HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Entwicklung eines KI-gestützten Tools zur Analyse und Zusammenfassung wissenschaftlicher Arbeiten

Inhaltsverzeichnis

1.Einleitung

1.1 Problemstellung

In der heutigen wissenschaftlichen Welt stehen Studierende und Dozenten vor einer Vielzahl an Herausforderungen im Umgang mit der stetig wachsenden Menge an Fachliteratur. Sowohl Studierende als auch Dozenten müssen oft in kurzer Zeit eine große Anzahl an Studien und Publikationen lesen, um den aktuellen Stand der Forschung zu einem bestimmten Thema zu verstehen oder sich auf Vorlesungen vorzubereiten. Studierende hingegen verfügen häufig nicht über ausreichend Zeit oder Muse, um alle notwendigen Publikationen vollständig zu lesen und zu verstehen.

Dabei kann es leicht passieren, dass wichtige Informationen übersehen werden oder sie Schwierigkeiten haben, die wesentlichen Erkenntnisse einer Arbeit zu extrahieren und in den eigenen Forschungs- oder Lernprozess zu integrieren. Darüber hinaus fehlen ihnen oft geeignete Werkzeuge, um komplexe wissenschaftliche Texte effizient zu analysieren und zusammenzufassen.

Auch für Dozenten ergibt sich eine ähnliche Problematik. Sie müssen sich in einem dynamischen Forschungsfeld schnell und fundiert über neue Entwicklungen informieren, um ihre Lehrinhalte auf dem neuesten Stand zu halten. Des Weiteren müssen Sie die wissenschaftlichen Arbeiten ihrer Studierenden sichten und bewerten. Die manuelle Durchsicht und Analyse ist jedoch zeitaufwändig und kann zu einem erheblichen Arbeitsaufwand führen.

Vor diesem Hintergrund wird ein zunehmender Bedarf an automatisierten, KI-gestützten Tools zur Analyse und Zusammenfassung wissenschaftlicher Arbeiten deutlich. Solche Werkzeuge könnten Studierende und Dozenten gleichermaßen unterstützen, indem sie effizient Schlüsselinformationen aus großen Textmengen extrahieren und aufbereiten. Ziel ist es, den Zugang zu wissenschaftlichen Erkenntnissen zu erleichtern und den Arbeitsaufwand im akademischen Alltag zu reduzieren, ohne dabei die Qualität und Tiefe des Verständnisses zu beeinträchtigen.

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines KI-gestützten Tools, das wissenschaftliche Arbeiten in Form von PDF-Dokumenten effizient analysieren und zusammenfassen kann. Hierbei soll eine webbasierte Anwendung erstellt werden, die folgende Kernfunktionen integriert:

1. Schnittstelle zum PDF-Parser: Zunächst wird eine PDF-Parser-Schnittstelle implementiert, die es ermöglicht, wissenschaftliche Artikel im PDF-Format zu verarbeiten. Der Parser extrahiert den Textinhalt der Dokumente und bereitet diesen für weitere Analysen vor. Dabei wird darauf geachtet, dass der Parser auch mit komplexen PDF-Strukturen, wie Tabellen, Diagrammen oder Fußnoten, umgehen kann, um eine möglichst vollständige und korrekte Textextraktion zu gewährleisten.

2. Schnittstelle zur KI-API: Um die Extraktion und Zusammenfassung der wesentlichen Informationen aus dem Text zu automatisieren, wird eine Schnittstelle zu einer KI-API integriert. Diese API ermöglicht es, mithilfe von Natural Language Processing die wichtigsten Inhalte der wissenschaftlichen Arbeit in kurzer Form zusammenzufassen. Ziel ist es, dass die KI in der Lage ist, automatisch relevante Abschnitte zu identifizieren und in verständlicher, kompakter Form darzustellen.

3. Benutzeroberfläche: Die Anwendung soll eine intuitive Benutzeroberfläche bieten, um die Aufgabe der PDF-Zusammenfassung möglichst schnell und einfach umzusetzen.

Letztlich soll die entwickelte Anwendung sowohl technisch ausgereift als auch benutzerfreundlich sein, um eine breite Akzeptanz an der Hochschule zu gewährleisten.

1.3 Struktur der Arbeit

Die vorliegende Arbeit ist in fünf Hauptkapitel gegliedert, die eine systematische Vorgehensweise bei der Entwicklung des KI-gestützten Tools zur Analyse und Zusammenfassung wissenschaftlicher Arbeiten in PDF-Form beschreiben.

Das zweite Kapitel, Anforderungsanalyse, bildet den Ausgangspunkt. Es definiert die Zielgruppe des Tools und geht auf deren spezifische Bedürfnisse ein. Im Anschluss werden die funktionalen und nicht-funktionalen Anforderungen detailliert beschrieben. Diese Anforderungen betreffen sowohl die technischen als auch die nutzungsbezogenen Aspekte des Tools. Darüber hinaus werden Anwendungsfälle dargestellt, die die geplante Nutzung des Tools veranschaulichen.

Im dritten Kapitel, Technologiestack, werden die wesentlichen Technologien, die für die Entwicklung des Tools benötigt werden, beschrieben. Dazu gehören die Programmiersprachen, die Auswahl geeigneter Frameworks für das Frontend und Backend. Ein besonderer Fokus liegt auf der Integration externer APIs und Bibliotheken, insbesondere der KI-API zur Textzusammenfassung und der PDF-Parser-Technologie.

Das vierte Kapitel, Implementierung, beschreibt detailliert den technischen Aufbau der Anwendung. Zunächst wird die Backend-Logik erläutert, die den Upload und die Verarbeitung von wissenschaftlichen Arbeiten in PDF-Form ermöglicht. Hierbei wird zunächst der Ablauf der Dokumentenanalyse durch PDF-Parser erklärt. Anschließend geht es um die Textzusammenfassung durch die KI-API, gefolgt von der Generierung der Zusammenfassungen und der entsprechenden Quellenangaben. Im Anschluss wird eine Benutzeroberfläche erstellt, die eine intuitive Interaktion mit der Anwendung ermöglicht.

Das fünfte Kapitel, Rekapitulation und Fazit, bewertet den Erfolg der Zielerreichung anhand der zu Beginn definierten Anforderungen. Es beleuchtet die während der Implementierung aufgetretenen Herausforderungen und bietet einen Ausblick auf mögliche Erweiterungen des Tools. Hier wird diskutiert, welche zusätzlichen Funktionen in der Zukunft integriert werden könnten, um die Anwendung noch nützlicher und vielseitiger zu machen.

2. Anforderungsanalyse

2.1 Zielgruppe

Die Zielgruppe für das geplante KI-gestützte Tool zur Analyse und Zusammenfassung wissenschaftlicher Arbeiten besteht hauptsächlich aus Studierenden und Dozenten. Beide Gruppen stehen im akademischen Umfeld vor der Herausforderung, effizient mit der wachsenden Menge an Fachliteratur umzugehen.

