UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Higher Polytechnic School of Alcoi

# Developing a cross-platform mobile application using Flutter

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Lahtinen, Arttu

Tutor: Pérez Llorens, Rubén

ACADEMIC YEAR: 2023/2024

## Abstract

The increasing popularity of mobile devices has led to an increase in the demand for mobile applications. However, developing applications that work across different platforms can be challenging, as each platform has its unique development requirements. In this thesis, we present the development of a cross-platform mobile application using Flutter.

Keywords: Flutter, Dart, Cross-platform mobile development

## Resumen

La creciente popularidad de los dispositivos móviles ha llevado a un aumento en la demanda de aplicaciones móviles. Sin embargo, desarrollar aplicaciones que funcionen en diferentes plataformas puede ser desafiante, ya que cada plataforma tiene sus requisitos de desarrollo únicos. En esta tesis, presentamos el desarrollo de una aplicación móvil multiplataforma utilizando Flutter.

Palabras clave: Flutter, Dart, El desarrollo móvil multiplataforma

# Contents


Abstract ........ 1
Table of Figures ........ 3
1 Introduction ........ 1
2 Technologies and tools ........ 3
2.1 Cross-platform mobile development overview ........ 3
2.1.1 Native Development ........ 3
2.1.2 Cross-platform Development ........ 3
2.2 Comparison of Cross-Platform Development Frameworks ........ 4
2.3 Flutter ........ 5
2.4 Dart ........ 9
2.5 Backend ........ 10
2.5.1 Node.js and Express Framework ........ 10
2.5.2 MongoDB ........ 12
2.5.3 JSON Web Token (JWT) ........ 13
2.5.4 Bcrypt ........ 15
2.5 Visual Studio Code ........ 17
2.5.1 Extensions in Visual Studio Code ........ 17
2.6 Postman ........ 18
2.7 Git ........ 18
2.7.1 GitHub ........ 19
2.8 Selection Criteria for Technologies and Tools ........ 19
3 Development of the mobile application ........ 22
3.1 System Architecture and Design ........ 22
3.2 User interface design and Implementation ........ 23
3.3 Backend development with Node.js and Express ........ 33
3.4 Frontend development using Flutter ........ 42
4 Summary and reflection ........ 57
References ........ 58
Appendices ........ 60

## Table of Figures

*Figure 1 Flutter Architectural Overview (Flutter n.d.)* ........ 7
*Figure 2 Structure of Flutter Project* ........ 8
*Figure 3 Default main.dart generated by Flutter* ........ 9
*Figure 4: Example of a basic application using node.js and express.* ........ 11
*Figure 5 MongoDB user data* ........ 13
*Figure 6 Generating JWT using JSON web token.* ........ 15
*Figure 7 Hashing password using Bcrypt* ........ 16
*Figure 8 Architectural structure of the application* ........ 22
*Figure 9 Login screen* ........ 24
*Figure 10 Register screen* ........ 24
*Figure 11 All notes (grid view)* ........ 25
*Figure 12 All notes (list view)* ........ 25
*Figure 13 Notes screen* ........ 26
*Figure 14 Adding a new note* ........ 26
*Figure 15 Adding a todo* ........ 27
*Figure 16 Adding a tag* ........ 27
*Figure 17 Adding a contact* ........ 28
*Figure 18 All contacts screen* ........ 29
*Figure 19 Adding a contact to note* ........ 30
*Figure 20 Note with contact* ........ 30
*Figure 21 All tags screen* ........ 31
*Figure 22 Deleting a note* ........ 31
*Figure 23 Search function* ........ 32
*Figure 24 Sidebar* ........ 33
*Figure 25 Server.js* ........ 33
*Figure 26 Noteroutes* ........ 34
*Figure 27 User model* ........ 35
*Figure 28 User controller* ........ 38
*Figure 29 Backend file structure* ........ 39
*Figure 30 getNotes* ........ 40
*Figure 31 updateNote* ........ 40
*Figure 32 User model* ........ 43
*Figure 33 auth provider* ........ 46
*Figure 34 login screen code part1* ........ 48
*Figure 35 login screen code part2* ........ 50
*Figure 36 ApiService* ........ 52
*Figure 37 main.dart file* ........ 56
*Figure 38 Structure of frontend* ........ 56

# 1 Introduction

Mobile applications have become essential in our daily lives, providing convenience, entertainment, and access to information on the go. The increasing popularity of mobile devices has led to a rise in the demand for applications that can run seamlessly on multiple platforms, such as iOS and Android. This challenge has prompted the development of cross-platform mobile frameworks that allow developers to write code once and deploy it across various platforms.

This thesis focuses on developing a cross-platform mobile application using the robust Flutter framework. Flutter has gained significant popularity for creating high-performance, visually appealing, and feature-rich applications for both Android and iOS platforms.

The primary motivation behind this project is to explore the potential of Flutter and its programming language, Dart, which has become prominent in the mobile development field. My academic experience at UPV involved a mobile application course using Xamarin instead of Flutter. However, Flutter caught my interest during an internship at Casamedia, leading to the decision to use it for this project.

The internship provided a basic understanding of mobile programming with Flutter and impressed me with its capabilities and flexibility. Building upon previous group project experience with the MERN stack (MongoDB, Express, React, Node.js), the inspiration to create a full-stack mobile application using Flutter as the frontend and Node.js, Express, and MongoDB for the backend emerged due to its efficiency.

This thesis has two main objectives. The first is to develop a simple yet functional mobile application with common components found in many apps, serving as a foundation for further expansion and customization to meet diverse user needs. The second is to gain valuable experience in general mobile development, becoming proficient in Flutter and Dart for frontend development and honing skills in Node.js and Express for backend development.

This project will explore mobile app design, frontend and backend development, database integration, and best practices in cross-platform mobile application development. The focus is on creating a comprehensive and well-rounded application, addressing both functional and design aspects.

Through this thesis, I aim to contribute to the dynamic landscape of mobile app development while expanding my knowledge and expertise in cutting-edge technologies. I believe that

the culmination of my efforts will result in an engaging and user-friendly mobile application showcasing the potential of Flutter-based cross-platform development.

# 2 Technologies and tools

This chapter provides an overview of the technologies and tools utilized in the development of the cross-platform mobile application.

## 2.1 Cross-platform mobile development overview

In today's world, a significant majority of mobile users have either Android or iOS device. According to statistics on Mobile Operating System Market Share Worldwide, Android holds the largest share at 68.79%, while iOS accounts for 30.44% (StatCounter, 2023).

Developers have two primary approaches for developing mobile applications: native development and cross-platform development.

Cross-platform mobile development enables developers to build applications that run seamlessly on multiple platforms.

### 2.1.1 Native Development

Native development involves building mobile applications specifically for a particular mobile operating system, such as Android or iOS. Developers use platform-specific programming languages and tools to optimize the app for each platform. Native apps offer high performance and responsiveness, adhering to platform-specific design guidelines for an intuitive user experience. They also have direct access to the device's hardware features. However, native development requires separate codebases and teams for each platform, increasing costs and coordination efforts. Maintaining two codebases can be time-consuming and prone to more errors. Furthermore, platform-specific logic can lead to inconsistencies between Android and iOS apps. (Kotlin, 2023.)

### 2.1.2 Cross-platform Development

Cross-platform development creates apps that can run on multiple operating systems. By sharing code across platforms, developers can reduce development time and costs. Cross-platform frameworks like Flutter, React Native, and Kotlin Multi-platform Mobile facilitate code sharing and provide access to common device features. However, cross-platform apps may have performance issues due to the abstraction layer introduced by the frameworks. Customizing access to platform-specific APIs requires additional effort. Ensuring a consistent user interface across platforms can be challenging, as shared UI components may not perfectly match native design patterns. When choosing between native and cross-

platform development, it is crucial to consider project requirements, target audience, and available resources. (Kotlin, 2023.)

## 2.2 Comparison of Cross-Platform Development Frameworks

| Feature/Aspect | Flutter | React Native | .NET MAUI |
|---|---|---|---|
| Development Language | Dart | Javascript, React | C# |
| User Interface | Customizable widgets, Skia rendering engine | Customizable, native look and feel, community-driven | Native-like across different platforms |
| Performance | Impressive, 60FPS | Good, native modules for performance | Excellent, native UI, .NET libraries |
| Development Tools | Strong community, Flutter DevTools, IDE support | Community support, hot reloading, IDE integration | Robust tools, Visual Studio, VS Code |
| Platform Support | iOS, Android, web, macOS, Windows | iOS, Android, web, macOS, Windows | iOS, Android, macOS, Windows |
| Community Support | Active and growing, Google | Large community, Facebook | Strong, Microsoft-backed |
| Learning Curve | Moderate | Accessible | Relatively gentle for C# developers |
| Long-term Viability | Promising | Established, open-source | Strong support, Microsoft |

In the realm of cross-platform mobile app development, choosing the right framework is crucial. This comparison table provides a snapshot of key aspects of three popular cross-platform frameworks: Flutter, React Native, and .NET MAUI. Each framework has its unique strengths and is suitable for different scenarios, ranging from performance-focused to ease of development. (Kanini, 2023).

## 2.3 Flutter

Flutter is a free and open-source framework developed by Google, designed to build natively compiled multi-platform mobile, web, and desktop applications from a single codebase. Flutter uses Dart programming language and provides a rich set of pre-designed widgets, offering a fast and efficient way to build visually appealing and interactive user interfaces (Flutter, n.d.).

Flutter is a good option for companies of all sizes, but especially for small teams and companies that can use Flutter to create an app that works with iOS, Android, Windows, Mac, and Linux. Some famous companies that use Flutter: Alibaba, BMW, eBay, Toyota, and Tencent (Flutter, n.d.).

Some of the features that make Flutter popular are fast development using Flutter's "hot reload" feature that allows developers to see the changes made in the code immediately, easy-to-learn Dart programming language that offers features like strong typing, null safety, and asynchronous programming, Native-like performance by utilizing a compiled programming language and a high-performance rendering engine, and access to native features using platform-specific plugins.

In a typical Flutter project, the application is organized into a modular and hierarchical structure, which promotes code reusability and separation of concerns. The main components of a Flutter project include:

1. **Main.dart**: This is the entry point of the Flutter application. It is where the app starts executing. The main function runs the **runApp** function, which takes the root Widget as its argument.

2. **pubspec.yaml**: This file manages the assets and dependencies for a Flutter app. In this file, all the packages that the project depends on are listed, and specific assets, like images, audio files, etc., are configured.

3. **lib folder**: This is where the Dart code lives. It usually includes the following subdirectories:

    - **screens** or **views**: Contains the UI code for different screens in the app.

    - **widgets**: Contains the code for smaller reusable UI components.

- **models**: Contains the data model for the app.
- **services**: Contains the code for business logic and data manipulation.

4. **assets folder**: Serves as a storage location for assets like images, fonts, or data files.
5. **test folder**: Contains the unit test files for testing the application.
6. **android and ios folders**: These folders encompass essential files required for building the app on Android and iOS platforms, respectively. These folders are generally kept the same unless platform-specific modifications, such as altering the app icon or initial loading screen, become necessary.

In terms of creating a basic Flutter project, it is as simple as running the command **Flutter create project_name** in the terminal. This command creates a new Flutter project with the above structure and includes a simple demo app you can build on.

The IDE used in this project also provides an even easier way to do this by pressing ctrl+shift+p and then choosing Flutter: New Project.

The power of Flutter lies in its widgets. Everything in Flutter is a widget, from structural elements like buttons and text fields to stylistic elements like fonts and colors. These widgets are combined to build the complete user interface, and each widget can be customized to achieve the desired look and feel. This widget-based approach makes Flutter highly flexible and intuitive, as developers can create complex UIs from a combination of simple, reusable widgets.

