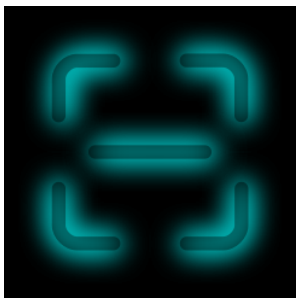


jjscan

Jörg J. Buchholz

10. September 2025

Teil I

Bedienungsanleitung

Kapitel 1

Einführung

jjscan [1] ist eine Android-App, die die Seiten eines Papierdokuments fotografiert, zuschneidet, in einem PDF zusammenfasst und in einen definierten Ordner auf dem Smartphone schiebt. Von da wird das PDF mit Syncthing (Abschnitt 1.2) auf einen Server synchronisiert und dort mit Paperless-ngx (Abschnitt 1.1) in ein Dokumentenmanagementsystem integriert.

1.1 Paperless-ngx

Paperless-ngx [2], [3] ist ein freies, quelloffenes Dokumentenmanagementsystem (DMS), das physische Dokumente in ein durchsuchbares digitales Archiv umwandelt und das in einem Docker-Container nativ unter Linux läuft. Paperless ist ideal, um die täglich hereinkommende Papierpost direkt in Paperless' `consume`-Ordner zu scannen, von wo das Programm die Dokumente automatisch liest, Texterkennung (OCR) durchführt, den Korrespondenten und den Dokumenttyp ermittelt und das Dokument mit Tags versieht (Abbildung 1.1).

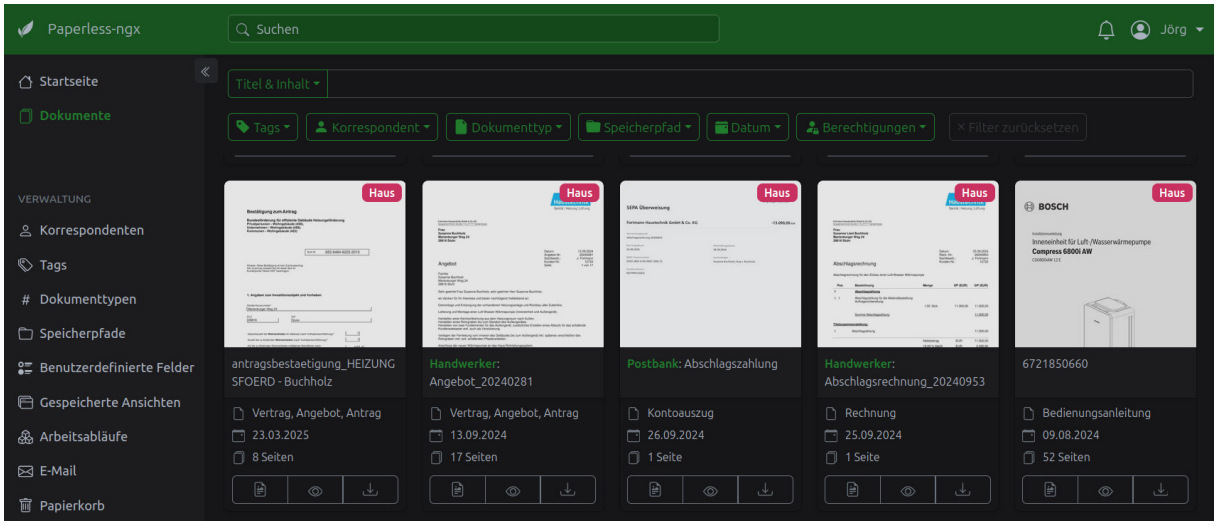


Abbildung 1.1: Paperless-ngx

Als Alternative zu einem guten teuren Einzugsscanner, der auch dünne, kleine Papiere zerstörungsfrei einscannt, bietet es sich heutzutage an, die Papierdokumente einfach mit

dem Smartphone zu „fotografieren“ und direkt an den `consume`-Ordner des Paperless-Servers zu schicken.

1.2 Syncthing

Zur automatischen Übertragung vom Smartphone zum Paperless-Server eignet sich beispielsweise Syncthing [4], [5]. Wikipedia schreibt dazu:

> *Syncthing ist eine quelloffene Software zum Datenabgleich zwischen Geräten. Der Abgleich erfolgt auf Basis von Peer-to-Peer-Übertragungen in einem lokalen Netzwerk oder über das Internet.*

Es reicht also, in Syncthing auf dem Smartphone (Abbildung 1.2) einen Quellordner und auf dem Server einen Zielordner freizugeben

