



INSTITUTO POLITÉCNICO DE LISBOA
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

PostChat
"Bring Back Postcards"

António Carvalho n.º 48347

e-mail: a48347@alunos.isel.pt

Supervisor: Nuno Leite

Report

Projeto e Seminário - June 2023

Abstract

The digital postcard application developed in this project aims to bring back the charm of sending postcards in the digital age. With the rise of instant messaging and social media, the traditional practice of sending postcards has diminished. However, we believe that there is still value in the personal touch and sentimentality that postcards offer, and our application seeks to bridge the gap between generations and provide a modernized yet traditional means of communication.

By leveraging technology, we have created a platform where users can create, personalize, and send digital postcards to their loved ones. The application captures the essence of traditional postcards by providing customizable templates, allowing users to add their own messages and images. Users can unleash their creativity and design unique postcards that reflect their personality and emotions.

The application is built on a client-server architecture, with the Android application serving as the client and a web *Application Programming Interface* (API) as the server. This architecture allows for seamless communication between the user's device and the server, enabling quick and reliable delivery of postcards. The use of modern technologies, such as Android Compose, ensures a smooth and visually appealing user interface, enhancing the overall experience.

Through this project, we aim to address the generation gap by providing a platform that appeals to both younger and older generations. The digital nature of the postcards caters to the tech-savvy younger generation, while the traditional sentimentality of the practice resonates with older generations who appreciate the nostalgic value of postcards.

In conclusion, this project showcases the power of technology to revive and adapt traditional practices for the modern era. By combining the convenience of digital communication with the emotional connection of handwritten postcards, our application offers a unique and meaningful way to stay connected with loved ones. As we continue to refine and expand this project, we hope to inspire more people to embrace the beauty of

postcards and foster meaningful connections in an increasingly digital world.

Keywords: Written Postcards, Digital Postcards, Generation Gap, Client-Server Application, Android Compose.

Acronyms

AI Artificial Intelligence

API Application Programming Interface

HTR Handwriting Text Recognition

HTTP Hypertext Transfer Protocol

JVM Java Virtual Machine

PNG Portable Network Graphics

RDBMS Relational Database Management System

REST REpresentational State Transfer

SMS Short Messaging Service

SVG Scalable Vector Graphics

TTS Text To Speech

Contents

Abstract	iii
Acronyms	v
List of Figures	xi
1 Introduction	1
2 State of the art	3
3 Requirements	5
3.1 Functional Requirements	6
3.1.1 Mandatory Requirements	6
3.1.2 Optional Requirements	6
3.2 Overview	6
3.2.1 Project Architecture	6
3.2.2 Signing In	7
3.2.3 Connecting to Friends	7
3.2.4 Obtaining Templates	7
3.2.5 Creating Chat Groups	7
3.2.6 Sending Messages (Postcards)	8
3.2.7 Handwriting Text Recognition	8
4 Web API	9
4.1 Implementation	11
4.1.1 Profiles	12
4.1.2 Interceptors	12

4.1.3	Controllers	13
4.1.4	Services	13
4.1.5	Repository	14
4.1.6	Transaction Manager	14
4.1.7	Error Handling	16
4.2	Postcard	16
4.2.1	Message	17
4.2.2	Messaging Flow	17
4.2.3	SVG Manipulation	18
4.3	Database Representation	19
4.4	Application Runner and Storing Templates	20
4.5	Verifying Phone Numbers	20
4.6	Performing HTR	21
5	HTR Model	23
5.1	Introduction	24
5.1.1	Offline HTR	24
5.1.2	Online HTR	25
5.2	Implementation	25
5.2.1	Why Offline HTR	25
5.2.2	Pipeline	26
5.2.3	Detector	26
5.2.4	Recognizer	27
5.2.5	Natural Text Ordering	31
5.3	Limitations	32
6	Client	33
6.1	Android System and Compose Framework Overview	34
6.1.1	Android Manifest	34
6.1.2	Android Activity	35
6.1.3	Data Storing	35
6.1.4	ViewModel	36
6.1.5	Canvas	36
6.1.6	Making HTTP Request	36
6.2	Activities	37
6.2.1	Permissions	37
6.2.2	Sign In	38

6.2.3	Home	42
6.2.4	Create Chat	44
6.2.5	Chat View	45
6.2.6	Draw Postcard	48
6.2.7	View Postcard	50
7	Conclusions and Future Work	53
	Appendices	55
A	Other Definitions	57
A.1	Technologies	57
	References	59

List of Figures

3.1	Project Architecture	7
3.2	Application Flow	8
4.1	PostChat API endpoints.	10
4.2	API Structure.	11
4.3	Authentication Interceptor.	12
4.4	Transaction Manager.	15
4.5	Postcard Representation.	17
4.6	Postcard POST Flow	18
4.7	Postcard GET Flow	19
4.8	SVG Merging	19
4.9	Entity Association Model	20
5.1	HTR System	26
5.2	Detection Flow	27
5.3	Recognizer Flow	28
5.4	Postcard Main Sections	31
5.5	Average height calculation for left section	32
6.1	Android Activity Navigation Graphic	34
6.2	Permission Activity	37
6.3	Prompt Permission Activity	38
6.4	Signin Activity	39
6.5	Signin Activity invalid password size	40
6.6	Signin Activity missing invalid password	41
6.7	Home Activity	43
6.8	CreateChat Activity	44

6.9	CreateChat Activity name prompt	45
6.10	Chat Activity	46
6.11	Chat Activity bottom templates list	47
6.12	Chat Activity pick template	48
6.13	CreateChat sent postcard	49
6.14	Draw Activity	50
6.15	Draw Activity draw	51
6.16	Postcard View	52

Chapter 1

Introduction

PostChat is an innovative Android application that aims to bring back the charm of postcards and bridge the gap between the older and younger generations through the use of modern technology. The app serves as a client-server platform that enables users to create, personalize, and send digital postcards to their loved ones, fostering meaningful connections in today's digital age.

In today's fast-paced world, people have become increasingly reliant on technology to communicate with their friends and family. However, the traditional art of sending postcards has lost its appeal over the years, particularly among the younger generation. With PostChat, we strive to revive this age-old practice and make it relevant again, while also making it accessible and convenient to use for everyone.

The PostChat application operates on a client-server architecture, where the Android app serves as the client and a robust web API functions as the server. This design allows for seamless communication between users and the server, enabling the creation and transmission of digital postcards with ease.

The PostChat app is meticulously designed to cater to the needs of all age groups, with a user-friendly interface that is easy to navigate. Users have access to a diverse range of postcard templates, allowing them to personalize their messages and express their creativity. Additionally, the app integrates advanced features, such as AI tools tailored for the visually impaired, ensuring an inclusive and empowering experience for all users.

Furthermore, PostChat goes beyond the confines of traditional postcards, as it facilitates real-time communication and community building. Users can create chat groups within the app, enabling them to engage in conversations and share postcards with multiple recipients simultaneously. This feature encourages meaningful interactions and strengthens connections among friends and family.

This comprehensive report aims to delve deeper into the workings of the PostChat application, analyzing its features, design, usability, and the integration between the Android app and the web API server. By examining its architecture and functionalities, we aim to provide a holistic understanding of PostChat's unique capabilities and its potential to revolutionize the way people connect and communicate with each other in today's digital age.

Through the harmonious blend of modern technology and the timeless concept of postcards, PostChat stands as a testament to innovation and human connection. By bridging generational gaps and embracing the power of digital communication, PostChat offers a modernized yet nostalgic means of fostering relationships and preserving the cherished tradition of sending heartfelt messages.

Chapter 2

State of the art

In today's digital landscape, several messaging apps compete to provide users with a platform for connecting and sharing personalized messages. This chapter aims to compare PostChat with messaging and postcard apps such as *WhatsApp* (2023), *Telegram* (2023), and *MyPostcard* (2023), highlighting PostChat's state-of-the-art qualities and distinct features over these apps. By analyzing their design, usability, and user experience, we can gain insights into PostChat's unique position, distinct idea, and what it borrows from its contemporaries.

PostChat, much like WhatsApp or Telegram, prioritizes a user-friendly design and intuitive navigation. Its interface is simple and streamlined, ensuring a seamless postcard creation and sending experience for users of all ages. Also like those apps, it allows to create groups but instead of sending messages it sends postcards.

Unlike WhatsApp or Telegram, PostChat is focused on sending postcards and has a set of simple and straight forward drawing tools, high quality postcard templates and AI assisting tools that distinguish it from those other apps.

MyPostcard has a different approach as it allows users to fully customize a postcard via a digital interface but send the postcard via physical mail, that is, no chat groups, no way to find friends in the service, just plain address form filling. PostChat approach allows for both formats as it send the digital postcards and can export them in order to printed onto physical paper.

In conclusion *PostChat* borrows elements from different apps and makes a unique format for itself opening doors for a new way of communication and a new possible business opportunity.

Chapter 3

Requirements

Contents

3.1	Functional Requirements	6
3.1.1	Mandatory Requirements	6
3.1.2	Optional Requirements	6
3.2	Overview	6
3.2.1	Project Architecture	6
3.2.2	Signing In	7
3.2.3	Connecting to Friends	7
3.2.4	Obtaining Templates	7
3.2.5	Creating Chat Groups	7
3.2.6	Sending Messages (Postcards)	8
3.2.7	Handwriting Text Recognition	8

3.1 Functional Requirements

In this section we will delve in to the requirements needed to implement this project.

3.1.1 Mandatory Requirements

The mandatory functional requirements are the following:

- Editing and personalizing a postcard
- Sending and receiving postcards;
- Creating chat groups;
- Connecting to all friends in the service via user's stored phone number;
- Export a postcard in high quality;
- Build and train a Handwriting Text Recognition model.

3.1.2 Optional Requirements

- Use a *Text To Speech* (TTS) implementation to read the postcard out loud;
- Using *Short Messaging Service* (SMS) based verification;

3.2 Overview

PostChat has simple and intuitive functionalities. In this chapter is shown a overview of the service, detailed explanation of every component will be provided later in this report.

3.2.1 Project Architecture

PostChat has a client made for Android using Jetpack Compose framework a server programmed in Kotlin using the Spring framework and running in the *Java Virtual Machine* (JVM). The same server creates processes to call the *Artificial Intelligence* (AI) *Handwriting Text Recognition* (HTR) model made in Python and uses a PostgreSQL database to store data.

