Universität Trier

Implementing Collaboration Applications with an Unconventional Software Architecture

Implementation approach using DART, Flutter & Firebase

FB IV
Master: Wirtschaftsinformatik

Name:	Lionel Schroeder
Addresse:	46, rue de la Resistance
	L-4942 Bascharage

Erster Betreuer: Prof. Dr. Peter Sturm
Zweiter Betreuer: Prof. Dr. Stephan Diehl

Matrikelnummer : 1494510

Abgabedatum : XX.XX.XXXX

Abstract

Education is a key to a successful life. Increasing the enjoyment of learning is a difficult task that is constantly developing. How often does a teacher ask in his class if a student has a question without getting a response from them? This can have a simple reason, namely anxious students. Allowing anonymous questions during lectures increases the student-teacher interaction during a lecture.

On the other hand, introducing gamification into lectures is more familiar with the help of quizzes. These quizzes allow recapping previous lectures, which increases the students' grades significantly.

The Crayon applications merge both problems into a single application. These applications use the newest flagship technologies from Google. One of them is called Flutter, a framework that uses Dart as a programming language and is used to develop multi-platform applications. The data consumed or created by Crayon must be stored in a database. Therefore Google has a new database system called Firestore. Firestore is a schemaless database that automatically propagates data changes to the listening applications. Firebase's automatic data propagation on changes increases the user experience of applications.

Acknowledgements

First, I want to thank the University of Trier to teach me advanced information technology skills to circumvent the challenges of developing the Crayon applications. Especially Prof. Dr. Peter Sturm to allow me to write my own education application and giving me feedback during the project/Master thesis.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivations

Increasing student grades and participation in courses is crucial in today's educational system. The majority of teachers ask in their lectures if students have questions. Most of the time, there are no responses. There can be multiple reasons which can cause a student not to ask a question over the course material. One of them can be that a student feels that their question is not significant enough. Another possibility can be that the student is anxious to ask a question. According to Perry Samson, an atmospheric science professor at the University of Michigan, the questions asked during lectures increases tremendously if the students can ask their questions anonymously [10]. In addition, he states that providing an anonymous way of asking questions increases the interaction and understanding of the course material.

Moreover, students tend not to revise their previous lecture material. Students who revise the material of the previous lecture have better grades than a student who does not, according to Mehmet Akif Ersoy from the University of Turkey [2]. Increasing the motivation of students to revise their old course material can be achieved with gamification. Gamification adds game mechanics to an environment that does not have a gaming environment. Adding gamification to lectures can be done by using quizzes. The book [9] states that there are significant benefits of testing students. One of them is to test the students frequently, which encourages them to study. Quizzes during lectures can be seen as a test and thus can solve the problem of lousy student grades.

Kahoot is an application that gives the possibility to the teacher to start a quiz during the lecture. Using such a quiz application requires the student to install an application that only lets the user participate in quizzes. However, applications such as Kahoot have one major drawback: they can only do one thing: creating/starting a quiz. Kahoot can only do the quizzes. Messaging applications can be used for anonymous questions asking to the teacher where each student also requires the same application. In addition, the teacher might require an application to present his lecture slides. The number of applications required to increase the learning benefits comes with costs for the teacher and the student.

The teacher must manage multiple applications and has to "force" the students to use multiple applications instead of one. Furthermore, students can have multiple teachers, and each one of them can have other applications which allow creating quizzes or anonymous question asking. Thus creating one application that merges these features can make life easier for the student and the teacher.

1.2 Objectives and Contributions

The Crayon applications were developed to regroup working techniques in education into one to improve the interactions and student grades. The Crayon applications are divided into two parts, management and student application. Both applications were developed with the Flutter Framework and are currently not a standard in the developing industry.

The teacher uses the management software, which is in charge of creating the content for the student application. The content or data created by the management application must be stored in a database. The database chosen is called Firebase and is currently one of the most advanced databases in the world.

Using new technologies requires deep research of each technology. This research or analysis allows one to find out the benefits and the pitfalls for each one of them. Finding out the different pitfalls prevents them from occurring during the development process. Firebase has one pitfall, namely cost. Traditionally data is normalised to reduce data redundancies which is not the case in Firebase. Firebase data can be modelled to reduce the production cost, which creates the field of data modelling by cost reduction. Moreover, Flutter applications performance can be harmed significantly by not using state management techniques. These techniques allow rebuilding only parts of the screen instead of the whole screen. Depending on the screen content, this can lead to a slow application.

2 Basics

Developing software for different devices required application creators to know multiple programming languages. The development world is subdivided into multiple development fields. Web Pages have two main fields, namely front-end and back-end. Front-end describes the visual aspect of a web page; it requires the knowledge of at least three programming languages, namely:

- HyperText Markup Language (HTML)
- Cascading Style Sheets (CSS)
- Javascript

If a web page only uses these three types, it is called a static webpage. A static web page is a page that does not change its content. These types of web pages are commonly used for small businesses.

More complex web pages use a back-end application to serve data to the front-end application. The back-end application is used for data retrieval and might also be used for heavy computational tasks to prevent the front-end application from slowing down. Back-end applications can be written in multiple programming languages. The most common ones are:

- Java
- Python
- Php

Thus, building a more complex web page requires building two applications. However, nowadays, developing only a webpage might not be enough. One example of this are email services such as Gmail from the company Google. Google not only developed Gmail for web pages but also for smartphones. According to GSMA intelligence [8], 66.5% of the global population have smartphones. Thus developing a specific application for these devices increases the customer base for Google and other companies. Smartphone applications benefit from being faster since the visual aspect of the application is already stored on the device, thus allowing faster operation. In addition, mobile applications can accept notifications from a back-end application. Email service applications notify the user as soon as a new email is available. Building mobile apps requires different programming languages depending on the device's operating system. An operating system manages computer hardware and software resources and provides a service for computer programs.

The most used Operating Systems on smartphones are currently Android and iOS. Creating an application for those devices requires the developer to learn at least two programming languages. IOS uses Swift, and Android uses Java or Katolin as a programming language. Developing software for both operating systems is a requirement due to the extensive user reach. Statcounter [13] states that the market share for Android devices is 63% and iOS 38.5% in Germany. Complex mobile applications also use a back-end application as in web development.

2.1 Dart

Dart is an ECMA-standardised programming language and is mainly developed by Google. ECMA standards for European Computer Manufacturers Association have a role in standardising different technology fields. They set the standards for inter compatibility for different web browsers.

Dart is a modern alternative to the JavaScript programming language. Dart code can be used for both front-end and back-end applications. Currently, the main focus of Dart is front-end applications. The coding style and syntax of Dart is similar to other Object-Oriented Programming languages such as Java. Applying the same syntax, it is easier for object-oriented programmers to use this new language.

Dart code is much faster than JavaScript due to both compilation approaches JIT and AOT. JIT stands for Just-In-Time compiler and has one benefit it does not compile the whole code ahead of time. This means that the startup process of a dart application can significantly be increased by not compiling the complete code. On the other hand, AOT or ahead of time compiler compiles the complete code, which gives a longer starting time but increases the overall performance.

The Dart documentation states that Dart code is a type-safe programming language [3]. This means that Dart variables can have a type, but they do not have to. Furthermore, Dart has a null safety feature, a great asset over other programming languages. The Dart documentation explains the null safety feature as follows:

"For us, in the context of null safety, that means that if an expression has a static type that does not permit null, then no possible execution of that expression can ever evaluate to null. The language provides this guarantee mostly through static checks, but there can be some runtime checks involved too [5] ."

Usually, developers who have a bug in their code try to search for the problem and find out if another developer has already solved this issue. The higher the rate of developers for a programming language is, the higher the chance of finding the solution to the developer's bug. Statista states in [14] that over 64% of the developers know/use the JavaScript programming

language, and Dart is only represented with 6%. This is due to the age of each programming language. JavaScript has existed since 1995, which results in a vast developer pool. The counterpart Dart was released in November 2013 and thus is a new programming language and therefore has a smaller developer size. The smaller developer size makes running into already solved problems less likely than in JavaScript. This makes the development process of Dart application more complicated. However, the gains of the Dart programming language are significant in terms of speeds of an application and for developing applications for multiple devices concurrently.

2.2 Flutter

Firebase is Google's "All-In-One Backend Solution" for mobile and web applications. Firebase provides Software Development Kit (SDK) tools and infrastructure, allowing developers to use efficient APIs to ease the development process. By using Firebase, the developer cannot build a custom backend application for data retrieval. However, Firebase can only be used on their devices and can not be downloaded for personal use. In other words, a Firebase customer must use Google's infrastructure. Not every company can buy servers and know how to maintain them. In addition, a Firebase customer does not need to write scalable backend code. As soon as Firebase detects that more processing power is required for an application, Firebase automatically assigns more processing power to the applications without the need of a developer. The key features from Firebase which will be used in the Crayon applications are:

- Authentication
- Database
- Cloud Storage
- Scale

Authentication is one of the critical security issues of modern applications. Firebase provides a particular service for authentication purposes which is called Firebase Auth. Firebase makes the security updates to maintain the highest security standards for users' data. In addition, there are multiple ways to authenticate with Firebase:

- Authentication
- Database
- Cloud Storage

- Scale

Phone number authentication allows an individual to authenticate with his phone number. Firebase will send a verification code to the user's phone, and in return, the user has to enter the verification code to access the restricted content of the application. On the other hand, email authentication is the most commonly used. Email authentication is usually accompanied by a password. Emails are verified by sending an email to the registration email with a specific activation link. This link needs to be opened by the user, and Firebase automatically validates the email. Both processes allow verification to discard non-human interaction with a given application. The final authentication method is the username with a password that is currently dying out. This is due to the not available verification process to check whether the user is a computer program or a human.

Firebase has two different database types for storing data. These two are Cloud Firestore and Realtime Database. Both Cloud Firestore and Realtime Database are cloud-hosted NoSQL databases. These NoSQL databases are specifically designed to create schemaless databases. SQL databases have a predefined structure. In other words, they can be described as tables. Thus, SQL databases require to be predefined by a data analyst or a developer. NoSQL solves this issue by storing the data in a JavaScript Object Notation format which acronym is JSON.

Realtime Database stores the JSON Objects in one big JSON object, which the developers can control in real-time. Figure 1, is an example of a big JSON file. Naturally, developers would split up chat messages into multiple files to create a hierarchical data structure and benefit from a better overview of the data. However, Realtime database does not allow another structural format except for one big JSON object. Realtime database also gives the possibility to listen to changes to the data in real-time. A major drawback of Realtime database is that the queries are not shallow. For example, querying for a specific JSON object in the big Json Object also returns all sub-objects of the searched object. This means that more data can be requested from the database than needed. Moreover, Realtime database does not allow querying over multiple fields, meaning the data needs to be denormalized, meaning that a JSON object requires an additional field where the query needs to be performed.