Studierende sind eine der primären Zielgruppen, da sie regelmäßig wissenschaftliche Arbeiten lesen und analysieren müssen, um sich auf Prüfungen, Seminararbeiten oder Abschlussarbeiten vorzubereiten. Oftmals haben sie begrenzte Zeit und müssen sich schnell einen Überblick über den Stand der Forschung in ihrem Fachgebiet verschaffen. Die spezifischen Anforderungen und Herausforderungen, denen Studierende gegenüberstehen, sind die effiziente Zeitnutzung, das Verständnis von komplexen Inhalte und die Informationsüberflutung. Das Tool hilft Ihnen dabei, relevante Quellen schnell zusammenzufassen und in den Kontext ihrer eigenen Arbeit zu stellen.

Dozenten sind die zweite Hauptzielgruppe dieses Tools, da sie neben der Vermittlung von Lehrinhalten auch für ihre eigene Forschung und die Betreuung von Studierenden verantwortlich sind, wobei sich ihre Anforderungen nur leicht von denen der Studierenden unterscheiden, insbesondere in Bezug auf Zeitmanagement, Effizienz, die Bewertung wissenschaftlicher Arbeiten und die kontinuierliche Aktualisierung ihrer Lehrinhalte.

Sowohl Studierende als auch Dozenten teilen einige gemeinsame Bedürfnisse, die bei der Entwicklung des Tools berücksichtigt werden sollten. Das Tool muss einfach zu bedienen sein, mit einer intuitiven Benutzeroberfläche und leicht verständlichen Funktionalitäten, um eine schnelle und effiziente Nutzung zu gewährleisten. Ein weiteres wichtiges Kriterium ist die Präzision und Qualität der Zusammenfassungen. Beide Gruppen legen großen Wert auf die Genauigkeit der Informationen, weshalb das Tool wissenschaftliche Arbeiten präzise analysieren und qualitativ hochwertige Zusammenfassungen generieren muss, die den Kern der Inhalte verständlich wiedergeben. Zudem sollte das Tool flexibel einsetzbar sein, um verschiedene Arten von wissenschaftlichen Arbeiten effektiv zusammenzufassen. Diese breite Anwendbarkeit stellt sicher, dass es in vielen verschiedenen Fachbereichen genutzt werden kann.

Insgesamt adressiert das Tool die spezifischen Herausforderungen, denen sich Studierende und Dozenten im akademischen Alltag gegenüberstehen und bietet eine Lösung, die sowohl die Effizienz steigert als auch die Qualität der wissenschaftlichen Arbeit verbessert.

2.2 Funktionale Anforderungen

Die funktionalen Anforderungen an das KI-gestützte Tool umfassen zentrale Funktionen, die das Tool zur Erfüllung seiner Aufgaben bieten muss. Zunächst muss das Tool die Möglichkeit bieten, wissenschaftliche Arbeiten im PDF-Format hochzuladen und den Textinhalt zuverlässig zu extrahieren, auch wenn die Dokumente komplexe Elemente wie Tabellen, Diagramme oder Fußnoten enthalten. Ein weiterer wichtiger Aspekt ist die Integration einer Schnittstelle in Form einer KI-API, um die Texte automatisiert zu analysieren und präzise Zusammenfassungen generieren zu können. Die generierten

Zusammenfassungen sollen dem Nutzer direkt in einer benutzerfreundlichen Darstellung angezeigt werden. Eine zentrale Anforderung ist dabei, dass das Tool die Zusammenfassung als PDF-Datei zum Download anbietet, die der Nutzer lokal auf seinem Gerät speichern kann.

2.3 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen an das KI-gestützte Tool konzentrieren sich auf Aspekte der Benutzerfreundlichkeit, Leistung und Sicherheit, die für eine reibungslose und effiziente Nutzung sorgen. Eine der wichtigsten Anforderungen ist die Benutzerfreundlichkeit. Dazu muss die Navigation klar strukturiert sein, um das Hochladen von wissenschaftlichen Arbeiten sowie das Herunterladen von Zusammenfassungen schnell und einfach zu gestalten.

Ein weiterer nicht-funktionaler Aspekt betrifft die Leistungsfähigkeit des Tools. Es sollte in der Lage sein, große PDF-Dokumente innerhalb kurzer Zeit zu verarbeiten und die Textzusammenfassung effizient zu generieren. Zudem ist die Zuverlässigkeit von hoher Bedeutung – das Tool muss robust und stabil arbeiten, um Fehler oder Abstürze während des Verarbeitungsprozesses zu vermeiden.

Hinsichtlich der Sicherheit muss sichergestellt werden, dass die hochgeladenen Dokumente vertraulich behandelt und nach der Verarbeitung nicht gespeichert werden, um den Datenschutz zu wahren. Da die Ergebnisse ausschließlich lokal als PDF-Dateien gespeichert werden, sollte das Tool zudem eine sichere Download-Funktion bieten, ohne dass Daten auf externen Servern verbleiben.

2.4 Anwendungsfälle

Im folgenden Abschnitt werden zwei Anwendungsfälle aus Sicht der Studierenden und der Dozierenden beschrieben, um einen besseren Einblick in den Ablauf des Tools zu vermitteln.

Ein Student bereitet sich auf eine Seminararbeit vor und benötigt einen Überblick über die aktuelle Forschung zu einem spezifischen Thema. Anstatt mehrere wissenschaftliche Arbeiten in voller Länge zu lesen, lädt er PDF-Dokumente von relevanten Artikeln in das Tool hoch. Das Tool extrahiert den Text aus den PDF-Dateien, analysiert diese und generiert automatisch Zusammenfassungen der wesentlichen Inhalte. Anschließend kann die generierte Zusammenfassung heruntergeladen und lokal auf dem eigenen Gerät gespeichert werden.

Eine Dozentin muss mehrere Abschlussarbeiten ihrer Studierenden begutachten. Da diese Arbeiten umfangreich sind, nutzt sie das Tool, um sich schneller in die jeweiligen Themen einzuarbeiten. Sie lädt die PDF-Dateien der Abschlussarbeiten in das Tool, welches die Dokumente analysiert und Zusammenfassungen erstellt. Die Dozentin erhält so schnell einen Überblick über die Kerninhalte und Argumentationsstrukturen der Arbeiten und kann sich darauf basierend besser auf die detaillierte Bewertung und das Feedback vorbereiten.

3. Technologiestack

3.1 Programmiersprache

Dart wurde als Hauptprogrammiersprache für die Entwicklung des Backends des KI-gestützten Tools gewählt, da es eine Reihe entscheidender Vorteile für dieses Projekt bietet. Besonders hervorzuheben ist die breite Palette an verfügbaren Bibliotheken, die effiziente Verarbeitung von Textdaten und die nahtlose Integration von Künstlicher Intelligenz ermöglichen. Darüber hinaus bietet Dart eine solide Grundlage für die Entwicklung von plattformübergreifenden Anwendungen in Kombination mit Frameworks wie Flutter, was die Vielseitigkeit und Effizienz des Projekts weiter steigert.