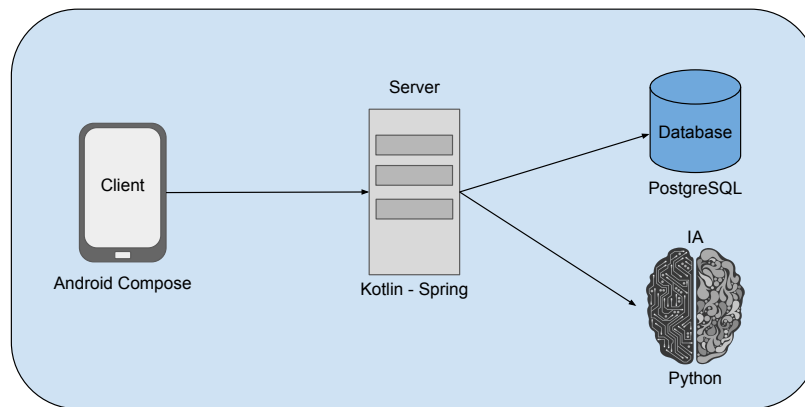


Figure 3.1: Project Architecture

3.2.2 Signing In

To be able to use any of the provided functionalities its necessary for the user to sign in the service. Signing In can be either a login operation or a register operation. Its required for the user to have a valid phone number in order to perform either one of the operations. Once registered the user will have access to all of the service functionalities.

3.2.3 Connecting to Friends

Connecting to friends in the service is simple, no need to find for a specific username simply give access to the user's phone numbers and the service will provide you with the contacts registered in it.

3.2.4 Obtaining Templates

A template represents a potential postcard background. Every application developed to use this Web API needs to have consistency, that is, it needs to keep the templates up-to-date with the ones stored in the server.

3.2.5 Creating Chat Groups

Once registered the user can now create chats groups. These chat groups are made up of all the people the users is meant to send a postcard and/or receive from. When sending a postcard to someone a chat group is required.

3.2.6 Sending Messages (Postcards)

A message is referenced in this report and throughout the application development as a holder for the postcard. A message contains more information besides the postcard itself (we will delve deeper about messages in the report 4.2.1). Sending a postcard involves choosing a postcard template (the background of the postcard) and writing to it. Internally the postcard is represented by two layers, the template and the written content, both of which are SVG's *W3C (2023)*, allowing for high quality postcards.

3.2.7 Handwriting Text Recognition

For each message a request can be made to extract the drawn text contained in it to computer digits. To perform such operation only the drawn content is needed.

Application Flow The application flow can be seen in Figure 3.2.

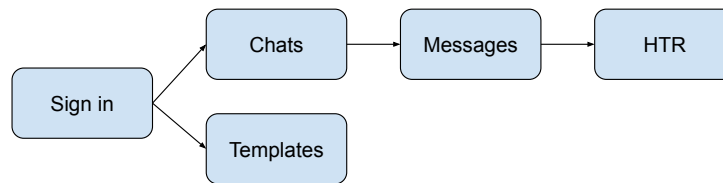


Figure 3.2: Application Flow

Web API

Contents

4.1	Implementation	11
4.1.1	Profiles	12
4.1.2	Interceptors	12
4.1.3	Controllers	13
4.1.4	Services	13
4.1.5	Repository	14
4.1.6	Transaction Manager	14
4.1.7	Error Handling	16
4.2	Postcard	16
4.2.1	Message	17
4.2.2	Messaging Flow	17
4.2.3	SVG Manipulation	18
4.3	Database Representation	19
4.4	Application Runner and Storing Templates	20
4.5	Verifying Phone Numbers	20
4.6	Performing HTR	21

The web API follows a *REpresentational State Transfer* (REST) approach over *Hyper-text Transfer Protocol* (HTTP) which allows for a standardized and scalable architecture for building robust and interoperable systems.

The REST API offers 1) Simplicity - as it leverages existing HTTP methods and status codes, making it easy to understand. 2) Scalability - designed to be stateless, meaning each request contains all the necessary information to be processed independently. This allows for horizontal scaling so we can add more servers to handle increased traffic without impacting the functionality of the API. 3) Flexibility - Uses JSON format for responses, which is widely supported and easy to parse in various programming languages. This flexibility makes it easier for clients to consume the API and integrate it into their applications.

In the current version of the API a overview of the endpoints is represented in Figure 4.1.

Group	Description	Operation	Uri
Home	Register on the API	POST	/register
	Login	POST	/login
	Logout	POST	/logout
User	Get users registered in the API	GET	/user?phoneNumbes=[]
	Get user information	GET	/user/{id}
Chat	Get all of user's chats	GET	/chat
	Create a chat	POST	/chat
	Send a message to a specific chat	POST	/chat/{id}
	Get a information from a chat (users and chat info)	GET	/chat/{id}
Message	Get all messages (postcards)	GET	/message
Template	Get all templates (postcard backgrounds)	GET	/template
HTR	Perform HTR operation to postcard	POST	/htr

Figure 4.1: PostChat API endpoints.

4.1 Implementation

Structure The API uses Spring MVC framework, is written in kotlin and runs in the JVM. The structure of the API is described in Figure 4.2

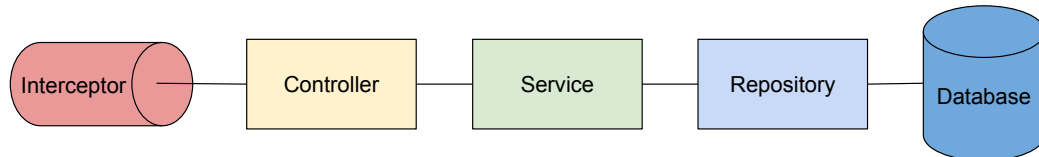


Figure 4.2: API Structure.

Spring simplifies the process of implementing a Web API by providing a comprehensive and feature-rich framework that abstracts away much of the complexity associated with building APIs. The main benefits of using Spring in this project are:

- **Dependency Injection:** Spring's dependency injection (DI) container allows for easy management and wiring of components within the API. By leveraging DI, developers can declaratively define and inject dependencies into their API classes, promoting loose coupling and enhancing testability.
- **Inversion of Control (IoC):** Spring's IoC container takes care of instantiating and managing the lifecycle of API components, relieving developers from manually managing object creation and destruction. This simplifies the development process and helps maintain a consistent and predictable application state.
- **Annotation-driven Development:** Spring encourages the use of annotations for defining API endpoints, mapping request methods, and handling request parameters.
- **Exception Handling:** Spring offers robust exception handling mechanisms, allowing developers to define custom exception handlers that gracefully handle and transform exceptions into appropriate HTTP responses. This simplifies the process of handling errors and communicating meaningful error messages to API clients.
- **Security and Authentication:** Spring Security offers a comprehensive set of features for implementing authentication and authorization mechanisms in an API. It provides declarative security configurations, authentication providers, and authorization rules, simplifying the implementation of secure APIs.

A request flow is defined in the Figure 4.2 in order from left to right.

4.1.1 Profiles

Spring Profiles are used to simplify configuring the API for production and testing. Several components are annotated with *Profile(<profile name>)* which determines what component to use for each profile. One good use for the Profiles is to determine which database source to use depending on the profile name chosen (for production or testing).

4.1.2 Interceptors

An interceptor is a component that has the ability to intercept and process requests and responses, it provides a way to customize the behavior of the request/response processing pipeline. In the PostChat API three interceptors are implemented:

- **Logger Interceptor** - The logger interceptor is used for every request/response, it's use is mainly for logging and debug. For every request, a entry is logged containing information about the request's method and URI and the response's status code;
- **Content Interceptor** - The content interceptor guaranties that every request body has application-json Content-Type;
- **Authentication Interceptor** - The authentication interceptor is a crucial part of the API it automates the authentication process by parsing and verifying the token cookie or the Bearer-Token header.

Authentication Interceptor

The server generates a session token when login-in or registering. The function of the authentication interceptor is to verify this token. The authentication interceptor intercepts every request that is processed by a handler containing a parameter annotated by the *Authenticate* java annotation. Figure ?? illustrates a request needing authentication.

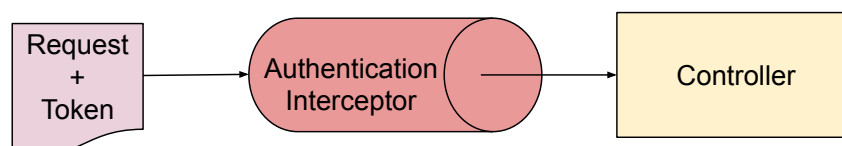


Figure 4.3: Authentication Interceptor.

A handler having a parameter annotated by the *Authenticate* is marked for an authenticated operation, that is, only users signed in the application are supposed to use these operations, an implementation of the annotation and a controller handler is shown below.

```
@Target(AnnotationTarget.VALUE_PARAMETER)
annotation class Authenticate

@GetMapping
@ResponseStatus(HttpStatus.OK)
fun getChats(@Authenticate user: User): ChatList =
    service.getChats(user.phoneNumber)
```

4.1.3 Controllers

A controller is a key component in Spring MVC that plays a crucial role in handling HTTP requests and generating appropriate responses. Acting as the middleware between the user interface and the business logic (service), the controller facilitates the flow of data and interactions within the web application.

A controller class encapsulates a collection of handler methods, each of which defines a specific HTTP operation (such as GET, POST, PUT, DELETE) and its corresponding URI. These handler methods are responsible for processing the incoming requests and executing the necessary business logic.

By mapping URLs and HTTP methods to the appropriate handler methods, the controller determines which code should be executed to handle a particular request. This mapping is typically defined using annotations like `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc., which allow developers to specify the desired URL patterns and HTTP methods for each handler method.

4.1.4 Services

Services refer to the components responsible for implementing the business logic and functionality required to fulfill the API's purpose. Services encapsulate specific operations and actions that can be performed on the data.

Services contain the implementation of the business logic that defines how the API handles and processes requests. They encapsulate the specific operations, algorithms, and rules necessary to perform the desired functionality.

Services interact with the database to retrieve, modify, or persist data required by the API. They may encapsulate data access logic and perform CRUD (Create, Read, Update,

Delete) operations on the underlying data. Are responsible for handling data validation, transformation, and manipulation to ensure data integrity and consistency. The interaction with the database is done using JDBI, which is a top level implementation of the JDBC library.

For input data validation that doesn't require database operations, a object *Check* was implemented. Every method defined in the object was implemented following a blockchain pattern, that is, every method return a reference to the same object leading to the ability to call successive methods in a chain.

4.1.5 Repository

A repository is a component that encapsulates the logic for interacting with data storage systems, such as databases, file systems, or external services. It provides a set of methods and operations to perform common data operations, such as retrieving, creating, updating, and deleting entities. The primary purpose of a repository is to abstract the complexities of data access and provide a clean, domain-specific interface for the business logic to interact with the data.

The repository component abstracts the underlying data storage by providing a consistent and unified interface. It shields the business logic from the specifics of the data storage system, such as the query language or the data access technology being used. By encapsulating data access logic within the repository, it promotes modular and reusable code, allowing the business logic to be decoupled from the implementation details of the data storage.

Repository offers a set of standard data access operations, such as querying entities based on specific criteria, inserting new entities, updating existing entities, and deleting entities from the data storage. These operations are implemented within the repository, utilizing appropriate data access technologies and SQL queries, to interact with the data storage efficiently. By providing a consistent interface for these operations, the repository simplifies the implementation of data access and persistence logic.

Queries were written in raw format so all information about a specific query can be found in the method that implements the repository operation.