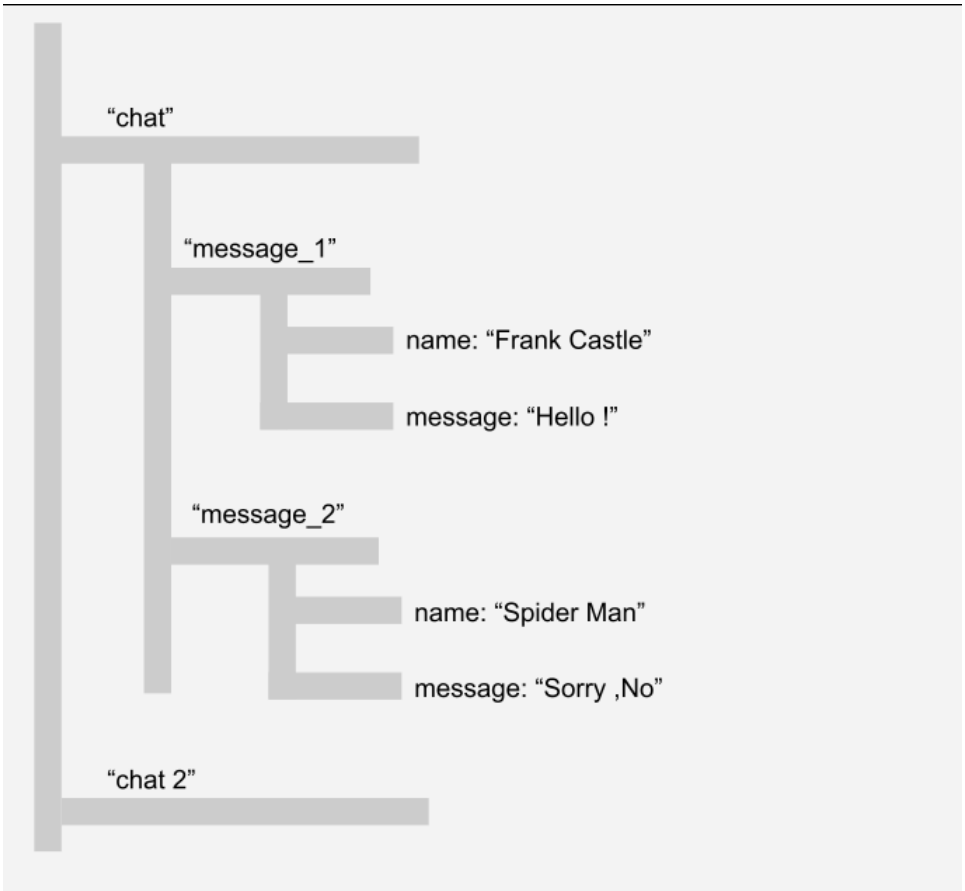

Figure 1: Real Time Database Data format

On the other hand, Firestore, the newest product from Google, has more functionalities than Realtime database. The data is not stored in one huge JSON object but in a more structured approach. The data is subdivided into collections and documents. Documents can have a maximum size of 1 MB. If the size is exceeded, the JSON object can not be stored. Collections contain documents, and the documents contain data and can point to other sub-collections. Figure 2, describes how the data can be structured in Firestore. The chat collection contains all the chats documents. In this case, chat 1 and chat 2 are documents that contain the different messaging information of two individuals. This additional data structure allows Firestore to have an important advantage by performing queries. Firestore queries are shallow, which allows the retrieval of a single document without retrieving any other linked subcollections. Firestore does not need a denormalizing process of data as the Realtime database counterpart and thus allows querying over multiple fields. However, querying multiple fields in Firestore can not be done by default. Firestore, by default, indexes each field to make querying for a specific document instantaneous. Composite indexing is required to perform multiple fields querying, which the developer must set up in Firestore.

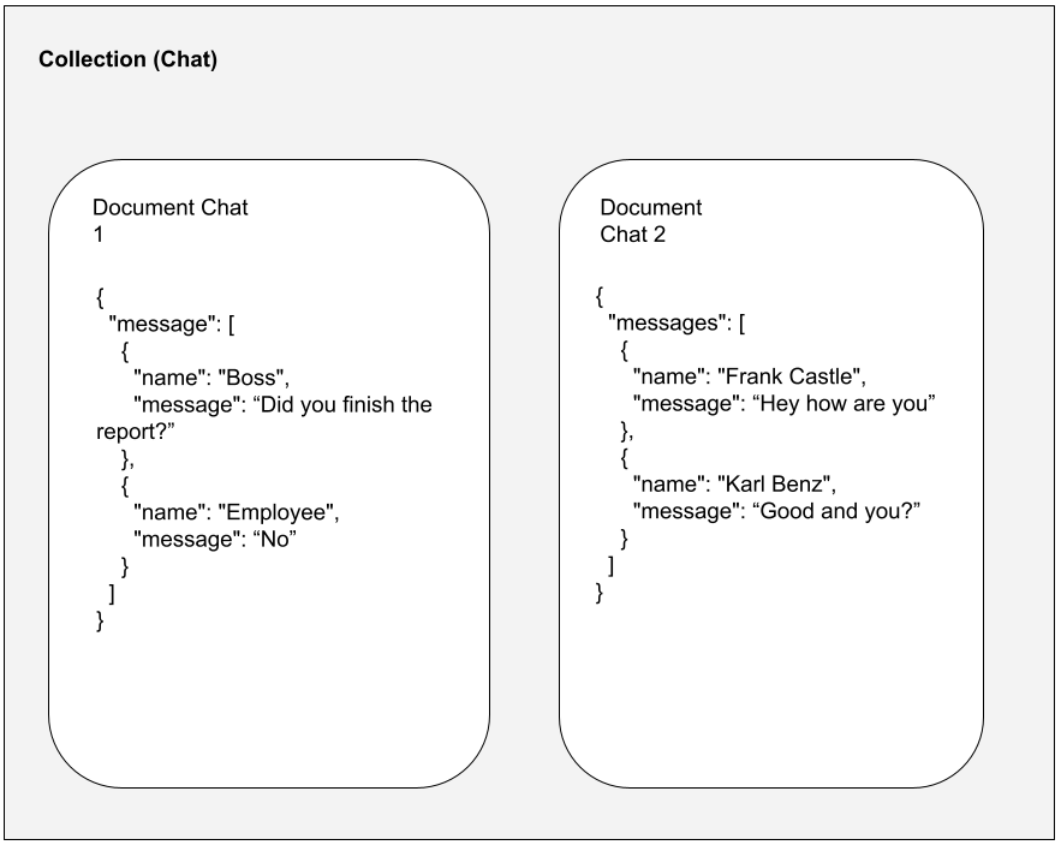

Figure 2: Firebase Model of a chats

2.3 Environment

The environment describes the different tools and versions of the source code used during the development process. Both projects use the identical versions of Flutter, namely 2.5.3, the most recent version at the start of development. Since Flutter uses Dart as a programming language, the Dart version 2.14.4 was used. Visual Studio Code from Microsoft was used for developing the code for both applications. Android Studio Code from IntelliJ was used to download different virtual mobile devices. Having access to multiple devices allows testing the application on different screen sizes. This can prevent bugs in the design of an application.

The editor Android Studio Code provides a simple way to download different Android firmwares. These firmwares describe the different Android or iOS versions of a mobile phone. Usually, everything should work on every device, no matter the version. However, an old Android version might not have functionalities available to newer versions in some cases. Therefore, it is required in Android to specify the minimal supported Android version. The simplest way to have a working application is to make the minimal version the newest one.

However, the Crayon applications use 20 as a minimum for Android devices. This minimum Android versions name was called KitKat and was added in 2014. Statcounter states that the significant majority or 95%+ of Android users use a higher version than our minimum version set [12].

The Android documentation from Google suggests that an application should at least cover 90% of the devices [6] which the Crayon applications overreach significantly.

It is recommended to use a version control system during the development process. Version control systems allow to track changes on source code and collaborate with other developers. GitHub is the most used version control system in software development. GitHub has a repository which is a data structure used to store files or directories. Each upload/commit from a developer gives a new version to the project. This allows downgrading to the older version if the new code contains anomalies or bugs. Storing the project into a repository has another vital asset: if a computer gets damaged, the code will still be available on GitHub, and the current work will not be lost.

3 Preliminary Analysis

The preliminary analysis's primary goal is to identify the core requirements for an application. This analysis creates the blueprints of the software. The core components of the blueprints are:

- Mockup
- Requirements
- Data
- Software architecture
- State management

Mockups visualise what the final application applications might look like. These mockups can have a mid to high fidelity to the final product. The requirements are closely related to the mockup since the requirements describe which core functionalities an application must-have. In addition, the data must be known before developing the software. Knowing which data is required before implementing the software allows designing the data in a certain way. For example, to reduce query costs on Firestore. Finally, creating the software architecture allows implementing different functionalities of an application the same way without reinventing the wheel for each function. This architectural process is required to increase the manageability and consistency of a programm. The Flutter framework, however requires additional analysis. Proper Flutter applications have 60 frames per second. The frames per second describe how fast an application updates the view. Flutter by default, rebuilds the whole screen each time a value is updated, which decreases significantly the frames per second and harms the performance of a Flutter application. State management thus introduces an essential complexity that must be treated before the application is developed.

3.1 Mockup & Requirements

The requirements analysis describes what functionalities an application requires to be successful. The iconic image 3 in information technology describes the process of the creation of an application. This image shows the problems in developing an application. The client usually knows what he wants but can not describe it adequately. On the other hand, the developer/engineer creates the software how he understands it. The client, in the end, did not want what the engineer developed and explained in more detail what he wanted. Preventing such errors can not be entirely omitted, but some can be prevented.