Zudem wird Dart direkt in nativen Code kompiliert, was eine schnelle Ausführung und geringe Latenzzeiten ermöglicht. Ein weiterer Vorteil ist, dass Dart bereits in vorherigen studentischen Projekten erfolgreich eingesetzt wurde, was eine aufwändige Einarbeitung überflüssig macht und den Entwicklungsprozess von Beginn an effizient gestaltet. Zukünftig betrachtet bietet Dart zudem verschiedene Entwicklungsplattformen an und eignet sich sowohl für kleinere Projekte als auch für hochskalierbare, cloudbasierte Anwendungen. Diese Flexibilität macht Dart zu einer zukunftssicheren Wahl für unterschiedlichste Anforderungen.

3.2 Framework

Für das Frontend wird Flutter als Framework verwendet, da es eine moderne, plattformübergreifende Lösung bietet, um eine benutzerfreundliche Oberfläche zu entwickeln, die auf verschiedenen Geräten (Web, Desktop und Mobile) funktioniert. Zudem kann mit einer einheitlichen Codebasis sowohl für Web- als auch für mobile Anwendungen gearbeitet werden. Dies ermöglicht es, das Tool flexibel und auf verschiedenen Geräten zugänglich zu machen, ohne separate Applikationen entwickeln zu müssen. Des Weiteren verwendet Flutter Dart als Programmiersprache und ermöglicht somit eine schnelle Entwicklung, da bereits Kenntnisse durch andere Vorlesungen mit diesem Framework gemacht wurden.

Flutter bietet außerdem eine umfangreiche Sammlung vorgefertigter Widgets, die anpassbare und pixelgenaue Benutzeroberflächen ermöglichen. Diese Widgets funktionieren plattformübergreifend, können aber auch plattformspezifisch angepasst werden. Zusätzlich macht es Flutter besonders einfach, komplexe Animationen und Übergänge zu erstellen, die nahtlos und leistungsstark laufen.

Zusammengefasst bietet die Kombination aus Dart für das Backend und Flutter für das Frontend eine leistungsstarke, flexible und zukunftssichere Technologiebasis, um ein effizientes und benutzerfreundliches KI-gestütztes Tool zur Analyse und Zusammenfassung wissenschaftlicher Arbeiten zu entwickeln.

3.3 Externe APIs

Die Integration externer APIs ist ein zentraler Bestandteil der Entwicklung des KI-gestützten Tools, da sie die Funktionalitäten bereitstellen, die für die Analyse, Verarbeitung und Zusammenfassung wissenschaftlicher Arbeiten notwendig sind.

4. Konzeption & Implementierung

4.1 Komponentendiagramm

Zunächst wurde ein Komponentendiagramm erstellt, um einen umfassenden Überblick über alle genutzten Frameworks und APIs zu gewährleisten. Gleichzeitig dient es dazu, sicherzustellen, dass sämtliche benötigten Funktionen vollständig integriert sind.

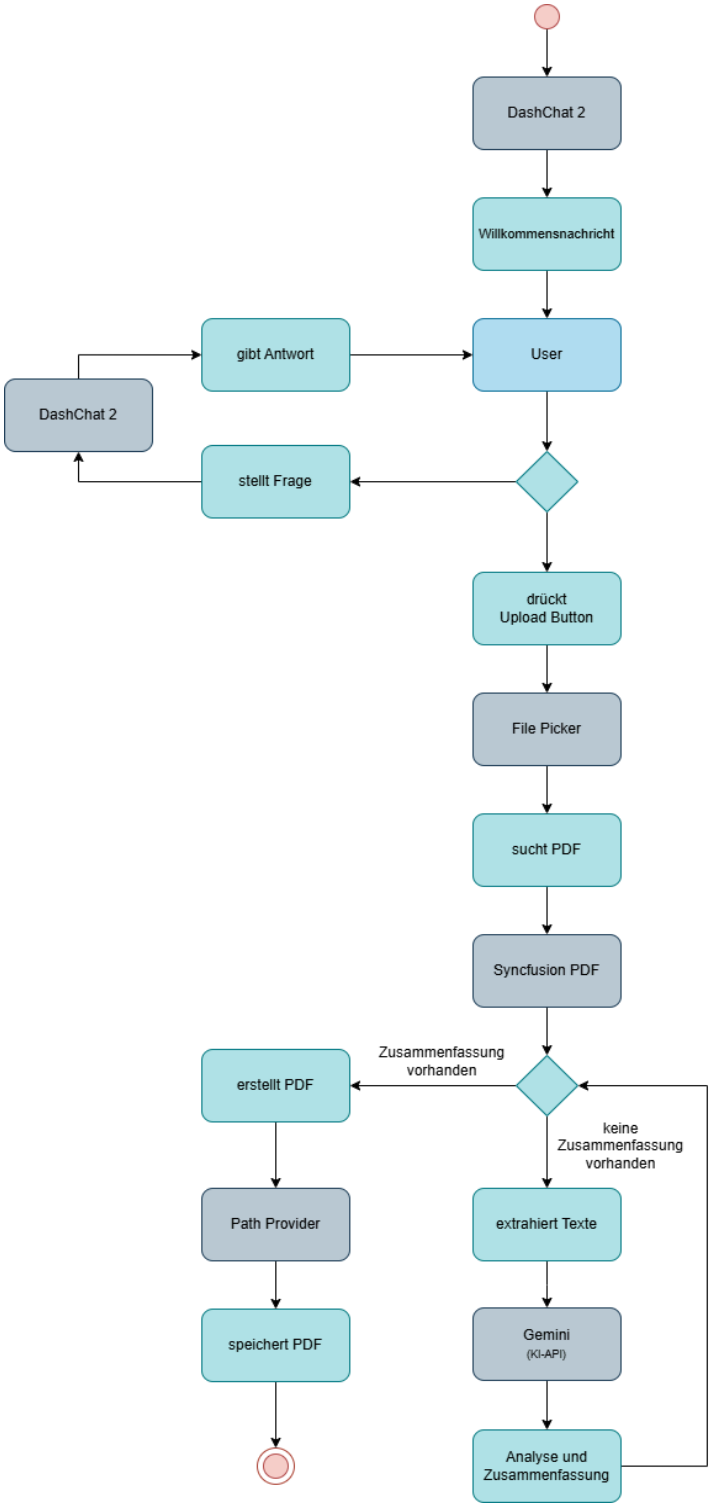

Abbildung 4.1: Komponentendiagramm
(eigene Darstellung)

4.2 Aufbau der Benutzeroberfläche

Bei der Implementation des Tools wurde mit der Startseite und dem Aufbau einer Benutzeroberfläche begonnen. Um die Benutzerfreundlichkeit des Tools zu erhöhen, wurde sich für einen ChatBot entschieden, der den User anleitet und somit das Verständnis des Tools erhöht. Durch eine Willkommensnachricht wird kurz erklärt, wie die App funktioniert (Abbildung 4.2). Des Weiteren soll der ChatBot Fragen zum Tool beantworten.