4.1.6 Transaction Manager

For database operations a transaction manager is defined. A transaction manager, as the name suggests, manages transactions made in the repository component by encapsulating instances of the repositories and exposing them to the Service in a lambda function. It

automatically closes and opens connections to the database. The Figure 4.4 illustrates how the transaction manager works.

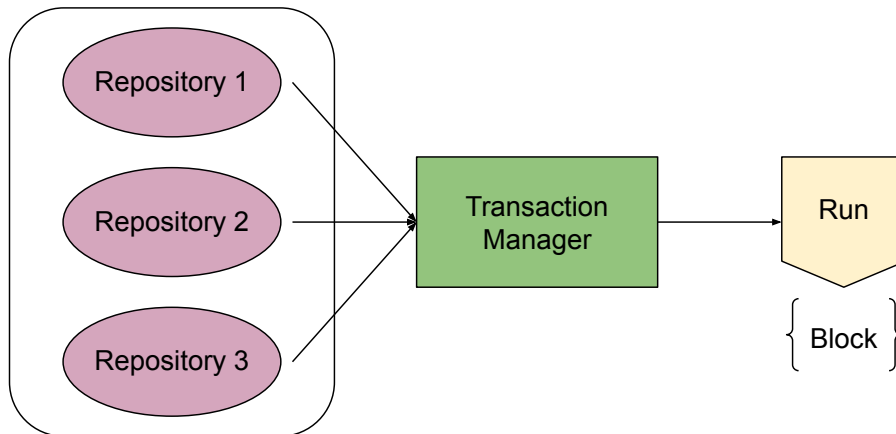


Figure 4.4: Transaction Manager.

The transaction manager receives the services instances and passes transaction handlers to it. The manager offers a singular method called *run*, this method exposes the repositories and encapsulates the transaction.

4.1.7 Error Handling

Errors are handled in the application by using a *ControllerAdvice* annotated class called *ErrorHandler*.

There are API specific defined Exceptions that are thrown when an error occurs. The error responses are returned in a body with application/problem+json Content-Type.

The *ControllerAdvice* annotation is used to define a global exception handler. This class plays a crucial role in centralizing the error handling logic across multiple controllers within the application. It intercepts and handles exceptions that occur during request processing, providing a consistent approach to handling errors.

The *ErrorHandler* has a method called *handleException* that is annotated with `@ExceptionHandler(Exception::class)`, indicating that it is responsible for handling any uncaught exceptions that occur within the application. This method takes the exception as an argument and returns a *ResponseEntity* wrapping a *ProblemJSON* object.

For any other unhandled exceptions, a default response entity is created with an internal server error status code and a problem type detail that includes the exception message. This generic error handling ensures that unexpected exceptions are appropriately handled, preventing the exposure of sensitive information to API clients.

By utilizing the `@ControllerAdvice` annotation and implementing a centralized error handling approach, this code enhances the robustness and reliability of the web API. It ensures consistent error responses across all controllers, simplifies exception handling logic, and promotes maintainability of the application. Additionally, the logging of exceptions aids in debugging and monitoring the application's health.

4.2 Postcard

A postcard is represented as a combination of layers:

- Handwritten content - Represents the text (paths) drawn by the user as a SVG encoded in a base 64 string.
- Template - Represents a postcard background as a SVG encoded in a base 64 string.

This approach contributes for smaller sized SVG's when sending a message and improved results from the Detector in the AI pipeline.

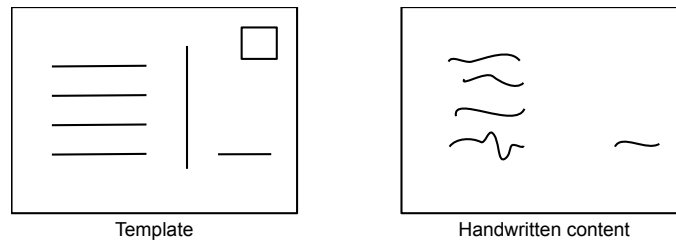


Figure 4.5: Postcard Representation.

4.2.1 Message

A message is a data structure that connects a postcard to a chat element. It is the holder of the postcard information and contains all necessary information to perform further operations in the API. Its composition is:

- The phone number of the user that sent the postcard;
- The message id;
- The chat id;
- The handwritten content;
- The merged content 4.2.3;
- The template name;
- The timestamp of when it was created.

4.2.2 Messaging Flow

After creating a chat user can now send a postcard, to send a postcard there are three things needed 1) the name (identifier) of the template used, 2) the handwritten SVG in base 64 string and 3) the timestamp of when it was sent. The Figure 4.6 illustrates the messaging send flow.

The server responds back with a body containing the saved message. The same response body applies for the list of messages when getting all messages.

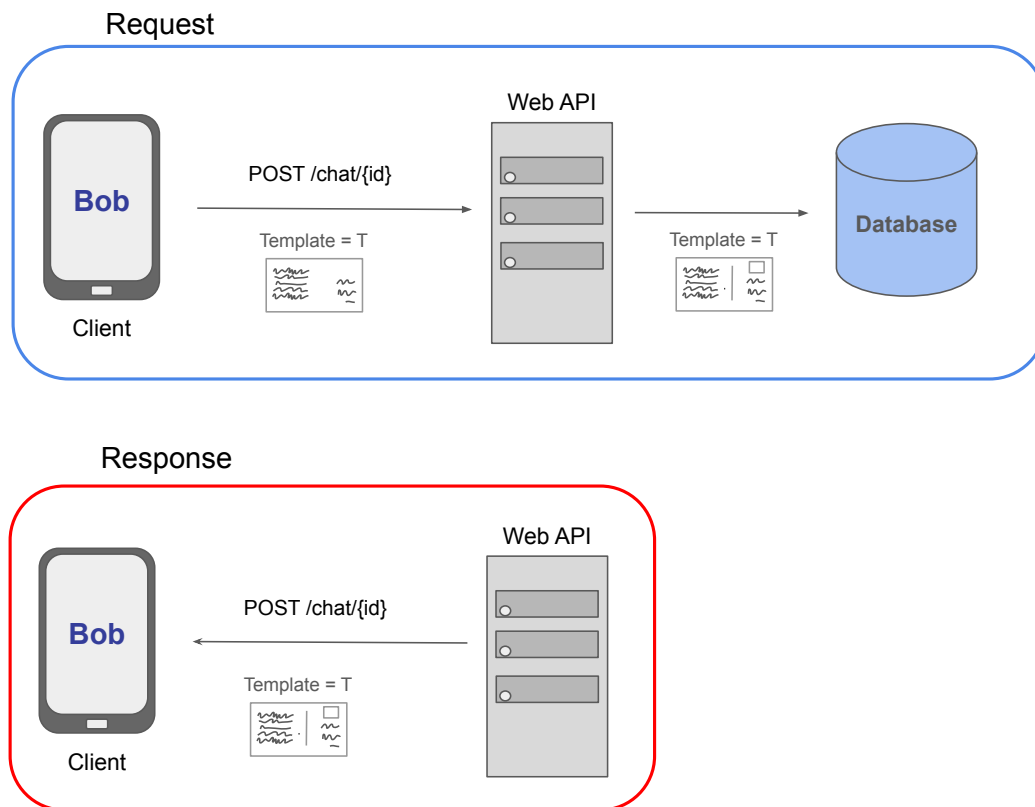


Figure 4.6: Postcard POST Flow

Getting all messages Messages are defined in a different URI from Chats, the reason of that is because we want to get all the messages across all chats, so to make a concise URI definition the Message URI was added (look into Figure 4.1 for all endpoints). Figure ?? illustrates getting all messages.

As soon as the user receives all pending messages, the same are delete from the server's database and can no longer be retrieved.

4.2.3 SVG Manipulation

When retrieving a message we want to keep it simple for any front end application to display the postcard, for that reason the template and the the handwritten content are merged into one new *Scalable Vector Graphics* (SVG) document and encoded into base 64. Figure 4.8 illustrates the merging process of a SVG

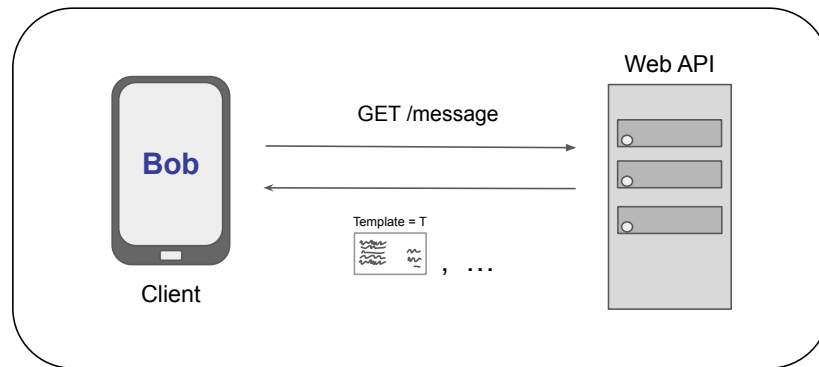


Figure 4.7: Postcard GET Flow

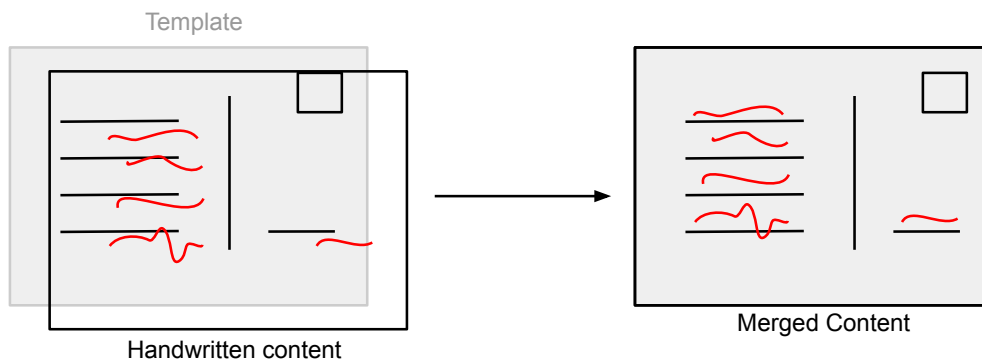


Figure 4.8: SVG Merging

4.3 Database Representation

Database uses PostgreSQL, an open-source relational database management system (*Relational Database Management System* (RDBMS)) known for its reliability, robustness, and extensive feature set. The database EA model can be found in Figure 4.9

Notice that *chat_group_member* is what defines the relation between a Chat and a User, as it is a many to many relation, and that its key is defined by the User's identifier plus the Group's identifier.

Templates are relatively small sized SVG's (normally never exceeding the KB's region) and so the same are saved in the database as a base 64 encoded string.

Message's *obtained* works as a flag to allow the message to be later removed.

