can be easily customized to fit the needs of your application [**?**].
- **CategoryGridItem Widget**: Designed for visually displaying categories in a grid layout, this widget enhances user interaction by providing clickable category items, each styled with relevant imagery and text. It's intended to make navigation through various pub or beverage categories both intuitive and engaging [**?**].
- **MainDrawer Widget**: This widget would act as a slide-out menu, accessible from multiple areas within the app. It's aimed at efficiently organizing navigation links to various app sections like user profiles and settings, enhancing accessibility without cluttering the main interface [**?**].
- **FormContainerWidget**: Essential for forms within the app, this widget would organize form fields such as text inputs and buttons into a coherent layout. It ensures that user data entry is streamlined and visually consistent across different forms [**?**].
- **GridView Widget**: Utilized for displaying items in a structured grid, this widget would be adaptable for various content types, including pub listings or menu items. Its customization options would allow for tailored spacing, scrolling, and alignment, making it versatile for different display needs [**?**].
- **BottomNavigationBar Widget**: Positioned at the bottom of the app's interface, this widget would facilitate easy navigation between the app's major sections, such as home, search, and profile. It's designed to be user-friendly and customizable, ensuring that essential functionalities are readily accessible [**?**].
- **AppBar Widget:** The AppBar widget serves as the top navigation bar, providing quick access to drawer menus, search functionalities, and other page-specific actions. Highly customizable, it can be tailored to include

branding elements, navigation icons, and additional actions, ensuring a cohesive user interface and enhancing overall usability [?].

3.3 Managing State in Flutter Applications

State management constitutes a pivotal facet of Flutter application development, encompassing methodologies for storing, updating, and disseminating data and UI states across an application. Flutter offers a plethora of techniques and architectures to facilitate state management, tailored to diverse project sizes and complexities.

3.3.1 Basic State Management Techniques

- **Local State Management**: Predominantly employed within Stateful Widgets, this approach confines state management to the widget itself. Leveraging the `setState()` method, developers can effectuate state updates and trigger UI rebuilds, rendering it ideal for rudimentary interactions such as UI element toggling or text updates [?].

3.3.2 Intermediate State Management Techniques

- **InheritedWidget and InheritedModel**: These foundational Flutter constructs streamline data distribution throughout the widget tree. Particularly advantageous for disseminating data required by multiple widgets concurrently, such as user settings, themes, or login states [?].

3.3.3 Advanced State Management Solutions

- **Provider**: Building upon InheritedWidget, Provider streamlines data flow within applications, curtailing code redundancy. It enables widgets to subscribe to model changes and selectively trigger rebuilds, efficacious for medium to large-scale projects characterized by frequent and pervasive state alterations [?].

- **Riverpod**: Evolving from Provider, Riverpod advocates for a clear separation of concerns, decoupling state management from the widget tree to enhance testability and modularity. Its refined API supports both global and local state management, with enhanced dependency handling relative to Provider [?].

- **Bloc Pattern (Business Logic Component)**: The Bloc pattern segregates business logic from the user interface, managing state and events through streams. Particularly advantageous for applications harboring complex states or substantial business logic, fostering responsive and scalable UIs adept at reacting to asynchronous data efficiently [**?**].
- **Redux**: Renowned for its predictability, Redux centralizes the entire application state within a single immutable store, with alterations triggered by actions processed through reducers. Suited for large applications necessitating high testability and predictability, albeit introducing more boilerplate than some counterparts [**?**].
- **MobX**: Leveraging observables and reactions, MobX orchestrates state changes reactively, mitigating boilerplate and automating UI updates in response to state alterations. Its reactive nature appeals to developers seeking uncomplicated state management with minimal overhead [**?**].

The choice of a state management strategy hinges on the specific requisites of the application. While rudimentary applications may suffice with basic state management, sophisticated systems can derive benefits from advanced solutions such as Bloc or Riverpod. Effective state management underpins the performance, responsiveness, maintainability, and scalability of Flutter applications. Thus, the selection of an apt state management tool is pivotal, delineating the seamlessness of feature integration and the robustness of the application's response to user interactions and data updates [**?**].

3.4 Flutter's Testing Framework

Flutter provides a robust testing framework that supports unit, widget, and integration tests, each crucial for ensuring different aspects of app functionality. Unit tests focus on testing individual functions and classes for correct logic, widget tests check

if widgets render correctly and handle user interactions as expected, while integration tests verify the interaction between different parts of the app, ensuring seamless operation. Flutter's testing framework is well-documented and integrates seamlessly with popular testing libraries, making it convenient for developers to write and execute tests during the development process [**?**].