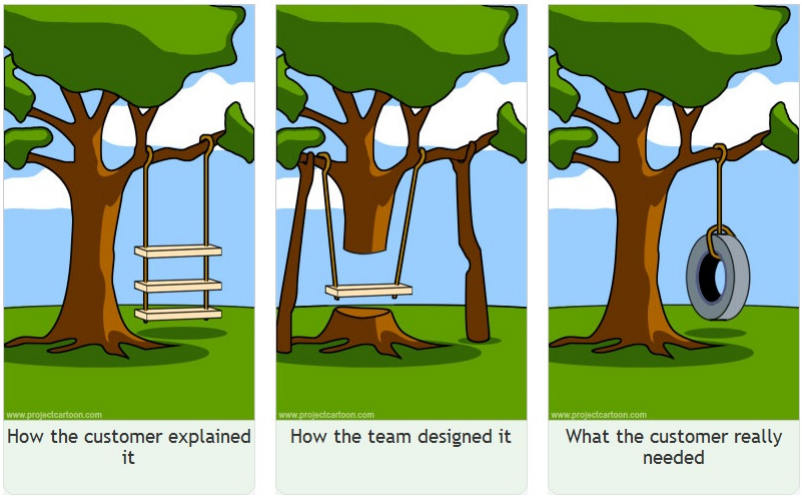

Figure 3: Project development process

Preventing mis-creating an application requires detailed documentation on what the application should do. This documentation of the application's functionalities allows the developer to play by a script and not implement what he wants but what the client wants. This script gives the client reinsurance that he should get what he wants, and both sides have a written version of what they agreed on in case of conflicts. The Crayon application is divided into two separate applications. One application for the lecture giver and one which consumes the content provided by the lecture giver. The lecture giver application is called Crayon Management, and the student's application is called Crayon. The management application requires to have the following main functionalities:

- Lecture creation
- Presentation mode
- Questions
- Quizzes creation/start

A lecture giver must at least create one lecture before performing other critical functionalities of the application. The presentation screen allows the teacher or lecture giver to present his pdf slides. These pdfs have to be stored in the Firebase file system, allowing access to the pdfs over the web on different devices. The question feature is available as soon as a student asks a question in the student application. The system notifies the teacher that a student asked a question. However, the teacher decides when the question should be answered. The teacher also has the possibility of starting a quiz during his lecture. By

starting a quiz the student must automatically be notified that a quiz started for this specific lecture. The student application must have the following key functionalities:

- Enroll
- Ask Questions
- Participate to quiz

The students must enrol into a lecture to participate in quizzes and ask questions. The enrollment process must be done over a QR-code. A lecture id is a 24 digit long character sequence and entering 24 digits to enrol in a lecture significantly harms the user experience. Therefore, a more user-friendly way to enrol in a lecture is required. A QR-code is a perfect solution for such a problem. As soon as the student is enrolled in such a lecture, he gains access to the other features. The application must detect Real-time changes. For example, if a lecture date or the room of a lecture changes, the user must see the changes in the application. Finally, as soon as the teacher starts a quiz, the application must show that the teacher started a quiz.

Illustration 4 shows the exact process of how to start a quiz. First of all, the teacher requires to select a previously created quiz. Before a teacher can open a lobby that allows the users to join the quiz session, the teacher must set how much time the quiz should take. After setting up the time, the Crayon management application modifies the lecture document in Firestore. This modification will automatically be propagated to the student's application. The student application processes this new information and displays that a student can join the new quiz session. Therefore a pop-up is opened, and the student can enter a username. This username gets sent back over Firestore to the Crayon management application and displays the student's username. As soon as the teacher starts the quiz, the quiz information with the quiz's time will be sent to the student application. The student application will automatically open the quiz mode, and the student has to answer the quiz's questions. After the quiz is completed, the student application will automatically send the responses to the management application over Firestore. The teacher can move to the quiz responses screen if he decides to do so or if the quiz time elapses. The teacher can respond to each question individually and see the students' responses on this screen.

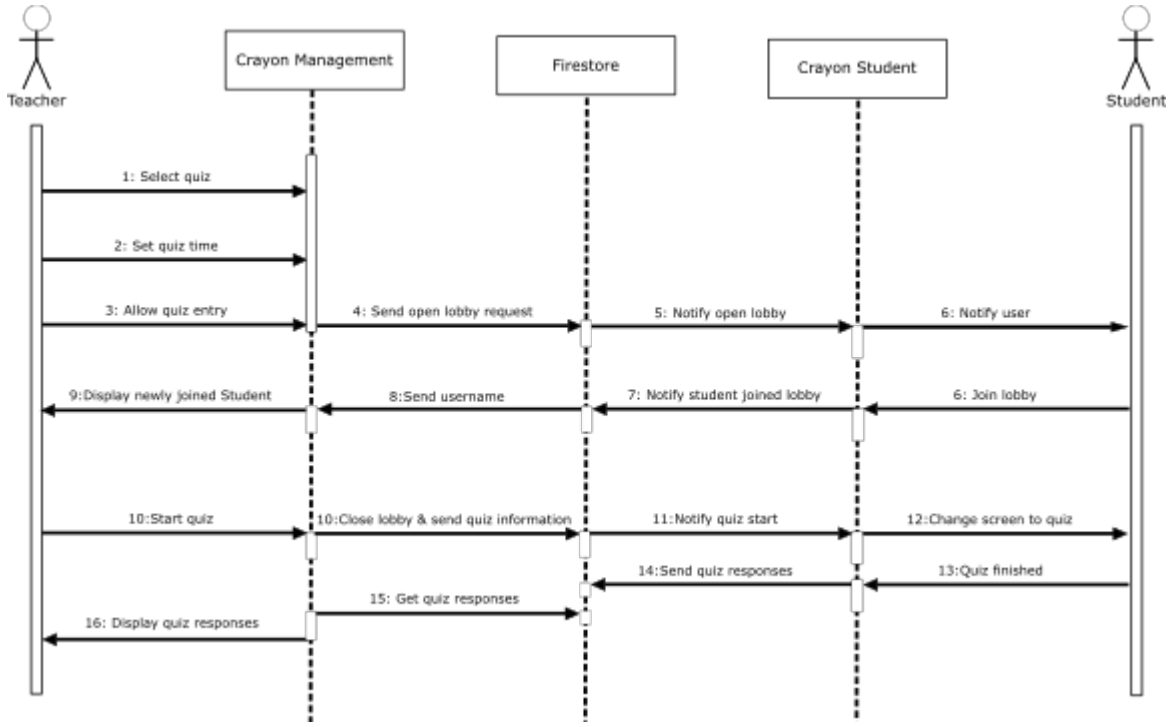

Figure 4: Sequence diagramm of a quiz process

The second key is allowing a student to ask anonymous questions, which is described in 5. The teacher requires to be on the presentation screen to see the questions asked by the student. However, the system still stores the questions from the students even if the teacher is not on the presentation screen. In step three, the student asks his question, which gets sent to Firestore. Firestore notifies the management system that the student asked a question. As soon as the teacher retrieves the questions manually, a deletion request of the question or questions will be sent to Firestore.

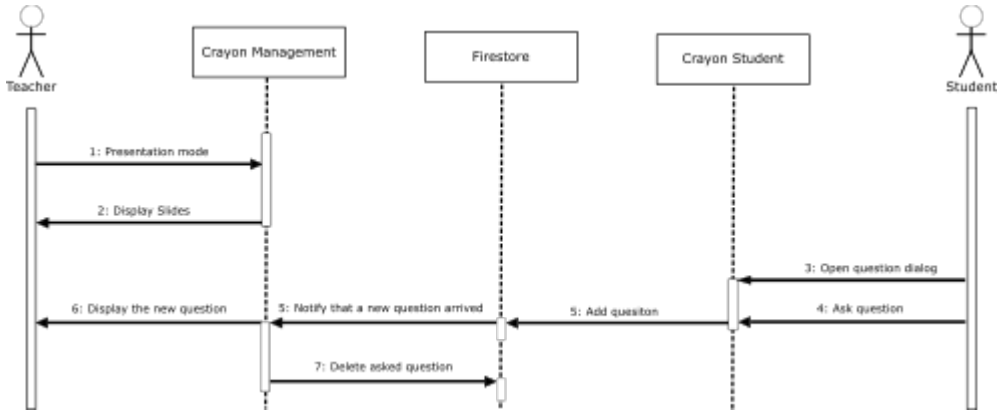

Figure 5: Sequence diagramm of a question process

Mockups can further decrease a mis-implementation of an application. The mockup is usually not created by the developer but by a UI/UX expert who decides how the application looks and feels. Mockups allow the developer to implement the application as it states on these images provided by the UI expert. The mockups of the different screens of an application should include:

- Content Layout
- Color Scheme
- Typography
- Spacing
- Navigation visuals

Content layouts relate to how the content should be displayed. These layouts can be, for example, a table. The colour scheme states which colour or shades should be used over the whole application. The main objective of this process is to keep one primary colour over the whole application. The typography states which font style should be used and at which size the different text paragraphs should be. The spacing aspect describes that a given content should not overlay another and should not seem cluttered. Navigation in an application is a crucial component that allows seeing other application contents. Thus emphasising how to navigate through the application makes the application more user friendly.

Mockups can be created in multiple applications available online. The state-of-the-art application for mockups is called Figma. Figma has a lot of built-in functionalities to go beyond of just creating pictures. Figma allows creating "proto-types", which creates animations for navigating between screens. Small functions can be included to display dialogues

and more. Creating advanced mockups as it is possible in Figma allows showing a given person or the contract giver of an application to see how the final application will look and which functionalities the application will have.

3.2 Data Modeling by Cost reduction

The Firebase Database system is an efficient database, but it costs. From the table 1, the different CRUD operations such as reading, writing and deleting a document have different pricing ranges. Reading is free up to 50.000 documents per day, and writing or deleting is free for 20.000 documents per day. If the application has a higher consumption, the administrator/application owner must pay for additional reading or/and writing operations tranches. Thus, adding additional document reads (50.000) will cost 0.06 $ and afterwards 100.000 for each extra tranche. The writing to documents is three times higher than document reads. Finally, the cheapest operation is deleting files which only costs 0.02$. Preventing unnecessary requests to the Firestore thus must be included in the data modelling process for both applications to reduce unnecessary costs.

Operation	Free quota per day	Price beyonthe free quota	Price unit
Document reads	50,000	$0.06	per 100,000 docs
Document writes	20,000	$0.18	per 100,000 docs
Document deletes	20,000	$0.02	per 100,000 docs

Table 1: Firebase pricing

Both applications require authentication. The authentication process creates a unique identifier for the newly registered user. This unique identifier allows the storage of additional information for only that specific user. There are several ways to implement an authentication process. First of all, phone numbers are increasingly used in modern-day applications. It is easier for the user since he does not need to remember a password. A password study was conducted by HYPR, which states that 78% of people had to reset a password they forgot in the past 90 days [7]. Phone authentication would allow a majority of users not to remember a password. The European Data Protection supervisor states in [15] that an application should only store required personal data. Phone Numbers do not add any benefit for the user except that they forget their passwords. Both crayon applications thus require an email and a password to be stored in Firebase. Only persons who use the management application must add a first name and last name. The final data model in JSON format is stated in 1.

A lecture can be given on multiple dates, have specific types, have a title and have a teacher. The dates describe when a course starts and ends and on which day of the week

```
{
  "UID": "Unique_ID",
  "firstName": "Nicolas",
  "lastName": "Tesla",
  "email": "nicola_tesla@edu.com",
  "password": "ElonUsesMyTeck13"
}
```

Listing 1: JSON model for Registration in the Management app

the lecture occurs. Lectures have a specific day in the week and time when they happen. In addition, lectures can have three distinct types of lecture:

- Exercise (Practice session)
- Lecture (Theoretical session)
- Other

There are two solutions to store lectures with their respective type and occurrence. The less powerful method would be to store the lectures by type and day of occurrence into different documents. This would increase the requests to Firestore and thus increase the price for retrieving the lectures. Another more favourable possibility is storing the lecture and their different types into the same file to prevent unnecessary Firebase charges.