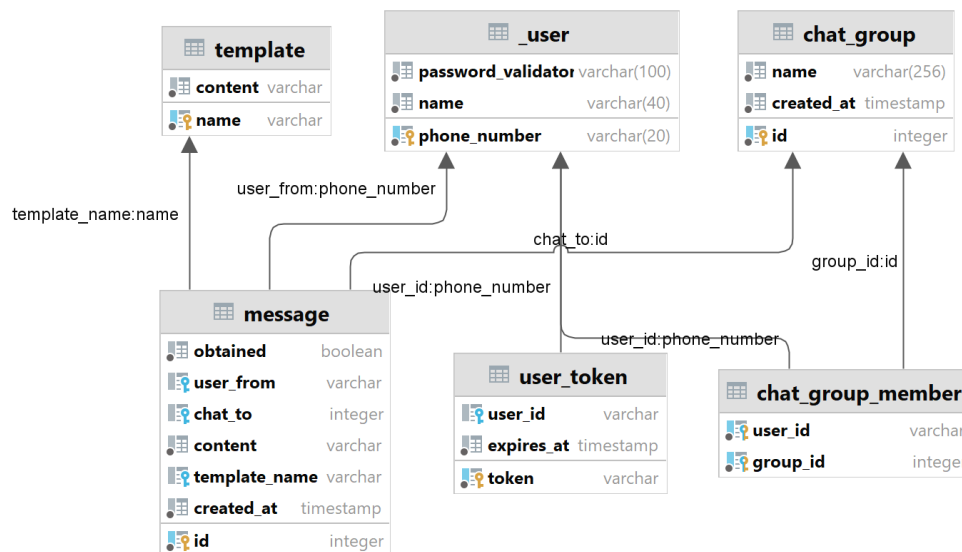


Figure 4.9: Entity Association Model

4.4 Application Runner and Storing Templates

Application Runner In Spring Framework, an `ApplicationRunner` is an interface that allows us to perform certain tasks after the Spring Boot application has started. It provides a convenient way to execute code logic that needs to run during the initialization phase of the application.

Storing Templates In order to automatically add templates to the database a Spring element `RunnerInsertTemplates` is defined, it implements the `ApplicationRunner` interface. When running the server the `RunnerInsertTemplates` will search for files in a specified `templates` folder and automatically insert all new templates. It is important to say that the templates names must be unique as it is the identifier for the database.

4.5 Verifying Phone Numbers

Verifying phone numbers is an important aspect of many applications and services to ensure the accuracy and legitimacy of user-provided contact information. While the implementation of SMS phone verification is not currently available, the application utilizes Google's `Libphonenumber` library for phone number validation.

The `Libphonenumber` library is a widely used and trusted library for working with phone numbers. It provides functionality to parse, format, and validate phone numbers

based on international standards and regional conventions. By utilizing this library, the application can perform various operations related to phone numbers, such as extracting the country code, validating the number's format, and determining the type of phone number (e.g., mobile, landline).

Although *Short Messaging Service* (SMS) phone verification is not yet implemented, the *Libphonenumber* library offers valuable features to ensure that the provided phone numbers are in a valid format and conform to the expected structure. This helps in reducing errors and preventing the submission of incorrect or invalid phone numbers.

When verifying phone numbers using the *Libphonenumber* library, the application can perform checks such as validating the length, format, and potential existence of the number. However, it's important to note that the library's validation does not guarantee the actual ownership or availability of the phone number, as it primarily focuses on the technical aspects of number validation.

To implement SMS phone verification in the future, the application would need to integrate with a suitable SMS gateway or service provider that can send verification codes to the provided phone numbers and handle the verification process. This additional functionality would allow the application to send SMS messages containing verification codes and verify them against user-provided inputs.

4.6 Performing HTR

Because our *Handwriting Text Recognition* (HTR) model is written in python, the source code folder must be known by the server. The *PYTOOLS_POSTCHAT* environment variable should be defined and must contain the absolute path to the python source folder.

Our python model takes a *Portable Network Graphics* (PNG) file as input therefore the SVG content needs to be converted to a temporary PNG file, this is done by using Apache's Batik Image converter.

The HTR operation is performed by the *htr* method. It takes the handwritten content encoded in base 64 as input and returns the recognized text as a String.

In order to perform such operation the server needs to call python code, to do so, it uses Java's *ProcessBuilder*.

The *ProcessBuilder* takes "**python**", "**main.py**", "**-source**", <png file path> as the program arguments and the source folder as default directory, we then proceed to start the process and wait (in a separate thread) for the process to finish. When done it will print in the standard output the recognized text, we then catch that output and return it.

Chapter 5

HTR Model

Contents

5.1	Introduction	24
5.1.1	Offline HTR	24
5.1.2	Online HTR	25
5.2	Implementation	25
5.2.1	Why Offline HTR	25
5.2.2	Pipeline	26
5.2.3	Detector	26
5.2.4	Recognizer	27
5.2.5	Natural Text Ordering	31
5.3	Limitations	32

5.1 Introduction

Handwriting recognition *Handwriting Text Recognition* (HTR) is a technology that converts handwritten text into digital format. Its purpose is to enable the efficient processing, storage, and manipulation of handwritten content through automated recognition algorithms.

5.1.1 Offline HTR

Offline handwriting recognition refers to the technology and process of converting static images or scans of handwritten text into digital text or characters. Unlike online handwriting recognition, which interprets handwriting in real time as it is being written, Offline HTR analyzes pre-existing images or documents that have been captured or scanned.

The goal of Offline HTR is to accurately recognize and convert the handwritten text in images into editable and searchable digital format. This technology finds applications in digitizing historical documents, handwritten notes, forms, and any other handwritten content that needs to be converted into machine-readable text.

The process of Offline HTR typically involves several steps:

- **Detection:** In the detection phase, the handwritten text regions within the scanned or captured images are identified. This can be achieved through techniques such as text detection algorithms, connected component analysis, or contour-based methods. The goal is to localize and extract the regions containing the handwritten content for further processing.
- **Image preprocessing:** The scanned or captured images are enhanced, filtered, and prepared to optimize the quality of the handwriting for recognition. This may involve noise reduction, binarization, deskewing, and other techniques.
- **Recognition:** Machine learning algorithms, such as neural networks or Hidden Markov Models (HMM), are trained using labeled samples of handwritten characters or words. These models are then used to recognize and classify the extracted features into corresponding textual representations.
- **Post-processing:** The recognized text is further refined and processed to improve accuracy, correct errors, handle ambiguities, and align with language-specific rules and dictionaries.

5.1.2 Online HTR

Online handwriting recognition refers to the technology and process of converting handwritten input into digital text or characters in real time. It involves capturing and interpreting the movements and patterns made by a user while writing using a stylus or a digital pen on a touch-enabled device, such as a tablet or a smartphone.

Unlike offline handwriting recognition, which analyzes static images of handwritten text, online handwriting recognition takes advantage of the temporal information obtained during the writing process. This allows for real-time interpretation and immediate feedback as the user writes, making it suitable for applications where instant recognition is required, such as note-taking, electronic signature verification, form filling, and interactive whiteboards.

Online handwriting recognition systems typically use various techniques, including pattern recognition algorithms, machine learning, and neural networks, to analyze the dynamic information provided by the user's pen strokes. These algorithms analyze factors such as stroke speed, direction, pressure, and sequence to identify and interpret the handwritten characters or gestures. The recognized text can then be further processed, stored, or used in various applications and systems that require digital text input.

5.2 Implementation

In this section we will talk about the decisions made and explain the implementation of each component.

5.2.1 Why Offline HTR

In a project like this, it initially seems like a no-brainer to implement Online HTR instead of Offline HTR. However, the lack of complex yet comprehensive documentation for beginners pose a significant roadblock. Without readily available resources and clear guidance, the implementation process becomes more complex and time-consuming. This led to frustration and delays as we struggled to navigate the complexities of Online HTR without proper guidance.

Furthermore, the absence of open-source implementations of Online HTR hinders collaboration and knowledge sharing within the community. Open-source projects often serve as valuable starting points, providing code samples, libraries, and frameworks that accelerate the development process. Without such resources, we must start from scratch or rely on proprietary solutions, limiting the ability to customize and adapt the HTR system

to the specific project requirements. The potential benefits of enhanced accuracy, real time text recognition, and dynamic input support opportunities may be overshadowed by the steep learning curve and lack of accessible resources for offline HTR implementation.

As a result, the decision to implement Offline HTR becomes less straightforward yet possible thanks to widely available open-source implementations and good guidance examples such as the one followed to implement our model *A_K_Nain (2023)*.

5.2.2 Pipeline

The HTR system follows a pipeline-based approach to process images and extract text. The pipeline consists of two main stages: text detection and text recognition. The Figure 5.1 represents a overview of the implemented HTR system. For our needs it takes a PNG containing the postcard drawn content and returns the text as a String.

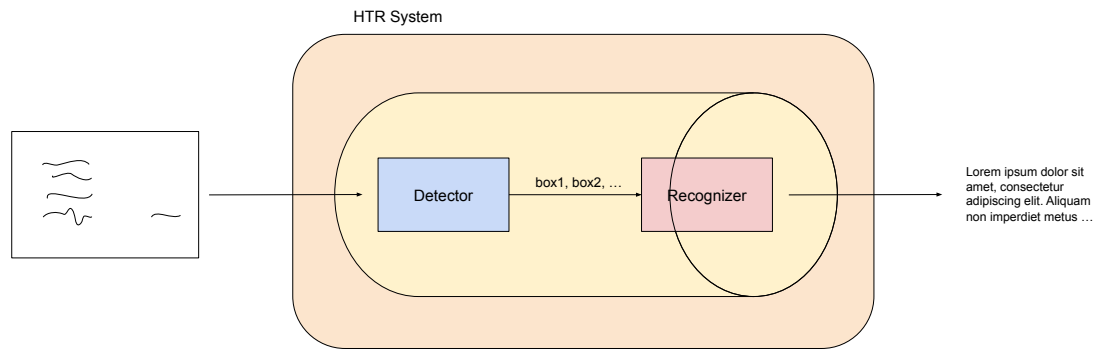


Figure 5.1: HTR System

Both components use the Tensorflow framework and are written in python.

5.2.3 Detector

The detection is made possible by using CRAFT-Text *clovaai (2023)* detection implementation from *faustomorales (2023)*. The particular detector was trained for machine-generated characters based on fonts but still manages to give good results for handwritten text, big part of it's good results is the absence of visual clutter in the background as we only provide the drawn content to the model.

The Figure 5.2 illustrates how the detector works, keeping in mind that the figure is simplified for explanatory purposes and the detector detects words and not lines.

The returned value from the detector model is a list containing four points (x and y) defining a box that represents where the word is relative to the image.