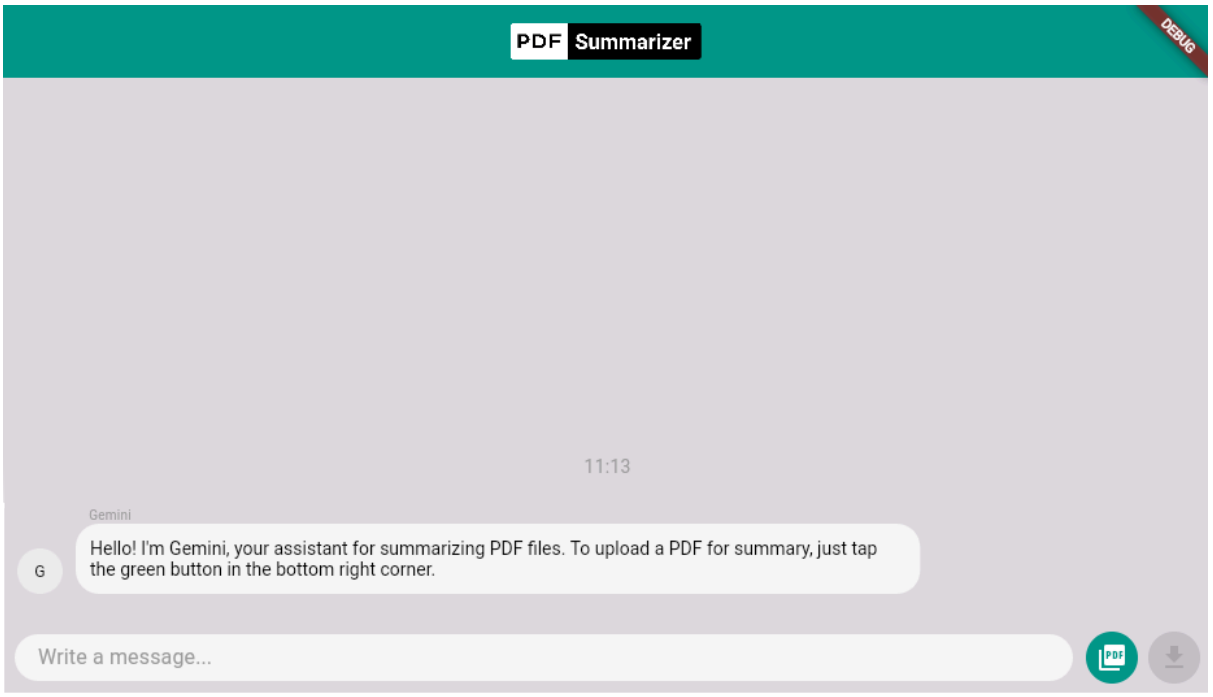

Abbildung 4.2: Startseite mit Willkommensnachricht
(eigene Darstellung)

Um dies zu realisieren, wurde die Bibliothek DashChat2 benutzt. Diese Bibliothek bietet Widgets für die Erstellung von ansprechenden Chat-Oberflächen. Sie unterstützt sowohl Textnachrichten als auch Dokumente und speichert zudem die Zeitstempel und Benutzerinformationen. Dies hilft, um bessere Kontrolle über den Nachrichtenverkehr zu erhalten. Die Verwendung der Bibliothek im Code erfolgt über einen einfachen Import-Aufruf.

```
import 'package:dash_chat_2/dash_chat_2.dart';
```

Anschließend wird eine leere Liste von Nachrichten erstellt sowie 2 ChatUser. Die leere Liste speichert die geschriebenen Nachrichten zwischen *currentUser* und *geminiUser*. In diesem Fall ist *currentUser* der eigene Benutzer und *geminiUser* der Chatbot.

```
List<ChatMessage> messages = [];
ChatUser currentUser = ChatUser(id: "0", firstName: "You");
ChatUser geminiUser = ChatUser(id: "1", firstName: "Gemini");
```

addWelcomeMessage Methode

Mit der *addWelcomeMessage* Methode wird der leeren Liste eine Nachricht hinzugefügt. Diese sorgt dafür, dass beim Starten des Tools im Chat eine Willkommensnachricht im Chat angezeigt wird (Abbildung 4.2.1).

```
void _addWelcomeMessage() {
 ChatMessage welcomeMessage = ChatMessage(
   user: geminiUser,
   createdAt: DateTime.now(),
   text:
       "Hello! I'm Gemini, your assistant for summarizing PDF files. To
upload a PDF for summary, just tap the green button in the bottom right
corner."
 );
```

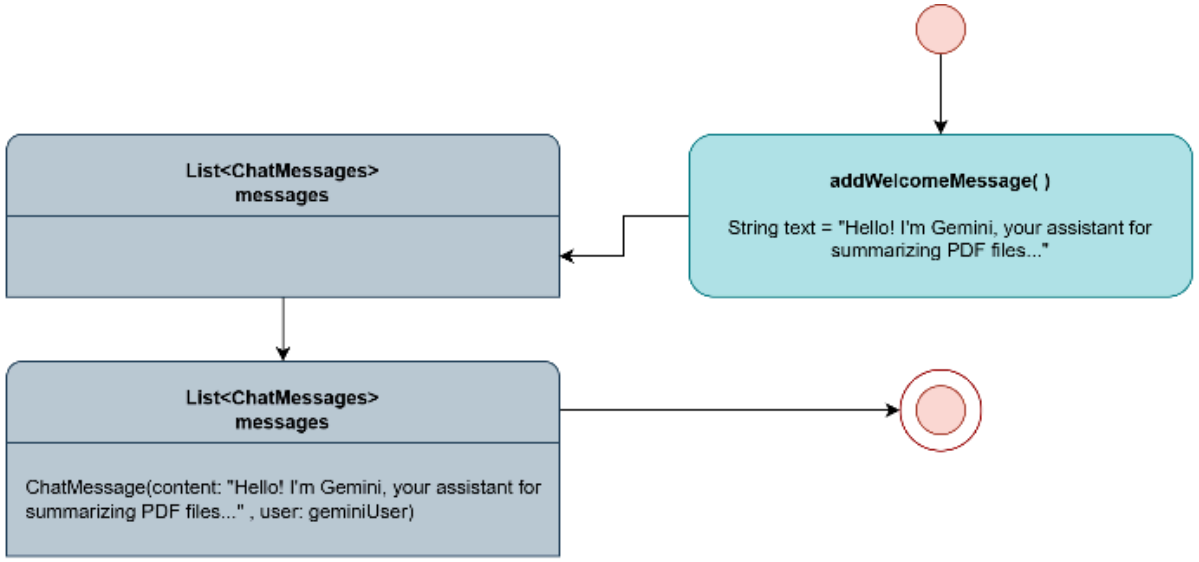

Abbildung 4.2.1: Hinzufügen der Willkommensnachricht
(eigene Darstellung)

Um den nun vorbereiteten Chat anzuzeigen, muss lediglich das DashChat-Widget in Flutter eingebunden werden um einen funktionstüchtigen Chatbot zu erhalten.

```
Widget _bodyUI() {
 return DashChat(
        currentUser: currentUser,
        onSend: _sendMessage,
        messages: messages,
        )
}
```

4.3 sendMessage Methode

Die *sendMessage* Methode wird beim Versenden einer Nachricht innerhalb von DashChat aufgerufen. Ihre Hauptaufgabe besteht darin, den Inhalt der Nachricht, den zugehörigen Nutzer sowie den Zeitstempel in der Liste *messages* zu speichern. Anhand des Ablaufplans (Abbildung 4.3) soll nun die *sendMessage* Methode im folgenden Beispiel genauer erklärt werden.