The applications allows to create/join quizzes. Quizzes are connected to a specific lecture, allowing only to show the quizzes relevant to a specific lecture. A lecture can have multiple quizzes and are not directly added into the lecture document. The student Crayon application does not require knowing every quiz available, only the one the teacher started. Adding the quizzes into another document is thus a preferred way of storing the quizzes. A quiz has a title and can contain multiple questions which can have multiple responses. A response is completed by a Boolean value that describes if a response is correct or a wrong answer. The data model of a quiz can be seen in 2.

As soon as a teacher starts a quiz for a specific lecture, the document with the lecture information is modified by adding the selected quiz data to the lecture document. This approach allows the student application to only listen to one document change instead of multiple.

Moreover, a lecture requires slides for presentation purposes which are stored on the Firebase file system. Accessing these files over the web can only be achieved with their respective URLs, which are required to be stored in the lecture document.

```
{
  "id": "Unique_QUIZ_ID",
  "title": "Quiz_TITLE",
  "questions": [
    {
      "question_ID": "Unique_ID",
      "title": "Question 1",
      "responses": [
        {
          "response": "Response 1",
          "is_Response": true
        },
        {
          "response": "Response 2",
          "is_Response": false
        }
      ]
    }
  ]
}
```

Listing 2: JSON example of a quiz

The teachers profile document must also include the lectures he created. There are two ways of solving these issues. The first way is to query each lecture document and check if the PID match the teacher's id. This is a valid method and the most common approach. However, the Crayon applications use a custom snippets method. This method reduces significantly document reads requests to Firebase. Currently, ten read requests are performed if the teacher gives ten courses. The new method will drop the value to two document read requests and does not depend on the number of courses the teacher gives. The teacher's profile gets small portions of the lecture stored in his document. These small portions only include the name and the dates of a the lectures he created. After, if the teacher requires more lecture information, he can request more data from the actual lecture document.

The snipped method also has one minor drawback: when a lecture requires to be updated. For example, if the teacher changes the classroom of a lecture, the snipped in the teacher's profile also requires to be changed and thus resulting in two document writes instead of one.

On the other hand, the student Crayon application does not create additional data except for questions and responses to a quiz. The students profile document does not use the snipped method since the students must have real-time updates of the data of a course. This is required since the application needs to detect if a course has an available quiz or if a lecture room has changed.

Asking questions is a simple string added to the questions document of a lecture. The response of a quiz by the user contains the strings of the question and if the question was rightfully answered or not. In addition, a score variable is added, which describes if the student performed good or bad in the quiz. The final data model of the student's responses can be seen in 3.

```
{
  "responses": [
    {
      "userName": "Flying_Car",
      "UID": "Unique_ID",
      "responses": [
        {
          "question_ID": "QID",
          "time_taken": "10s",
          "was_timed_out": "false",
          "response": "(a): Response 1"
        }
      ]
    }
  ]
}
```

Listing 3: JSON example of students response to a quiz

3.3 Code Architecture

Designing software is one of the most complex tasks in creating software. Lousy software design increases code redundancies and make an application harder to maintain. Software maintenance is the process of modifying the code after deployment. A common misconception is that software maintenance only occurs for corrective purposes. However, maintenance also includes:

- Perfective
- Adaptive
- Preventive

Preventive changes to the software include updates on the code documentation and code optimisation. Preventive thus allows the software to become more stable, scalable and understandable. Adaptive change happens if the operating system or the hardware where the software runs on changes. Perfective changes are improvements in features for the software.

Frontend frameworks typically force the developer into a specific architecture. For example, Angular uses the MVC or Model View Controller design Pattern, which Trygve Reenskaug developed. Figure 6 shows how the MVC design pattern can be described. The model

is in charge of storing and managing data. This is often a database. The view is the Graphical user interface, also known as GUI, which the users see. Finally, the controller is the brain of the application. The controller's goal is to convert the input from the user from the view and, depending on the action from the user, update the GUI or requests more data from the model. A Major benefit of using such an architecture is the separation of concerns.

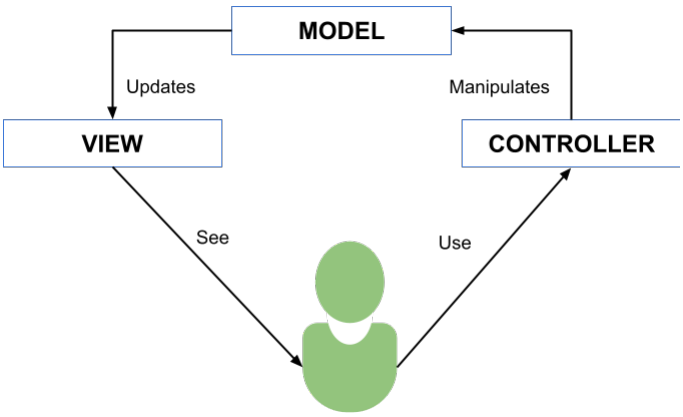

Figure 6: Model View Controller

The Flutter framework gives a lot more flexibility to the developer on what design pattern to use or even completely omit of using one. By not forcing the developer into a specific design criteria, each Flutter project can be written differently. Both Crayon applications code architecture have a close relation to the old but gold MVC design pattern. Illustration 7 shows the custom architecture of both Crayon applications for each screen.

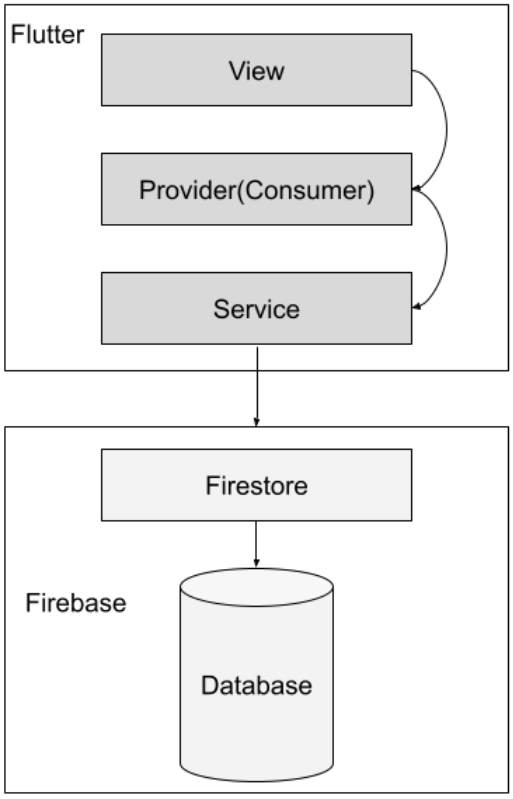

Figure 7: Crayon architecture

The view describes the user interface as in the MVC design pattern. The view contains the different widgets which describe the screen. The view does not contain any functional tasks. The view contains a widget which is called provider. The provider is in charge of the logic in the application and manages the states of the widgets by deciding which widget to rebuild if a widget changes. The provider can be seen as a controller however, he is also in charge of passing data to other widgets and can access the service layer for outside requests. The outside requests for the Crayon applications are operations on Firebase. The provider is a State management solution that Google recommends to increase the overall performance of a Flutter application.

3.4 State management

There is one proper way of implementing the provider state management solution in Flutter. The first solution uses widgets which are called Provider and Consumer. The Provider's objective is to make the Provider himself available to all children widgets on the same route/screen. The Provider widgets can only be accessed within the same route except

if they are declared at the App starting level. If a screen requires multiple providers, the widget MultiProvider can be used. The consumer widget is used to consume the data from the Provider, and if the data changes, the Provider can also notify the consumer that the data has changed.

The counter-app provided by Flutter is inefficient because the whole widget tree will be rebuilt as soon as the counter value changes. The illustration 8 with the title Provider 1 shows a wrong implementation of the Provider. The Provider widget was added and wraps the MyHomePage widget. In addition, the counter variable was moved outside the MyHomePageWidget to the Provider widget. The Provider itself shares the counter value to all the child widgets. Thus, the button and the text widget now have access to the counter value. However, as soon as the counter is incremented, all the sub widgets are rebuilt and cause inefficiencies since only the Text Widget needs to be rebuilt. This inefficiency can be countered by using the consumer widget, as shown in Provider 2. The Consumer widgets retrieve the Provider's data and give the counter value to the text widget. Moreover, as soon as the counter is incremented, the Provider can notify the consumer that the counter value has changed. The notification of a change has to be managed by the developer.

Coding a provider which allows notifying consumers of data changes requires extending the base class ChangeNotifier, which can be observed in the code 4. This allows the Provider to notify the respective consumers with the newly available function notifyListeners. A provider requires a registration, usually at the start of the screen where the Provider is needed. If a provider is accessed from another screen, it will return an error due to different build contexts. The code in 5 describes a registration of a provider. However, the widget MultiProvider is used instead of the normal Provider widget. This Multiprovider widget adds better flexibility by providing a solution to add multiple single providers instead of nesting them. In addition, the StateCounterProvider is initialised with the Provider ChangeNotifierProvider.

To preview the necessity of applying a state management solution compared to no state management solution can bee seen in the following video [1].

[1]Link to the State management Video `https://github.com/SchroederLionel/CrayonVideos/blob/main/2022-02-08%2016-50-09.mp4`.