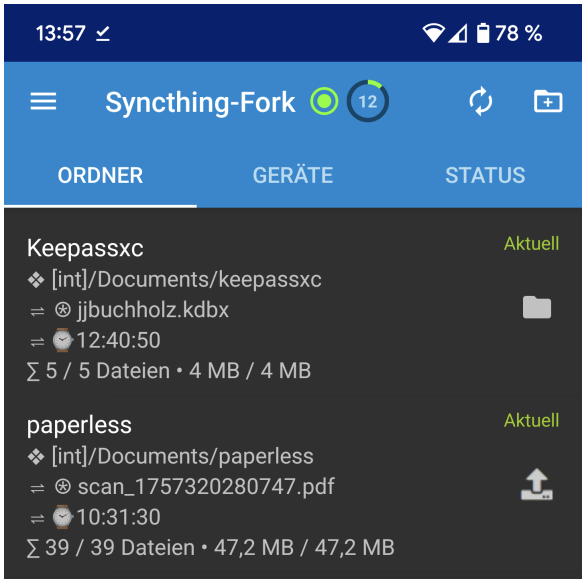


Abbildung 1.2: Syncthing auf dem Smartphone

Syncthing kümmert sich dann selbstständig um die Übertragung aller Dateien[1] aus dem Quellordner des Smartphones in den Zielordner des Paperless-Servers.

1.3 jjscan

Dann brauchen wir jetzt also nur noch ein kleines Programm auf dem Android-Smartphone, das dessen Kamera nutzt, um die Seiten eines neuen Papierdokumentes zu fotografieren, diese dann möglichst intelligent zuschneidet, in einem PDF zusammenfasst und dieses in den angegebenen Ablageordner schiebt.

Natürlich gibt es genau das in den unterschiedlichen Varianten im Google Play Store. Allerdings hat jede der getesteten Anwendungen ihre eigenen Nachteile: Entweder

- es kann der Ablageordner nicht fest verdrahtet werden oder

[1] Aus Abbildung 1.2 geht hervor, dass der Autor Syncthing auch nutzt, um Kennwörter auf all seinen Geräten synchron zu halten.

- das Zuschneiden oder das Fotografieren lässt sich nicht automatisieren oder
- es wird nicht automatisch ein PDF erzeugt oder
- es wird ständig um Geld gebettelt oder ...
- oder ... oder ...

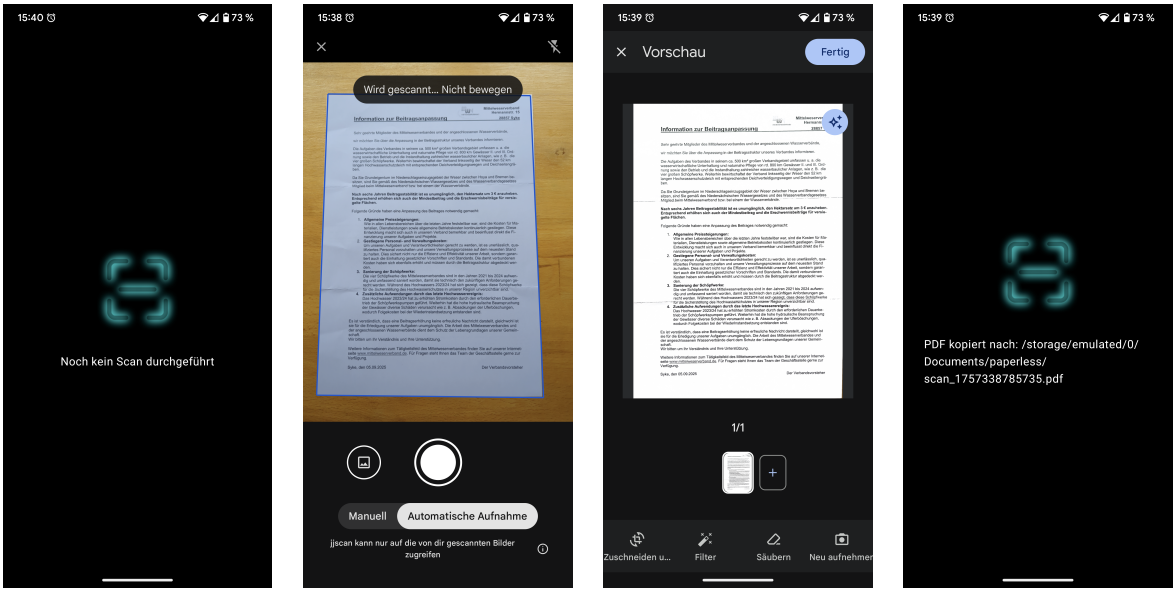
Ergo – selbst ist der Mensch! Wir nutzen diese Gelegenheit, um erstmalig[2] eine native Android App mit Android Studio zu erstellen (Teil II), die exakt das leistet, was wir uns wünschen.

[2]Die bisherigen Smartphone-Anwendungen aus der Feder des Autors waren browser-basierte Apps, die zwar den Vorteil haben, dass sie damit auf jedem beliebigen Smartphone (also auch unter iOS) laufen, aber eben doch nicht effizient auf alle Systeme eines Android-Phones zugreifen können.

Kapitel 2

Wie geht das?

Die Bedienung von jjscan ist denkbar einfach. Abbildung 2.1 zeigt die vier[1] Seiten, über die wir mit jjscan interagieren.



(a) Startseite (b) Scanseite (c) Vorschauseite (d) Speicherseite

Abbildung 2.1: jjscan besteht aus vier Seiten.