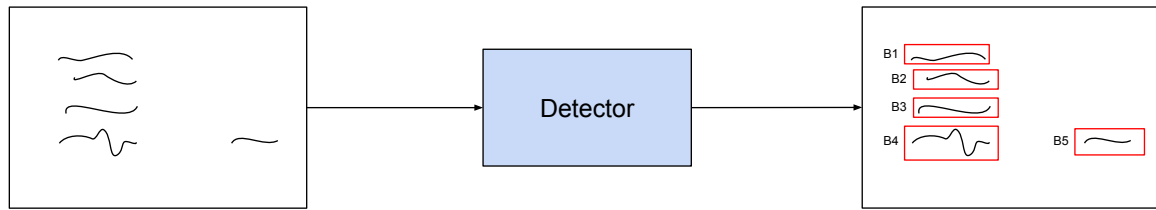


Figure 5.2: Detection Flow

5.2.4 Recognizer

The text recognition stage takes the boxes from the detector and processes them to extract the actual text content creating a temporary file for each box. It applies character segmentation, feature extraction, and sequence modeling techniques to recognize and convert the text regions into machine-readable format. A layer by layer summary based on online documentation:

- **Input Layer:** The model takes an input image with dimensions (128, 32, 1), representing a grayscale image. It also takes input labels, which are sequences of characters.
- **Convolutional Layers:** The model starts with two convolutional blocks. Each block consists of a 2D convolutional layer followed by a max-pooling layer. The convolutional layers learn local image features and the max-pooling layers downsample the feature maps.
- **Reshaping and Dense Layer:** After the convolutional layers, the feature maps are reshaped to a new shape that is compatible with the recurrent part of the model. This reshaping operation prepares the data for input to the recurrent layers. The reshaped features pass through a fully connected dense layer with 64 units and ReLU activation.
- **Dropout Regularization:** A dropout layer is applied to reduce overfitting by randomly setting a fraction of the input units to 0 during training.
- **Recurrent Layers:** Two bidirectional LSTM layers are stacked on top of each other. Bidirectional LSTMs process the input sequence in both forward and backward directions, allowing the model to capture dependencies in both directions. The LSTM layers have dropout applied to them to prevent overfitting.

- **Output Layer:** A dense layer with a softmax activation is used as the output layer. The number of units in this layer corresponds to the vocabulary size (number of characters) plus two special tokens introduced by the CTC loss. The softmax activation produces a probability distribution over the characters.
- **CTC Loss Layer:** The output of the softmax layer is passed to the Connectionist Temporal Classification (CTC) layer. The CTC layer calculates the CTC loss, which measures the difference between the predicted sequence and the ground truth labels. It takes both the labels and the output of the softmax layer as inputs.
- **Model Compilation:** The model is compiled with the Adam optimizer. The specific learning rate and other optimizer parameters can be further customized if needed.
- **Model Output:** The output of the model is the output of the CTC layer, representing the CTC loss.

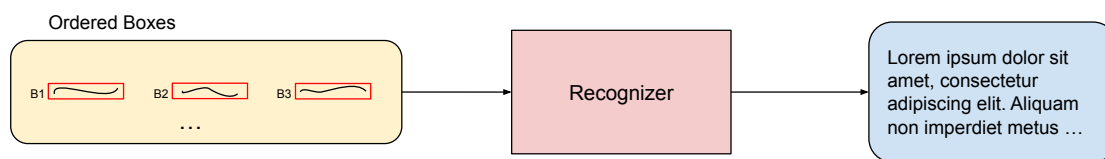


Figure 5.3: Recognizer Flow

Image Preprocessing

Image preprocessing is a critical step in offline HTR systems as it aims to enhance the quality of scanned or captured handwritten images before they are fed into the recognition model. The objective is to optimize the images for accurate recognition by applying various transformations and adjustments.

In the context of our project, the provided code snippet offers a foundational approach to image preprocessing.

Reading and Decoding Image:

The first step involves reading the image file using `tf.io.read_file` and decoding it using `tf.image.decode_png`. By decoding the image as grayscale (`decode_png(image, 1)`), we ensure that only the necessary information is retained. Distortion-Free Resizing:

To achieve consistent input sizes, the `distortion_free_resize` function is employed for resizing the image to the desired dimensions (`img_size`). During resizing, the `preserve_aspect_ratio=True` parameter ensures that the original aspect ratio of the image is maintained. This function intelligently pads the image symmetrically to ensure uniformity and prevent distortion. Normalization:

After resizing, the image is cast to `tf.float32` and normalized to a range between 0 and 1 by dividing each pixel value by 255.0. Normalization facilitates improved convergence during training and enhances the overall performance of the recognition model. By utilizing the `preprocess_image` function, we establish a solid foundation for handling image preprocessing in our offline HTR system. This function accepts an image path as input and performs resizing, padding, and normalization operations to prepare the image for subsequent processing.

Training

The recognizer was trained using the *IAM (2023)* words dataset, which is a widely used benchmark dataset for HTR research. The IAM dataset is a comprehensive collection of handwritten English text samples contributed by different writers. It consists of 86810 training samples, 4823 validation samples and 4823 test samples.

To get good results the recognizer was trained for roughly 60 iterations in Google's Colab Notebook servers.

Tweaks After each training iteration, the Tensorflow training process incorporates additional code through callbacks to enhance its functionality. In this particular scenario, two tweaks have been added to optimize the training process:

- **Early Stopping Callback:** An early stopping callback is implemented to monitor the model's performance during training and determine if it is not improving. The `patience` parameter is set to 3, indicating that if the model does not show improvement for 3 consecutive iterations, the training process will stop early. By setting `restore_best_weights` to `True`, the callback ensures that the best weights achieved during training are restored before stopping, allowing for optimal model performance. The `verbose` parameter is set to 1, enabling the callback to display informative messages about its operations.
- **Checkpoint Callback:** A checkpoint callback is added to periodically save the model's weights during training. The `filepath` parameter specifies the path where the weights will be saved. By setting `save_weights_only` to `True`, only the weights

of the model will be saved, reducing storage requirements. The verbose parameter is set to 1, enabling the callback to display informative messages about the saving process.

These tweaks improve the training process by introducing early stopping criteria based on the model's performance and by providing checkpoints to save the model's weights at different stages.

Postprocessing

To be able to decode the output produced by the Output Layer (dense 2) a function called *decode_batch_predictions* is implemented in the guide *A_K_Nain (2023)*.

The function takes the model predictions *pred* as input. These predictions are usually in the form of probability distributions over the characters in the vocabulary for each time step.

The variable *input_len* is created to specify the length of the input sequences for each prediction in the batch. It is set to be the same for all predictions and is equal to the number of time steps in the predictions.

The function utilizes the CTC (Connectionist Temporal Classification) decoding method to convert the predictions into sequences of characters. It applies the *ctc_decode* function from Keras backend, passing the predictions, input length, and using greedy search (other methods like beam search can be used for more complex tasks). The *ctc_decode* function returns the decoded sequences.

The results variable stores the decoded sequences. It selects the first element *[0][0]* from the *ctc_decode* output, which represents the decoded labels for the batch. It also truncates the sequences to a maximum length *max_len* if necessary.

The function iterates over each decoded sequence in *results*. For each sequence, it applies several operations to convert the numerical labels to actual text.

First, it uses *tf.where* to find the positions where the labels are not equal to -1 (a special token often used in CTC decoding to represent blank or no label).

It then uses *tf.gather* to gather the non-equal elements from the labels.

The gathered labels are passed through *num_to_char* function, which maps the numerical labels to their corresponding characters.

Next, *tf.strings.reduce_join* is applied to concatenate the characters into a single string representation.

Finally, *numpy().decode("utf-8")* is used to convert the string from a TensorFlow tensor to a regular Python string, and the resulting string is appended to the *output_text* list.

After iterating over all the sequences in results, the function returns the `output_text` list, which contains the decoded texts for each prediction in the batch.

In summary, the `decode_batch_predictions` function takes the model predictions, performs CTC decoding to convert the predictions into sequences of characters, and applies additional post-processing steps to obtain the final text representations for the predictions.

5.2.5 Natural Text Ordering

One of the challenges in HTR is maintaining the natural ordering of text when dealing with multi-line or multi-column documents. To address this challenge, we came up with a algorithm that uses the average box size to calculate a margin of error for each point in a line. In our implementation there is always two fixed boxes Figure 5.4

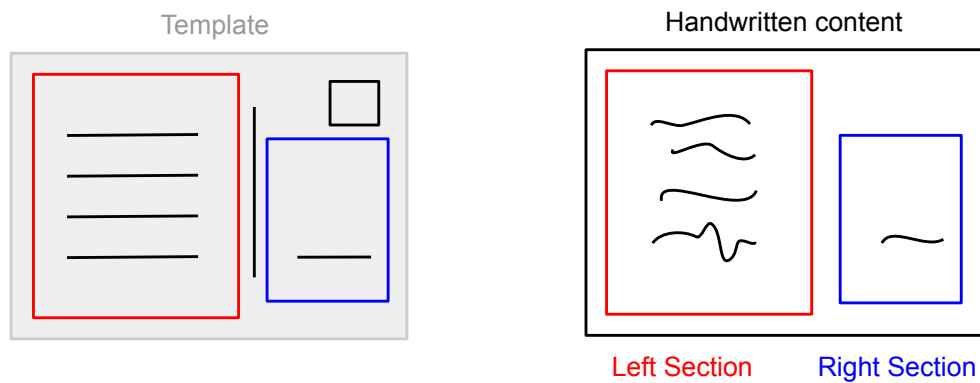


Figure 5.4: Postcard Main Sections

The ordering algorithm works like this:

We start by calculating the height of every box, using vector calculation. We do this for every box and divide by the number of boxes obtaining the average box height.

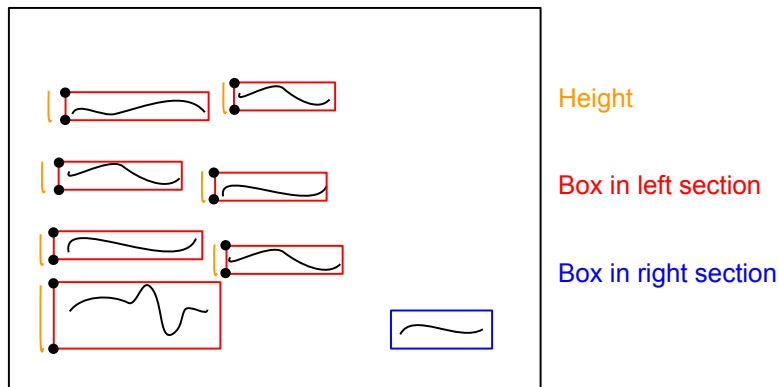


Figure 5.5: Average height calculation for left section

Now all we have to do is pick the same located point in every box (top left or right left, etc...) and test the y value \pm the calculated average box/2. If the y value is inside the range $y - \text{average height}/2$ to $y + \text{average height}/2$ then compare the x values else compare the y values. A lower x value means it comes earlier in the natural ordering. A higher y value than the current one means it comes later in the natural ordering.

5.3 Limitations

IAM dataset vocabulary is primarily focused on the English language. It includes a wide range of alphanumeric characters, including uppercase and lowercase letters (A-Z, a-z), digits (0-9), and common punctuation marks. However, it does not cover the entire spectrum of possible characters that can exist in different languages or writing systems.

This vocabulary limitation means that the IAM dataset may not be suitable for recognizing text in languages other than English or for dealing with specialized symbols or characters that are outside the dataset's predefined set. For example, if the dataset does not include characters specific to a particular language or domain, the HTR model trained on IAM may struggle to accurately recognize and transcribe such characters.