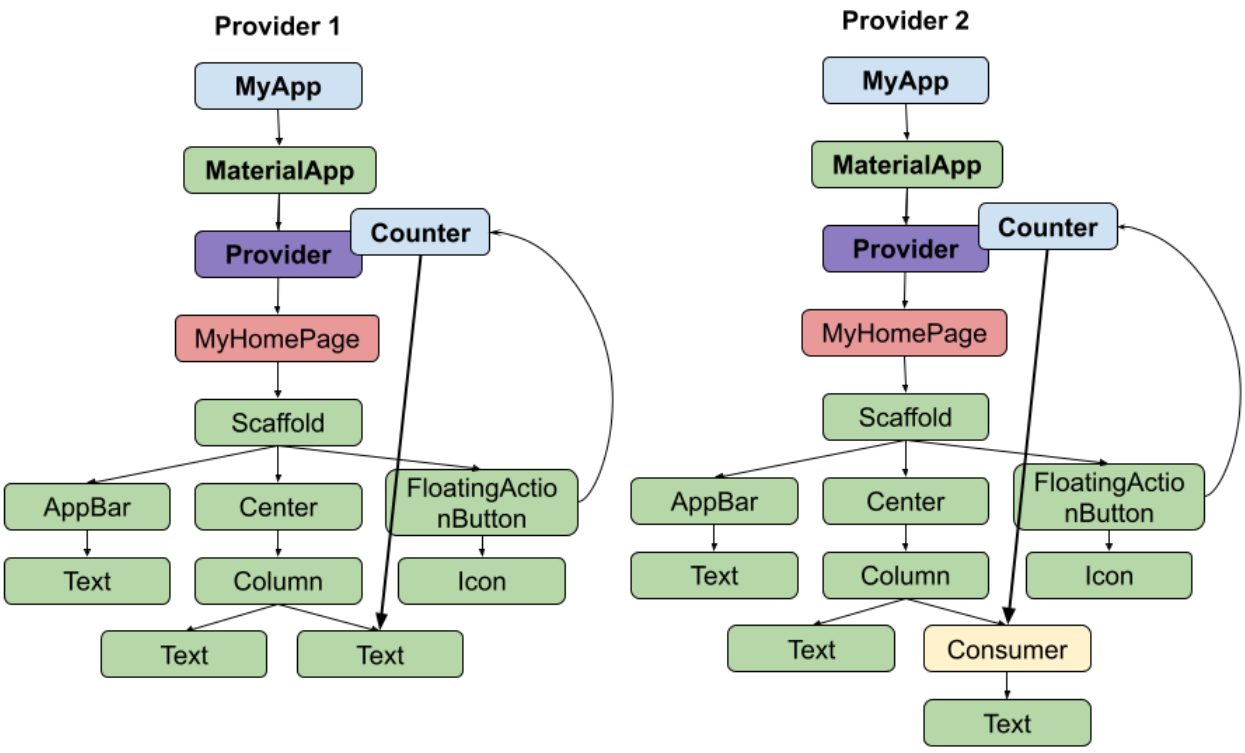

Figure 8: Counter-app width Provider

```
class StateCounterProvider extends ChangeNotifier {
    int count = 0;
    int get getCount => count;
    incrementCount(){
        count++;
        // Notify consumers
        notifyListeners();
    }
}
```

Listing 4: Counter provider with change notification

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
        providers: [
          ChangeNotifierProvider(
            builder: (_) => StateCounterProvider(),
          )
        ],
        child: MaterialApp(
          home: MyHomePage(title: "Flutter Counter Page"),
        ));
  }
}
```

Listing 5: Counter provider with change notification

4 Implementation

The Crayon applications have a high standard of code. These standards not only include state management but also include improved error handling. Each subsection describes how the Crayon applications go beyond the usual way of implementing Dart code.

4.1 Packages & Folder Structure

Packages are features that are not by default available in Flutter. The definition of packages, according to the Flutter-docs, state the following:

"shared packages contributed by other developers to the Flutter and Dart ecosystems" [4].

This means that every developer can add extensions to the Flutter/Dart ecosystem. This can include poorly developed packages. A badly developed package might consist of no documentation, or not every platform is supported. In addition, packages can also be out of date. Flutter 2.0 introduced a null safety feature in march 2021. The null safety feature improves the developer's productivity by eliminating the whole range of exceptions of accessing null values. Null values which do not contain any data and cause an error by accessing them. To improve the code base, both applications use this breakthrough feature. By allowing Null safety in both applications, old code bases are not compatible anymore. The drawback of requiring the new safety feature means that old code bases will not work anymore. On the other hand, the benefit is that there is no mismatch between different code versions. Moreover, the applications cannot be automatically updated if the package gets updated to the new Flutter version with new features or bug fixes. Adding packages to a Flutter project requires only adding the package name to the pubspec.yaml file. The pubspec file specifies the project's dependencies and the project name with a respective description. These dependencies include:

- Font styles
- Images
- Language files
- Packages

A proper folder structure is required to maximise the visibility of the different components of the applications to other developers. Illustration 9 a shows the general folder structure of the Crayon applications. The data models folder contains the entities the application uses.

```
firebase_core: ^1.7.0
firebase_auth: ^3.1.3
cloud_firestore: ^2.5.3
firebase_storage: ^10.0.5
cupertino_icons: ^1.0.2
provider: ^6.0.1
shared_preferences: ^2.0.8
validators: ^3.0.0
dartz: ^0.10.0
qr_code_scanner: ^0.6.1
connectivity: ^3.0.6
```

Listing 6: Packages used in the Crayon applications

The l10n is in information technology, often referred to as internationalising. The internationalising folder contains the files which are required to perform translations. The provider folder contains the overall providers of the applications. The route folder includes the routing information or, in other terms, the navigation of the application. The screens folder contains the view of the different screen widgets of the applications. The service folder contains the logic for outside requests to Firebase. The state folder contains information on which state a given provider currently is. More information will be available in the following subsection. The widgets folder contains reusable widgets. Widgets that are used multiple times can be found under this folder. Image 9 b displays which widgets are commonly reused in the application. To give an example, the Loading widget will be used every time an operation is performed on the database. Using such a widgets folder makes it easier for other developers to see reusable widgets. Changing the design of one of these widgets will propagate to the complete application instead of multiple times.

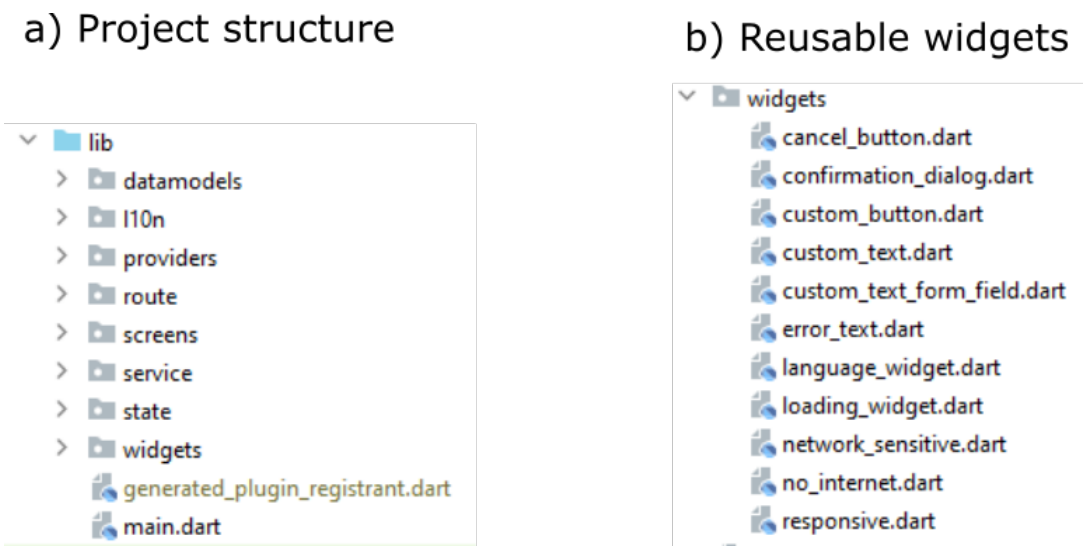

Figure 9: Crayon application folder structure

4.2 Advanced Provider implementation

The Crayon applications use a more advanced provider implementation strategy. One addition to the provider is the states. States describe in which situation the provider currently is. The visual updates in both applications are mostly done due to operations on Firestore. An example of such an operation can be retrieving lectures from Firestore. Outside operations from the application require some time to be executed. Functions that require time must be shown to the user with a loading widget. The loading state is, therefore, the first addition to the provider. On completion of the time-sensitive function, the final loaded widget should be displayed, and the provider should swap to the loaded state. Increasing the initial built speed of a screen can be done with low budget widgets. These widgets are Containers or Sizedboxes. Placing an initial widget that requires less computation time than a widget that requires data from Firebase prevents one significant error. This error occurs when the initial built function of a screen is interrupted. This interruption occurs when the widget which requires outside data calls the setstate or notifyListeners function from the provider before the initial built function was completed. This can happen when the data request is faster than the initial build. This leads the advanced provider to his final state, namely the initial state. The three primary states a provider can be coded as enumeration as shown in 7.

```
enum NotifierState { initial, loading, loaded }
```

Listing 7: Possible provider states

The complete implementation of an advanced Provider for the retrieval of a specific lecture

is shown in 8. The requirement to extend ChangeNotifier is still required to notify the consumer widget. The state variable displays in which state the provider is currently in. Each advanced provider should be in the initial state to benefit from a fast initial build and must prevent the interruption of the built function. The setState function changes the state of the provider and notifies the consumer. The lecture variable is in charge of storing the retrieved lecture from Firebase with the function getLecture. As soon as the provider requests the specific lecture, the getLecture method gets called. The first step of the function is to change the provider's state with the loading state. Afterwards, the LectureService gets called to retrieve the specific lecture. As soon as the course is retrieved, the state gets changed to loaded, and the lecture gets set.

The usage of the advanced provider over the Consumer widget can be seen in 9. Moreover, the code contains an initstate function. This initstate function is fired first before and after the built function, even if the initstate function is not completed. The initstate function contains a function named addPostFrameCallback, which allows the execution of the inner function after the initial build. By not using the addPostFrameCallback function, the provider would change the state before the built function of the screen has completed execution. Thus, the request to the Firebase will initially be postponed until the initial build is completed.

On the other hand, the Consumer widget checks which state the provider currently is with the respective if and else statements. At the start of the screen, the provider is instantiated with the initial state, and thus, an empty Container will be returned. After the first built completion, the Lecture request from the inner function of the addPostFrameCallback method will be executed. This changes the state of the provider directly to loading. The Consumer widgets gets notified by the state change and returns the loading state. Upon completing the request, the state is changed into the loaded state, and the Consumer widget returns the LectureWidget.

4.3 Theming & Translation

Theming and translations in applications are crucial to increase the applications' user experience. Students and teachers can speak multiple languages, and a good application can handle multiple languages. Theming describes in which mode the application should be displayed. These modes are commonly called dark or light modes. There are numerous ways of implementing translations and theming in Flutter applications. However, both Crayon applications use a custom solution to solve the theming and translation problem by using provider.

The Flutter team provided a new solution for translations. This solution automatically

```dart
class DetailedLectureProvider extends ChangeNotifier {
  /// Initial state of the Provider
  NotifierState _state = NotifierState.initial;
  NotifierState get state => _state;


  /// Function which allows to change the state of the provider.
   void _setState(NotifierState state) {
    _state = state;
    notifyListeners();
  }

  late Lecture? _lecture;
  Lecture? get lecture => _lecture;

  void _setLecture(Lecture? lecture) {
    _lecture = lecture;
  }
  /// Function which allows to retrieve a specific Lecture
  /// from Firebase
  void getLecture(String lectureId) async {

    /// Change the Provider state to loading
    _setState(NotifierState.loading);

    /// Retrieve data from Firebase
    Lecture lectureFromFirebase = LectureService.getLecture(lectureId)

    /// Set the lecture
    _setLecutre(lectureFromFirebase);