2.1 Startseite

Auf der Startseite (Abbildung 2.1a) gibt es über der Information, dass noch kein Scan durchgeführt wurde, das jjscan-Logo als Schaltfläche. Ein Anklicken der Schaltfläche führt unmittelbar zur Scanseite (Abschnitt 2.2).

[1] Eigentlich hätten wir auf die erste und vierte Seite auch verzichten können. Die eigentliche Magie findet auf den mittleren beiden Seiten statt. Und eigentlich ist die vierte Seite auch wieder die erste Seite; nur dass der Informationstext ein anderer ist.

2.2 Scanseite

Die Scanseite (Abbildung 2.1b) basiert zum Großteil auf Googles Machine Learning Kit (ML-Kit) [6], das freundlicherweise unter anderem auch eine Schnittstelle (API) zur Dokumentenerfassung auf Smartphones zur Verfügung stellt. Google schreibt dazu:

> *Once the document scanner flow is triggered from your app, users retain full control over the scanning process. They can optionally crop the scanned documents, apply filters, remove shadows or stains, and easily send the digitized files back to your app.*

In der Tat ist es auf der Scanseite sehr einfach möglich, eine Seite eines Papierdokuments automatisch zu scannen, da „Automatische Aufnahme“ voreingestellt ist. Es reicht also, das Smartphone ungefähr waagerecht über die Seite zu halten, sodass die Seite den Scanrahmen in etwa ausfüllt. Eine KI versucht dann selbstständig interaktiv, das Dokument vom Hintergrund abzugrenzen, indem sie einen blauen Rahmen um die erkannte Seite legt. Häufig ist es dabei sinnvoll, das Telefon ein wenig zu bewegen (auch hoch und runter) und zu kippen, bis die Seite korrekt erkannt wird. Das Fotografieren und Ausrichten des Dokuments selbst findet dann automatisch statt und die Anwendung wechselt selbstständig auf die Vorschauseite (Abschnitt 2.3).

In den meisten Fällen ist es vorteilhaft, zum Scannen alle Deckenleuchten auszuschalten, und die Smartphone-eigene Beleuchtung (Blitzzeichen oben rechts auf der Scanseite) einzuschalten[2], um Schatten der Hände und des Smartphones selbst zu vermeiden. Gegebenenfalls können wir auf der Vorschauseite versuchen, Schatten[3] wieder zu entfernen, was unserer Erfahrung nach nicht immer gut funktioniert.

Wenn wir tatsächlich auf die KI-gesteuerte *Automatische Aufnahme* verzichten wollen (wofür wir eigentlich noch keinen Grund gefunden haben) können wir den links daneben angeordneten *Manuell*-Button anklicken und den Scanvorgang dann händisch auslösen. Außerdem haben wir auf der Scanseite die Möglichkeit, nach Anklicken des Foto-Symbols (links) ein schon vorher abgespeichertes Foto in das aktuelle Dokument einzufügen.

2.3 Vorschauseite

Auf der Vorschauseite (Abbildung 2.1c)

- können wir das Foto nachträglich zuschneiden[4], wenn wir – warum auch immer – nicht die gesamte Seite in Paperless speichern möchten,
- können wir diverse Filter (Automatisch, Farbe, Graustufen, S/W, Schatten) verwenden, um zu versuchen, die Qualität des Scans zu optimieren,
- können wir den *Säubern*-Button ausprobieren, um die KI zu motivieren, mit den Fingern markierte Bereiche zu „säubern“, was allerdings selten zum gewünschten Ergebnis führt,

[2] Leider können wir in der aktuellen Version von [6] den Blitz nicht voreingestellt aktivieren.

[3] Wie immer im Leben ist es aber natürlich sinnvoll, Fehler erst gar nicht zu machen, anstatt später zu versuchen, ihre Auswirkungen zu kompensieren ...

[4] Die *Zuschneiden*-Seite bietet auch die Möglichkeit, das Bild um Vielfache von 90° zu drehen, was aber erfahrungsgemäß sehr selten notwendig ist, da die KI auch ein Drehen des Smartphones (Portrait ↔ Landscape) recht zuverlässig erkennt. Wenn wir feststellen, dass die KI den Ausschnitt nicht korrekt gewählt hat, ist es in den meisten Fällen einfacher, den *Neu-aufnehmen*-Button zu verwenden, um erneut auf die Scanseite zu gelangen.

- können wir – wenn wir mit dem Scanergebnis nicht zufrieden sind – die Aufnahme dieser Seite wiederholen, indem wir auf den *Neu-aufnehmen*-Button anklicken.

Seit Neuestem gibt es auf der Vorschauseite rechts oben einen voreingestellt aktivierten *Drei-Sterne-Zauber*-Button, der in [6] noch nicht dokumentiert ist, der aber tatsächlich zu recht ansehnlichen Verbesserungen des Scans führt.