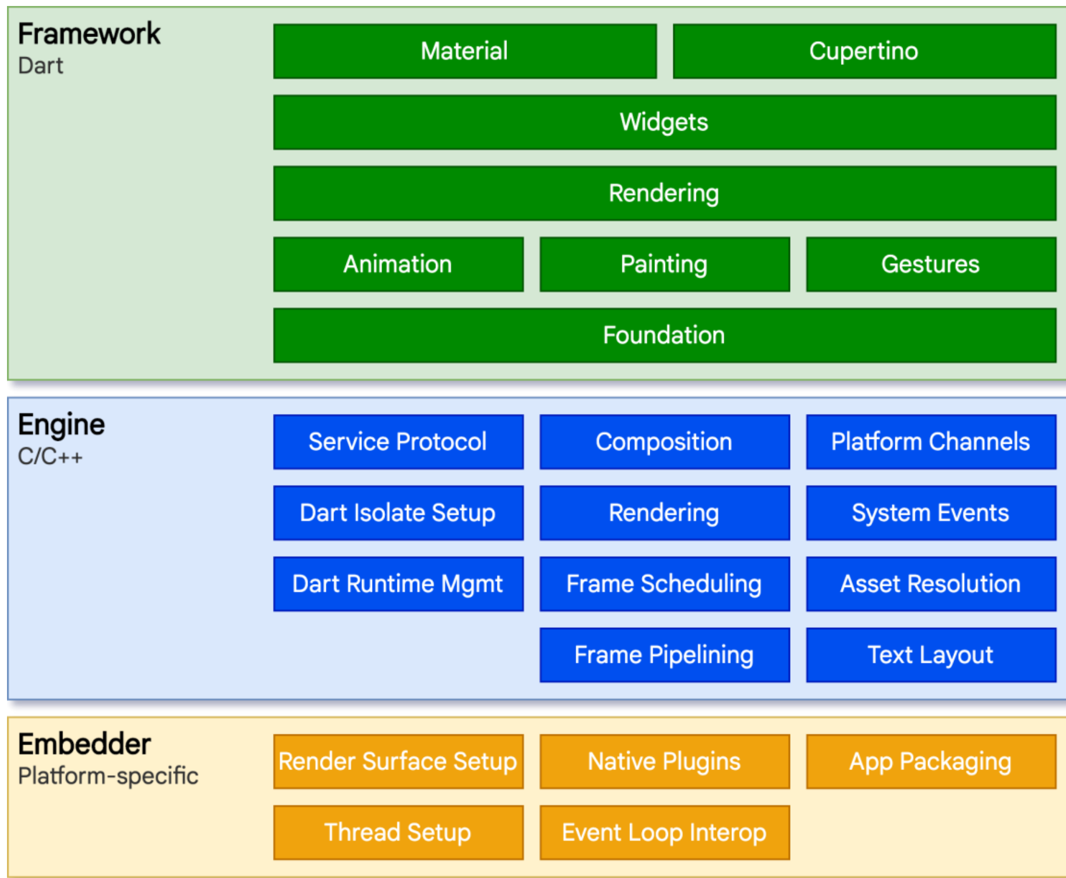


*Figure 1 Flutter Architectural Overview (Flutter n.d.)*

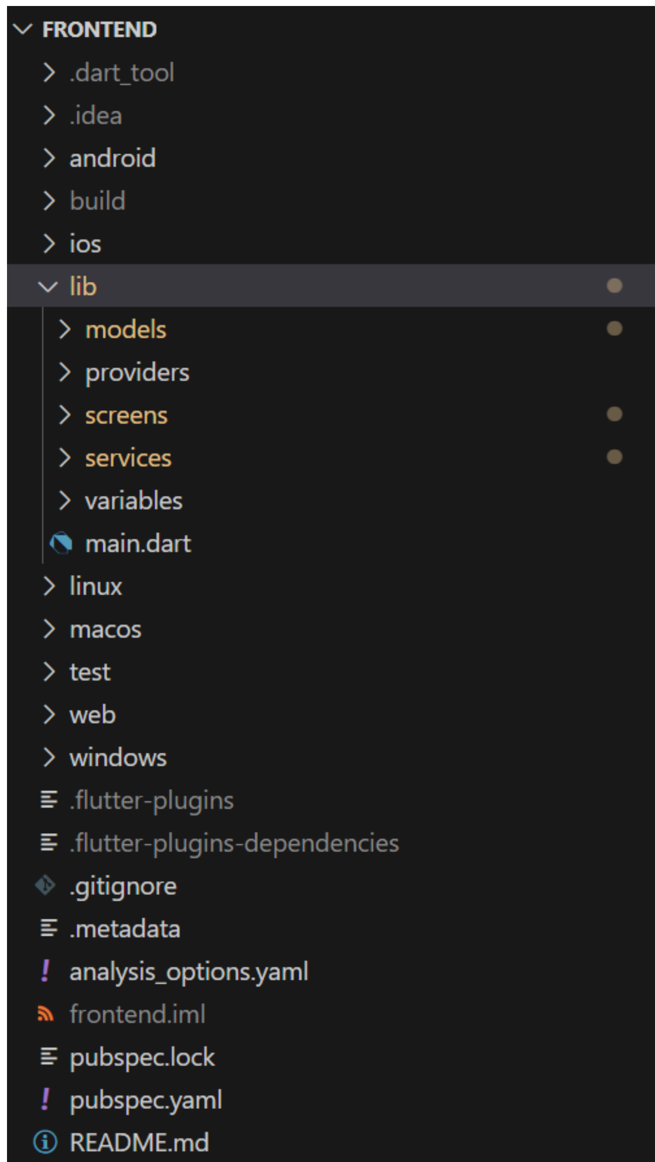


*Figure 2 Structure of Flutter Project*

```
lib > main.dart > MyHomePage
1   import 'package:flutter/material.dart';
2
    Run | Debug | Profile
3   void main() {
4     runApp(const MyApp());
5   }
6
7   class MyApp extends StatelessWidget {
8     const MyApp({super.key});
9
10    // This widget is the root of your application.
11    @override
12    Widget build(BuildContext context) {
13      return MaterialApp(
14        title: 'Flutter Demo',
15        theme: ThemeData(
16          // This is the theme of your application.
17          //
18          // Try running your application with "flutter run". You'll see the
19          // application has a blue toolbar. Then, without quitting the app, try
20          // changing the primarySwatch below to Colors.green and then invoke
21          // "hot reload" (press "r" in the console where you ran "flutter run",
22          // or simply save your changes to "hot reload" in a Flutter IDE).
23          // Notice that the counter didn't reset back to zero; the application
24          // is not restarted.
25          primarySwatch: Colors.blue,
26        ), // ThemeData
27        home: const MyHomePage(title: 'Flutter Demo Home Page'),
28      ); // MaterialApp
29    }
30  }
```

*Figure 3 Default main.dart generated by Flutter*

## 2.4 Dart

Dart is a client-optimized programming language designed for developing fast applications on various platforms. It aims to provide a productive and flexible programming language for multi-platform development and a versatile execution runtime platform for application frameworks. Dart is tailored explicitly for client development, prioritizing efficient development processes with features like sub-second stateful hot reload and high-quality production experiences across different compilation targets, including web, mobile, and desktop.

Dart serves as the foundation of Flutter, powering Flutter apps with its language and runtimes. Additionally, Dart supports essential developer tasks such as code formatting, analysis, and testing.

The Dart language emphasizes type safety, utilizing static type checking to ensure variables always match their static types. Type inference allows for optional type annotations, while the typing system provides flexibility using dynamic types combined with runtime checks when needed.

Dart incorporates sound null safety, which means variables are non-nullable by default unless specified otherwise. This feature protects against null exceptions through static code analysis, ensuring non-nullability is maintained at runtime. (Dart n.d.)

## 2.5 Backend

The backend of the mobile application is the powerhouse behind the scenes, responsible for data storage, processing, and serving APIs to the client-side application. This subsection delves into two essential components of the backend: Node.js and Express framework, which together form a potent duo for efficient server-side development, and MongoDB, a NoSQL database renowned for its versatility in data storage.

### 2.5.1 Node.js and Express Framework

Node.js is a server framework that utilizes JavaScript on the server, allowing developers to build and run software applications. It is designed to be efficient with non-blocking I/O, so you can proceed with the following tasks without waiting. This feature is one of the reasons Node.js is well-regarded for its efficiency. Node.js is also open-source and cross-platform, which means it can run on various platforms like Windows, Linux, and macOS.

With Node.js, you can generate dynamic content, manage files on the server, and handle data in the database. Node.js's single-threaded event loop abstracts I/O from external requests, leading to better control and efficiency in managing operations. Some main reasons for using Node.js include its speed, the ability to keep data in native JSON format in databases, and the wide availability of modules and community support. (Kursova, 2017.)

Express.js, on the other hand, is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is unopinionated, allowing developers to choose their own tools and plugins for developing APIs. With

Express.js, you have a myriad of HTTP utility methods and middleware at your disposal, making the creation of robust APIs quick and easy. (Express n.d.)

According to Stack Overflow's last year's developer survey (Stack Overflow, 2022), JavaScript was the most commonly used programming language for the tenth year in a row, and Node.js was the most commonly used web technology used by Professional Developers and those learning to code.

Creating a basic node.js project with express is as simple as running the command **npm init** in the terminal. This command initializes a new node.js project and creates a 'package.json' file. The 'package.json' file contains essential information about the project, such as its name, version, dependencies, and other configuration details. Then installing express as a dependency by running the command **npm i express** and creating a file 'server.js' for example, and adding the code below creates a basic app that returns 'Hello World!' when navigating to localhost:3000/ if run locally.

```
const express = require('express' 4.18.2 )
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

*Figure 4: Example of a basic application using node.js and express.*

Nodemon is a helpful tool for Node.js development as it monitors changes in the source code and automatically restarts the server whenever modifications are detected, providing a more streamlined and efficient development experience.

### 2.5.2 MongoDB

MongoDB is a popular NoSQL database known for its flexibility, scalability, and ease of use. It employs a document-oriented data model, making it highly suitable for managing large amounts of diverse and complex data. One of the main reasons MongoDB is favored in modern applications is due to its compatibility with JavaScript Object Notation (JSON) data format, which seamlessly integrates with JavaScript-based technologies like Node.js and Express (MongoDB, n.d., Features). This subsection introduces MongoDB and discusses its suitability for storing and retrieving data in cross-platform mobile application development.

According to Stack Overflow's last year's developer survey (Stack Overflow, 2022), MongoDB was the second most popular database for those learning to code but with a similar percentage for Professional Developers.

MongoDB offers several advantages over popular databases like MySQL and PostgreSQL, making it a viable choice for many applications, especially those requiring flexibility, scalability, and compatibility with JavaScript technologies.

Compared to MySQL, a popular relational database management system (RDBMS), MongoDB adopts a non-relational (or NoSQL) approach to storing data, representing information as a series of JSON-like documents, in contrast to MySQL's table and row format. This flexibility allows the structure of key/value pairs in a MongoDB collection to vary from document to document, unlike MySQL, which requires data matching a predefined schema. (MongoDB, n.d., MongoDB vs MySQL).

In terms of user-friendliness, MongoDB's flexible approach to storing data is particularly suitable for developers who may not be database experts, while MySQL requires an understanding of principles like normalization, referential integrity, and relational database design to exploit its potential fully. MongoDB is designed for easy scalability, allowing data to be distributed across many servers, while MySQL's options for scalability are more limited. (MongoDB, n.d., MongoDB vs MySQL).

For performance, MySQL is generally faster at selecting many records, while MongoDB is significantly faster at inserting or updating many records. This is partly due to MongoDB's support for a specific **insertMany()** API for rapidly inserting data. (MongoDB, n.d., MongoDB vs MySQL).

Compared with PostgreSQL, another popular RDBMS, MongoDB offers a more flexible approach to data storage. MongoDB's document-oriented model allows the storage of JSON objects in binary format (BSON), while PostgreSQL requires a table structure. MongoDB and PostgreSQL offer good performance, but MongoDB is generally faster when handling large amounts of data. Both databases offer scalability, but MongoDB has an advantage in horizontal scalability, enabling data distribution across multiple servers. When it comes to ease of use, MongoDB is generally considered more beginner-friendly than PostgreSQL due to PostgreSQL's steep learning curve. (MongoDB, n.d., MongoDB vs PostgreSQL).

MongoDB Data Example

Below is a sample of data stored in MongoDB for user collection in our cross-platform mobile application:

```
_id: ObjectId('641e0057e194ab09c4cb7aeb')
name: "exampleuser"
email: "user@mail.com"
password: "$2a$10$AkuKWNrF/rb.aSm8AGrfL.iUDaIzt5aiFA6kk56BLgxSfT83LcrUW"
createdAt: 2023-03-24T19:56:07.062+00:00
updatedAt: 2023-03-24T19:56:07.062+00:00
__v: 0
```

*Figure 5 MongoDB user data*

In this example, we have a user with the name "exampleuser" and the email "user@mail.com." The password is securely hashed for user authentication using the bcrypt hashing algorithm. Additional fields, such as createdAt and updatedAt, store the timestamp when the user document was created and last updated.

This example showcases how MongoDB stores data in a document-oriented format, where each document represents a user entity with various fields containing their information. The document's structure allows flexibility in accommodating additional user-specific data as needed.

### 2.5.3 JSON Web Token (JWT)

JSON Web Token (JWT) is a widely used open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs are commonly used to authenticate users and share information between the

client-side and server-side applications. The structure of a JWT typically consists of three parts: header, payload, and signature.

1. **Header:** The header contains information about the type of token (JWT) and the cryptographic algorithm used for signature generation.
2. **Payload:** The payload contains the claims or statements about the user and additional data. It can include information like user ID, roles, permissions, and token expiration time.
3. **Signature:** The signature is a hash value generated using a secret key and the header and payload data. It ensures the integrity of the token and prevents tampering.

The use of JWTs in mobile applications, especially those built using Node.js and Express, offers several advantages:

**Stateless Authentication:**

JWTs allow for stateless authentication, meaning the server does not need to maintain user-session information. Instead, the necessary user data is embedded within the token itself. This approach reduces the load on the server and simplifies the scaling of applications.

**Improved Security:**

Since JWTs are digitally signed, the server can verify the token's authenticity. Any tampering attempts will result in the signature verification failure, ensuring the token's data remains secure.

**Cross-Platform Compatibility:**

JWTs are represented as simple JSON objects, making them compatible with a wide range of platforms, including JavaScript-based technologies like Node.js and Express, which are used in the backend of mobile applications.

**Flexibility and Customization:**

Developers can add custom claims to the payload, allowing them to store application-specific data within the token. (JWT, n.d.).

In Node.js and Express applications, libraries like JSON web token simplify the generation and validation of JWTs. Here is an example of how to generate a JWT using a JSON web token in a Node.js/Express application:

```
const jwt = require('jsonwebtoken');

const secretKey = 'your_secret_key_here';
const user = { id: 123, username: 'example_user' };

const token = jwt.sign(user, secretKey, { expiresIn: '1h' });
console.log('Generated JWT:', token);
```

*Figure 6 Generating JWT using JSON web token.*

In this example, we use the JSON web token library to sign the user object with a secretKey and set an expiration time of one hour (expiresIn: '1h'). The resulting token can be sent to the client-side application, and subsequent requests can include the token in the headers for authentication.

It is essential to handle JWTs securely and avoid storing sensitive information in the payload. Additionally, tokens should be transmitted over secure channels (HTTPS) to prevent interception.

### 2.5.4 Bcrypt

Bcrypt is a widely used password-hashing function designed to store and protect passwords from unauthorized access securely. Passwords are a common target for attackers, and storing them in plaintext is a severe security risk. Bcrypt addresses this issue by employing a computationally expensive hashing algorithm that adds a layer of security to password storage.

The process of hashing a password with Bcrypt involves several steps:

1. **Salt Generation:** Bcrypt generates a random salt, an additional piece of data unique for each password hash. The salt is combined with the password before hashing, ensuring that their hashes will be different even if two users have the same password.

2. **Key Stretching:** Bcrypt applies a key stretching function that repeatedly hashes the salted password. The number of iterations is configurable, and more iterations result in a more secure hash.

3. **Final Hash Generation:** After the specified number of iterations, Bcrypt produces the final hash stored in the database.

Bcrypt offers several advantages for securing passwords in a mobile application backend:

**Protection Against Brute Force Attacks:**

Bcrypt's key stretching and computational intensity make it resistant to brute-force attacks. The time it takes to generate the hash deters attackers from attempting to guess passwords.

**Unique Salts for Each Password:**

With Bcrypt, each password has its unique salt, preventing attackers from using precomputed tables (rainbow tables) to crack multiple passwords simultaneously.

**Configurable Complexity:**

Developers can adjust the number of iterations (work factor) based on the application's security needs. As hardware improves, the work factor can be increased to maintain the same level of security. (bcrypt, n.d.).

In Node.js and Express applications, the bcrypt library is commonly used to hash passwords using Bcrypt. Here is an example of how to hash a password using bcrypt:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;

const plainPassword = 'user123';
bcrypt.hash(plainPassword, saltRounds, (err, hash) => {
  if (err) {
    console.error('Error hashing password:', err);
  } else {
    console.log('Hashed Password:', hash);
  }
});
```

*Figure 7 Hashing password using Bcrypt*

In this example, we use the bcrypt library to hash the plainPassword with a saltRounds value of 10. The resulting hash is the secure password representation that can be stored in the database.