3.4.1 Unit Testing in Flutter

Unit tests in Flutter are written using the built-in testing library, which provides a set of functions and matchers for writing tests. These tests are typically used

to test individual functions, methods, or classes in isolation, ensuring that they behave as expected under different conditions. By isolating the code being tested, unit tests help identify and fix bugs early in the development process, improving code quality and maintainability [**?**].

3.4.2 Widget Testing in Flutter

Widget tests in Flutter focus on testing UI components, ensuring that they render correctly and respond to user interactions as expected. These tests are written using the Flutter testing library and allow developers to simulate user interactions such as taps and scrolls to verify the behavior of UI components. Widget tests are crucial for detecting UI-related bugs and ensuring a smooth user experience across different devices and screen sizes [**?**].

3.4.3 Integration Testing in Flutter

Integration tests in Flutter verify the interaction between different parts of the app, including UI components, state management, and external dependencies. These tests are written using the Flutter driver API, which allows developers to simulate user interactions and verify the behavior of the entire app. Integration tests are essential for detecting integration issues and ensuring that the app functions correctly as a whole [**?**].

3.4.4 Test-Driven Development (TDD) in Flutter

Test-Driven Development (TDD) is a development approach that prioritizes writing tests before writing code. In Flutter, TDD is supported by the built-in testing framework and encourages developers to define test cases based on expected behavior before implementing the corresponding code. By following a TDD approach, developers can ensure that their code is thoroughly tested and meets the specified requirements, leading to higher code quality and fewer bugs [**?**]

Chapter 4

System Design

The purpose of the system design section in this dissertation is to provide a comprehensive analysis of the architectural framework and the underlying technologies of the "Pub-Wiser" application. This overview will detail how each component of the application interacts within the system to deliver a seamless and efficient user experience. Understanding the system design is crucial for appreciating the technical complexities and the strategic decisions that contribute to the application's functionality and scalability.

4.1 Purpose of System Design

The system design encapsulates the blueprint and the technical strategy behind the application. It lays out the structural foundation that supports all functional requirements of the application, addressing both the immediate and future needs of the system. The primary goals of exploring the system design in this context are to:

- **Cohesion of Components** The "Pub-Wiser" application demonstrates effective integration across its components to provide a seamless user experience. Developed with Flutter, the frontend utilizes dynamic widgets and the Provider package for efficient state management, enabling responsive updates based on backend changes. The backend, powered by Firebase, supports real-time data management and user authentication, while integration with Google Places API enhances functionality by providing detailed location information.

- **Scalability and Maintainability** The system's design is modular, using Flutter for the frontend to simplify updates and expansions. Firebase handles backend scalability automatically, adapting to increased user loads without

manual intervention. This scalable architecture ensures the application can grow and adapt over time without compromising performance.

- **Robustness and Security** Security is paramount, with Firebase Authentication safeguarding user identities and sessions. Firestore's stringent security rules ensure data integrity and privacy. Comprehensive error handling and regular security updates help maintain the application's robustness and protect against emerging threats.

4.2 Architecture

The architecture of the Pub Wiser app is organised around several directories and files within the `/lib` directory. Each file or group of files serves a specific purpose in the project's structure. Let's break down the main components:

Main Components

1. `main.dart`: This is the entry point of the application. It sets up the app environment and the initial route.
2. `const.dart`: Contains constant values used throughout the app such as the Google Maps API.
3. `global/common/toast.dart`: Provides utilities for showing toast messages across the app, used for feedback on user actions.

Features

The app's features are split into different categories and pages:

App Pages and Widgets

- `features/app/pages/home_page.dart`: The home page of the app.
- `features/app/pages/categories.dart`: Manages the categories page.
- `features/app/pages/filters.dart`: Contains the filters page for sorting or searching.
- `features/app/pages/pub_details.dart`: Displays details of a specific pub.
- `features/app/pages/pubs_page.dart`: Page to list pubs.

- `features/app/pages/tabs.dart`: Handles the tab navigation.
- `features/app/splash_screen/splash_screen.dart`: The initial splash screen shown on app start.
- `features/app/widgets/`: Widgets used across various app pages, like `category_grid_ite` `main_drawer.dart`, `pub_item.dart`, `pub_item_trait.dart` for different UI components.