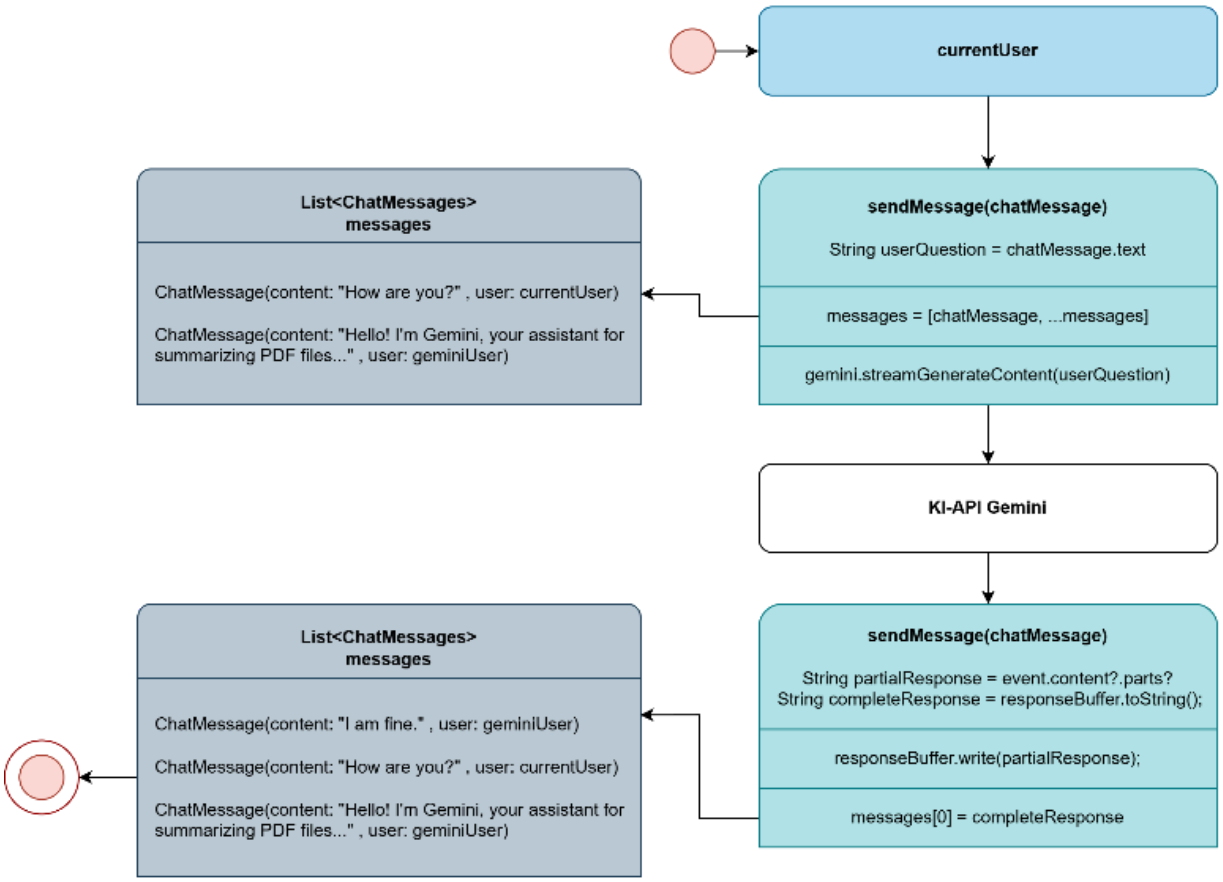

Abbildung 4.3: Ablaufplan der sendMessage Methode
(eigene Darstellung)

Beispiel sendMessage

Nach der Willkommensnachricht des *geminiUser* sendet der *currentUser* eine Chat-Nachricht vom Typ *chatMessage* mit der Frage: "How are you?". Diese *chatMessage* wird an der ersten Position der Liste *messages* eingefügt. Die aktualisierte Liste *messages* sieht nun wie folgt aus:

```
List<ChatMessage> messages = [
ChatMessage(content: "How are you?" , user: currentUser),
ChatMessage(content: "Hello! I'm Gemini, your assistant for summarizing
PDF files..." , user: geminiUser)]
```

Im nächsten Schritt wird der Text aus chatMessage in der Variablen *userQuestion* gespeichert. Danach wird ein *responseBuffer* erstellt um die Antworten von der KI-API *Gemini* zu erhalten.

```
String userQuestion = chatMessage.text;
StringBuffer responseBuffer = StringBuffer();
```

Anschließend wird die *userQuestion* mit folgendem Befehl an die KI-API *Gemini* übergeben und verarbeitet.

```
gemini.streamGenerateContent(userQuestion)
```

Ein Teil der Antwort wird von Gemini zunächst im String *partialResponse* gespeichert. Die *partialResponse* wird daraufhin in den *responseBuffer* geschrieben. Sollten weitere Antworten eintreffen, werden diese ebenfalls in den *responseBuffer* geschrieben.

```
String partialResponse = event.content?.parts?
responseBuffer.write(partialResponse);
```

Sobald keine weiteren Antworten mehr eintreffen, wird der Inhalt des *responseBuffer* mit allen gesammelten Antworten in den String *completeResponse* transformiert.

```
onDone: () {
 String completeResponse = responseBuffer.toString();
```

Im nächsten Schritt wird geprüft, ob die erste Nachricht in der Liste *messages* vom Benutzer *geminiUser* stammt. Falls dies der Fall ist, wird die *completeResponse* an die erste Position der Liste eingefügt. Dadurch wird die vorherige Antwort der KI-API Gemini durch die *completeResponse* ersetzt. Diese Vorgehensweise stellt sicher, dass die letzte Antwort von Gemini im Chatverlauf als eine zusammenhängende Nachricht dargestellt wird. Ohne diese Prüfung könnte es passieren, dass die letzte Antwort von Gemini in mehrere separate Nachrichten aufgeteilt wird und den Chatverlauf somit unübersichtlicher machen.

```
setState(() {
 if (messages.isNotEmpty && messages.first.user == geminiUser) {
   messages[0] = ChatMessage(
     user: geminiUser,
     createdAt: DateTime.now(),
     text: completeResponse,
   );
 }
```

Die aktualisierte Liste sieht nach dem Aufruf der sendMessage Methode wie folgt aus:

```
List<ChatMessage> messages = [
ChatMessage(content: "I am fine." , user: geminiUser),
ChatMessage(content: "How are you?" , user: currentUser),
ChatMessage(content: "Hello! I'm Gemini, your assistant for summarizing
PDF files..." , user: geminiUser)
]
```

Auf der Startseite wird der Chatverlauf aus der Liste *messages* übersichtlich und korrekt dargestellt. Der Benutzer hat anschließend die Möglichkeit, entweder eine neue Frage zu stellen oder eine PDF-Datei über den Upload-Button hochzuladen. Falls sich der User für das Stellen einer neuen Frage entscheidet, beginnt die *sendMessage* Methode erneut. Wenn der User hingegen eine PDF-Datei hochlädt wird die *sendPDFMessage* Methode getriggert.