Using Bcrypt to store passwords securely ensures that even in a data breach, attackers will find it extremely challenging to retrieve the original passwords.

## 2.5 Visual Studio Code

Visual Studio Code is a lightweight and extensible source code editor developed by Microsoft that has gained popularity among developers. This subsection highlights the features and advantages of using Visual Studio Code as an integrated development environment (IDE).

VS Code is used with various programming languages, including C, C#, C++, Fortran, Go, Java, JavaScript, Node.js, Python, Rust, Julia, and Dart. It has debugging support, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can customize the editor's theme, keyboard shortcuts, and preferences and install extensions that enhance functionality. It is based on the Electron framework, employed to develop Node.js web applications that run on the Blink layout engine. The editor component of VS Code, codenamed "Monaco," is the same as the one used in Azure DevOps. (Wikipedia, n.d.).

One of the key aspects of VS Code is its ability to provide essential support for most common programming languages. This includes syntax highlighting, bracket matching, code folding, and configurable snippets. However, the support for additional languages and features can be extended by freely available extensions on the VS Code Marketplace. (Wikipedia, n.d.).

According to Stack Overflow's last year's developer survey (Stack Overflow, 2022), VS Code was ranked the most popular developer environment tool.

### 2.5.1 Extensions in Visual Studio Code

Extensions add functionality to VS Code, enabling it to support more programming languages, add new themes, debuggers, and perform static code analysis. Some extensions used in this project include Dart, Flutter, and GitHub Copilot.

**Dart Extension**: The Dart extension for VS Code supports the Dart programming language and tools for effectively editing, refactoring, running, and reloading Flutter mobile apps. It offers features such as automatic hot reloads for Flutter, refactorings, and code fixes, syntax highlighting, code completion, real-time errors/warnings/TODOs, and the ability to switch between devices for Flutter. As of now, it has been installed over 6.8 million times. (Visual Studio Marketplace, n.d.).

**Flutter Extension**: This extension supports effectively editing, refactoring, running, and reloading Flutter mobile apps. It depends on and automatically installs the Dart extension for Dart programming language support. This extension has been installed over 6.2 million times as of today. (Visual Studio Marketplace, n.d.).

**GitHub Copilot**: GitHub Copilot is an AI pair programmer that provides autocomplete-style suggestions as code is written. Suggestions from GitHub Copilot can be received by writing the desired code or by providing a natural language comment describing the intended code functionality. This tool can convert comments into actual code, create unit tests, and generate SQL queries. Powered by OpenAI Codex, GitHub Copilot operates in real-time within the editor, supporting multiple programming languages seamlessly. Its integration into the editor enables continuous usage as code is being typed. As of the current date, this extension has been installed over 300,000 times (Visual Studio Marketplace, n.d.).

## 2.6 Postman

Postman is a widely-used API client that simplifies creating, testing, and documenting APIs. Launched in 2012 by Abhinav Asthana, it provides a user-friendly interface for making HTTP requests and viewing responses. It supports various APIs, including REST, SOAP, and GraphQL. Key features include organizing APIs into collections, automating tests, and generating detailed API documentation. Postman's functionality makes it an essential tool for modern API development and testing. (Postman, 2023)

## 2.7 Git

Git is a distributed version control system created by Linus Torvalds in 2005. It facilitates collaborative work among multiple developers on a project without causing conflicts due to

simultaneous changes. Git monitors alterations made to files in a repository, enabling a comprehensive view of modifications, their authors, and the reasons behind them. Additionally, it offers functionalities like branching and merging, allowing developers to work on separate branches for development and subsequently integrate them back into the main codebase (Git, 2023).

### 2.7.1 GitHub

GitHub, founded in 2008, is a web-based hosting service for Git repositories. It provides a platform for collaboration, allowing developers to contribute to projects, track issues, and manage changes. GitHub extends the functionality of Git with features like pull requests, which facilitate code review and discussion, and GitHub Actions, which automate software workflows. It is widely used in open-source projects and by companies for private repositories. (GitHub 2023)

## 2.8 Selection Criteria for Technologies and Tools

Choosing the right technologies and tools for a cross-platform mobile application requires careful consideration of various factors. In this subsection, we will explain the rationale behind selecting Flutter for the frontend and Node.js, Express framework, and MongoDB for the backend. We will discuss key criteria such as community support, documentation, performance, ecosystem, and compatibility with project requirements.

**Flutter for the frontend**: Flutter is a popular open-source UI toolkit developed by Google. It was chosen for the frontend due to the following reasons:

- **Cross-platform compatibility**: Flutter allows building applications that can run seamlessly on both Android and iOS platforms, reducing development effort and time.
- **Fast development**: Flutter's hot-reload feature enables quick code changes and immediate visual updates, resulting in a shorter development cycle.
- **Rich UI and smooth performance**: Flutter provides a rich set of pre-built widgets and a customizable UI, enabling developers to create visually appealing and responsive user interfaces.

- **Strong community support**: Flutter has a large and active community of developers, which ensures continuous improvement, frequent updates, and extensive support through online forums, tutorials, and packages.
- **Documentation**: Flutter has comprehensive and up-to-date documentation, making it easier for developers to learn and utilize its features effectively.

**Node.js and Express framework for the backend**: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, while express is a popular web application framework for Node.js. The choice of Node.js and Express framework for the backend was based on the following considerations:

- **JavaScript ecosystem**: Since Flutter uses Dart programming language, selecting Node.js allows developers to leverage their existing JavaScript skills and maintain a consistent tech stack.
- **Performance and scalability**: Node.js is known for its event-driven, non-blocking I/O model, which provides high performance and scalability, making it suitable for handling concurrent requests in real-time applications.
- **Large package ecosystem**: Node.js has a vast ecosystem of packages and modules available through npm (Node Package Manager), simplifying backend development by providing ready-to-use solutions for various functionalities.
- **Express framework**: Express is a lightweight and flexible framework that simplifies the development of robust APIs and web applications, offering features like routing, middleware support, and a modular structure.

**MongoDB as the database**: MongoDB is a popular NoSQL database known for its flexibility and scalability. It was chosen for the project based on the following factors:

- **Flexible document structure**: MongoDB's document-based data model allows for easy storage and retrieval of complex data structures, making it suitable for flexibly storing various data types.
- **Scalability**: MongoDB's distributed architecture and automatic sharding capabilities enable seamless scaling as the application's data grows.
- **Compatibility with Node.js**: MongoDB has an official Node.js driver and offers excellent integration with Node.js applications, making it a natural choice for the backend technology stack.

**Visual Studio Code as the IDE**: Visual Studio Code (VS Code) is a lightweight and feature-rich code editor developed by Microsoft. It was chosen as the IDE for development due to the following reasons:

- **Extensive ecosystem**: VS Code has vast extensions and plugins that enhance productivity, support various programming languages, and provide integrations with popular tools and frameworks.
- **Intuitive user interface**: VS Code offers a clean and intuitive interface with features like code highlighting, IntelliSense, debugging capabilities, and built-in version control.
- **Active community**: VS Code has a large and active community that contributes to continuous improvement, provides support, and develops valuable extensions.
- **Flutter and Dart support**: Visual Studio Code provides excellent support for Flutter and Dart development. It offers various features and extensions designed explicitly for Flutter development.

By considering these selection criteria, we ensure that Flutter, Dart, Node.js, Express framework, MongoDB, and Visual Studio Code align with the project's goals, offer the necessary features and support, and enable efficient and seamless development of the cross-platform mobile application.

# 3 Development of the mobile application

## 3.1 System Architecture and Design

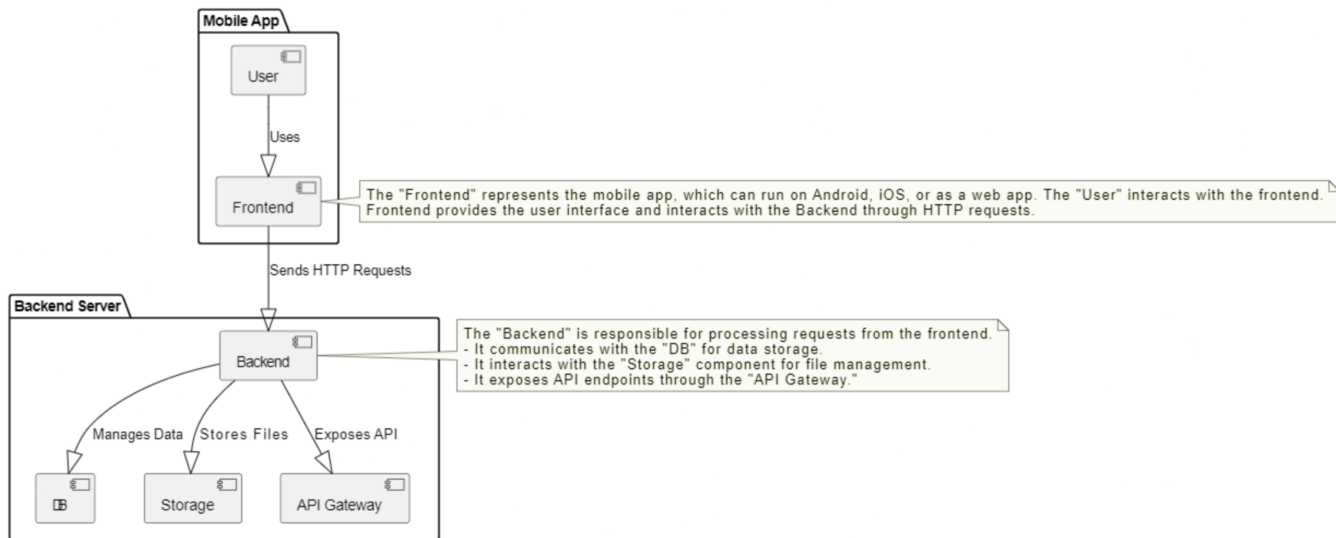


*Figure 8 Architectural structure of the application*

The "Frontend" represents the mobile app, which can run on Android, iOS, or as a web app. The "User" interacts with the frontend, which provides the user interface for the application. Users can view and manage notes, contacts, and other app features through the frontend. Frontend communicates with the "Backend" to perform various tasks and retrieve data by sending HTTP request.

The "Backend" serves as the core of the application. It processes requests sent by the frontend, handling actions such as creating, reading, updating, and deleting data. It is divided into multiple components:

DB (Database): This component manages data storage, including user profiles, notes, contacts, projects, and more.

Storage: Used for file storage, such as images, audio recordings, and other attachments.

API Gateway: The API Gateway component exposes API endpoints that allow the frontend to interact with the backend. It plays a crucial role in facilitating communication between the frontend and backend, enabling various app functionalities.

## 3.2 User interface design and Implementation

**Login screen:**

This is where the application starts if the user is not logged in. The login screen lets existing users sign in using their registered email and password. Once authenticated, users can access their saved notes.

**Register screen:**

The register screen allows new users to create an account by providing their name, email, and password. Upon successful registration, users gain access to the app's features.