User Authentication

- `features/user_auth/UI/pages/login_page.dart` and `sign_up_page.dart`: Pages for user login and sign up.
- `features/user_auth/UI/widgets/form_container_widget.dart`: Reusable widget for form inputs.
- `features/user_auth/firebase_auth_implementation/firebase_auth_services.dart` Services for managing Firebase authentication.

Models

Defines the data models used across the application:

- `models/*.dart`: Files like `bar_details.dart`, `category.dart`, `drink.dart`, `filter_enum.dart`, `filter_model.dart`, `pub.dart` define the structure of data objects used.

Data Handling

Data Access

- `data/*.dart`: Such as `pub_data.dart` for mock data access.
- `data/firebase_store_implementation/firestore_pubData.dart`: Manages Firebase Firestore interactions for pub data.

Providers

- `providers/*.dart`: Files like `filter_provider.dart` and `pubs_provider.dart` manage state for filters and pub listings using the Provider pattern.

Services

- `services/*.dart`: Such as `firestore_service.dart` and `google_places_client.dart` for external data interactions and API management.

Communication & Data Flow

- **Firebase Firestore**: Handles data storage and real-time updates.
- **Google Places API**: Integrates external data for pub locations.
- **State Management**: Uses Provider for state management across the app.

This section describes the application architecture, which is composed of the following components:

- **Frontend**: Developed using Flutter and Dart, providing a responsive UI for the pint price comparison application.
- **Authentication**: Handled through Google Firebase, ensuring secure access for users.
- **Database**: User data, categories, pub data, and pint prices/suggestions are stored in Google Firestore.
- **Location Based Tracking**: Implemented with Google Maps to display information for each pub on the map.

4.3 Frontend Design

The "Pub-Wiser" application's frontend is developed using Flutter, enabling a uniform and efficient user experience across both iOS and Android platforms. Flutter facilitates rapid UI development with its extensive widget library, allowing for high-performance, customizable interfaces that are compiled directly to native code.

4.3.1 Flutter as the Development Framework:

Flutter's ability to handle complex, responsive UIs through a rich collection of widgets is integral to the application's design, ensuring smooth and responsive interactions.

4.3.2 Responsive UI

The application employs GridView and ListView for structured content display, along with custom widgets like CategoryGridItem and PubItem to enhance user interaction and display efficiency.

4.3.3 State Management with Provider

"Pub-Wiser" implements an MVVM (Model-View-ViewModel) architecture using the Provider package, which separates the UI (View) from the business logic (ViewModel). This pattern enhances maintainability and scalability by ensuring the UI updates dynamically in response to data changes managed by the ViewModel, without direct dependency on the business logic.

4.4 Backend

The backend infrastructure of the "Pub-Wiser" application is built on Firebase, a comprehensive development platform by Google that provides a suite of cloud-based tools and services. This robust backend solution enhances the application's capability to handle data securely and efficiently, providing seamless integration with the frontend.

4.4.1 Firebase in Backend Services

Firebase acts as the backbone for the "Pub-Wiser" app's backend, providing a range of services that support real-time data synchronization, user authentication, and dynamic data storage. Its serverless architecture allows developers to focus on creating rich user experiences without the complexities of server management. Firebase's scalable infrastructure ensures that as the app grows in user base or data complexity, the backend can effortlessly scale to meet increased demands.

4.4.2 Firestore for Real-Time Data Handling

Firestore is a NoSQL database included within Firebase that offers live data synchronization across user devices in milliseconds. It is pivotal in managing the dynamic and real-time aspects of "Pub-Wiser," such as user reviews, pub ratings, and profile updates. Firestore's structure is document-oriented, allowing for flexible, schema-less data storage that is ideal for the varied and evolving data needs of the app. Its powerful querying capabilities ensure efficient data retrieval, updates, and deletions, enhancing the responsiveness of the application.

4.4.3 Firebase Authentication

Firebase Authentication is integrated into "Pub-Wiser" to provide a secure and hassle-free user authentication process. This service supports authentication using emails, phone numbers, and popular third-party providers like Google, Facebook, and Twitter. Firebase Authentication also handles user sessions and secures user data with robust security protocols, ensuring that user information is protected. This layer of security is critical for maintaining trust and integrity within the "Pub-Wiser" app, allowing users to manage their accounts and personal information safely.