Abbildung 4.3.1: Startseite Chatverlauf
(eigene Darstellung)

4.4 sendPDFMessage Methode

Die Methode *sendPDFMessage* wird aufgerufen, sobald eine PDF-Datei über den Upload Button hochgeladen wird. Ihre Hauptfunktion besteht darin, den Text aus der hochgeladenen PDF zu extrahieren und diesen anschließend an die KI-API *Gemini* zu übermitteln. *Gemini* fasst den extrahierten Text zusammen und liefert eine Antwort zurück. Anhand des Ablaufplans (Abbildung 4.4) wird im Folgenden die Funktionsweise der Methode *sendPDFMessage* beispielhaft erklärt.

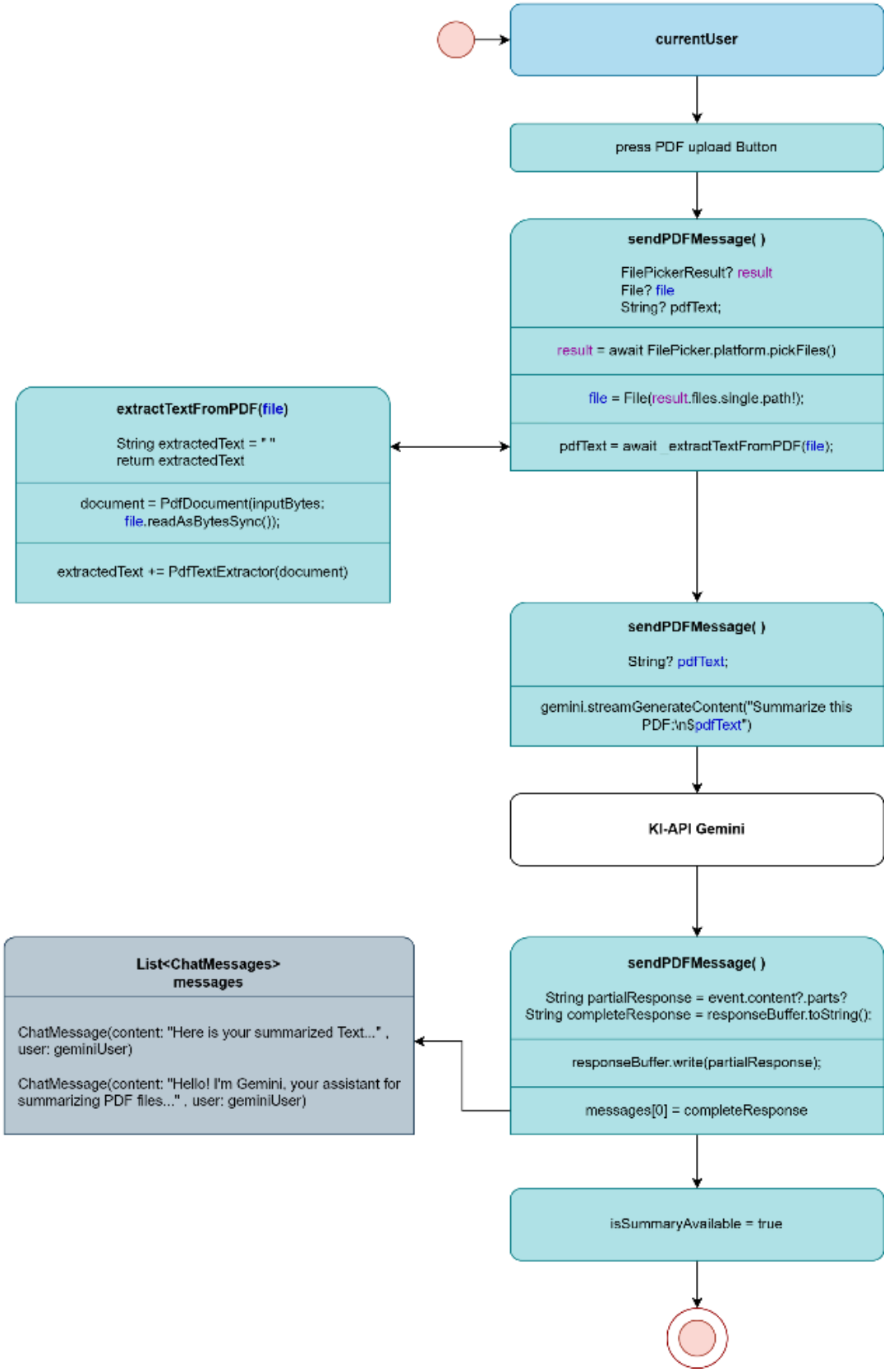

Abbildung 4.4: Ablaufplan der sendPDFMessage Methode
(eigene Darstellung)

Beispiel sendPDFMessage

Die Funktionalität zum Hochladen von PDF-Dateien wird mithilfe der in Flutter verfügbaren Bibliothek File Picker realisiert. Diese Bibliothek ermöglicht es, Dateien oder Verzeichnisse direkt vom Gerät des Nutzers auszuwählen. Für die Verwendung der Bibliothek genügt ein einfacher import-Befehl:

```
import 'package:file_picker/file_picker.dart';
```

Mit dem *FilePicker* kann der Nutzer eine PDF-Datei auswählen, die anschließend in der Variablen *result* gespeichert wird. Sollte der Upload fehlschlagen, wird die Fehlermeldung mithilfe der von Flutter bereitgestellten Methode *SnackBar( )* angezeigt. Die Fehlermeldung erscheint unterhalb des Chatverlaufs auf der Startseite und informiert somit den Nutzer klar und sichtbar über den Fehler. Wenn der Upload erfolgreich war, wird die PDF-Datei in der Variable *file* gespeichert und steht anschließend für die weitere Verarbeitung zur Verfügung.

```
void _sendPDFMessage() async {
 FilePickerResult? result;
 File? file;
 String? pdfText;

 try {
   result = await FilePicker.platform.pickFiles();

   if (result == null) {
     // Close loadingDialog
     Navigator.of(context).pop();
     // Show SnackBar ErrorMessage
     ScaffoldMessenger.of(context).showSnackBar(
       const SnackBar(content: Text('No file selected')),
     );
     print("No file selected");
     return;
   } else {
     file = File(result.files.single.path!);
   }
 } catch (e) {
   Navigator.of(context).pop();
   ScaffoldMessenger.of(context).showSnackBar(
     const SnackBar(content: Text('Error during FilePicking')),
   );
   print("Error during FilePicking: $e");
   return;
 }
```