Wenn wir mit dem aktuellen Scan zufrieden sind, haben wir zwei Möglichkeiten:

1. Wir können das Scannen des gesamten Dokuments abschließen, indem wir ganz oben rechts den *Fertig*-Button verwenden, was uns unmittelbar auf die Speicherseite (Abschnitt 2.4) führt.
2. Wir können weitere Scans zum Gesamtdokuments hinzufügen, indem wir den *Plus*-Button rechts neben dem aktuellen Scan anklicken, um ein weiteres Mal die Scanseite (Abschnitt 2.2) zu besuchen.

2.4 Speicherseite

Auf der Speicherseite (Abbildung 2.1d) wird entweder eine Fehlermeldung ausgegeben, wenn das Abspeichern des Dokuments nicht geklappt hat, beispielsweise, weil wir den Scan abgebrochen haben oder es wird als Erfolg vermeldet, dass das PDF in den Ordner `/storage/emulated/0/Documents/paperless` abgespeichert wurde. Um sicherzustellen, dass der Name der Datei eindeutig ist, beinhaltet er, unter Verwendung der Unixzeit [7], die Anzahl der Sekunden, die seit dem 1. Januar 1970 UTC vergangen sind (beispielsweise `scan_1678886400000.pdf`).

Teil II

Unter der Haube

Kapitel 3

Einführung

In früheren Projekten hatten wir jede einzelne Zeile selbst programmiert und mit unseren eigenen erklärenden Kommentaren versehen. In jjscan haben wir uns wieder massiv von künstlicher Intelligenz unterstützen lassen:

Während des Programmierens hat uns Gemini[1] in Android Studio [8] (bzw. der Github Copilot [9]) hilfreich über die Schulter geschaut und eigene Ideen und Verbesserungen des Codes angeboten, auf die wir sehr häufig eingegangen sind.

Nach Abschluss der Programmierung haben wir Gemini genutzt, um alle Methoden und jede einzelne Codezeile kommentieren zu lassen und ein paar einleitende Worte über die einzelnen Funktionen der Anwendung zu schreiben. Das klappt erstaunlich gut. Dabei müssen wir – nicht ganz ohne Neid – eingestehen, dass die Kommentare manchmal treffsicherer und prägnanter formuliert sind, als wir das ohne langes Nachdenken hinbekommen hätten. Wir sparen uns daher in den folgenden Kapiteln 4 und 5 jegliche eigene Erklärungen und zeigen ausschließlich die von der KI erzeugten Erläuterungen und Kommentare.

[1] Das von Google bereitgestellte Gemini scheint – verständlicherweise – etwas besser die spezifischen Androidprobleme lösen zu können als die anderen KIs des Github Copilots.

Kapitel 4

Geminis App-Geschichte

Wir haben Gemini in Android Studio aufgefordert, eine allgemeinverständliche[1], ausführliche Beschreibung der einzelnen Bestandteile der Anwendung zu erstellen. Um diese dann aus Android Studio formatiert in dieses LyX-Dokument übernehmen zu können, haben wir Gemini des Weiteren aufgefordert, seine Beschreibung als schön formatiertes Markdown auszugeben und uns auch gleich den passenden `pandoc`-Befehl mitzuteilen, um das Markdown-Dokument in Linux Mint nach LaTeX zu konvertieren:

```
pandoc --from markdown-auto_identifiers --to latex --output
   gemini_jjscan.tex --wrap=none --standalone --variable=mainfont:
   "DejaVu Serif" --variable=sansfont:"DejaVu Sans" --variable=
   monofont:"DejaVu Sans Mono" --variable=mathfont:"Latin Modern
   Math" --variable=lang:de-DE --variable=documentclass:article -M
    listings gemini_jjscan.md
```

Der Befehl erzeugt eine direkt kompilierbare LaTeX-Datei, die wir in ein weiteres LyX-Dokument importieren und von dort ohne weitere Anpassungen in dieses Dokument kopieren können. Hier nun also Geminis Märchenstunde:

4.1 Die Reise einer gescannten Seite: Wie deine Android-App Dokumente digitalisiert und verwaltet

Stell dir vor, du hast ein wichtiges Papierdokument – vielleicht eine Rechnung, eine Notiz oder eine Zeichnung – und möchtest es sicher und schnell auf deinem Smartphone speichern, idealerweise als PDF oder Bild. Genau das ermöglicht die Android-App, deren Funktionsweise wir hier beleuchten. Sie nutzt moderne Technologien, um diesen Prozess so einfach und benutzerfreundlich wie möglich zu gestalten.

4.1.1 Der Startpunkt: Die `Activity` und das Aussehen (`Theme`)

Jede Android-App hat einen oder mehrere „Bildschirme“, mit denen der Nutzer interagiert. In der Android-Welt nennt man einen solchen Bildschirm eine **`Activity`**. Unsere

[1] Der Begriff „allgemeinverständlich“ hat Gemini offensichtlich motiviert, einen etwas peinlichen Onkel-Hotte-Stil anzuschlagen; egal, die technischen Erklärungen selbst sind trotzdem erste Sahne!