4.5 Incorporating Google Places API

The Google Places API is a key component of the "Pub-Wiser" app. It supplies detailed information about pubs, including their locations, user ratings, reviews, and photos. By using this API, "Pub-Wiser" offers users up-to-date and thorough information about pubs, helping them make well-informed choices based on location, popularity, and user feedback. This feature is particularly valuable for a discovery-focused app like "Pub-Wiser," as accurate and rich content greatly enhances user experience and engagement.

4.6 Implementation of Dynamic Pint Prices

To enhance the user experience and provide real-time information, the "Pub-Wiser" application incorporates dynamic pricing on the PubDetailsPage. This feature is designed to display up-to-date pricing information for drinks at various pubs, allowing users to make informed decisions based on current offers and pricing trends. Here's how this functionality was implemented:

4.6.1 Data Retrieval and Management

The application utilizes Firebase Firestore to store and manage data about drink prices at different pubs. Firestore's real-time capabilities ensure that any changes made by pub owners to their pricing are immediately reflected in the app. When a user navigates to the PubDetailsPage, the app queries Firestore to fetch the latest pricing data for the selected pub

The implementation of dynamic real-time pint prices in the "Pub-Wiser" application is a prime example of effectively leveraging Firebase Firestore to enhance user interactivity and provide up-to-date information. Here's a description of the implementation based on the provided code snippet:

Initialization of Data Retrieval: Upon navigating to a pub's detail page within the app, the 'initState' method triggers the retrieval of the current drink prices from Firestore using the 'FirestoreService().getDrinkPrices(widget.pub.id)' method. This asynchronously fetches a list of 'Drink' objects, which includes the name and current price of each drink available at the selected pub.

Displaying Prices: The 'FutureBuilder' widget listens for the completion of the drinks future and builds the UI accordingly. It displays a loading indicator while the data is being fetched and handles any potential errors by printing log messages and displaying appropriate error messages to the user.

Updating Prices: Users can update the price of drinks by tapping on a listed drink, which brings up a dialog with an input field for the new price. The input is processed by the 'updateDrinkPrice' method from the 'FirestoreService', which updates the 'price' field of the corresponding document in the Firestore 'drinkPrices' collection for that pub.

User Interaction - Price Suggestions: Users are also able to suggest new prices for drinks. When a user submits a new price suggestion, it is added to the 'PriceSuggestions' subcollection under the specific 'drinkPrices' document in Firestore. This suggestion includes details about the user, the suggested price, and a timestamp, allowing other users to view and vote on it.

Real-Time Price Updates: The app incorporates a listener for changes in the 'PriceSuggestions' subcollection. When new suggestions are added or votes are updated, the app fetches the top-voted suggestion and updates the displayed drink price if the suggested price differs from the current price.

Optimizing User Experience: To maintain a seamless user experience, any changes in the drink prices due to updates or new top-voted suggestions are reflected in the UI in real-time. This ensures that all users viewing the pub details see the most current prices without needing to manually refresh the page.

Through the implementation of these features, the "Pub-Wiser" app provides a dynamic and real-time pricing system. It not only keeps the users engaged by allowing them to contribute to the content but also ensures the information they receive is current and community-verified. This interactive approach demonstrates the app's commitment to real-time data handling and user collaboration.

4.7 Implementation of Google Maps and Location Features

In the "Pub-Wiser" application, Google Maps and location features play a central role in enriching the user experience by providing interactive maps and location-based services. The implementation leverages Google Maps API, Flutter's '$google_maps_flutter$'

package, and the '*location*' package to provide real-time location updates and display nearby pubs or bars.

Utilizing '$flutter_polyline_points$', the app can draw routes on the map to guide users to their selected destinations. This integration of Google Maps and location services enhances the functionality of the Pub-Wiser app, allowing users to navigate to nearby pubs with ease.

Google Maps Integration: The integration begins with setting up the 'GoogleMap' widget within the app's 'HomePage' stateful widget. This widget is configured with an initial camera position centered around a predefined location, such as Galway City. Markers for each bar are dynamically added to the map using the data fetched from the Google Places API, allowing users to visually navigate and select pubs within their vicinity.

Fetching Location Data: To display relevant pub information based on user location, the app initiates a 'fetchBars' method, which sends a request to the Google Places API. The request includes parameters such as latitude, longitude, and search radius, specifying that we're interested in locations categorized as bars or pubs. The API response is processed, and for each result, a 'Place' object is created, which includes details like name, latitude, longitude, and place ID.