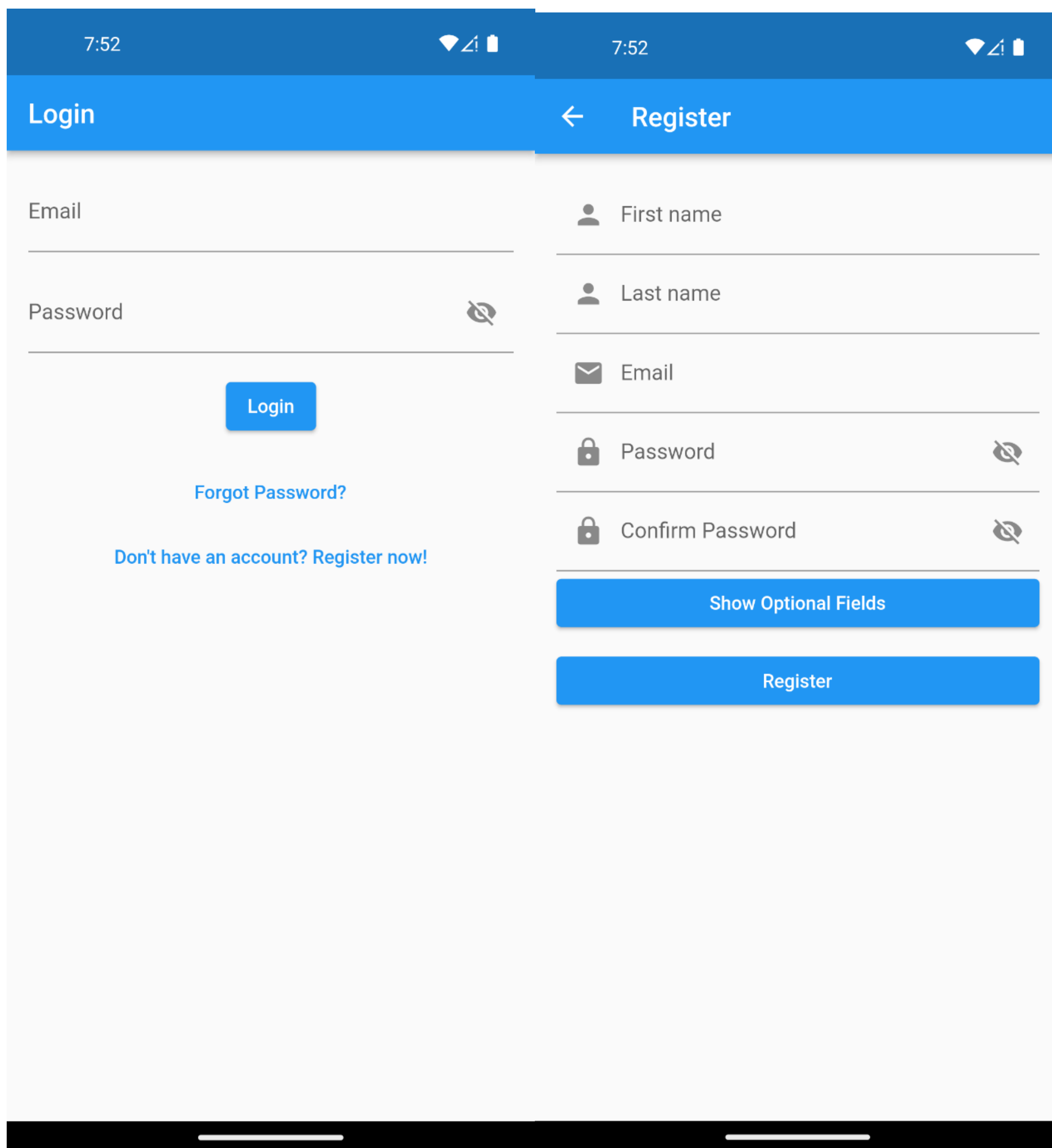


*Figure 9 Login screen*

*Figure 10 Register screen*

**All Notes Screen:**

The All Notes screen in grid view displays a visually appealing arrangement of users' notes in a grid format. A card represents each note, showcasing its title and a preview of its content.

The All Notes screen in the list view presents users' notes in a vertical list format. Each note entry displays its title and a brief snippet of the note's content, facilitating easy scrolling and navigation.

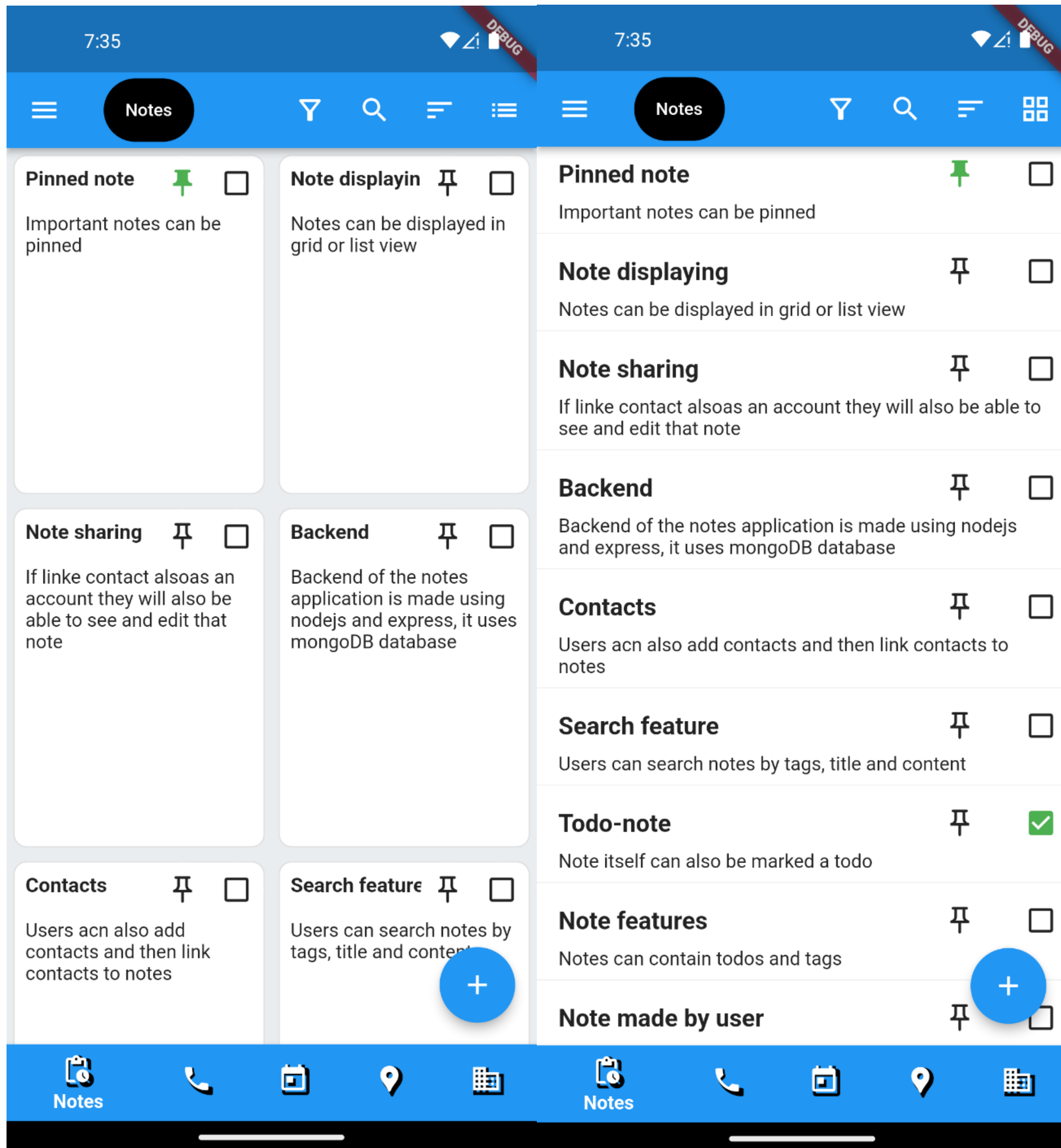


*Figure 11 All notes (grid view)*

*Figure 12 All notes (list view)*

**Note View Screen:**

The Note View screen provides a detailed view of a selected note. Users can read the note's full content, along with additional information such as todos and tags. Users can also edit the note in this view. Adding a new note utilizes this same screen.

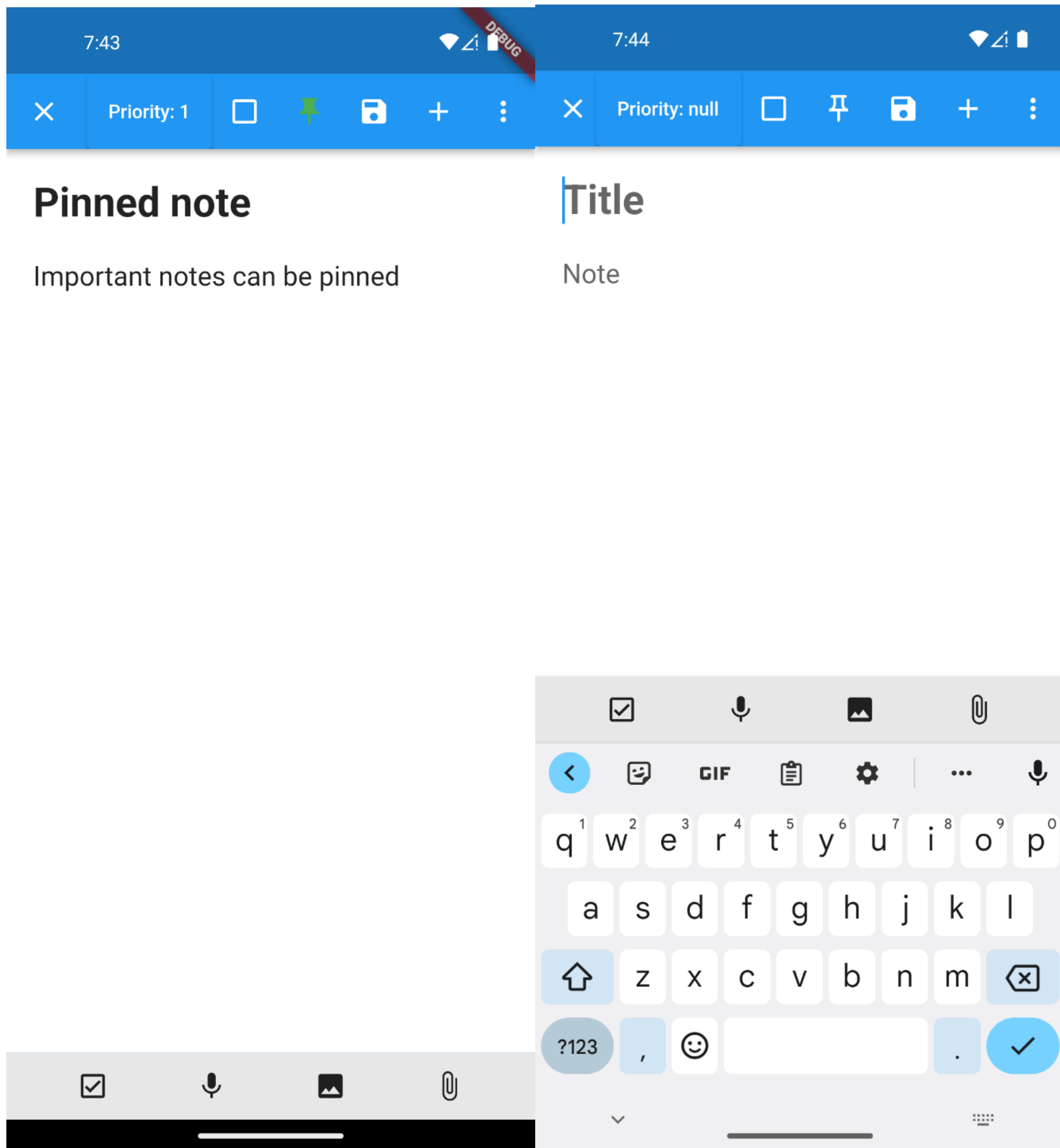


*Figure 13 Notes screen*

*Figure 14 Adding a new note*

**Adding tags and todos to note:**

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Users can also assign relevant tags to a specific note in the notes view screen. Tags help categorize and organize notes, making it easier to find related content quickly.

It also allows users to create a checklist of todo items within a note. Users can add, edit, or delete individual tasks, keeping track of their progress.

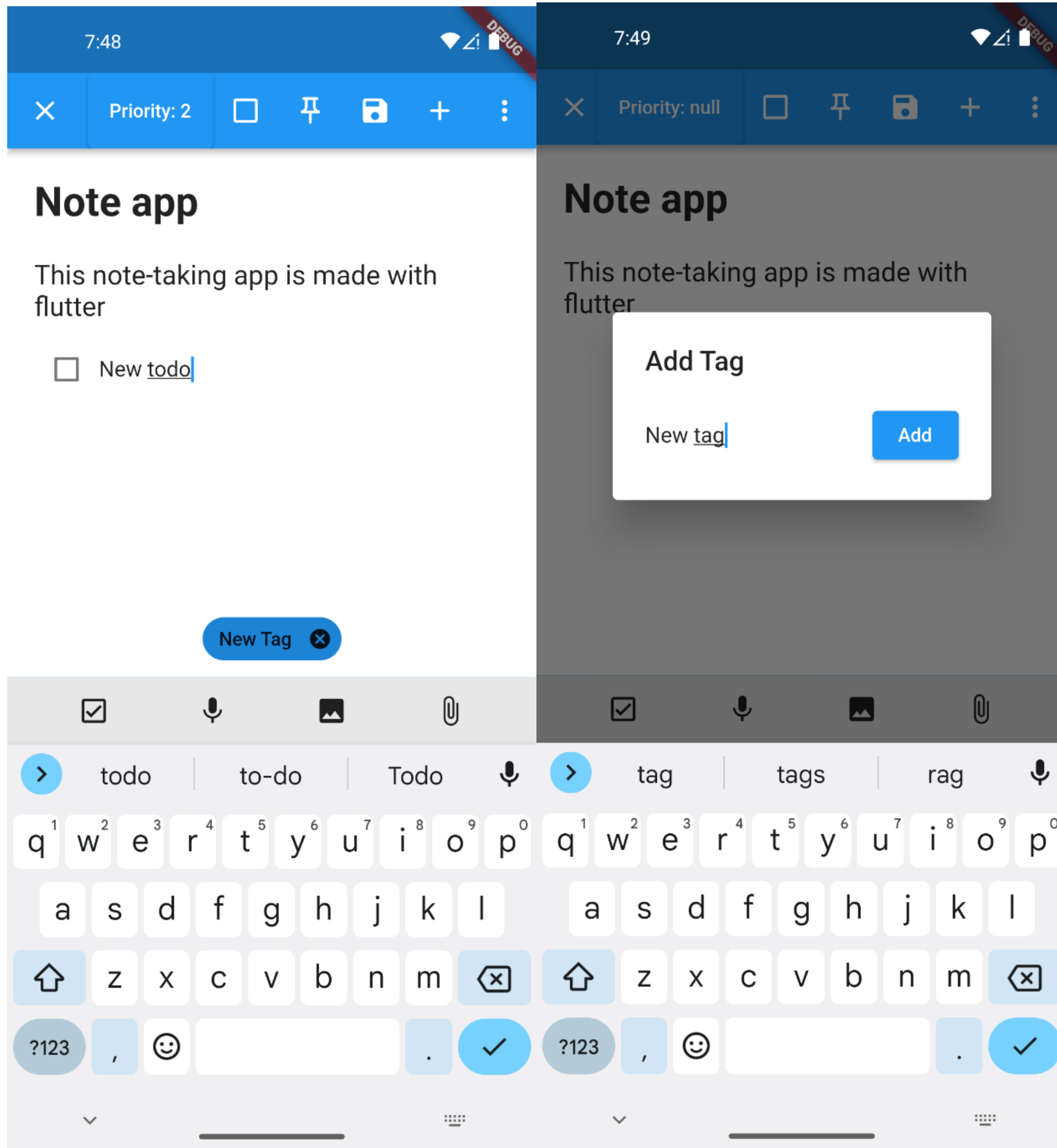


*Figure 15 Adding a todo*

*Figure 16 Adding a tag*

**Adding contacts:**

The "Add Contact" screen allows users to add contacts, which can then be later linked to notes if the user enters another registered user's email address and then links that user to a note; that note will also be shared with that user. This feature fosters collaboration and sharing of notes among users within the app. Users can also choose "Import from phone" and choose a contact from the phone's contact book.