Im nächsten Schritt wird die PDF-Datei, welche in der Variable *file* gespeichert ist, weiterverarbeitet. Um die PDF-Datei *file* verarbeiten zu können, wird die Bibliothek Syncfusion PDF verwendet. Diese Bibliothek ermöglicht das Extrahieren von Texten aus PDF-Dateien, selbst wenn die Dokumente komplexe Elemente wie Tabellen, Diagramme, Fußnoten oder spezielle Layouts enthalten. Die Bibliothek wurde für hohe Performance- und Speicheroptimierung entwickelt, was sie auch für umfangreiche Dokumente geeignet macht. Damit die Bibliothek in der Helfer-Methode *extractTextFromPDF* genutzt werden kann, wird wieder ein import-Befehl angegeben.

```
import 'package:syncfusion_flutter_pdf/pdf.dart';
```

Zu Beginn wird die *file* der Methode übergeben. Anschließend wird ein leerer String *extractedText* initialisiert, der als Speichervariable für den extrahierten Text dient. Im nächsten Schritt wird die *file* byteweise eingelesen und in der Variable *document* gespeichert. Danach wird für jede Dokumentenseite in *document* der Text extrahiert und an die Variable *extractedText angehängt*. Hierfür wird eine for-Schleife verwendet, die den Prozess so lange wiederholt, bis der Text aller Dokumentenseiten vollständig extrahiert wurde. Am Ende enthält der String *extractedText* den gesamten Inhalt der PDF-Datei *file*. Dieser String wird am Ende an die Methode *sendPDFMessage* zurückgeben.

```
Future<String> _extractTextFromPDF(File file) async {
 try {
   String extractedText = "";
   final PdfDocument document =
       PdfDocument(inputBytes: file.readAsBytesSync());

   for (int i = 0; i < document.pages.count; i++) {
     extractedText += PdfTextExtractor(document)
             .extractText(startPageIndex: i, endPageIndex: i) ??
         "";
   }
   return extractedText;
```

In der *sendPDFMessage* Methode angekommen, wird der zurückgegebene String *extratedText* in der Variable *pdfText* gespeichert. Anschließend wird überprüft, ob die Variable pdfText leer ist. Ist dies der Fall, wird dem Nutzer erneut eine Fehlermeldung in Form einer SnackBar angezeigt.

```
pdfText = await _extractTextFromPDF(file);
 if (pdfText.isEmpty) {
   Navigator.of(context).pop();
   ScaffoldMessenger.of(context).showSnackBar(
     const SnackBar(content: Text('No text in PDF to extract.')),
   );
}
```

Andernfalls wird der String *pdfText* an die KI-API *Gemini* weitergeleitet, um den Inhalt zusammenzufassen. Die Schnittstelle (API) beinhaltet die Kommunikation mit der Künstlichen Intelligenz *Gemini* von Google. Die API nutzt leistungsstarke Sprachmodelle für Natural Language Processing (NLP), eine Technologie, die Maschinen ermöglicht, menschliche Sprache zu verstehen, zu analysieren und zu verarbeiten. Durch die fortschrittlichen Algorithmen von NLP wird die Analyse und Zusammenfassung des gesendeten *pdfText* effizient und präzise. Die KI-API *Gemini* stellt die komplexen Inhalte in einer klaren und kompakten Form dar. Die Integration der Gemini-API wird wieder mit einem import-Befehl ausgeführt:

```
import 'package:flutter_gemini/flutter_gemini.dart';
```

Mit der Integration der Gemini-API kann nun eine Frage an die Künstlichen Intelligenz Gemini über die Methode *gemini.streamGenerateContent* gestellt werden. Wird eine PDF-Datei hochgeladen, wird automatisch die vordefinierte Anfrage *"Summarize this PDF: \n$pdfText"* an die Gemini-KI gesendet. Dadurch erhält der Nutzer eine Zusammenfassung der hochgeladenen PDF-Datei, ohne dass zusätzliche Eingaben erforderlich sind, was die Benutzererfahrung vereinfacht.

```
gemini.streamGenerateContent("Summarize this PDF:\n$pdfText")
```

Das Empfangen und Speichern der Antwort von Gemini erfolgt exakt wie im Beispiel der Methode sendMessage aus Abschnitt 4.3 und wird aus diesem Grund nicht weiter im Detail beschrieben. Die bestehende Funktionalität wird nahtlos übernommen, um eine konsistente Verarbeitung der Antworten sicherzustellen. Die aktualisierte Liste *messages* sieht nach dem Methodenaufruf von *sendPDFMessage* wie folgt aus:

```
List<ChatMessage> messages = [
ChatMessage(content: "Here is your summarized Text..." , user:
geminiUser)
ChatMessage(content: "Hello! I'm Gemini, your assistant for summarizing
PDF files..." , user: geminiUser)]
```

Als letzte Aktion wird das Flag *isSummaryAvailable* auf *true* gesetzt. Somit wird die Möglichkeit aktiviert, die generierte Zusammenfassung als PDF herunterzuladen.

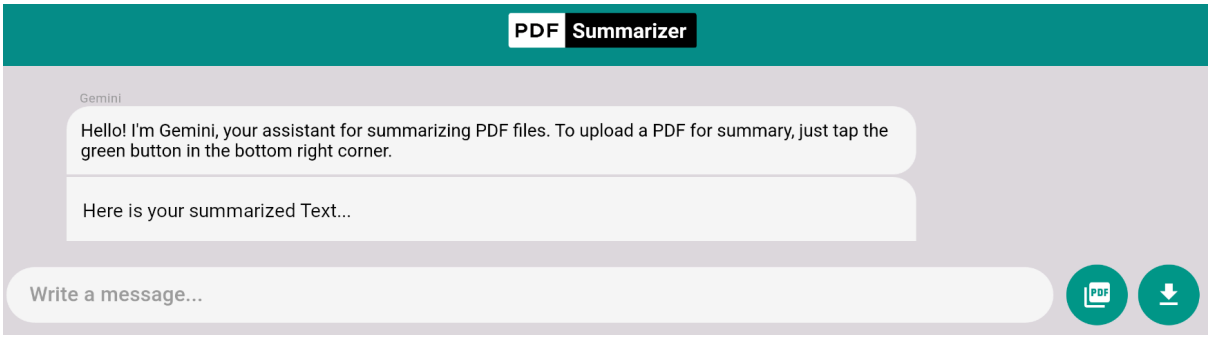

Abbildung 4.4.1: Startseite Chatverlauf mit Download Button aktiviert
(eigene Darstellung)

4.5 downloadResponseAsPDF Methode

Sobald der Download-Button getriggert wurde, wird die Methode *downloadResponseAsPDF* aktiviert. Im ersten Schritt der Methode wird überprüft, ob die letzte Chatnachricht nicht leer ist. Anschließend wird sich die letzte Chatnachricht vom User *geminiUser* in der Liste *messages* gesucht und in der Variable *lastGeminiResponse* gespeichert. Dies ist wichitg da somit sichergestellt wird, dass nur die letzte Nachricht von *Gemini* in die im nächsten Schritt erstellte PDF geschrieben wird.

```
void _downloadResponseAsPDF() async {
 if (messages.isNotEmpty) {
   ChatMessage? latestGeminiResponse =
       messages.firstWhere((message) => message.user == geminiUser);
```