Displaying Markers on the Map: For each 'Place' object retrieved, a marker is created and added to the map, represented by a 'Marker' object in Flutter. The marker includes details such as the bar's name and its geographic coordinates, enabling users to tap on the markers to view more details or get directions.

User Location Updates: The app uses the 'location' package to track the user's current location. The location updates are streamed via a 'StreamSubscription', updating the user's position on the map in real time. The app also includes functionality to center the map on the user's current location, enhancing the navigational experience.

Polyline Routes: Utilizing '$flutter_polyline_points$', the app can draw routes on the map. The 'getPolyline' method fetches points between two locations, which are then used to create a 'Polyline' object. This polyline represents the path between the user's location and the selected bar, providing visual guidance for navigation.

Customizing the Experience: Custom icons and additional map controls are implemented for a more personalized user experience. For instance, a custom marker is used to denote the user's current location, and additional UI controls could be added for functionalities like changing the map view or filtering displayed pubs.

The Google Maps integration in "Pub-Wiser" is a complex feature that significantly contributes to the app's functionality, offering users an intuitive and interactive way to discover pubs. It demonstrates the seamless combination of

multiple services and packages to create a rich, location-aware app component.

Chapter 5

System Evaluation

Evaluate your project against the objectives set out in the introduction. This chapter should present results if applicable and discuss the strengths and weaknesses of your system. This is a clear opportunity for you to demonstrate your critical thinking in relation to the project.

5.1 Working with Tables

This chapter evaluates the project and the application developed against the literature review and the project objectives.

5.2 Application of the Flutter Mobile Application Development Framework

The literature review identified the Flutter Framework as a popular open-source framework for building aesthetically pleasing mobile apps, known for its ease of use, flexibility, and fast development capabilities. The practical application of these findings revealed that:

- Flutter's ease of use and fast development capabilities were consistent with the literature review, particularly for those with experience in Java or JavaScript, due to the similarity with Dart, Flutter's programming language.

- The framework supports an extensive range of UI components without the need for deep knowledge of Dart, coupled with a strong technical community and a multitude of high-quality tutorials and articles.

- Flutter's integration with Visual Studio Code (VS Code) enhances development speed with numerous helpful extensions and the Hot Reload feature, which offers almost instant rendering of code changes aiding in testing and debugging.
- While Flutter is well-supported in Android Studio, it requires significant RAM, suggesting that VS Code is more suitable for development on machines with less RAM.
- The availability of various Flutter/Dart packages facilitates feature integration, although package updates often lead to compatibility issues, highlighting a need for better version management resources.

5.2.1 Integration with Other Software Development Products

Integrating Flutter with various software products proved generally straightforward but with some complexities:

- **Firebase Authentication:** The integration process was straightforward, utilizing pre-built UI components and the Flutter `firebase_auth` package to connect the application with Firebase Authentication efficiently.
- **Google Cloud Firestore:** Integration was simple to set up but complex to implement, requiring detailed coding to manage database operations effectively.
- **Google Dialogflow Chatbot:** The `dialog_flowtter` package facilitated the integration of Dialogflow with Flutter, although setting up the Dialogflow agent and intents presented challenges.
- **Stripe Payments:** Using the `stripe_payment` package along with Firebase functions enabled secure server-side payment processing. The necessity for a Firebase billing account to use Cloud Functions, even though no charges are incurred, was a significant setup step.

5.2.2 State Management

The choice of state management solution proved critical, with over 50 available packages providing minimal official guidance. The GetX package was selected for its simplicity, effectively addressing the state management needs of this project.

5.3 Summary

This chapter discussed the application's performance against the literature review and project objectives, highlighting the effective use of Flutter and its ecosystem to build a robust mobile application. The integration of Firebase services, Google Dialogflow, and Stripe payments within the Flutter framework demonstrated a high level of compatibility and performance, aligning with the project's goals.

This chapter evaluates the project and the application developed against the literature review and the project objectives.

5.4 Flutter vs Ionic

Flutter and Ionic are two of the most popular frameworks for developing mobile applications on Android and iOS from a single codebase. Flutter, developed by Google, and Ionic, developed by Facebook, both utilise a reactive programming model but have distinct characteristics.