The "Contacts" screen displays a list of the user's contacts. Users can view and manage their contacts' information, facilitating easy collaboration and sharing of notes.

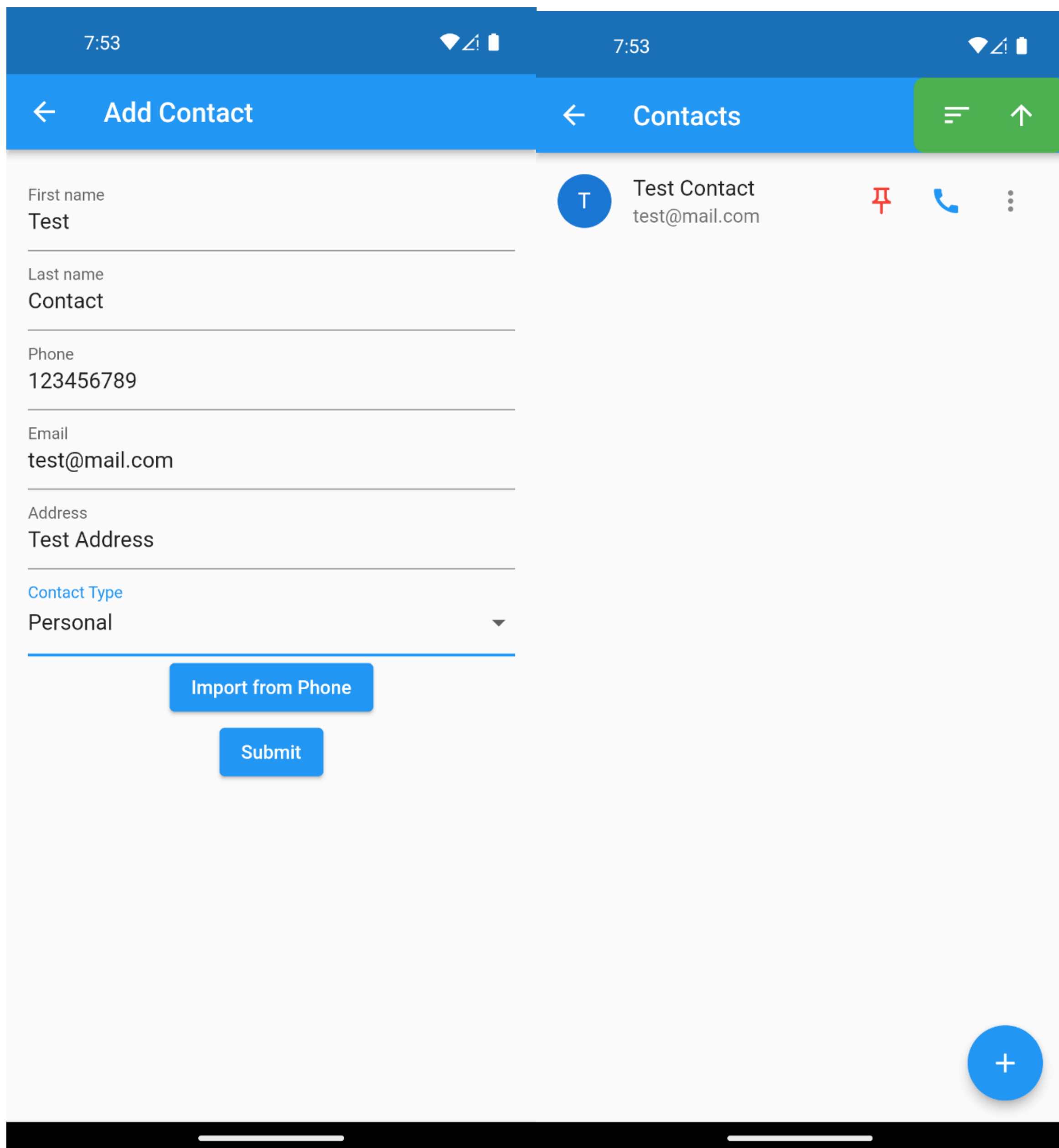


*Figure 17 Adding a contact*

*Figure 18 All contacts screen*

**Linking a contact to note:**

The "Adding Contact to Note" screen enables users to associate a specific contact with a particular note. Once added the associated contact gains access to the note, allowing collaboration and shared viewing.

The "Note with Contact" screen showcases a note associated with a specific contact. When a contact is linked to a note, they gain access to the note's content, promoting collaborative work and shared information.

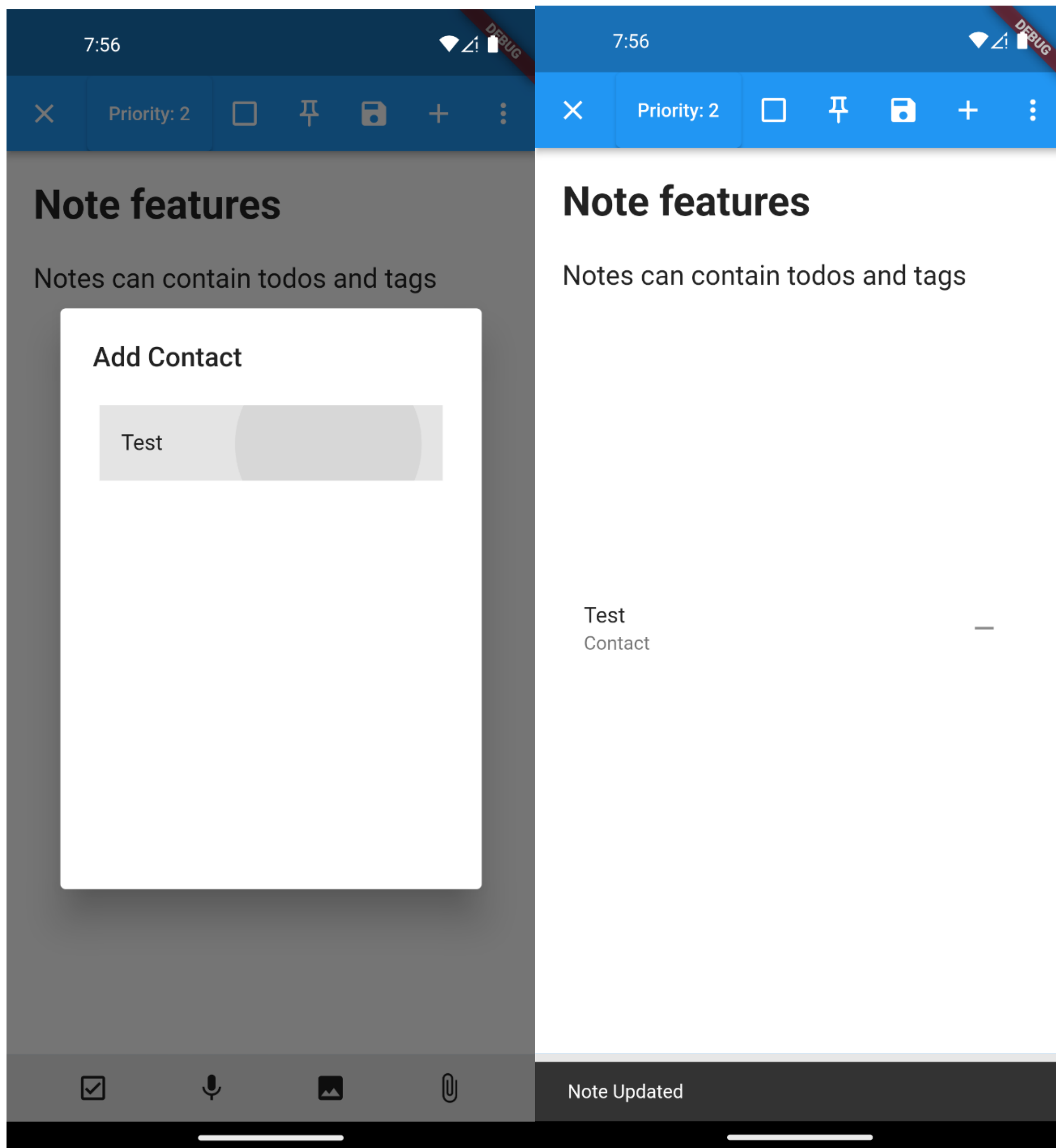


*Figure 19 Adding a contact to note*

*Figure 20 Note with contact*

**Adding tags:**

In the "Add Tags" screen, users can create and assign tags to categorize their notes effectively. Tags help users organize their notes based on specific themes or topics, making it easier to locate them later. Users can also add tags on the notes view screen; however, here, users can see a list of all the tags and have the option to remove unnecessary tags.

**Deleting notes:**

The "Deleting Note" screen confirms the user's decision to delete a specific note from their collection. Users are prompted to confirm the deletion to avoid the accidental removal of important content.

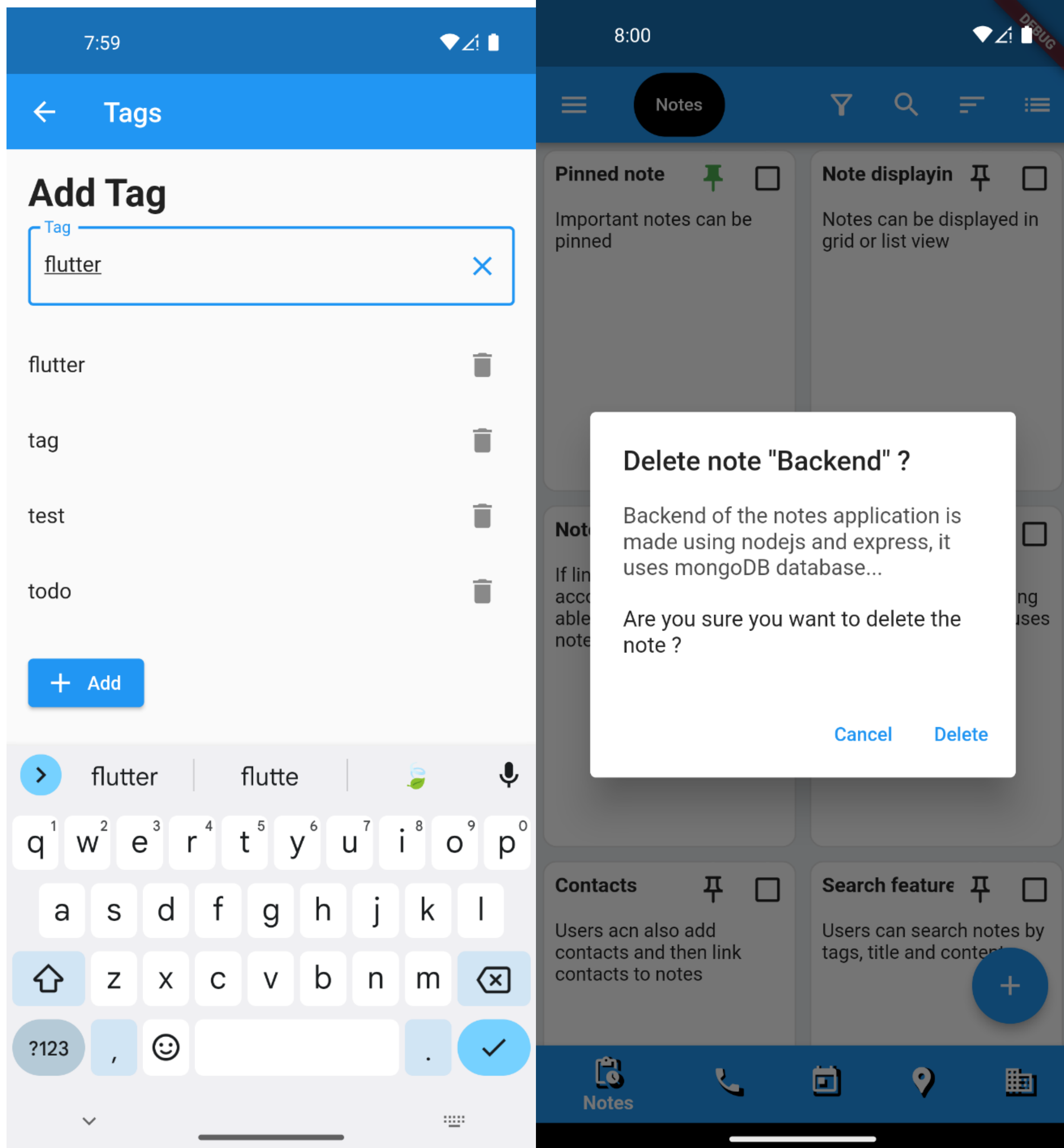


*Figure 21 All tags screen*

*Figure 22 Deleting a note*

**Search function:**

The search function allows users to find specific notes quickly by entering relevant keywords. The search feature enhances note retrieval and promotes efficient organization of notes.