App startet mit einer Haupt-`Activity`. Sobald diese `Activity` gestartet wird, muss festgelegt werden, wie sie aussieht und was sie anzeigt. Hier kommt **Jetpack Compose** ins Spiel. Compose ist ein modernes Toolkit von Google, um Benutzeroberflächen (kurz **UI** für User Interface) auf deklarative Weise zu erstellen. Anstatt Schritt für Schritt zu beschreiben, *wie* etwas gezeichnet wird, beschreibt man mit Compose, *was* auf dem Bildschirm erscheinen soll.

Damit die App ein einheitliches Erscheinungsbild hat – also definierte Farben, Schriftarten und Formen – verwendet sie ein **`Theme`**. Das ist wie eine Design-Vorlage, die sicherstellt, dass alle Elemente der App zueinander passen und professionell aussehen. Die `setContent`-Funktion innerhalb der `Activity` ist der Startschuss, um die mit Compose definierte UI, umhüllt vom `Theme`, auf dem Bildschirm darzustellen.

4.1.2 Die Benutzeroberfläche: Schaltflächen und Informationsanzeige mit Compose

Die Hauptansicht unserer App ist recht übersichtlich gestaltet. Wir sehen einen Titel, eine große Schaltfläche (einen `IconButton`, der ein Icon anzeigt) und einen Textbereich, der dem Nutzer Rückmeldung gibt (z.B. „Noch kein Scan durchgeführt“). All diese Elemente werden als **Composable-Funktionen** definiert.

- **`Surface` und `Column`**: Die Basis bildet oft eine `Surface`, eine Art Leinwand, die den Hintergrund setzt. Darauf werden Elemente in einer `Column` (Spalte) vertikal untereinander angeordnet.
- **`Text`**: Für die Anzeige von Text (Titel, Nachrichten) wird die `Text`-Composable verwendet.
- **`IconButton` und `Image`**: Der zentrale Knopf zum Starten des Scans ist ein `IconButton`. In ihm befindet sich eine `Image`-Composable, die das eigentliche Icon anzeigt.
- **`Modifier`**: Jedes dieser UI-Elemente kann mit einem **`Modifier`** angepasst werden. Ein `Modifier` ist wie ein Satz von Anweisungen, der einem Element sagt, wie es sich verhalten oder aussehen soll – zum Beispiel seine Größe (`Modifier.size()`), den Abstand zu anderen Elementen (`Modifier.padding()`) oder wie es den verfügbaren Platz ausfüllen soll (`Modifier.fillMaxSize()`).
- **`Spacer`**: Um Abstände zwischen Elementen zu schaffen, werden `Spacer` verwendet.

4.1.3 Der Star der Show: Der ML Kit Document Scanner

Das Herzstück der App ist die Fähigkeit, Dokumente zu scannen. Hierfür greift die App auf eine spezielle Bibliothek von Google zurück: den **ML Kit Document Scanner**. Bevor dieser Scanner verwendet werden kann, muss er konfiguriert werden.

- **`GmsDocumentScannerOptions` (Optionen)**: Man erstellt ein **Objekt** (eine Instanz einer Klasse, die Daten und zugehörige Funktionen bündelt) vom Typ `GmsDocumentScannerOptions`. Dieses Objekt hält verschiedene Einstellungen fest:
  - **Scan-Modus**: Soll der Nutzer eine volle Bearbeitungsoberfläche bekommen (`SCANNER_MODE_FULL`) oder einen einfacheren Modus?
  - **Ergebnisformate**: Soll das Ergebnis als PDF (`RESULT_FORMAT_PDF`), als Bild (`RESULT_FORMAT_JPEG`) oder beides gespeichert werden?

  - **Seitenlimit**: Wie viele Seiten dürfen maximal gescannt werden?

- **Der Scanner-Client**: Mit diesen **Optionen** wird dann ein „Client“ für den Scanner über `GmsDocumentScanning.getClient()` geholt. Dieser Client ist unsere Schnittstelle zum eigentlichen Scan-Vorgang.

4.1.4 Der Scanvorgang: Intents, Launcher und Ergebnisse

Wenn der Nutzer auf den Scan-Button tippt, passiert Folgendes:

1. **Intent anfordern**: Die App bittet den Scanner-Client, einen `IntentSender` zu liefern. Ein `Intent` in Android ist eine Art Nachricht, die eine Aktion beschreibt, die ausgeführt werden soll – in diesem Fall das Starten der Scan-Oberfläche. Der `IntentSender` ist ein spezielles Token, das es unserer App erlaubt, diesen `Intent` im Namen einer anderen Anwendung (der Scanner-App) zu starten.
2. **Der `ActivityResultLauncher`**: Unsere App muss darauf vorbereitet sein, das Ergebnis des Scanvorgangs (also die gescannten Dateien) zu empfangen. Hierfür wird ein `rememberLauncherForActivityResult` verwendet. Das ist eine clevere Funktion in Compose, die einen „Starter“ registriert. Dieser Starter weiß, wie man eine andere `Activity` (die Scan-Oberfläche) startet und wie man das Ergebnis verarbeitet, wenn diese `Activity` beendet wird. „Remember“ bedeutet hier, dass dieser Launcher auch dann noch existiert und funktioniert, wenn sich Teile der UI aufgrund anderer Zustandsänderungen neu aufbauen.
3. **Scan starten**: Mit dem erhaltenen `IntentSender` wird der `scannerLauncher` angewiesen, die Scan-Oberfläche zu starten. Der Nutzer sieht nun die Scan-Ansicht, kann Dokumente erfassen, zuschneiden, Filter anwenden etc.
4. **Ergebnisverarbeitung**: Wenn der Nutzer den Scan abschließt (oder abbricht), sendet die Scan-Oberfläche ein Ergebnis zurück an unsere App. Der `scannerLauncher` fängt dieses Ergebnis ab.
   - **Erfolg oder Misserfolg**: Zuerst wird geprüft, ob der Scan erfolgreich war (`Activity.RESULT_OK`).
   - **Daten extrahieren**: Bei Erfolg werden die Daten aus dem Ergebnis-`Intent` extrahiert. Das Ergebnis (`GmsDocumentScanningResult`) enthält entweder eine `Uri` zur erstellten PDF-Datei oder eine Liste von `Uris` zu den einzelnen gescannten Bildseiten. Eine `Uri` (Uniform Resource Identifier) ist wie eine eindeutige Adresse zu einer Datei.
   - **Nachricht aktualisieren**: Basierend auf dem Ergebnis (PDF, Bilder oder Fehler) wird eine Zustandsvariable `message` aktualisiert. Diese Variable ist mit `remember { mutableStateOf(...) }` deklariert. Das bedeutet, sie ist ein veränderlicher Zustand, der von Compose „erinnert“ wird. Wenn sich der Wert von `message` ändert, weiß Compose, dass es die `Text`-Composable, die diesen Wert anzeigt, neu zeichnen muss. Die Verwendung von `by` (z.B. `var message by remember ...`) ist eine **Delegat**-Eigenschaft in Kotlin. Sie delegiert das Speichern und Verwalten des Zustands an die `remember` und `mutableStateOf` Funktionen, macht den Code aber lesbarer.

4.1.5 Dateiverwaltung: Kopieren in den „paperless“-Ordner

Nachdem die gescannten Dateien als `Uris` vorliegen, möchte die App sie an einem organisierten Ort speichern, nämlich im öffentlichen „Documents“-Verzeichnis des Geräts, in einem Unterordner namens „paperless“.

- **`copyToPaperlessFolder` Funktion**: Diese Hilfsfunktion nimmt den Android-**`Context`** (ein Objekt, das Zugriff auf globale App-Informationen und Systemdienste bietet), die Quell-`Uri` und einen gewünschten Dateinamen entgegen.
- **`ContentResolver` (Resolver)**: Um auf den Inhalt zuzugreifen, auf den die `Uri` zeigt (denn eine `Uri` ist nur eine Adresse, nicht die Datei selbst), wird ein **`ContentResolver`** verwendet. Der `Resolver` kann einen **`InputStream`** für die Quelldatei öffnen. Ein `InputStream` ist ein Datenstrom, aus dem Bytes gelesen werden können.
- **Zielordner und Datei**: Die Funktion überprüft, ob der „paperless“-Ordner existiert, und erstellt ihn gegebenenfalls. Dann wird ein `File`-Objekt für die Zieldatei erstellt.
- **Kopiervorgang mit `copyStream`**:
  - Ein `FileOutputStream` wird für die Zieldatei geöffnet. Dies ist ein Datenstrom, in den Bytes geschrieben werden können.
  - Die eigentliche Kopierarbeit leistet die `copyStream`-Funktion. Sie liest Daten in Blöcken aus dem `InputStream` in einen temporären Speicherbereich, einen sogenannten **`Buffer`** (Puffer). Ein `Buffer` ist einfach ein Byte-Array. Das blockweise Lesen und Schreiben ist effizienter als Byte für Byte.
  - Die gelesenen Bytes aus dem `Buffer` werden dann in den `FileOutputStream` geschrieben, bis die gesamte Quelldatei kopiert ist.
  - Wichtig ist hier die Verwendung von `.use { ... }`. Dies ist eine Kotlin-Konstruktion, die sicherstellt, dass die **`Stream`**-Objekte (Input und Output) nach Gebrauch automatisch geschlossen werden, auch wenn Fehler auftreten. Das ist wichtig, um Ressourcen freizugeben.
- **Pfad zurückgeben**: Die Funktion gibt den absoluten Pfad zur neu kopierten Datei zurück, damit dieser dem Nutzer in der `message` angezeigt werden kann.

4.1.6 Lambda-Ausdrücke für kompakten Code

An vielen Stellen, z.B. bei der `onClick`-Aktion des Buttons oder bei der Ergebnisverarbeitung des `scannerLauncher`, werden **Lambda**-Ausdrücke verwendet. Ein Lambda-Ausdruck ist eine kurze, anonyme Funktion (eine Funktion ohne Namen), die direkt an Ort und Stelle definiert werden kann. Zum Beispiel `onClick = { /* Code, der bei Klick ausgeführt wird */ }`. Das `->`-Symbol trennt dabei die Parameter des Lambdas (falls vorhanden) vom Funktionskörper. Lambdas machen den Code oft kürzer und lesbarer, besonders für kleine, einmalige Aktionen.