- **Flutter** is known for its fast development cycle due to the hot reload feature and its ability to produce customised, highly responsive interfaces using a modern UI toolkit.
- **Ionic** boasts a larger community and a wider range of third-party libraries, which can simplify finding support and resources. However, Ionic generally offers better performance, particularly in rendering complex UIs, compared to Flutter.

5.5 The Development Process

5.5.1 Agile Development Approach

The project adopted a hybrid agile methodology which proved effective. Key aspects included:

- Using user stories to streamline requirements documentation.
- Organising work into two-week sprints, which helped maintain project momentum and focus.
- Using Jira as a project management tool to track progress and manage tasks efficiently.

5.5.2 Testing

The manual testing approach was initially sufficient, but considering the benefits of automated testing could enhance future project updates. Key points include:

- Automated testing can significantly reduce time and effort, especially beneficial for repetitive tests or multi-platform compatibility checks.
- Expanding testing to multiple users and including load and stress testing are critical next steps to ensure the application's robustness under heavy usage.

5.5.3 Deployment

Deploying the application to the Google Play Store involved several steps that were particularly challenging for new developers, including:

- Creating a Google Play Developer account.
- Generating and submitting a signed APK file.
- Complying with Google Play Console guidelines and requirements.

5.6 Functional Evaluation

The application was developed to facilitate food ordering through a mobile platform, leveraging various technologies to ensure functionality, usability, reliability, performance, and security.

5.6.1 Functionality and Usability

The application features included user registration, menu browsing, and secure payment options. The user interface was designed to be intuitive, allowing for easy navigation and interaction.

5.6.2 Reliability and Performance

Initial tests showed the application to be stable and responsive. Future tests should include scenarios with multiple users to optimize performance and ensure the app's scalability.

5.6.3 Security

Security measures were implemented using Google's authentication and Firestore services along with a secure Stripe payment integration through Firebase functions to protect user data and transactions.

5.7 Summary

This chapter provided a comprehensive evaluation of the developed application against the initial literature review and project objectives. Strengths were highlighted, such as the effective use of Flutter and its integration with other technologies, and areas for improvement were identified, particularly in terms of scalability and multi-user support.

5.8 Conclusion

The "Pub-Wiser" project aimed to achieve several objectives:

- Master the Flutter framework for developing mobile apps, including integration with backend services like Firebase.
- Evaluate the support provided by Google and the Flutter community.
- Investigate using Flutter with Firebase's Cloud Firestore for handling real-time data.
- Enhance skills in mobile app development and compare Flutter with other frameworks, like React Native.
- Implement and test the integration of third-party services, such as Firebase Authentication and Google Places API.
- Apply agile development principles to manage the project effectively.
- Develop a user-focused app, "Pub-Wiser," that includes features like pub discovery, dynamic pricing updates, and user-generated content.

Key Findings

- **Flutter Proficiency:** Flutter proved effective for creating responsive and visually appealing interfaces. Its programming language, Dart, is easy for those with Java or JavaScript experience to learn.
- **Development Tools:** The project used VS Code for its lightweight nature and Android Studio for emulator testing, particularly useful for systems with limited resources.
- **Community and Google Support:** The strong community and Google's ongoing support provided extensive resources, enhancing the development process.
- **Framework Comparison:** Choosing between Flutter and other frameworks like React Native depended on the project's specific needs, such as UI design, customization, and performance.
- **Service Integration:** The app successfully integrated with key Firebase services and the Google Places API, although managing state in a reactive environment was challenging.

- **Data Security:** Using Firebase for authentication and storing user-generated content in Firestore ensured strong data security and privacy compliance.
- **Performance and Scalability:** Initial performance was satisfactory, but more extensive testing is needed to ensure the app can handle multiple users effectively.
- **Agile Development:** Using an agile approach, including user stories and iterative sprints, helped keep the project focused and adaptable.
- **Testing and Deployment:** The project primarily used manual testing. Future projects could benefit from more automated testing frameworks in Flutter. Deployment to app stores highlighted complexities that could challenge those new to app publishing.

Summary

Overall, the "Pub-Wiser" project successfully created a functional and user-friendly app for pub exploration and social interaction. While it met its primary goals, areas like multi-user testing and deployment processes present opportunities for further development and refinement. The conclusions are based on a thorough evaluation discussed in the System Evaluation chapter, acknowledging the project's successes and identifying areas for future improvement.

Bibliography