While having made all the possible optimizations for using the detector, if the user draws text right next to each other there's a good chance it will detect it all as a whole word.

Chapter 6

Client

Contents

6.1	Android System and Compose Framework Overview	34
6.1.1	Android Manifest	34
6.1.2	Android Activity	35
6.1.3	Data Storing	35
6.1.4	ViewModel	36
6.1.5	Canvas	36
6.1.6	Making HTTP Request	36
6.2	Activities	37
6.2.1	Permissions	37
6.2.2	Sign In	38
6.2.3	Home	42
6.2.4	Create Chat	44
6.2.5	Chat View	45
6.2.6	Draw Postcard	48
6.2.7	View Postcard	50

The client was implemented in Android and uses Android's Jetpack Compose UI toolkit.

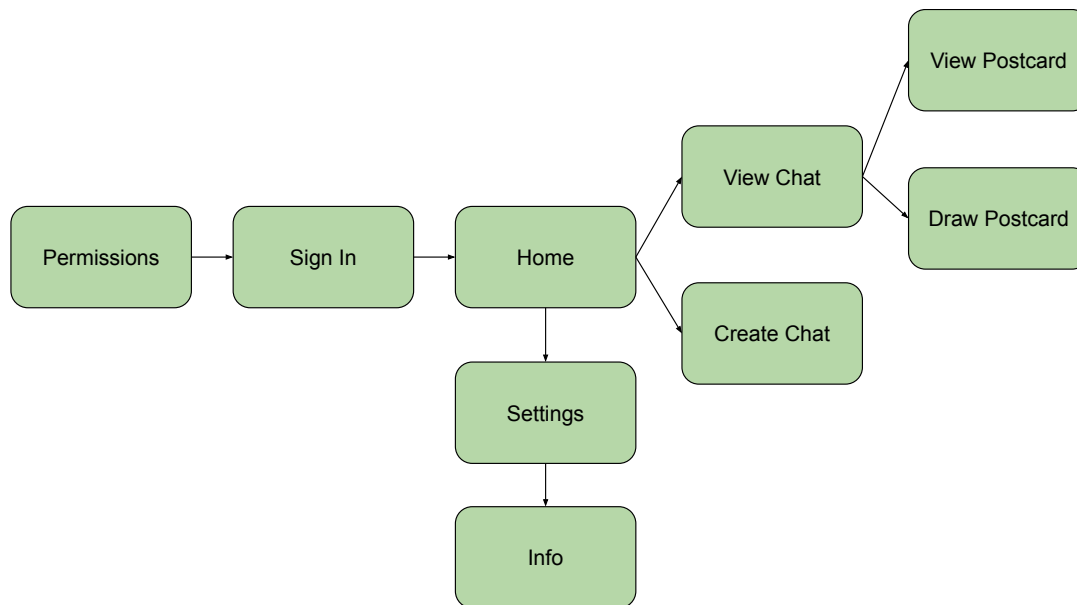


Figure 6.1: Android Activity Navigation Graphic

6.1 Android System and Compose Framework Overview

Before showing the client implementation its best to give some context and basic knowledge about the android system and the Compose framework.

6.1.1 Android Manifest

The `AndroidManifest.xml` file is an essential configuration file in Android development that provides essential information about the Android application to the Android system. It is located in the root directory of the Android project and is required for every Android application.

The Android manifest file contains important metadata about the app, including its package name, version number, permissions, activities, services, broadcast receivers, and more. It serves as a blueprint for the Android system to understand the structure and behavior of the application.

6.1.2 Android Activity

In the context of Android app development, an Activity is a fundamental component of an Android application that represents a single screen with a user interface. It is a crucial part of the overall app architecture and is responsible for handling user interactions and presenting visual elements to the user.

An Activity acts as a container for the user interface and provides a window in which the app's UI elements, such as buttons, text fields, images, and other widgets, are displayed. It manages the lifecycle of these UI components and handles user input events, such as button clicks or touch gestures.

Each Activity has a corresponding Java or Kotlin class that extends the Activity base class or its subclasses provided by the Android framework. This class contains methods that define the behavior of the Activity during different stages of its lifecycle, such as creation, starting, pausing, resuming, stopping, and destruction.

When an app is launched, typically, the main Activity is created and displayed to the user. The Activity is responsible for setting up the initial UI layout, interacting with data sources (e.g., retrieving data from a database or an API), and responding to user actions. It can also communicate with other Activities, such as starting a new Activity for a different screen or receiving results from a previously started Activity.

6.1.3 Data Storing

Android uses a file system that's similar to disk-based file systems on other platforms. The system provides several options for you to save your app data:

- App-specific storage: Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage. Use the directories within internal storage to save sensitive information that other apps shouldn't access;
- Shared storage: Store files that your app intends to share with other apps, including media, documents, and other files;
- Preferences: Store private, primitive data in key-value pairs;
- Databases: Store structured data in a private database using the Room persistence library.

6.1.4 ViewModel

The ViewModel class is a business logic or screen level state holder. It exposes state to the UI and encapsulates related business logic. Its principal advantage is that it caches state and persists it through configuration changes. This means that your UI doesn't have to fetch data again when navigating between activities, or following configuration changes, such as when rotating the screen.

6.1.5 Canvas

The Android Canvas is a fundamental graphics component provided by the Android framework. It serves as a drawing surface onto which we can render custom graphics, shapes, images, and text. The Canvas provides a set of drawing methods that allow us to create and manipulate visual elements within an Android application.

When working with the Canvas, we can perform various operations such as drawing lines, rectangles, circles, arcs, and paths. We can also apply transformations like translation, rotation, scaling, and skewing to manipulate the position and orientation of the drawn elements. Additionally, the Canvas supports the rendering of text, allowing us to display custom text with different fonts, sizes, colors, and styles.

6.1.6 Making HTTP Request

When developing Android applications, it is common to interact with web services and APIs to retrieve data or send data to a server. One popular library for making HTTP requests in Android is OkHttp.

OkHttp

OkHttp is an open-source HTTP client library for Java and Android applications. It is developed by the same team behind the widely-used Retrofit library and offers a simple and efficient way to make HTTP requests and handle responses. OkHttp is built on top of the Java standard library's `URLConnection`, providing a more convenient and powerful API.

6.2 Activities

In this section we will demonstrate all activities implemented.

6.2.1 Permissions

The Permissions activity handles all requests to permissions needed for the application to work. The Application needs permission to read contacts.

Figures 6.2 and 6.3 illustrate the implemented activity.

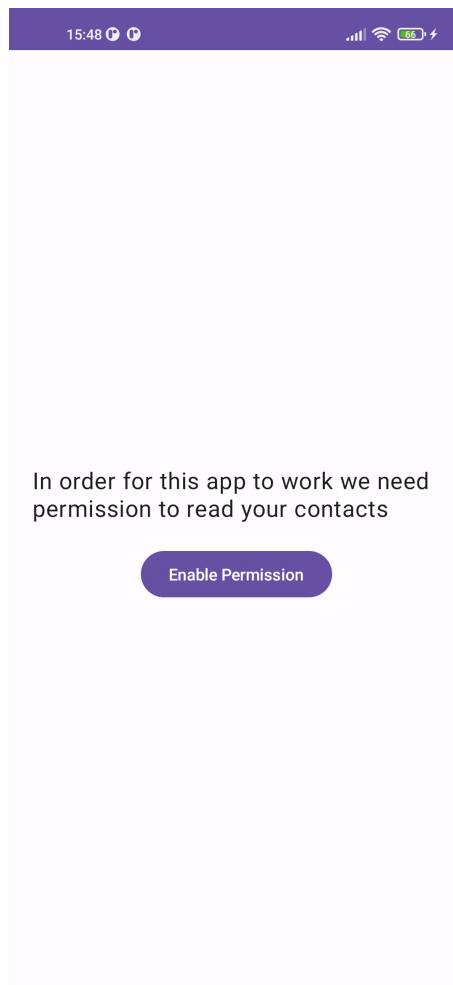


Figure 6.2: Permission Activity

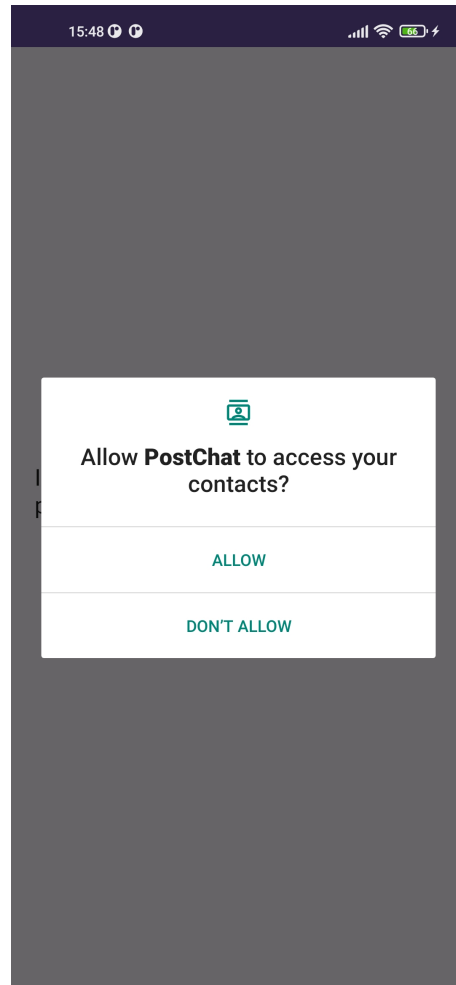


Figure 6.3: Prompt Permission Activity

6.2.2 Sign In

The Sign-in Activity serves as the component for managing the user's authentication process, containing both logging in and registering in the service. Additionally, it ensures the secure storage of the user's token by leveraging the `EncryptedSharedPreferences`.

Upon launching the Sign-in Activity, users are presented with a user-friendly interface where they can enter their credentials or choose to register as a new user. The activity handles the input validation and securely communicates with the server-side authentication API.

Once the user's credentials are verified, the Sign-in Activity retrieves the authentication token from the server's response. To ensure the token's confidentiality, it is stored using the `EncryptedSharedPreferences`. This specialized `SharedPreferences` implementation employs encryption algorithms to protect sensitive data from unauthorized access.

By utilizing the `EncryptedSharedPreferences`, the Sign-in Activity safeguards the user's authentication token, preventing it from being tampered with or exposed. This secure storage mechanism provides an additional layer of protection for user data, mitigating the risks associated with unauthorized token access.

In addition, the Sign-in Activity incorporates an automatic phone number region retrieval feature by leveraging the Carrier information.

Figures 6.4, 6.5 and 6.6 illustrate the implemented activity.