4.1.7 Importe (`import`)

Am Anfang der Datei sieht man viele `import`-Anweisungen. Da Kotlin und Android aus vielen verschiedenen Bibliotheken und Modulen bestehen, muss man explizit angeben, wel-

che Klassen und Funktionen man verwenden möchte. Ein `import com.example.jjscan-.ui.theme.JjscanTheme` sagt dem Compiler beispielsweise: „Ich möchte das `JjscanTheme` verwenden, das im Paket `com.example.jjscan.ui.theme` definiert ist.“

4.1.8 Zusammenfassung der Reise

Die App startet also eine `Activity` mit einer Compose-basierten UI. Der Nutzer tippt auf einen Button, was den ML Kit Document Scanner über einen `Intent` und einen `ActivityResultLauncher` startet. Nach dem Scan liefert der Scanner eine `Uri` zu einer PDF oder zu Bildern. Diese Dateien werden dann mithilfe von `ContentResolver`, `InputStream`, `FileOutputStream` und einem `Buffer` in einen speziellen Ordner kopiert. Während des gesamten Prozesses wird der Nutzer über eine `Text`-Composable auf dem Laufenden gehalten, deren Inhalt dynamisch durch einen `MutableState` aktualisiert wird. All dies wird durch die Struktur von Kotlin, die Mächtigkeit von Jetpack Compose und die Funktionalität der Android-Systemdienste und externen Bibliotheken wie ML Kit ermöglicht.