    /// Change the provider state to loaded
    _setState(NotifierState.loaded);
  }

}
```

Listing 8: Provider implementation for data retrieval

```
 @override
 void initState() {
   super.initState();
   WidgetsBinding.instance!.addPostFrameCallback((_) =>
       Provider.of<DetailedLectureProvider>(context, listen: false)
           .getLecture(widget.lecture.id));
 }

@override
Widget build(BuildContext context) {
   return Consumer<DetailedLectureProvider>(
               builder: (_, lectureNotifier, __) {
                 if (lectureNotifier.state == NotifierState.initial) {
                   return Container();
                 } else if (lectureNotifier.state == NotifierState.loading) {
                   return const Center(child: CircularProgressIndicator());
                 } else {
                   return LecutreWidget(lecture:lectureNotifier.lecture)
                 }
               },
             );
 }
```

Listing 9: Accessing the advanced Lecture provider

generates getter functions from arb files that contain the translations. Arb stands for Application Resource Bundle and is a JSON file on steroids specifically developed for localization. The arb files use key-value coding as a mechanism. In other words, the key is used to access the actual value. Flutter generates these getter methods to access the translation values based on the keys of the translation. The new method's benefit is that the Flutter framework prevents accessing translation values that do not exist or are null. However, the new method prevents using search by keyword. Thus, enumerations like Monday or Thursday can not be translated dynamically in the case of days. One specific example is making a get request on Firestore, which returns a day like Monday. In such a case, the developer must have if and else if statements to translate the specific Monday value. For weekdays, this results in 7 else if statements and 12 else if statements for months, which makes it a lot of code for only translating dates.

Therefore, both Crayon applications use a custom solution to resolve this issue. The first step is to create different JSON files for each language the application requires to support. Afterwards, the developer is required to make a class that allows retrieving the translated files, which can be seen in 10. The load function describes the process of importing the required file depending on the currently selected language by the user. In addition, this class also describes the supported languages of the application. Here, the custom locale is a custom class that inherits from local to override the compare to operator. This allows only to consider the language code and not the country code. This class also contains the required translation method by key.

```dart
class AppLocalizations {
  /// Locale describes the which language is currently selected
  final Locale appLocale;
  /// Specifies which languages are available
  static final List<CustomLocale> languages = [
    CustomLocale(languageCode: "en"),
    CustomLocale(languageCode: "fr"),
    CustomLocale(languageCode: "de"),
  ];
  Map<String, String>? _localizedStrings;

  AppLocalizations({required this.appLocale});

  static AppLocalizations? of(BuildContext context) {
    return Localizations.of<AppLocalizations>(context, AppLocalizations);
  }
  /// Load translated json file
  Future<bool> load() async {
    // Load JSON file from the "language" folder
    String jsonString = await rootBundle
        .loadString("assets/language/\${appLocale.languageCode}.json");
    Map<String, dynamic> jsonLanguageMap = json.decode(jsonString);

    _localizedStrings = jsonLanguageMap.map((key, value) {
      return MapEntry(key, value.toString());
    });
    return true;
  }

  // Translation by key
  String? translate(String jsonkey) {
    return _localizedStrings![jsonkey];
  }

}
```

Listing 10: Translation class file loader and search by key

The custom translation implementation can not be used directly due to the MaterialApp widget. The MaterialApp widget requires a Localisation Delegate to manage the translations. Therefore, a custom delegate class was crafted, which can be seen in 11. The AppLocalizations Delegate is used as a wrapper for the MaterialApp widget for managing the access for translated texts. The class extends the default LocalizationsDelegate and overrides two specific functions. One of them is the isSupported function which specifies which languages are supported by accessing the custom translation class from 10. In addition, the load function uses the custom translation class for loading the respective file depending on the current language of the application. Afterwards, the AppLoalization delegate can be added to the MaterialApp widget as a parameter to complete the custom translation process.

```
class AppLocalizationsDelegate extends LocalizationsDelegate<AppLocalizations> {
  const AppLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) =>
      AppLocalizations.languages.contains(locale);

  @override
  Future<AppLocalizations> load(Locale locale) async {
    AppLocalizations localizations = AppLocalizations(appLocale: locale);
    await localizations.load();
    return localizations;
  }
}
```

Listing 11: Custom App Localisation Delegate

The solution is not perfect as it currently stands since there are no checks if the translated values exist. To resolve this problem, the creation of a custom text widget is required, as seen in 12. The additional key feature to the custom text widget is the safety text. The safety text is a required parameter that is displayed in case the translation is missing. This solution has two significant benefits. It lets other developers know which texts are displayed without requiring them to check what translation comes out with the respective key in the JSON files, and there will never be access to a null value. Moreover, adding another translation now to the crayon apps only requires adding a JSON file with the respective translations and adding a custom locale to the languages array in 10.

Switching from dark theme to light theme in Flutter can be done by creating two Theme-

```dart
class CustomText extends StatelessWidget {

  /// Textcode for translation.
  final String? textCode;

  /// If the textCode is null or the transaltion of the text code is null
  /// the saftytext will be used.
  final String safetyText;

  const CustomText(
      {Key? key,
      this.textCode,
      required this.safetyText,
    }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    var appTranslation = AppLocalizations.of(context);
    String? text;
    if (textCode == null) {
      text = safetyText;
    } else {
      text = appTranslation!.translate(textCode as String);
    }
    return Text(text ?? safetyText);
  }
}
```

Listing 12: Custom text widget

Data variables inside the theme provider. One variable is for the dark theme, and the other is for the light theme. Theme Data has multiple parameters that specify the text colour font family, the application's colour theme, and more. The code snipped 13 shows the creation of such a theme. This allows streamlining the application theme to look for each individual text the same. In addition, it will enable fast changes in the design of the application since every widget is directly connected to the theme.

```
ThemeData light = ThemeData(
      fontFamily: "Poppins",
      primaryColor: Colors.orange,
      scaffoldBackgroundColor: Colors.white,
      textTheme: const TextTheme(
          headline1: TextStyle(
              fontFamily: "Poppins",
              fontSize: 42,
              fontWeight: FontWeight.bold,
              color: Colors.black),

          bodyText1: TextStyle(
              fontFamily: "Poppins",
              fontSize: 18,
              fontWeight: FontWeight.w400,
              color: Colors.black),
      ));
```

Listing 13: Theme setup

Accessing the provider as in 14 [1] allows rebuilding the whole application as soon as the theme changes. In addition, the MaterialApp widget requires to specify the available theme and put which theme is currently active 14 [2]. Finally, a function is necessary to change the theme inside the theme provider. In addition, it is preferred to store the preferred theme of the user in the shared preferences. By storing and retrieving the data from the shared preferences, the user is not required to specify the theme for every launch of the application 15.

```
@override
Widget build(BuildContext context) {
  [1] final themeProvider = Provider.of<ThemeProvider>(context, listen: true);
  final localeProvider = Provider.of<LocaleProvider>(context, listen: true);
  return MaterialApp(
      title: "Crayon",
     [2] themeMode: themeProvider.mode,
     [2] theme: themeProvider.light,
     [2] darkTheme: themeProvider.dark,
      onGenerateRoute: route.controller,
      initialRoute: route.splash,
      locale: localeProvider.getLocal,
      localizationsDelegates: const [
        AppLocalizationsDelegate(),
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
      ],
      supportedLocales: AppLocalizations.languages,
    );
}
```

Listing 14: Provider access

```
void main() async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  runApp(MultiProvider(
    providers: [
      ChangeNotifierProvider<ThemeProvider>(
          create: (_) =>
              ThemeProvider(isDarkMode: prefs.getBool("themeDark") ?? false)),
      ChangeNotifierProvider<LocaleProvider>(
          create: (_) => LocaleProvider(prefs.getString("language"))),
    ],
    child: const MyApp(),
  ));
}
```

Listing 15: Shared preference access

4.4 Exception & Validation handling

An exception in information technology is an "exceptional event" during a program's execution. Depending on the gravity of the exception, the application might crash. A study conducted by Compuware [1] states that users do not tolerate application crashes and would uninstall the application if a crash occurs. Therefore a proper exception handling solution is required to avoid uninstalls and increase the user satisfaction of the crayon applications.

Exceptions from an application can be prevented by sanitising user inputs. This means it is required to check if input fields contain the correct information. One primary example for input validation is for email form fields. These input fields require an email validation process. The validation process checks if the email contains an "@". If the user forgets to add the "@" symbol, the user should be notified that the given email is invalid. This process prevents an unnecessary request to Firestore, and an error will be thrown from firebase, which states that the email is not valid.

Flutter provides a built-in solution for sanitising user input which can be performed with the Form widget. In code snipped 16 is a custom implementation and is closely related to the default Form widget provided by Flutter. The custom form widget contains already a text field widget which allows keeping the same design over the whole application. This solution enables higher maintainability if the error requires to be displayed differently. The developer must only change it in one file instead of each input field individually. The validator is a vital function that returns an empty string if the string is valid or returns a string that de-

scribes the error. The errors have to be specified by the developer. For example, a non-valid string can be a password with less than eight characters. For grouping purposes and consistency, the different validator functions are grouped into a single class called ValidatorService. This allows the re-usability of other validator functions. Using such a validator prevents unnecessary requests to Firestore.

On the other hand, the applications require an active internet connection to work. This means the application must notify the user if he has a working internet connection. The management system does not have the connectivity feature since the web browser will automatically display a respective error screen that there is no internet connection available.

However, smartphone applications do not have this default error handling of a non-active internet connection. Therefore, the student application uses a stream provider that displays a respective icon if there is no internet connection. The stream provider allows the application to listen to the connectivity of the smartphone device continuously. The implementation of the connection provider can be seen in 17. The provider's constructor instantiates the connectivity stream and adds it to the stream controller. The Connectivity feature is not by default available in Flutter and requires to be added to the packages. The connectivity package allows checking if an internet connection is available. The function getStatusFromResult is not a necessary function, but it will enable to not be dependent on the Connectivity states. If the connectivity state name changes in the package based on an update, it would require the developer to change it in every file the connectivity package is used. However, the improved solution requires only changing it in the ConnectionProvider class. Since the ConnectionProvider requires access over the whole application, the provider must be registered at the initialisation phase of the application. Accessing stream providers data can be done the same way as a typical provider. Moreover, creating a widget for network sensitivity is required to prevent loading data without an active internet connection. The implementation of a network sensitivity widget wrapper can be seen in 18. This widget decides based on the internet connection which widget to return. The child widget parameter is the widget that requires loading data from Firestore. If there is no active internet connection, no internet widget will be displayed to the user instead of the widget that requires an active internet connection. The Network sensitivity widget was not wrapped on every widget required to have an internet connection. The reason for this is that the built-in Firestore API provides a solution to retrieve data from the cache automatically. If the user does not have a proper internet connection, the user can still see which courses he is enrolled in. If the room changes, the user will not see this change, but at least the user gets some information from the application, even if it is not the most actual data.