**Sidebar:**

The sidebar presents a convenient navigation menu accessible by sliding from the side of the screen. It provides quick access to essential app features, such as viewing all notes, adding tags, managing contacts, and accessing settings (not yet implemented).

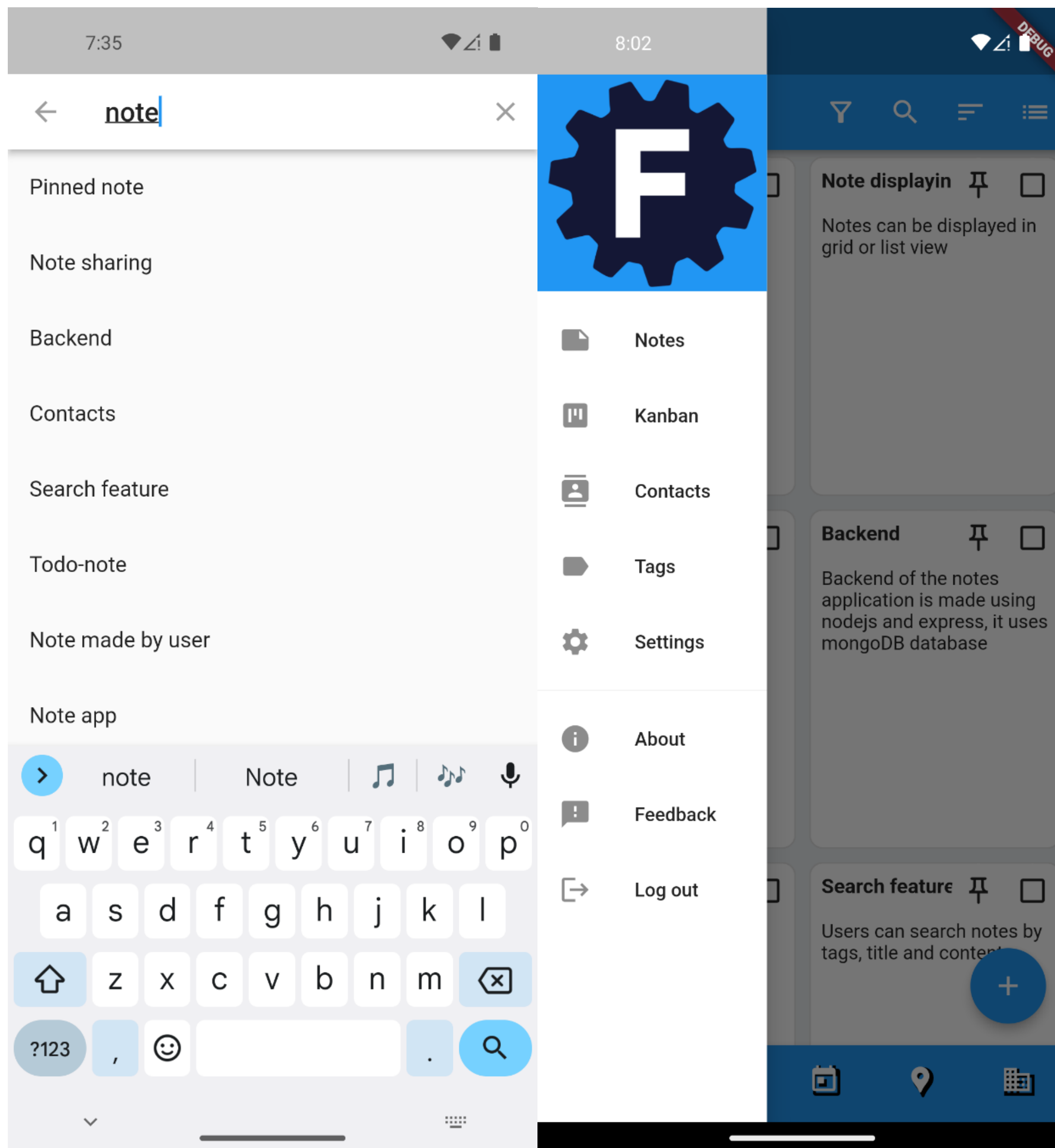


*Figure 23 Search function*

*Figure 24 Sidebar*

## 3.3 Backend development with Node.js and Express

In this section, we explore the implementation of the backend server for our cross-platform mobile application using Node.js and Express. The backend handles data storage, processing API requests, and managing user authentication.

Below is the main server file (server.js) that initializes our backend:

```
1   const express = require('express');
2   const cors = require('cors');
3   const colors = require('colors');
4   const dotenv = require('dotenv').config();
5   const { errorHandler } = require('./middleware/errorMiddleware');
6   const connectDB = require('./config/db');
7   const port = process.env.PORT || 3000;
8
9   connectDB();
10
11  const app = express();
12
13  app.use(cors({
14      origin: '*', // allow to server to accept request from different origin
15  }));
16  app.use(express.json());
17  app.use(express.urlencoded({ extended: false }));
18
19  app.use('/api/notes', require('./routes/noteRoutes'));
20  app.use('/api/users', require('./routes/userRoutes'));
21  app.use('/api/tags', require('./routes/tagRoutes'));
22  app.use('/api/todos', require('./routes/todoRoutes'));
23  app.use('/api/contacts', require('./routes/contactRoutes'));
24
25  app.use(errorHandler);
26
27  app.listen(port, () => console.log(`Server is running on port ${port}`));
28
```

*Figure 25 Server.js*

In the main server.js file, we initialize the backend with the following steps:

**Connect to MongoDB:** We establish a connection to the MongoDB database using the connectDB() function.

**Create the Express App:** We create an Express application and enable Cross-Origin Resource Sharing (CORS) to allow requests from different origins.

**Parse Incoming Data:** We parse incoming JSON and URL-encoded data to handle requests effectively.

**Define Routes:** We define separate route endpoints for notes, users, tags, todos, and contacts, making the code more organized and modular.

**Error Handling:** We implement custom error handling middleware to ensure consistent error management across the application.

**Start the Server:** The backend server listens on the specified port (defaulting to port 3000) using app.listen().

With this configuration, our backend is fully equipped to handle API requests, manage data, and provide a reliable foundation for our cross-platform mobile application.

Following good practices in software development, we have structured our backend codebase with separate folders and files to enhance readability, maintainability, and modularity.

**Routes:** We have organized our API routes into separate files within the routes folder. This approach allows us to manage endpoints for different resources independently, making the codebase more organized and easy to navigate.

Here is an example how a route file looks like; This is for notes, the endpoint for getting all notes, posting a note, and editing or deleting a note with id:

```
backend > routes > noteRoutes.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const { getNotes, postNote, updateNote, deleteNote } = require('../controllers/noteController');
4  const { protect } = require('../middleware/authMiddleware');
5
6  router.route('/').get(protect, getNotes).post(protect, postNote);
7  router.route('/:id').put(protect, updateNote).delete(protect, deleteNote);
8
9  module.exports = router;
```

*Figure 26 Noteroutes*

**Models:** In the models folder, we define Mongoose schemas that represent the data structure of our MongoDB collections. The use of models provides a clear abstraction of data and allows us to interact with the database consistently and structured.

Here is an example of what a model looks like; this is for users; the user has a name, email, and password and can have contacts:

```
backend > models > JS userModel.js > ...
1   const mongoose = require('mongoose');
2
3   const userSchema = mongoose.Schema({
4       name: {
5           type: String,
6           required: [true, 'Please add a name']
7       },
8       email: {
9           type: String,
10          required: [true, 'Please add an email'],
11          unique: true,
12      },
13      password: {
14          type: String,
15          required: [true, 'Please add a password'],
16      },
17      contacts: [
18          {
19              type: mongoose.Schema.Types.ObjectId,
20              ref: 'Contact'
21          }
22      ],
23  },
24      {
25          timestamps: true
26      }
27  )
28
29  module.exports = mongoose.model('User', userSchema);
30
```

*Figure 27 User model*

**Controllers:** Our route handlers are placed in the controllers folder, keeping the logic for each route separate. This separation of concerns enhances code readability and makes debugging and testing individual components easier.

Here is an example how controller looks like; This is for users and is responsible for user-related operations such as registration, login, and fetching user data. 'registerUser' function is responsible for handling the registration of a new user in the backend of our cross-platform mobile application. The function is triggered when an HTTP POST request is made to the /api/users endpoint, and it is accessible to the public, meaning anyone can use it to register as a new user.

The code follows these steps:

**Data Validation:** It extracts the name, email, and password from the request body. Then, it checks if all the required fields (name, email, and password) are provided. If any fields are missing, the server responds with a status code of 400 (Bad Request) and throws an error message indicating that all fields should be provided.

**Check for Existing User:** The code queries the database to check if a user with the provided email already exists. If a user with the same email is found, the server responds with a status code of 400 and throws an error message indicating that the user already exists.

**Password Hashing:** If the provided email is unique and the user is new, the code generates a random salt value using bcrypt.genSalt() with a cost factor of 10. This salt is then used to hash the password securely using bcrypt.hash(). The hashed password is stored in the database, protecting the original password.

**Create User Record:** After hashing the password, the code creates a new user record in the database using the user.create() method from the User model. The user's name, email, and hashed password are saved in the database.

**Response:** If the user is successfully created, the server responds with a status code of 201 (Created) and sends a JSON object containing the newly created user's _id, name, email, and a JSON Web Token (JWT) generated by the generateToken() function. This JWT can be used for future authentication.

**Error Handling:** If errors occur during the registration process, such as database errors or missing fields, the code catches them using the asyncHandler middleware and sends an appropriate error response with a status code of 400 (Bad Request) and an error message.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

backend > controllers > userController.js > <unknown>

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const asyncHandler = require('express-async-handler');
const User = require('../models/userModel');


// @desc Register a new user
// @route POST /api/users
// @access Public
const registerUser = asyncHandler(async (req, res) => {
    const { name, email, password } = req.body

    if(!name || !email || !password) {
        res.status(400);
        throw new Error('Please add all fields')
    }

    //check if user exists
    const userExists = await User.findOne({email})
    if(userExists) {
        res.status(400);
        throw new Error('User already exists')
    }

    // Hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Create user
    const user = await User.create({
        name,
        email,
        password: hashedPassword
    })

    if(user) {
        res.status(201).json({
            _id: user.id,
            name: user.name,
            email: user.email,
            token: generateToken(user._id)
        })
    } else {
        res.status(400);
        throw new Error('Invalid user data')
    }
})
```

*Figure 28 User controller*

**Middleware:** We have implemented custom middleware functions in the middleware folder to handle error management and authentication tasks. Middleware ensures that common functionalities are applied across the application without duplicating code.

Error Handling Middleware (errorHandler): This middleware function, defined in the error-middleware.js file, is crucial in managing errors throughout the application. When an error occurs, it captures the error message and sets an appropriate status code based on the severity of the error. If the application is in production mode, it omits the detailed error stack trace to prevent the exposure of sensitive information. This ensures that all errors are consistently handled and reported to the client in a user-friendly manner.

Authentication Middleware (protect): In the authmiddleware.js file, we have implemented a custom middleware function called protect to handle user authentication. This middleware protects routes that require a user to be authenticated. It checks for a JSON Web Token (JWT) in the Authorization header of the incoming request. If a valid JWT is found, the middleware decodes the token to extract the user's ID. Using this ID, it fetches the corresponding user data from the database, excluding sensitive information like the password. This user data is then attached to the request object as req. user, allowing subsequent route handlers to access the authenticated user's details.

The protect middleware enables secure access to specific routes by ensuring that only authorized users can proceed, while unauthorized users are denied access with an appropriate status code and error message.

By implementing these middleware functions, we achieve modularity and reusability across our backend codebase, streamlining common functionalities and enhancing the overall robustness of our cross-platform mobile application.

**Config:** The config folder contains configuration files, such as the connection to the MongoDB database using Mongoose and environment variables setup with dotenv. Centralizing configuration settings improves maintainability and allows for easy changes in the future.

By adopting this organized folder structure and adhering to best practices, we aim to build a scalable, maintainable backend that follows industry-standard coding conventions. This approach streamlines the development process and ensures that our cross-platform mobile application is built on a solid foundation.

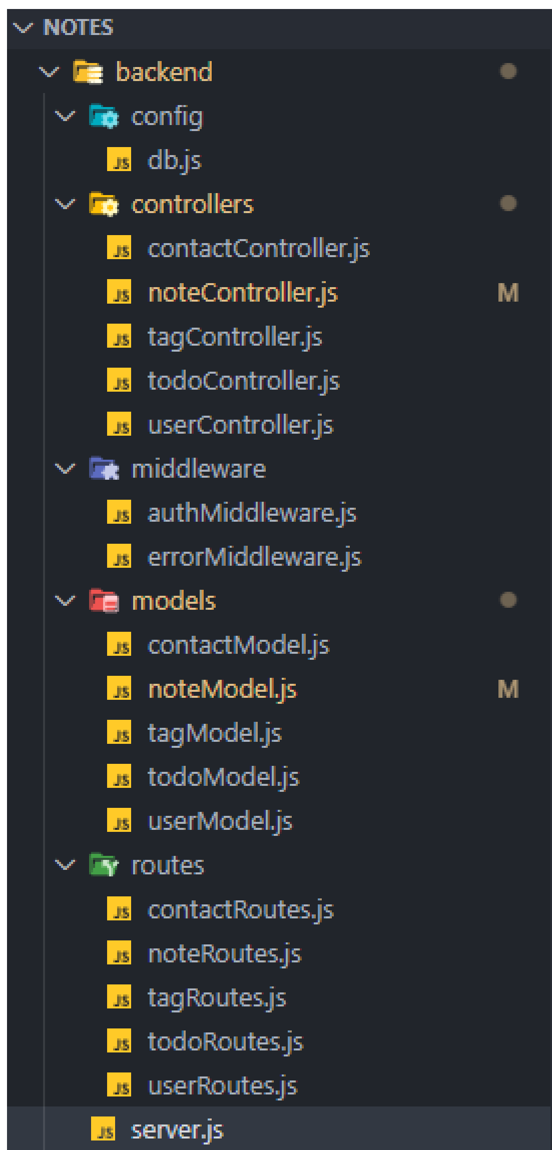


*Figure 29 Backend file structure*

The figure above illustrates the backend structure, demonstrating the organization of the main server file, routes, models, controllers, middleware, and configuration files. This structured backend architecture forms the backbone of our cross-platform mobile application, ensuring its efficiency, scalability, and reliability in handling user data and requests.

**Note sharing:**

The mobile application includes a note-sharing feature allowing users to collaborate and share notes. Users can add contacts by specifying their email addresses when they want to share a note. The backend handles the note-sharing process through the following steps:

1. Get All Notes:

To retrieve all notes associated with a specific user, the backend uses the following route and handler:

```
// @desc    Get all notes
// @route   GET /api/notes
// @access  Private
const getNotes = asyncHandler(async (req, res) => {
    const notes = await Note.find({ user: req.user.id });
    // added "sharedto" into note model so that we can get notes that are shared to the user
    const sharedNotes = await Note.find({ sharedto: req.user.id });
    notes.push(...sharedNotes);
    res.status(200).json(notes);
});
```

*Figure 30 getNotes*

In this code snippet, the getNotes function fetches all notes belonging to the authenticated user. Additionally, it retrieves notes that have been shared with the user. To facilitate note sharing, we have added a sharedto field in the note model to store the array of user IDs (emails) with whom the note has been shared.

2. Update Note for Sharing:

When a user updates a note and adds contacts to share it, the following logic is implemented in the updateNote function:

```
126    // Sharedto is an array of user emails
127    // We need to find the user ids of the users with the emails in sharedto
128    // Then we add the user ids to the sharedto field of the note
129    const sharedtoUserIds = [];
130    if (!sharedto) {
131        sharedto = [];
132    }
133    for (const email of sharedto) {
134        const user = await User.findOne({ email });
135        if(!user) {
136            res.status(404);
137            throw new Error(`User with email ${email} not found`);
138        }
139        sharedtoUserIds.push(user._id);
140    }
141
142    note.sharedto = sharedtoUserIds;
143
```

*Figure 31 updateNote*

In this code, we iterate through the provided sharedto array (which contains the email addresses of users to share the note with) and find the corresponding user IDs. An error is thrown if a user with a given email is not found. Otherwise, we collect the user IDs and assign them to the sharedto field of the note. This ensures that the note is correctly shared with the specified users.

By implementing note sharing in the backend, the application offers users a collaborative and efficient environment for sharing their notes seamlessly. This feature enhances user interaction and productivity, making the cross-platform mobile application a versatile tool for team collaboration and personal use.

**API**

The backend provides a REST API with various endpoints for user registration, authentication, note management, tag management, todo management, and contact management. Below is a table summarizing the available API endpoints, their descriptions, required parameters, and request types.

| Endpoint | Description | Parameters | Request Type |
|---|---|---|---|
| /api/users | Create new user | Name, email, password | POST |
| /api/users/login | Authorize user | Email, password | POST |
| /api/users/me | Get user data | None | GET |
| /api/notes/:id | Get all notes by user | id | GET |
| /api/notes | Create new note | Title, content, tags | POST |
| /api/notes/:id | Update note | Title, content, tags | PUT |
| /api/notes/:id | Delete note | Id | DELETE |
| /api/tags | Get all tags by user | Id | GET |
| /api/tags | Create new tag | Name, notes | POST |
| /api/tags/:id | Update tag | Name, notes | PUT |
| /api/tags/:id | Delete tag | Id | DELETE |

| /api/todos | Get all todos by user | Id | GET |
|---|---|---|---|
| /api/todos/:id | Get all todos by note | Id | GET |
| /api/todos | Create new todo | Name, notes | POST |
| /api/todos/:id | Update todo | Name, notes | PUT |
| /api/todos/:id | Delete todo | Id | DELETE |
| /api/contacts | Get all contacts by user | Id | GET |
| /api/contacts | Create new contact | Name, notes | POST |
| /api/contacts/:id | Update contact | Name, notes | PUT |
| /api/contacts/:id | Delete contact | id | DELETE |

## 3.4 Frontend development using Flutter

This section explores the frontend implementation for our cross-platform mobile application using Flutter. Below, we provide an overview of the frontend's main components and features, highlighting its development's key aspects.

Frontend File Structure contains the following folders:

**Models Folder:** Contains model classes (contact.dart, note.dart, tag.dart, todo.dart, user.dart) representing various entities in the application. These models help structure data retrieved from the backend API.

Here is an example of what the user model looks like:

```
class User {
  String name;
  String email;
  String password;
  String? id;
  String? token;

  User({
    required this.name,
    required this.email,
    required this.password,
    this.id,
    this.token,
  });

  factory User.fromJson(Map<String, dynamic> json) => User(
        name: json["name"],
        email: json["email"],
        password: json["password"],
      );

  Map<String, dynamic> toJson() => {
        "name": name,
        "email": email,
        "password": password,
      };
}
```

*Figure 32 User model*

The user model contains five properties: name, email, password, id, and token.

The class has a constructor that takes name, email, password, id, and token as parameters. When creating a new User object, you provide values for the required properties name, email, and password and optionally include values for the id and token.

Additionally, the class contains two methods:

1. factory User.fromJson(Map<String, dynamic> json):

   This factory constructor takes a JSON map as input and returns a new User object. It converts JSON data received from an API response into a User object. The JSON map is parsed, and its values are assigned to the corresponding properties of the User object.

2. Map<String, dynamic> toJson():

   This method converts the User object into a JSON map. It is used when sending data to an API in the request body. The method creates a JSON map with the name, email, and password properties and their respective values.

Using this User model class, you can easily represent user data in your Flutter application and efficiently convert it to and from JSON format for API interactions. This enhances code organization and simplifies working with user data throughout your application.

**Providers Folder:** Contains provider classes (auth_provider.dart, contact_provider.dart, notes_provider.dart, tag_provider.dart, todos_provider. Dart) that are responsible for managing state and data flow within the application. Each provider class handles state related to a specific entity and ensures seamless data propagation to UI components.

Here is an example of what auth provider looks like; it is a provider that manages user authentication state and interactions with the backend API for user login and registration.

Properties:

- _user: Represents the authenticated user and contains user-related information such as name, email, and password.
- apiService: An instance of the ApiService class that provides methods to interact with the backend API.
- _token: Contains the authentication token obtained after a successful login or registration.

Methods:

- login: Takes the user's email and password as parameters and attempts to log in the user by making an API request to the backend. If successful, it stores the authentication token in the _token property and notifies listeners of the authentication status change.
- register: Takes the user's email, password, and name as parameters and attempts to register the user by making an API request to the backend. If successful, it stores the user information and authentication token in the _user and _token properties, respectively, and notifies listeners of the authentication status change.

- _storeToken: Private method that stores the authentication token in the device's SharedPreferences for a persistent login state.
- initAuthProvider: Initializes the AuthProvider by retrieving the authentication token from SharedPreferences (if available) and setting it in the _token property. If a token is found, it sets it in the ApiService and notifies listeners of the authentication status change.

The AuthProvider class plays a critical role in managing user authentication and maintaining the user's login state throughout the application. It facilitates a smooth user experience by enabling users to log in, register, and remain authenticated while navigating various screens in the app. (provider, n.d.).

```
auth_provider.dart ×
frontend > lib > providers > auth_provider.dart > ...
1  import 'package:flutter/foundation.dart';
2  import 'package:frontend/models/user.dart';
3  import 'package:frontend/services/api_service.dart';
4  import 'package:shared_preferences/shared_preferences.dart';
5
6  class AuthProvider with ChangeNotifier {
7    late User _user;
8    final ApiService apiService;
9    String? _token;
10   String? get token => _token;
11
12   AuthProvider({required this.apiService});
13
14   User get user => _user;
15
16   bool get isAuthenticated => _token != null;
17
18   Future<String> login(String email, String password) async {
19     try {
20       final response = await ApiService.login(email, password);
21       if (response.containsKey('token')) {
22         final token = response['token'];
23         _storeToken(token); // Store the token in SharedPreferences
24         ApiService.setToken(token); // Set the token in the ApiService
25         notifyListeners();
26         return token;
27       } else {
28         throw Exception('Token not found in response');
29       }
30     } catch (error) {
31       rethrow;
32     }
33   }
34
```

*Figure 33 auth provider*

**Screens Folder:** Contains individual screen files (add_contact. Dart, add_note. Dart, add_tag.dart, contacts.dart, edit_contact.dart, login.dart, notes_homepage.dart, register. Dart) representing distinct user interface screens. These screens are constructed using Flutter widgets to create interactive UI components for different functionalities.

Here is an example of a login.dart file, which is the login screen. The LoginScreen class represents the user interface screen where users can log in to the application. It utilizes Flutter's StatefulWidget to manage its internal state and incorporates various Flutter widgets to construct an interactive login form.

Properties:

- _formKey: A GlobalKey identifies and manages the login form.
- _emailController: A TextEditingController that allows capturing and handling user input for the email field.
- _passwordController: A TextEditingController for capturing and handling user input for the password field.

Methods:

- _submit: A method triggered when the login form is submitted. It validates the form inputs, calls the login method from the AuthProvider to initiate the login process, and navigates to the NotesHomePage upon successful authentication. An error dialog is displayed to the user if any errors occur during the login process.

Widget Tree:

The LoginScreen widget tree consists of a Scaffold widget that provides the app's basic structure. The AppBar widget displays the title "Login" at the top of the screen, and the body of the screen contains a Form widget that holds the login form components.

The login form contains two TextFormField widgets for entering the email and password. The TextFormField widgets handle user input validation and display appropriate error messages if the input is invalid.

The "Login" ElevatedButton calls the _submit method when pressed, initiating the login process. Below the login button is a TextButton that navigates users to the RegisterScreen if they do not have an account yet.

```
import 'package:flutter/material.dart';
import 'package:frontend/providers/auth_provider.dart';
import 'package:provider/provider.dart';
import 'package:frontend/screens/register.dart';
import 'package:frontend/screens/notes_homepage.dart';

class LoginScreen extends StatefulWidget {
  static const routeName = '/login';

  const LoginScreen({Key? key}) : super(key: key);

  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final _formKey = GlobalKey<FormState>();
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();

  Future<void> _submit() async {
    if (_formKey.currentState!.validate()) {
      try {
        await Provider.of<AuthProvider>(context, listen: false)
            .login(
          _emailController.text.trim(),
          _passwordController.text.trim(),
        )
            .then((_) {
          Navigator.pushReplacement(
            context,
            MaterialPageRoute(builder: (context) => const NotesHomePage()),
          );
        });
      } catch (error) {
        showDialog(
          context: context,
          builder: (ctx) => AlertDialog(
            title: const Text('An error occurred!'),
            content: Text(error.toString()),
            actions: <Widget>[
              TextButton(
                child: const Text('Okay'),
                onPressed: () {
                  Navigator.of(ctx).pop();
                },
              ) // TextButton
            ], // <Widget>[]
          ), // AlertDialog
```

*Figure 34 login screen code part1*

```
55    @override
56    Widget build(BuildContext context) {
57      return Scaffold(
58        appBar: AppBar(
59          title: const Text('Login'),
60        ), // AppBar
61        body: Padding(
62          padding: const EdgeInsets.all(16),
63          child: Form(
64            key: _formKey,
65            child: Column(
66              children: [
67                TextFormField(
68                  controller: _emailController,
69                  keyboardType: TextInputType.emailAddress,
70                  decoration: const InputDecoration(
71                    labelText: 'Email',
72                  ), // InputDecoration
73                  validator: (value) {
74                    if (value == null || value.isEmpty) {
75                      return 'Please enter your email';
76                    }
77                    if (!value.contains('@')) {
78                      return 'Please enter a valid email address';
79                    }
80                    return null;
81                  },
82                ), // TextFormField
83                const SizedBox(height: 16),
84                TextFormField(
85                  controller: _passwordController,
86                  obscureText: true,
87                  decoration: const InputDecoration(
88                    labelText: 'Password',
89                  ), // InputDecoration
90                  validator: (value) {
91                    if (value == null || value.isEmpty) {
92                      return 'Please enter your password';
93                    }
94                    return null;
95                  },
96                ), // TextFormField
97                const SizedBox(height: 16),
98                ElevatedButton(
99                  onPressed: _submit,
100                 child: const Text('Login'),
101               ), // ElevatedButton
102               const SizedBox(height: 16),
103               TextButton(
104                 onPressed: () {
105                   Navigator.of(context).pushNamed(RegisterScreen.routeName);
106                 },
107                 child: const Text("Don't have an account? Register now!"),
108               ), // TextButton
109             ],
110           ), // Column
111         ), // Form
112       ), // Padding
113     ); // Scaffold
```

*Figure 35 login screen code part2*

**Services Folder:** Contains api_service.dart, a class responsible for managing API requests and facilitating data communication with the backend. It uses the http library to interact with backend API endpoints, fetching and sending data in JSON format.

The ApiService class handles communication with the backend server and manages API requests related to user authentication, notes, tags, todos, and contacts in the cross-platform mobile application. It utilizes the http package to make HTTP requests and interacts with the backend using RESTful API endpoints.

Properties:

- _baseUrl: The base URL of the backend server, where all API endpoints are located.
- _loginUrl: The URL for the login endpoint to authenticate users.
- _registerUrl: The URL for the user registration endpoint.
- _tagsUrl: The URL for the tags endpoint to manage tags.
- _contactsUrl: The URL for the contacts endpoint to manage contacts.
- _notesUrl: The URL for the notes endpoint to manage notes.
- _todosUrl: The URL for the todos endpoint to manage todos.
- token: A static variable representing the authentication token obtained after successful login. It is used to authorize API requests.
- _headers: A map containing the default headers to be included in API requests, such as the 'Content-Type' header.

Methods:

- login: Sends a login request to the backend with the provided email and password. If the login is successful, the response includes an authentication token stored in the token variable.
- logout: Clears the stored authentication token, effectively logging the user out of the application.

- setToken: Sets the provided token in the token variable for use in API requests.
- register: Sends a user registration request to the backend with the provided email, password, and name. If the registration is successful, the response includes user information.
- addNote: Sends a request to the backend to add a new note. It includes the note data and the authentication token in the request headers for authorization.
- deleteNote: Sends a request to the backend to delete a specific note by its ID. It includes the authentication token in the request headers for authorization.
- updateNote: Sends a request to the backend to update an existing note. It includes the updated note data and the authentication token in the request headers for authorization.
- fetchNotes: Retrieves a list of all notes from the backend. It includes the authentication token in the request headers for authorization.
- fetchTags, addTag, updateTag, deleteTag: Methods to manage tags, similar to the note-related methods.
- addTodo, fetchTodosByNoteId, fetchTodoByTodoId, updateTodo: Methods to manage todos, similar to the note-related methods.
- fetchContacts, addContact, updateContact, deleteContact, fetchContactByContactId: Methods to manage contacts, similar to the note-related methods.

The ApiService class does not directly interact with the UI. Instead, it is used by provider classes (e.g., AuthProvider, NoteProvider, TagProvider) to handle API requests and update state accordingly.

The ApiService class is custom-built for the cross-platform mobile application and is not based on external libraries. It demonstrates creating a data service to interact with the backend API using Dart and the http package.

Here is an example of what a login request to the backend's REST API server looks like using the ApiService class. This function is used to authenticate a user by sending their email and password to the backend and receiving an authentication token if the login is successful.