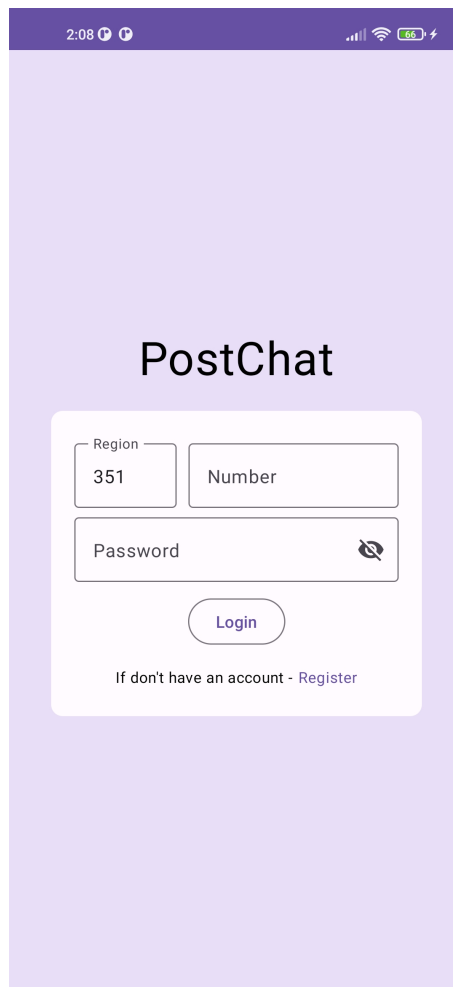
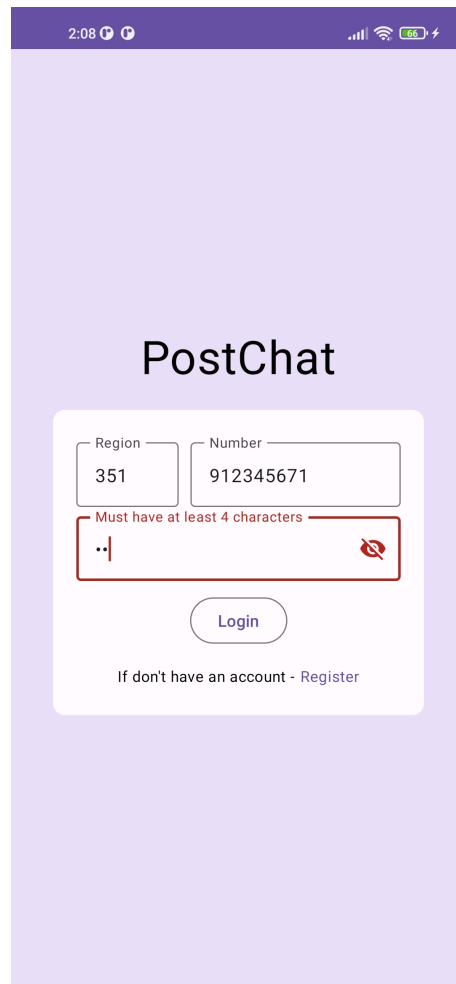


Figure 6.4: Signin Activity

It also does local verification's to user's input. Password and Phone Number validations are done.



The screenshot shows a mobile application interface for "PostChat". At the top, there is a status bar with the time "2:08", signal strength, Wi-Fi, and battery level at "66%". The main title "PostChat" is centered in a large, bold, black font. Below the title is a white login form with rounded corners. The form contains two input fields: "Region" with the value "351" and "Number" with the value "912345671". Below these fields is a password input field with a red border and a red error message "Must have at least 4 characters" above it. The password field contains two dots and a cursor. To the right of the password field is a red eye icon. Below the password field is a "Login" button with a purple outline and text. At the bottom of the form, there is a link that says "If don't have an account - Register".

Figure 6.5: Signin Activity invalid password size

The image shows a mobile application interface for 'PostChat'. At the top, there is a status bar with the time '2:09', signal strength, Wi-Fi, and battery level '66%'. The app title 'PostChat' is centered in a large, bold, black font. Below the title is a white login form with rounded corners. The form contains three input fields: 'Region' with the value '351', 'Number' with the value '912345671', and a password field. The password field is highlighted with a red border and contains the text 'At least 1 uppercase letter and 1 digit' in red, indicating an invalid password. The password field also shows four dots and a red eye icon. Below the password field is a 'Login' button with a purple outline. At the bottom of the form, there is a link that says 'If don't have an account - Register'.

PostChat

Region: 351 Number: 912345671

At least 1 uppercase letter and 1 digit

....

Login

If don't have an account - [Register](#)

Figure 6.6: Signin Activity missing invalid password

6.2.3 Home

The Home Activity serves as a central hub for connecting to the web API and retrieving essential information related to registered users, messages, and chats. Its primary purpose is to display all chats in a user-friendly manner, with the chats ordered based on the most recent message received.

By establishing a connection with the web API, the Home Activity can fetch the necessary data to populate the chat interface. It retrieves information about registered users, ensuring that the appropriate user profiles are displayed within the chat list. Additionally, it retrieves messages associated with each chat, allowing users to view their conversation history.

The Home Activity organizes the chats in a manner that prioritizes the most recent interactions. By ordering the chats based on the last message received, users can quickly identify and access their most recent conversations.

Moreover, the Home Activity provides intuitive controls and a simple interface, users can create new chat groups, within a button.

Figure 6.7 illustrate the implemented activity.

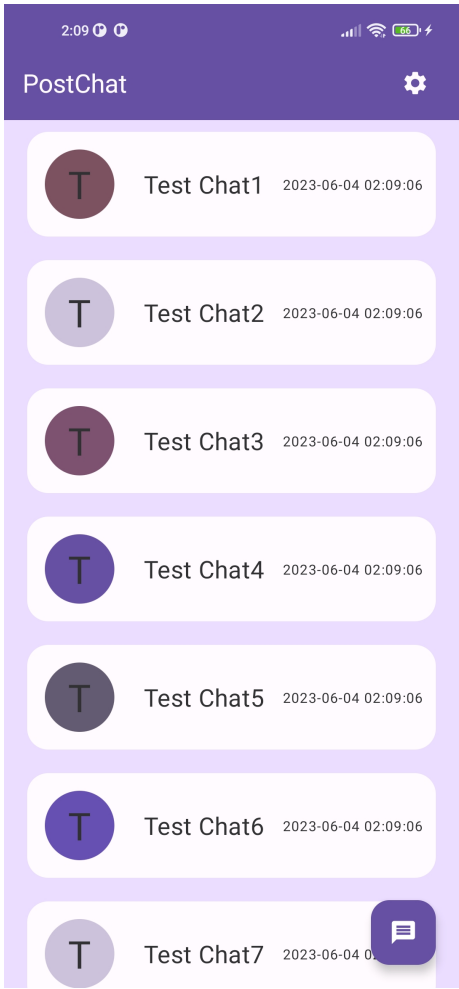


Figure 6.7: Home Activity

6.2.4 Create Chat

The CreateChat activity searches for the users stored in the local database and lets you pick the phone numbers you want to add to the chat. Every chat needs a name so a popup dialog input message shows when clicking the check button.

Figures 6.8 and 6.9 illustrate the implemented activity.

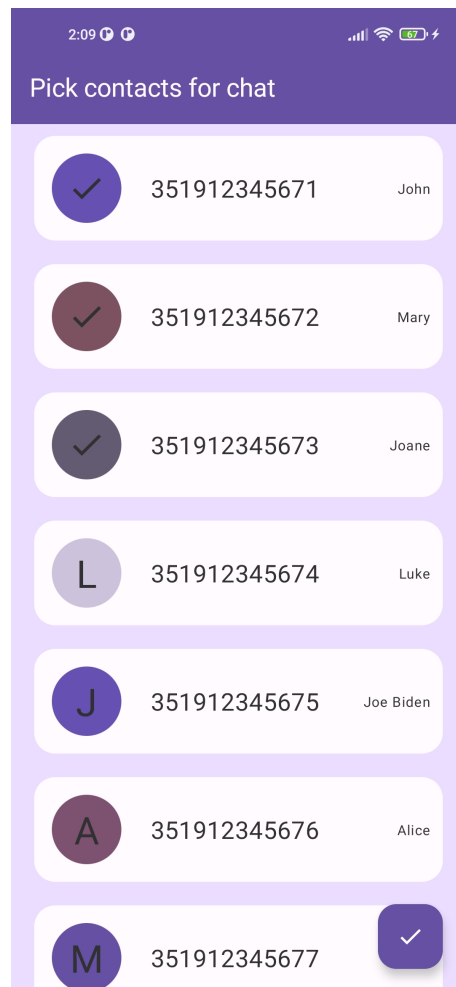


Figure 6.8: CreateChat Activity

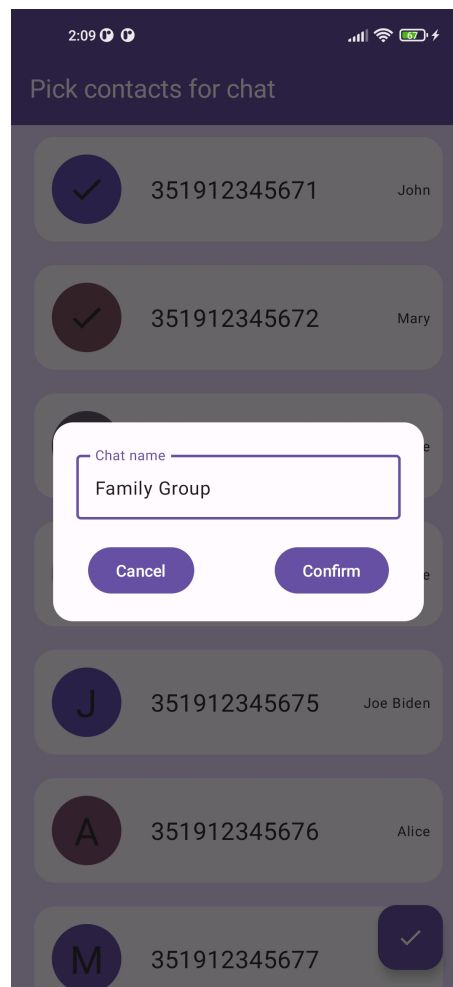


Figure 6.9: CreateChat Activity name prompt

6.2.5 Chat View

The Chat activity obtains the information about the current chat messages. It displays the postcards in order by the timestamp and above the same it shows the number from the person that sent the message. In the future this will be changed to query users in the local database and get their name.

Figures 6.10, 6.11 , 6.12 and 6.13 illustrate the implemented activity.



Figure 6.10: Chat Activity

On clicking the edit button the user is sent to the Draw Activity.