```dart
class CustomTextFormField extends StatelessWidget {
  /// Validator function which is used by the form widget
  /// to check if a string is valid
  /// (returns null if string is valid).
  final String? Function(String?)? validator;
  final String? labelCode;
  final String labelSafety;

  const CustomTextFormField(
      {required this.validator,
      required this.labelSafety,
      this.labelCode,
      Key? key})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    var appTranslation = AppLocalizations.of(context);
    String labelText = "";
    if (labelCode != null) {
      labelText = appTranslation!.translate(labelCode as String)
                ?? labelSafety;
    } else {
      labelText = labelSafety;
    }
    return Form(
      autovalidateMode: AutovalidateMode.onUserInteraction,
      child: TextFormField(
        validator: validator,
        style: Theme.of(context).textTheme.bodyText1,
        decoration: InputDecoration(
            border: const UnderlineInputBorder(),
            labelText: labelText),
      ),
    );
  }
}
```

Listing 16: Custom Textformfield widget

```
class ConnectionProvider {
  StreamController<ConnectivityStatus> connectionStatusController =
      StreamController<ConnectivityStatus>();
  ConnectionProvider() {
    Connectivity().onConnectivityChanged.listen((ConnectivityResult result) {
      // Convert result into Custom enum
      var connectionStatus = _getStatusFromResult(result);
      // Broadcast value
      connectionStatusController.add(connectionStatus);
    });
  }

  ConnectivityStatus _getStatusFromResult(ConnectivityResult result) {
    switch (result) {
      case ConnectivityResult.mobile:
        return ConnectivityStatus.cellular;
      case ConnectivityResult.wifi:
        return ConnectivityStatus.wifi;
      case ConnectivityResult.none:
        return ConnectivityStatus.offline;
      default:
        return ConnectivityStatus.offline;
    }
  }
}
```

Listing 17: Connection Provider

```dart
class NetworkSensitive extends StatelessWidget {
  final Widget child;
  const NetworkSensitive({Key? key, required this.child}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    var connectionStatus = Provider.of<ConnectivityStatus>(context);
    if (connectionStatus == ConnectivityStatus.wifi) {
      return child;
    } else if (connectionStatus == ConnectivityStatus.cellular) {
      return child;
    } else {
      return const NoInternet();
    }
  }
}
```

Listing 18: Network sensitive widget

Exceptions can also be thrown if an application makes an operation on the back-end. These exceptions can occur if Firestore is not available due to technical issues from Google or if the user bypasses the different checking mechanisms in the applications. To handle these errors, the standard is to handle these errors by throwing these early and catching them late. Another critical factor is not using exceptions as a form of flow control. According to a study from the Georgia Institute of Technology [11], states that some developers adopted the ignore-for now approach on the topic of exception handling. This means the developers think that exception handling is not a significant task in developing an application. However, the Crayon applications use a technique that requires the developer to handle these exceptions and is enforced by the compiler. In other words, if the developer decides to ignore the handling of an exception, the program will not run.

First of all, the dart programming language allows throwing objects without the need of extending another expectation which is relatively common in other programming languages. The code snipped 19 shows how simple the custom failure class implementation looks like. The variable code is used as a key-value for translating the error.

The code in 20 shows a deletion request to Firestore of a lecture. As most programming languages try and catch blocks, wrap the deletion request, and throw an exception provided by Dart, the Failure object is thrown with the respective error code. The magic for forcing

the developer to handle this exception is in the respective provider, which calls the function to request an operation in Firestore. The concrete implementation of this process is provided in 21. The dartz is a feature that is not directly provided in the Flutter framework and must be added. The dartz package allows adding functional programming into Dart. Functional programming is well known for its terseness and relatability. This package introduces types like Either and Task, which makes handling asynchronous errors easier. An operation on a database can either be a success or a failure. The Either type from dartz allows precisely this; it will enable a variable of type either to be a failure or a success, the return type of the API request. This means that in case of an exception on an operation on the Firestore database, the variable becomes a failure. In case of success, it becomes the returning type of the function performing the operation on the database. Afterwards, to access either type, the developer must fold the value. This means he is required to manage both outcomes. In case of a successful operation, both applications use SnackBars to display the operation's success. The Crayon student application shows SnackBars in case of database failures, and the crayon management application displays the texts due to the bigger screen size. A snack bar in applications is a bar that appears at the bottom of the screen, which shows a message for a given amount of time defined by the developer. This approach ensures the maintainability and the survivability of the application during errors by introducing the compiler as a helping factor to the developer to not forget to handle exceptions.

```dart
/// Failure is used as an exception.
class Failure {
  final String code;
  Failure({required this.code});

  @override
  String toString() => code;
}
```

Listing 19: Failure as an exception

```dart
/// returns a boolean if the deletion was successful.
/// Throws a Failure in case of an managed error.
Future<void> removeLecture(String lectureId) async {
  try {
    if (_auth.currentUser != null) {
      return FirebaseFirestore.instance
          .collection("users")
          .doc(_auth.currentUser!.uid)
          .update({
        "enrolled-lectures": FieldValue.arrayRemove([lectureId])
      });
    }
    throw Failure(code: "not-logged-in");
  } on FirebaseException catch (e) {
    if (e.code == "network-request-failed") {
      throw Failure(code: "no-internet");
    }
    throw Failure(code: "firebase-exception");
  } on SocketException {
    throw Failure(code: "no-internet");
  } on HttpException {
    throw Failure(code: "not-found");
  } on FormatException {
    throw Failure(code: "bad-format");
  }
}
```

Listing 20: Deletion request to Firestore of a Lecture

```
void removeLecture(String lectureId) async {
  /// Change visual my showing loading indicator.
  setState(NotifierState.loading);

  var result = await dartz.Task(() => api.removeLecture(lectureId))
      .attempt()
      .map(
        (either) => either.leftMap((obj) {
          try {
            return obj as Failure;
          } catch (e) {
            throw obj;
          }
        }),
      )
      .run();

  result.fold(
      (failure) => CustomSnackbar(
            text: failure.code,
            isError: true,
            context: context,
            saftyString: "Failed to remove lecture",
          ).showSnackBar(), (_) {
    CustomSnackbar(
      text: "lecture-removed-sucess",
      isError: false,
      context: context,
      saftyString: "Successfully removed lecture",
    ).showSnackBar();
    _user!.enrolledLectures.remove(lectureId);
  });