```
login.dart    api_service.dart 4 ×
frontend > lib > services > api_service.dart > ApiService > register
1   import 'dart:convert';
2   import 'package:frontend/models/note.dart';
3   import 'package:frontend/models/todo.dart';
4   import 'package:http/http.dart' as http;
5   import 'dart:core';
6   import '../variables/variables.dart';
7   import '../models/contact.dart';
8   import '../models/tag.dart';
9
10  class ApiService {
11    static const String _baseUrl = '$hostUrl/api';
12    static const String _loginUrl = '$_baseUrl/users/login';
13    static const String _registerUrl = '$_baseUrl/users/';
14    static const String _tagsUrl = '$_baseUrl/tags';
15    static const String _contactsUrl = '$_baseUrl/contacts';
16    static const String _notesUrl = '$_baseUrl/notes';
17    static const String _todosUrl = '$_baseUrl/todos';
18
19    static String? token;
20    static final Map<String, String> _headers = {
21      'Content-Type': 'application/json',
22    };
23
24    // Authentication
25    static Future<Map<String, dynamic>> login(
26        String email, String password) async {
27      final response = await http.post(
28        Uri.parse(_loginUrl),
29        headers: <String, String>{'Content-Type': 'application/json'},
30        body: jsonEncode(<String, String>{
31          'email': email,
32          'password': password,
33        }),
34      );
35
36      if (response.statusCode == 200) {
37        final Map<String, dynamic> responseData = jsonDecode(response.body);
38        token = responseData['token'];
39        return responseData;
40      }
41      if (response.statusCode == 401) {
42        throw Exception('Invalid email or password');
43      } else {
44        throw Exception('Failed to log in');
45      }
46    }
47
48    static void logout() {
49      ApiService.token = null;
```

Figure 36 ApiService

**Variables Folder:** Contains colors. Dart and variables.dart files define color palettes and other design-related variables used throughout the application. These variables ensure consistent and visually appealing design elements.

In this case, variables.dart only contains the backend server's URL. As the application grows, having dedicated variable files becomes even more beneficial. They provide a single source of truth for various configurations and settings, promoting code maintainability and scalability. When new features are added, or design changes are required, developers can easily update these variables, and the changes will propagate throughout the app seamlessly. This modularity enhances reusability and reduces the chances of introducing bugs due to inconsistent values.

And last but not least, the **main.dart** file, which is the entry point of the Flutter application. In this file, we configure and initialize the various providers used in the application and set up the main MyApp widget.

Here is a brief explanation of the main.dart file:

- main() Function: This is the entry point of the application. It calls WidgetsFlutterBinding.ensureInitialized() to ensure that the Flutter engine is initialized before running the application.
- AuthProvider Initialization: We create an instance of the AuthProvider class, passing an ApiService instance to it. The authProvider.initAuthProvider() function is then called to initialize the provider with the token retrieved from SharedPreferences (if available).
- MyApp Class: This is the main widget of the application, which extends StatelessWidget. It takes an instance of AuthProvider as a parameter.
- MultiProvider: We use MultiProvider to provide multiple instances of different providers to the widget tree. Providers like NotesProvider, TagProvider, TodoProvider, and ContactProvider are added as change notifiers.
- Consumer: The Consumer widget listens to changes in the AuthProvider, allowing the application to respond dynamically based on the authentication status.
- MaterialApp: This widget defines the core attributes of the application, such as the title, theme, and initial route.

- home and routes: The home property points to the NotesHomePage if the user is authenticated; otherwise, it shows the LoginScreen. The routes property maps route names to their corresponding widget classes for navigation purposes.

The main.dart file is a crucial starting point for the application, providing the necessary setup and configuration for the providers and routing. It ensures that the application starts with the appropriate screen based on the user's authentication status.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

login.dart | main.dart ×

frontend > lib > main.dart > ...

```
import 'package:frontend/providers/contact_provider.dart';
import 'package:frontend/providers/notes_provider.dart';
import 'package:frontend/providers/tag_provider.dart';
import 'package:frontend/providers/todos_provider.dart';
import 'package:frontend/screens/notes_homepage.dart';
import 'package:frontend/screens/login.dart';
import 'package:frontend/screens/register.dart';
import 'package:frontend/services/api_service.dart';
import 'package:provider/provider.dart';

Run | Debug | Profile
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  AuthProvider authProvider = AuthProvider(apiService: ApiService());
  await authProvider.initAuthProvider();
  runApp(MyApp(authProvider: authProvider));
}

class MyApp extends StatelessWidget {
  final AuthProvider authProvider;
  const MyApp({Key? key, required this.authProvider}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider<AuthProvider>(
          create: (_) => authProvider,
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<NotesProvider>(
          create: (_) => NotesProvider(),
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<TagProvider>(
          create: (_) => TagProvider(),
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<TodoProvider>(
          create: (_) => TodoProvider(),
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<ContactProvider>(
          create: (_) => ContactProvider(),
        ), // ChangeNotifierProvider
      ],
      child: Consumer<AuthProvider>(
        builder: (context, auth, _) {
          return MaterialApp(
            title: 'MyApp',
            debugShowCheckedModeBanner: false,
            theme: ThemeData(
              primarySwatch: Colors.blue,
            ), // ThemeData
            home: auth.isAuthenticated
                ? const NotesHomePage()
                : const LoginScreen(),
            routes: {
              LoginScreen.routeName: (ctx) => const LoginScreen(),
              RegisterScreen.routeName: (ctx) => const RegisterScreen(),
              NotesHomePage.routeName: (ctx) => const NotesHomePage(),
            },
          ); // MaterialApp
        },
```

*Figure 37 main.dart file*

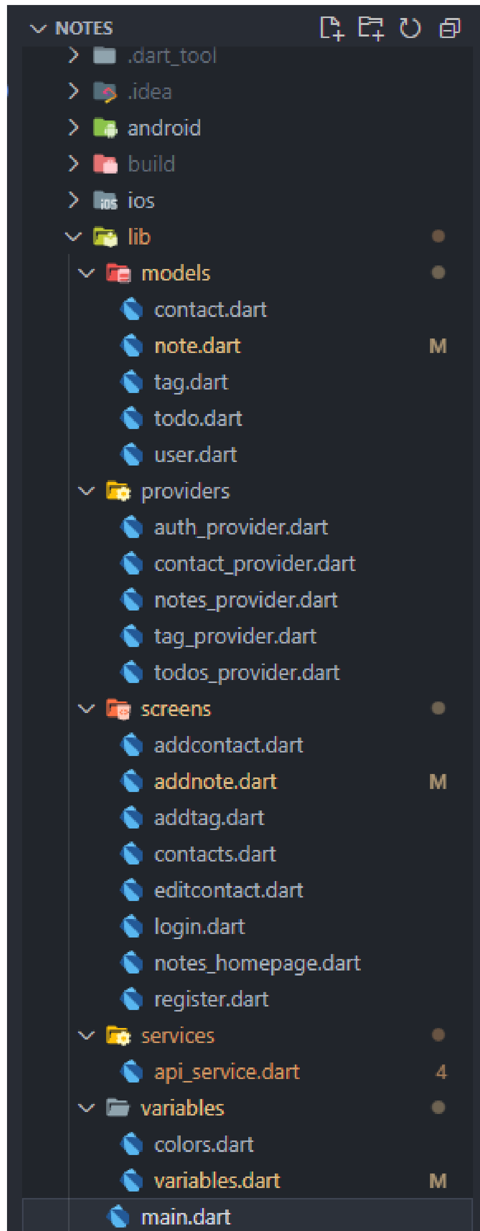


*Figure 38 Structure of frontend*

# 4 Summary and reflection

Creating a cross-platform mobile application using Flutter and Dart has been an exciting and learning-filled experience. Before starting this project, I had a small amount of experience with Flutter from an internship and some exposure to Node.js and Express through a group project at school. However, working on this application allowed me to dive deeper into both technologies and improve my skills significantly.

This project's main goal was to explore Flutter's potential and develop a functional app that runs seamlessly on Android and iOS devices. To achieve this, I carefully selected technologies like Flutter, Dart, Node.js, and Express, which proved to be an excellent combination for building a robust and responsive application.

During the development process, I started with planning and designing the app's structure. Flutter's rich ecosystem and ready-to-use widgets provided a smooth and efficient frontend development experience. On the backend, I leveraged Node.js and Express to handle data management and create a solid foundation for the app's functionality.

As of now, the application has yet to be published on Google Play or the App Store. However, I have plans to continue working on it in my free time. I intend to add more features, improve the user interface, and make the app visually appealing. After implementing these enhancements, I plan to publish it on both marketplaces to share it with a broader audience.

In conclusion, this project has been a valuable learning experience, allowing me to grow as a developer and gain expertise in Flutter and Node.js with Express. As I continue working on the app and further refining its features, I look forward to contributing to the ever-evolving world of mobile app development using Flutter. I aspire to create an app that users will love and find beneficial in their daily lives.

## References

StatCounter, 2023. Mobile Operating System Market Share Worldwide. Available at: https://gs.statcounter.com/os-market-share/mobile/worldwide Accessed 13.5.2023

Kotlin, 2023. Native and cross-platform app development: how to choose? Available at: https://kotlinlang.org/docs/native-and-cross-platform.html

Flutter, (n.d.). Flutter Docs, FAQ. Available at: https://docs.flutter.dev/resources/faq Accessed 13.5.2023

Dart, (n.d.). Dart overview. Available at: https://dart.dev/overview Accessed 13.5.2023

Postman, 2023. Postman | The Collaboration Platform for API Development. Available at: https://www.postman.com/ Accessed 26.5.2023

Git, (n.d.). Git – Fast, scalable, distributed revision control system. Available at: https://git-scm.com/ Accessed 26.5.2023

GitHub, (n.d.). GitHub: Where the world builds software. Available at: https://github.com/ Accessed 26.5.2023

freeCodeCamp, 2017, Dariya Kursova. What exactly is Node.js and why should you use it? Available at: https://www.freecodecamp.org/news/what-exactly-is-node-js-and-why-should-you-use-it-8043a3624e3c/ Accessed 27.5.2023

Express (n.d.), Express – Node.js web application framework. Available at: https://expressjs.com/ Accessed 27.5.2023

MongoDB (n.d.), MongoDB Features Available at: https://www.mongodb.com/features Accessed 27.5.2023

MongoDB (n.d.), MongoDB vs. MySQL Differences Available at: https://www.mongodb.com/compare/mongodb-mysql Accessed 27.5.2023

MongoDB (n.d.), Comparing MongoDB vs PostgreSQL Available at: https://www.mongodb.com/compare/mongodb-postgresql Accessed 27.5.2023

Stack Overflow, 2022, Stack Overflow Developer Survey Available at: https://survey.stackoverflow.co/2022/#technology Accessed 27.5.2023

Wikipedia (n.d.), Visual Studio Code. Available at: https://en.wikipedia.org/wiki/Visual_Studio_Code Accessed 27.5.2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Visual Studio Marketplace. Dart. Available at: https://marketplace.visualstudio.com/items?itemName=Dart-Code.dart-code Accessed 27.5.2023

Visual Studio Marketplace. Flutter. Available at: https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter Accessed 27.5.2023

Visual Studio Marketplace. Github Copilot. Available at: https://marketplace.visualstudio.com/items?itemName=GitHub.copilotvs Accessed 27.5.2023

JSON Web Token (JWT) Introduction. Available at: https://jwt.io/introduction Accessed 30.5.2023

Bcrypt GitHub Repository. Available at: https://github.com/kelektiv/node.bcrypt.js Accessed 30.5.2023

Provider package. Available at: https://pub.dev/packages/provider Accessed 26.7.2023

Kanini .NET MAUI vs. Flutter vs. React Native: A Comprehensive Comparison. Available at: https://kanini.com/blog/net-maui-vs-flutter-vs-react-native/ Accessed 05.11.2023

## Appendices

Project Code Repository

For the complete source code of the mobile application and backend development, please refer to the GitHub repository at the following link:

GitHub Repository - Mobile Application and Backend Code

The repository contains the entire codebase, including the Flutter application, Node.js backend with express, and the MongoDB database configuration. It provides a comprehensive view of the implementation details and allows interested readers to explore the project more deeply.

Please note that the repository may be updated with additional features, improvements, and bug fixes beyond the version presented in this thesis.