Im nächsten Schritt wird mithilfe der *Syncfusion PDF* Bibliothek ein neues PDF-Dokument erstellt. Anschließend wird dem Dokument eine neue Seite hinzugefügt, auf welcher der Inhalt von *lastGeminiResponse* eingefügt wird.

```
   final PdfDocument document = PdfDocument();
   final PdfPage page = document.pages.add();
   page.graphics.drawString(latestGeminiResponse.text)
```

Damit die Methode *downloadResponseAsPDF* vollständig funktioniert, muss zusätzlich noch die Bibliothek *Path Provider* in das Projekt integriert werden. Dies geschieht, wie gewohnt, durch Hinzufügen des entsprechenden import-Befehls:

```
import 'package:path_provider/path_provider.dart';
```

Die Path Provider Bibliothek bietet die Möglichkeit, auf das Dateiverzeichnis des Systems zuzugreifen. Ein großer Vorteil dieser Bibliothek liegt darin, dass keine plattformabhängigen Verzeichnispfade manuell angegeben werden müssen. Stattdessen übernimmt Path Provider automatisch die Verwaltung und sorgt dafür, dass Dateien stets an den richtigen Orten gespeichert werden. Mit der Variablen *directory* wird das Dokumentenverzeichnis des Zielsystems ermittelt. Dadurch wird sichergestellt, dass Dateien an einem passenden Speicherort abgelegt werden.

```
final directory = await getApplicationDocumentsDirectory();
```

Anschließend wird der vollständige Speicherpfad für die PDF-Datei in der Variablen *path* festgelegt. Der Dateiname wird dabei aus einem formatierten Datum, getrennt durch einen _ und dem Anhang GeminiResponse.pdf zusammengesetzt.

```
String path = "${directory.path}/${formattedDate}_GeminiResponse.pdf";
```

Als nächstes wird in der Variablen file der Speicherpfad angegeben, und das Dokument wird an diesem Ort gespeichert.

```
File file = File(path);
await file.writeAsBytes(await document.save());
```

Zum Abschluss erhält der User eine Speicherbenachrichtigung in Form einer SnackBar, um ihn darüber zu informieren, an welcher Stelle das Dokument gespeichert wurde.

```
ScaffoldMessenger.of(context).showSnackBar(
 SnackBar(content: Text("PDF saved at: $path")),
);
```

5. Rekapitulation

5.1 Evaluation der Zielerreichung

Die Zielsetzungen der Anwendung konnten erfolgreich umgesetzt werden. Die Integration der *Syncfusion-PDF* Bibliothek in Verbindung mit der Methode *extractTextFromPDF* ermöglicht eine effiziente und zuverlässige Extraktion von Inhalten aus PDF-Dokumenten. Diese Funktionalität wurde vollständig implementiert und bildet einen wichtigen Grundpfeiler der Anwendung.

Auch die Schnittstelle zur KI-API wurde erfolgreich realisiert. Mit Hilfe der Bibliothek *flutter_gemini* wird eine reibungslose Kommunikation zwischen dem Tool und der KI-API sichergestellt, sodass ein direkter und problemloser Austausch gewährleistet ist.

Darüber hinaus wurde das Ziel, eine intuitive und benutzerfreundliche Oberfläche zu schaffen, ebenfalls erreicht. Der Chatbot führt den Benutzer gezielt durch die Anwendung und bietet, abhängig von der jeweiligen Fragestellung, hilfreiche Unterstützung. Zusätzlich sorgen klar gestaltete Elemente und der Einsatz von Icons dafür, dass die Benutzeroberfläche übersichtlich, leicht verständlich und visuell ansprechend ist.

5.2 Herausforderungen während der Umsetzung

Während der Entwicklung traten einige Herausforderungen auf. Ein zentrales Problem war die Verwaltung des Chatverlaufs, welcher sich als sehr fehleranfällig erwies. Die Liste *messages*, welche den Chatverlauf repräsentiert, führte häufig zu unerwarteten Fehlern. So wurde zum Beispiel auf falsche Chatnachrichten geantwortet, Nachrichten wurden nicht angezeigt oder im schlimmsten Fall sogar überschrieben. Dies hing damit zusammen, wie die Chatnachrichten in der Liste *messages* gespeichert wurden. Erst durch die Implementierung des *responseBuffers* konnte dieses Problem erfolgreich behoben werden.

Ein weiteres, noch ungelöstes Problem betrifft die Formatierung der heruntergeladenen PDF-Datei. Während die Antworten im Chat korrekt als Markdown-Text angezeigt werden, wird diese Formatierung in der PDF-Datei nicht übernommen. Leider konnte hierfür bisher keine Lösung gefunden werden, wodurch die Darstellung in der PDF-Datei noch verbesserungswürdig bleibt.

5.3 Mögliche Erweiterungen und zukünftige Entwicklungen

Die Anwendung bietet Möglichkeiten für zukünftige Erweiterungen, die ihre Funktionalität und Benutzerfreundlichkeit weiter steigern könnten. Aktuell ist die App für Android verfügbar, doch durch leichte Anpassungen im Code ließe sie sich problemlos auch für Web- und iOS-Plattformen bereitstellen.

Zudem könnte die Darstellung der heruntergeladenen PDFs durch optische Verbesserungen und eine bessere Strukturierung ansprechender gestaltet werden.

Ein weiteres Potenzial liegt in der Weiterentwicklung der KI-Funktionalität. Die Gemini-KI könnte durch gezieltes Training mit zusätzlichen Restriktionen so optimiert werden, dass sie sich ausschließlich auf die Bearbeitung von PDF-Dokumenten konzentriert und auf die Unterstützung bei Fragen zur Verwendung des Tools spezialisiert.

Diese Erweiterungen bieten eine solide Grundlage, um die Anwendung zukünftig noch vielseitiger, effizienter und benutzerfreundlicher zu gestalten.

6. Fazit

Die Entwicklung des Tools stellt einen vielversprechenden Ansatz dar, um mit der stetig wachsenden Menge an wissenschaftlicher Literatur effizienter umzugehen. Durch die erfolgreiche Implementierung von Funktionen wie der Textextraktion aus PDF-Dokumenten und der Integration einer KI-API zur automatisierten Analyse und Zusammenfassung wissenschaftlicher Inhalte wurde ein kleiner Schritt zur Entlastung von Studierenden und Dozenten gemacht. Die Studierenden profitieren von der Zeitersparnis und der Möglichkeit, schnell relevante Informationen aus einer Vielzahl von Publikationen zu extrahieren. Dozenten hingegen erhalten durch die automatisierte Analyse weitere Unterstützung bei der Bewertung wissenschaftlicher Arbeiten.

Insgesamt zeigt sich, dass das Tool einen kleinen Beitrag zur Effizienzsteigerung im akademischen Umfeld leisten kann.

I. Abbildungsverzeichnis