  /// Change state to loaded
  setState(NotifierState.loaded);
}
```

Listing 21: Deletion of a Lecture inside the Provider

5 Results

The following subsections describe how the final applications look and what their different functionalities are. In addition, it contains my personal opinion about developing applications in Flutter.

5.1 Crayon student

Initially, a splash screen is shown to the user when the application starts. The objective of such a screen is to hide the loading process of opening an application. In native development, only one splash screen requires to be provided. However, Flutter requires two such screens. This is due to the initial loading of the app as in native and one for the Flutter engine at the start. In addition, the application requires a login screen. The user can change the language and switch the app into light or dark mode at this screen level. Both screens can be seen in 10.

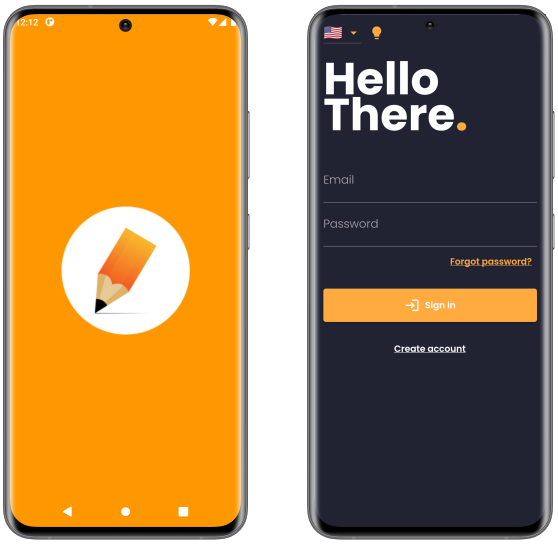

Figure 10: Splash and Login Screen

Figure 11 shows the dashboard and has following functionalities:

- Join/Delete a lecture
- Navigation
- Settings

- Join quiz
- Ask question

Joining a lecture can be done over the QR-code button. This button allows to open the camera mode and requires the student to scan the lecture QR-code. As soon as the user joins a lecture, he can also remove it. To remove a lecture, the user requires to long click the lecture. The navigation was specially designed for this applications. The navigation is subdivided by the days of the week. The navigation can be seen as a day filter. At the start of the application, the navigation directly goes to the current day to see which lectures a student has for that day. Joining a quiz can only be achieved if the management application allows it. By pressing on the lecture, the user can ask a question to the teacher. The settings icon allows opening the settings menu, which helps to change the language and theme mode.

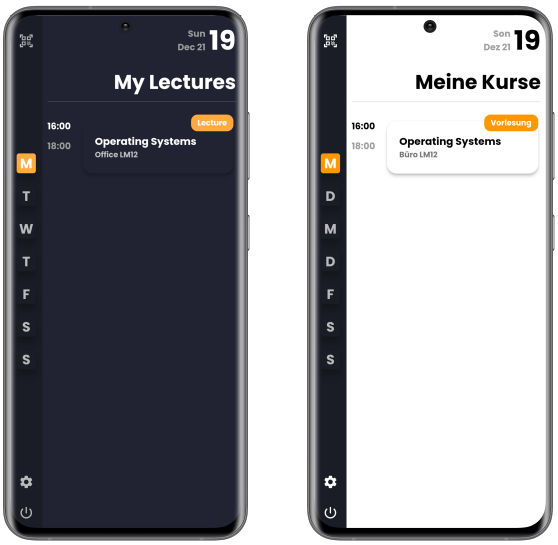

Figure 11: Dashboard

The screens in 12 show the quiz process of the crayon student application. The First screen shows how a question is displayed and how much time remains for the current quiz. The application will automatically transition to the new question when the student clicks on a response. During this transition, the user gets notified if the answer is right or wrong. As soon as all the questions are completed, a final score will be displayed 12. As mentioned in the implementation chapter, only one request is made to Firestore. Moreover, the video [2]

[2]Question asking process in video format `https://github.com/SchroederLionel/CrayonVideos/blob/main/2022-02-08%2016-58-51.mp4`.

describes visually how a student can ask a question and how the management system notifies the teacher that a question was asked by a student.

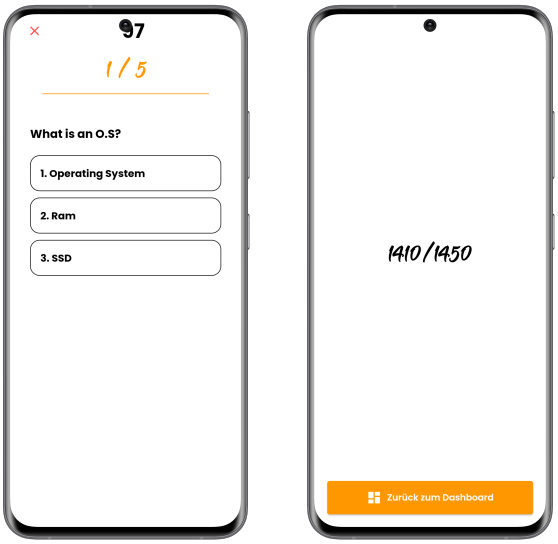

Figure 12: Quiz screens

5.2 Crayon management

The Crayon management application is currently a web page but can also be compiled into an executable. As in the student application, the management application also has a login as shown in 13. In addition, the dashboard shows which lectures the teacher has created 13. The dashboard also has the functionality to change the language and theme mode. The presentation mode of the application is shown in 14.The presentation screen is also in charge of handling the questions requested from a student. The yellow icon in the bottom left describes that a student has asked one question. This icon shows up in real-time soon as a student asks a question. The presentation screen can also work as a drawing board. The drawing board can display the current PDF page and draw over it for further explanation.

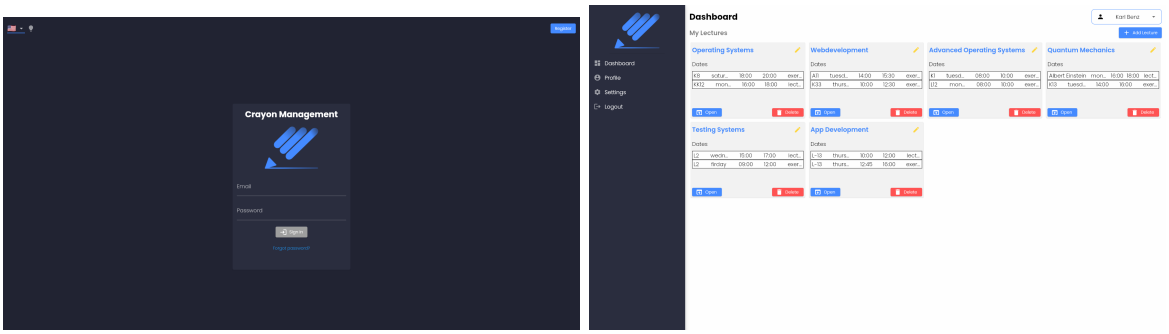

Figure 13: Login & Dashboard

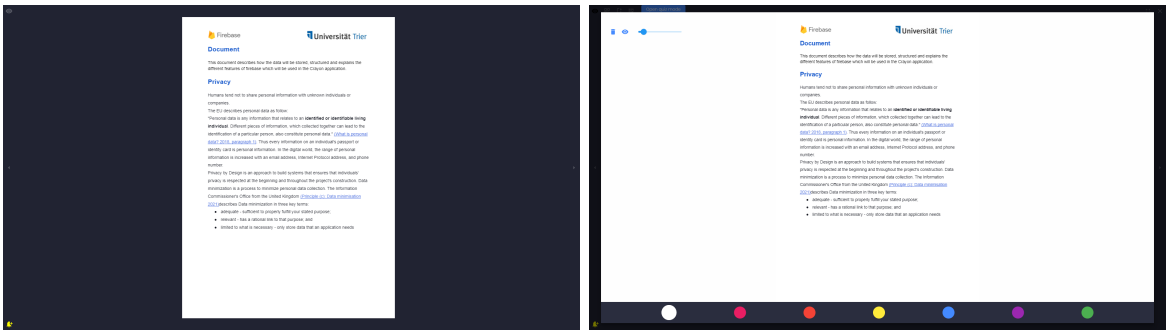

Figure 14: Presentation & Drawing mode

Moreover, the teacher can also start a quiz. Starting a quiz is a multiple-step process where no step can be omitted and is shown in illustration **??**. These steps are the following and are taken from the sequence diagram 4:

- Select quiz
- Set time for the quiz
- Lobby
- Timer
- Results
- Explanation

The first step is to select the quiz that the teacher wants to start. The time step describes how long a quiz should take. As soon as this time is over, the students will be kicked out of the quiz, and their answers will be automatically sent to Firestore. The third step is the

lobby step. The management application creates a lobby for the students/ quiz participants and notifies the student's application that a quiz has started. The lobby step also shows which students joined the quiz. As soon as the teacher decides that the quiz should start, the quiz participants will be placed into the quiz mod as shown in 12. The timer step is a simple countdown of the selected time. The teacher can early close the quiz or requires to wait until the timer is over to show the scoring board at the result step. The scoring board shows the five best players of the quiz. Finally, the teacher can explain the quiz. The explanation process also shows how many quiz participants answered the questions right or wrong. The video [3] shows an exmaple execution of a quiz with both applications.

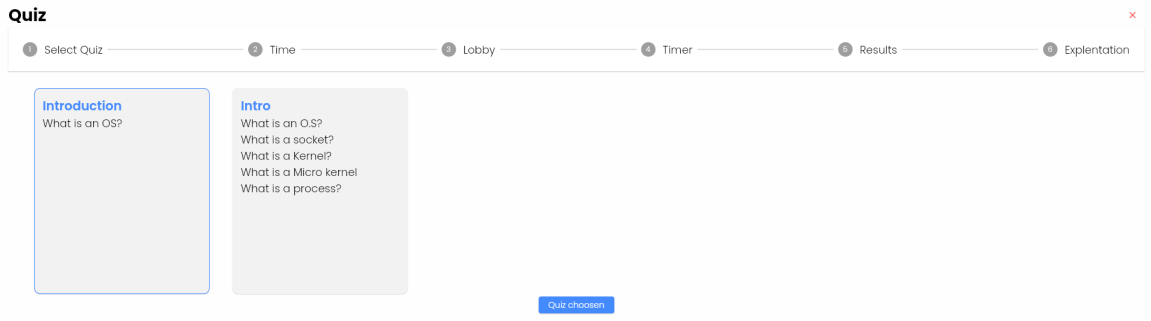

Figure 15: Start quiz

5.3 Future Work

Increasing the features in both applications is necessary to have an advantage against other competitors. One significant addition to the Crayon management application is implementing a new PDF storage system. This System should allow the teacher to add his drawing to the current PDF. At the current state of the application, this is not possible the drawing will be discarded except if the teacher makes a screenshot and adds it manually to the PDF. In addition, the teacher should be able to create home works in the management application. The homework task can easily be added to both applications. The Student application already filters by the date due to the navigation, which allows displaying the different home works without adding much code.

[3] Quiz process in video format `https://github.com/SchroederLionel/CrayonVideos/blob/main/Quiz_Process.mp4`.

6 Conclusion

Education is a key to the success of an individual. Improving the education process with game mechanics enhances the willingness of a student to listen to a lecture. These game mechanics are commonly known as quizzes. In addition, allowing a student to ask a question anonymously might take away the fear of asking a question during a lecture.

The teachers use the Crayon management application and create the different quizzes for a lecture and send them to the student's application. In addition, the application has an integrated presentation system that is also in charge of displaying the anonymously asked questions by students. The Crayon student applications consume the data from the management application. This includes the lecture content and quiz data to participate in the different quizzes a teacher wants to start.

Flutter and Firebase provide a great solution to decrease the development time of an application for multiple platforms. Flutter is a dynamic language currently which gets many updates. During the project development, the Flutter team provided three new Flutter versions. This means that Flutter code can faster deprecate than other programming languages. These dynamics require the developer to decide which external packages the applications require. Packages can be based on old dart code instead of the new one. One prominent case of drastic code base changes was the introduction of the Null Safety feature in Flutter in March 2021. These significant changes made old packages unusable, except the developer disabled this feature. However, the Null Safety Feature was a great addition to Flutter to remove the standard errors of accessing null values. In addition, the developer has to consider not using packages for a specific platform. These packages would eliminate the ability to create applications for multiple platforms.

On the other side, Firebase is a complete and fully functional product. Removing the requirement of writing a back-end is an excellent addition that speeds up developing an application. Using Firestore as a database system requires a proper data modelling procedure. Firebase charges the user per document operation and not by bandwidth or storage. Taking this to our advantage by creating redundancy in the different Documents allows reducing the costs of Firestore. Using Firestore as a back-end can not be installed on hardware other than Google's, thus making the applications dependent on Google services. However, an additional benefit is the scalability of Firestore.

Developing complex Flutter applications such as the Crayon applications requires a proper state management solution. State management allows rebuilding only the necessary widgets instead of the whole screen to increase their performance. Moreover, Flutter application requires an adequate architecture to not mismatch the logic in the application view. This separation of concerns allows increasing the maintainability of the application.

References

[1] Compuware. Users have low tolerance for buggy apps. https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/?guccounter=1, 2013.

[2] Mehmet Akif Ersoy. The effects of pre-lecture online quizzes on language students' perceived preparation and academic performance. Jan 2017.

[3] Google. Dart overview. https://dart.dev/overview, Oct 2018.

[4] Google. Using packages. https://docs.flutter.dev/development/packages-and-plugins/using-packages, Dec 2020.

[5] Google. Dart overview. https://dart.dev/null-safety/understanding-null-safety#:~:text=For%20us%2C%20in%20the%20context,some%20runtime%20checks%20involved%20too., Mar 2021.

[6] Google. Support different platform versions. https://developer.android.com/training/basics/supporting-devices/platforms, 2021.

[7] HYPR. New password. https://blog.hypr.com/hypr-password-study-findings, Oct 2019.

[8] GSMA intelligence. More than 5 billion people in the world own mobile devices. https://leftronic.com/blog/smartphone-usage-statistics/#:~:text=According%20to%20GSMA%20real%2Dtime,35.13%25%20of%20the%20world%27s%20population., Feb 2021.

[9] Henry Roediger, Adam Putnam, and Megan Sumeracki. *Ten Benefits of Testing and Their Applications to Educational Practice*, volume 55, pages 1–36. Jan 2011.

[10] Perry Samson. Can giving students anonymity help them engage in class? https://www.insidehighered.com/digital-learning/article/2019/12/06/students-may-benefit-anonymous-back-channel-communications, Juin 2019.

[11] Hina Shah, Carsten Görg, and Mary Harrold. Why do developers neglect exception handling? pages 62–68, 01 2008.

[12] Statcounter. Mobile android version market share worldwide. https://gs.statcounter.com/android-version-market-share/mobile/worldwide/, 2021.

[13] Statcounter. Mobile operating system market share germany. https://gs.statcounter.com/os-market-share/mobile/germany, Dec 2021.

[14] Statista. Most used programming languages among developers worldwide. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/, 2021.

[15] EUROPEAN DATA PROTECTION SUPERVISOR. Data minimization. https://edps.europa.eu/data-protection/data-protection/glossary/d_en, Oct 2018.