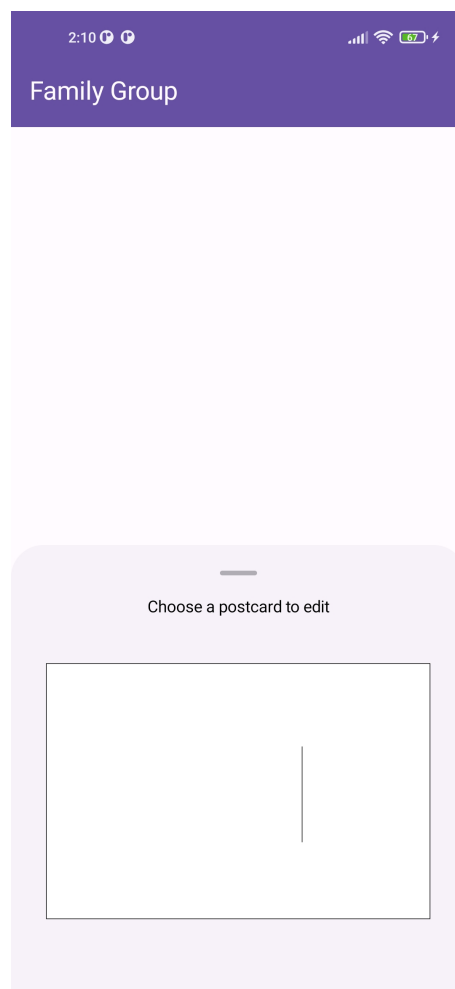


Figure 6.11: Chat Activity bottom templates list

This is a beta version some bugs still need to be fixed.

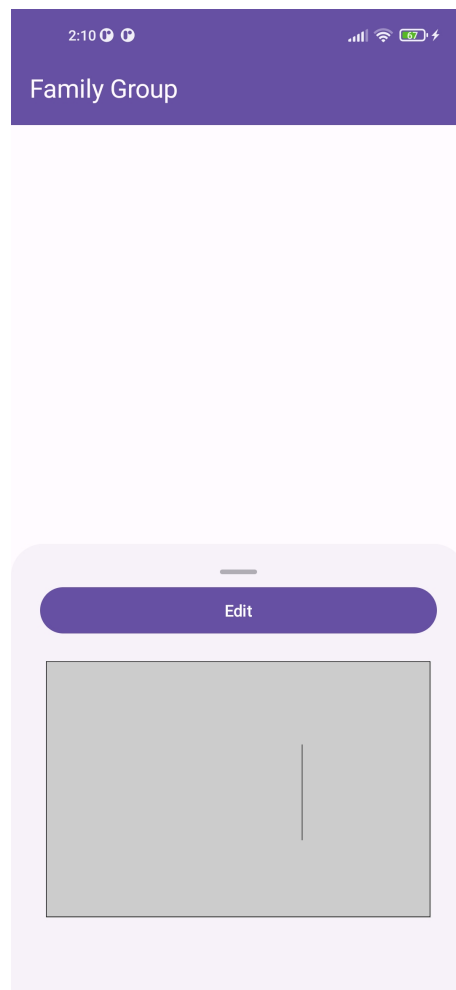


Figure 6.12: Chat Activity pick template

6.2.6 Draw Postcard

The Draw activity plays a crucial role in allowing users to edit and personalize a postcard using the Canvas. It involves implementing complex code to enable features such as drawing, zooming, and saving the postcard.

The challenge in implementing drawing and zoom features arise from the absence of a built-in two-finger zoom and one-finger touch draw function in the compose toolkit. To overcome this limitation, an in-depth analysis of the compose toolkit's inner code was conducted. By examining the underlying mechanisms of the compose toolkit, the necessary functionality for drawing and zooming was achieved.

To facilitate the saving of the canvas, a list is used to store the properties of each path. Each path property consists of the actual path data and the stroke style applied to it.

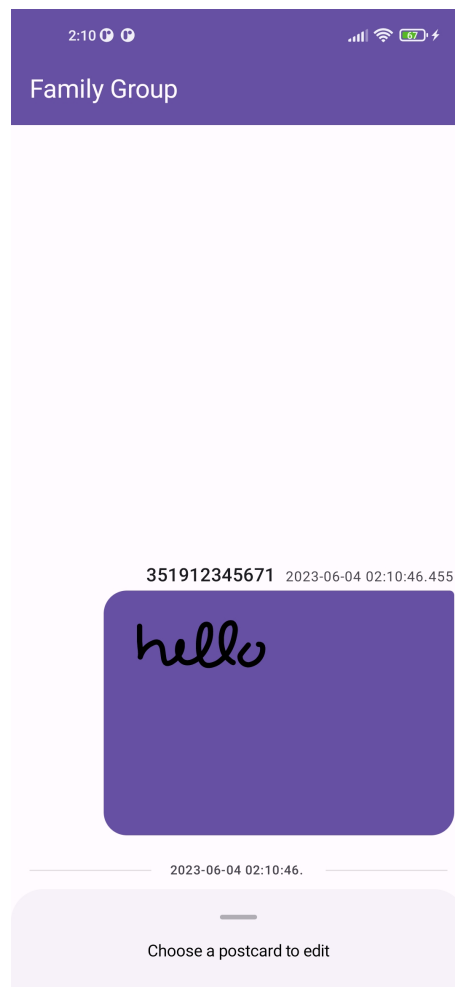


Figure 6.13: CreateChat sent postcard

When the canvas needs to be saved, the application iterates through each path in the list and generates an SVG (Scalable Vector Graphics) file.

During the SVG file creation process, the application adds the necessary path movements and stroke styles to accurately represent each drawn path. This ensures that the saved SVG file faithfully represents the visual appearance of the canvas.

Figures 6.14 and 6.15 illustrate the implemented activity.

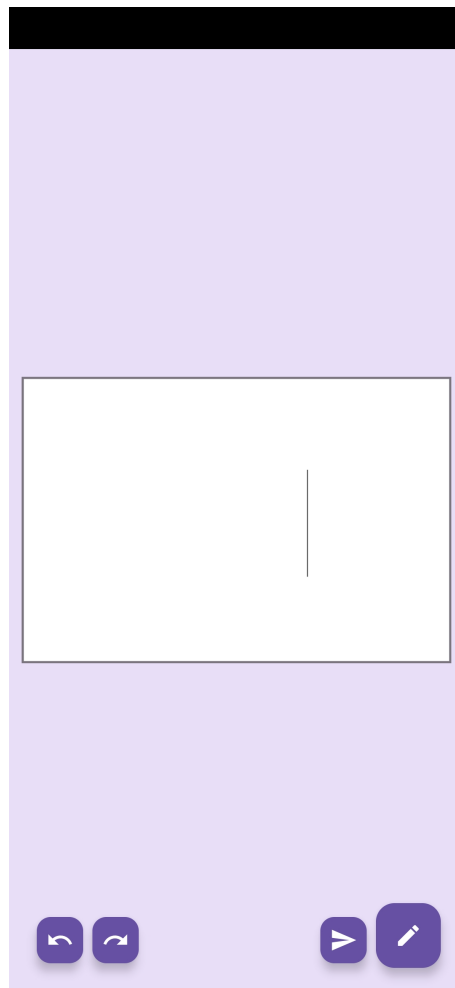


Figure 6.14: Draw Activity

6.2.7 View Postcard

The View Postcard Activity provides users with a dedicated interface to view postcards and offers the ability to save them with a simple button press. This activity focuses on delivering a seamless and immersive experience for users to enjoy the postcard content.

Figure 6.16 illustrate the implemented activity.

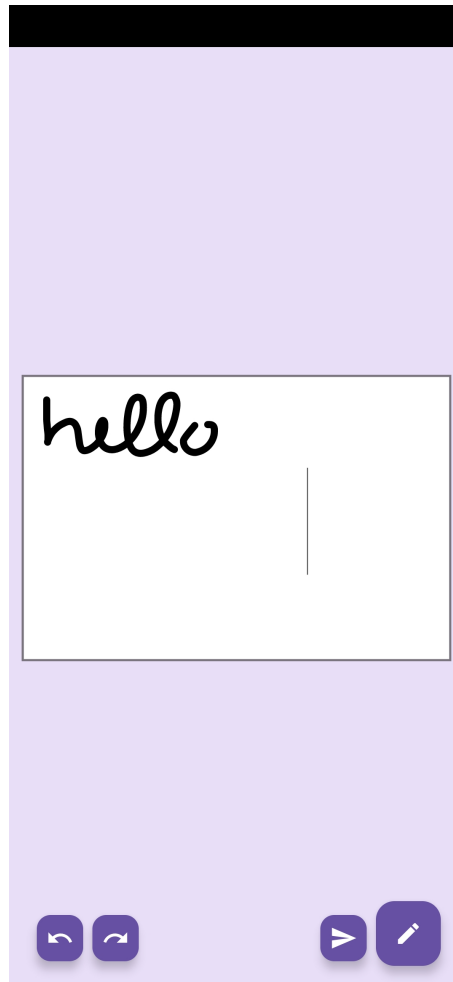


Figure 6.15: Draw Activity draw



Figure 6.16: Postcard View

Chapter **7**

Conclusions and Future Work

In this project, we have developed a comprehensive application that integrates various components, including a Handwritten Text Recognition (HTR) model, a web API, and an Android application frontend. The aim of the project was to provide users with a seamless and efficient solution for communicating using digital postcards and to expand our knowledge of the AI world as we go.

In terms of future work, there are several areas that can be further improved and expanded upon. Bug fixes and performance optimizations should be prioritized to ensure the stability and efficiency of the application. Additionally, further testing, including unit testing and integration testing, should be conducted to identify and address any potential issues.

Furthermore, the application can be enhanced by implementing additional functionalities and features. This may include integrating social sharing capabilities, enabling users to share their postcards on social media platforms.

Overall, this project has demonstrated the successful integration of Handwritten Text Recognition (HTR) technology, a web API, and an Android application frontend to create a comprehensive postcard sending system. By leveraging these technologies, users can easily personalize and send their handwritten postcards, making the process more efficient and convenient. With continued development and refinement, the application has the potential to provide users with a seamless and enjoyable communication process.

Appendices

Other Definitions

A.1 Technologies

- Spring Boot and MVC - Open Source Framework for Web Applications;
- JVM - Java Virtual Machine;
- Tensorflow - TensorFlow is a framework to create machine learning models for desktop, mobile, web, and cloud.
- Libphonenumber - Library to validate phone number format;
- Jetpack Compose - Android's recommended toolkit for building native UI;
- Kotlin - Programming Language that extends Java base language;

References

- A_K_Nain, S. P. (2023). Handwriting recognition. URL: https://keras.io/examples/vision/handwriting_recognition/.
- clovaai (2023). Craft: Character-region awareness for text detection. URL: <https://github.com/clovaai/CRAFT-pytorch>.
- faustomorales (2023). Keras ocr detector. URL: <https://keras-ocr.readthedocs.io/en/latest/api.html#core-detector-and-recognizer>.
- IAM (2023). Iam handwriting database. URL: <https://fki.tic.heia-fr.ch/databases/iam-handwriting-database>.
- MyPostcard (2023). Mypostcard. URL: <https://www.mypostcard.com/en/>.
- Telegram (2023). Telegram. URL: <https://telegram.org>.
- W3C (2023). Scalable vector graphics. URL: <https://www.w3.org/Graphics/SVG/>.
- WhatsApp (2023). Whatsapp. URL: <https://www.whatsapp.com/>.