Kapitel 5

Kommentierter Quelltext

Der Code des eigentlichen Kotlin-Programmes umfasst tatsächlich nur 154 Zeile Code, da die Hauptarbeit ja von Googles ML Kit Document Scanner durchgeführt wird. Wir haben Gemini allerdings aufgefordert, jede Codezeile ausführlich zu kommentieren, sodass das dargestellte Programm wesentlich umfangreicher anmutet:

```
// Definiert das Paket (package), zu dem diese Kotlin-Datei
   gehört.
// Pakete dienen zur Organisation von Code und zur Vermeidung von
    Namenskonflikten.
package com.example.jjscan

// Importiert die Klasse 'Activity' aus dem Android Framework.
// Eine Activity repräsentiert einen einzelnen Bildschirm mit
   einer Benutzeroberfläche.
import android.app.Activity
// Importiert die Klasse 'Uri' (Uniform Resource Identifier).
// Uris werden verwendet, um auf Ressourcen zu verweisen, z.B.
   Dateien im Dateisystem oder im Netzwerk.
import android.net.Uri
// Importiert die Klasse 'Bundle' aus dem Android Framework.
// Bundles werden verwendet, um Daten zwischen verschiedenen
   Android-Komponenten (z.B. Activities) zu übergeben oder um
   Zustände zu speichern und wiederherzustellen.
import android.os.Bundle
// Importiert die Klasse 'Environment' aus dem Android Framework.
// Diese Klasse bietet Zugriff auf Standard-Umgebungsvariablen
   und Verzeichnisse, z.B. das öffentliche Dokumentenverzeichnis.
import android.os.Environment
// Importiert die Klasse 'ComponentActivity', eine Basisklasse
   für Activities, die Jetpack-Bibliotheken wie ViewModel und
   Compose verwendet.
import androidx.activity.ComponentActivity
// Importiert die Composable-Funktion '
   rememberLauncherForActivityResult'.
// Diese Funktion wird in Compose verwendet, um einen Activity
   Result Launcher zu erstellen und zu speichern, der den
   Lifecycle der Composable-Funktion überlebt.
```

```
// Sie dient dazu, Ergebnisse von Activities zu erhalten, die für
    ein Resultat gestartet wurden (z.B. Dateiauswahl, Kamerabild).
import androidx.activity.compose.
   rememberLauncherForActivityResult
// Importiert die Erweiterungsfunktion 'setContent' für '
   ComponentActivity'.
// Diese Funktion wird verwendet, um den UI-Inhalt einer Activity
    mit Jetpack Compose zu definieren.
import androidx.activity.compose.setContent
// Importiert die Klasse 'IntentSenderRequest'.
// Diese Klasse kapselt eine Anfrage zum Starten eines '
   IntentSender', oft verwendet in Verbindung mit '
   ActivityResultLauncher' für komplexere Start-Szenarien.
import androidx.activity.result.IntentSenderRequest
// Importiert die Klasse 'ActivityResultContracts'.
// Diese Klasse stellt eine Sammlung von vordefinierten "
   Verträgen" für den Activity Result API bereit,
// die definieren, wie eine Activity gestartet wird und wie das
   Ergebnis zurückgegeben wird (z.B. für das Aufnehmen eines
   Bildes).
import androidx.activity.result.contract.ActivityResultContracts
// Importiert die Composable-Funktion 'Image', die zum Anzeigen
   von Bildressourcen in der Compose UI verwendet wird.
import androidx.compose.foundation.Image
// Importiert Layout-bezogene Composable-Funktionen und Modifier-
   Erweiterungen aus dem 'androidx.compose.foundation.layout'-
   Paket.
// Das Sternchen (*) ist ein Wildcard-Import, der alle
   öffentlichen Deklarationen aus diesem Paket importiert (z.B.
   Column, Row, Spacer, padding, fillMaxSize).
import androidx.compose.foundation.layout.*
// Importiert speziell 'Arrangement' aus dem Layout-Paket. Obwohl
    durch den Wildcard-Import oben bereits abgedeckt,
// kann ein expliziter Import die Lesbarkeit für dieses
   spezifische Element erhöhen oder bei Namenskonflikten helfen.
// 'Arrangement' wird verwendet, um die Anordnung von Kind-
   Elementen innerhalb von Layout-Composables wie Row oder Column
   zu steuern (z.B. Arrangement.Center).
import androidx.compose.foundation.layout.Arrangement
// Importiert UI-Komponenten und Hilfsklassen aus dem Material
   Design 3 Paket für Jetpack Compose.
// Das Sternchen (*) ist ein Wildcard-Import und inkludiert z.B.
   Button, Text, Surface, MaterialTheme.
import androidx.compose.material3.*
// Importiert die Annotation '@Composable'.
// Diese Annotation kennzeichnet eine Funktion als eine
   Composable-Funktion, die UI-Elemente in Jetpack Compose
   beschreibt.
// Composable-Funktionen können andere Composable-Funktionen
   aufrufen, um eine UI-Hierarchie aufzubauen.
import androidx.compose.runtime.Composable
```

```
// Importiert die 'getValue'-Delegatfunktion aus dem Compose
   Runtime.
// Sie ermöglicht es, den Wert eines 'State'-Objekts direkt zu
   lesen, als wäre es eine normale Variable (z.B. `val text by
   myState`).
// Änderungen am 'State' führen automatisch zu einer
   Neukomposition der betroffenen Composables.
import androidx.compose.runtime.getValue
// Importiert die Funktion 'mutableStateOf'.
// Diese Funktion erstellt ein 'MutableState'-Objekt, das einen
   veränderlichen Wert hält.
// Änderungen an diesem Wert lösen eine Neukomposition der
   Composables aus, die diesen Zustand beobachten.
import androidx.compose.runtime.mutableStateOf
// Importiert die Composable-Funktion 'remember'.
// 'remember' wird verwendet, um einen Wert über mehrere
   Neukompositionen hinweg im Speicher zu halten.
// Ohne 'remember' würde der Wert bei jeder Neukomposition auf
   seinen Initialwert zurückgesetzt.
import androidx.compose.runtime.remember
// Importiert die 'setValue'-Delegatfunktion aus dem Compose
   Runtime.
// Sie ermöglicht es, den Wert eines 'MutableState'-Objekts
   direkt zu setzen, als wäre es eine normale Variable (z.B. `
   myState = "neuer Wert"`).
import androidx.compose.runtime.setValue
// Importiert Klassen und Objekte für die Ausrichtung von UI-
   Elementen innerhalb ihrer Eltern-Container.
// Beinhaltet z.B. 'Alignment.Center', 'Alignment.Start', '
   Alignment.CenterHorizontally'.
import androidx.compose.ui.Alignment
// Importiert die Klasse 'Modifier'.
// Modifier werden verwendet, um Composable-Elemente zu
   dekorieren oder ihnen Verhalten hinzuzufügen.
// Beispiele sind das Setzen von Größe, Padding, Hintergrundfarbe
   , Klick-Listenern usw.
import androidx.compose.ui.Modifier
// Importiert 'LocalContext', ein CompositionLocal, das den
   aktuellen Android 'Context' bereitstellt.
// Der Context wird für viele Operationen benötigt, z.B. Zugriff
   auf Ressourcen, Systemdienste oder das Starten von Activities.
import androidx.compose.ui.platform.LocalContext
// Importiert die Funktion 'painterResource'.
// Diese Funktion wird verwendet, um eine Bildressource (z.B. aus
    dem 'drawable'-Ordner) als 'Painter'-Objekt zu laden,
// das dann von der 'Image'-Composable verwendet werden kann.
import androidx.compose.ui.res.painterResource
// Importiert die Erweiterungseigenschaft 'dp' (density-
   independent pixels).
// Sie wird verwendet, um Größenangaben in einer Weise zu
   definieren, die auf verschiedenen Bildschirmdichten konsistent
```

```
   aussieht.
import androidx.compose.ui.unit.dp
// Importiert das benutzerdefinierte Compose-Theme 'JjscanTheme'.
// Theme-Dateien (oft in ui/theme/Theme.kt) definieren
   typischerweise Farbschemata, Typografie und Formen für die App.
import com.example.jjscan.ui.theme.JjscanTheme
// Importiert die Optionsklasse für den ML Kit Document Scanner.
// Mit dieser Klasse können verschiedene Einstellungen für den
   Scanvorgang konfiguriert werden (z.B. Modus, Ergebnisformate).
import com.google.mlkit.vision.documentscanner.
   GmsDocumentScannerOptions
// Importiert die Hauptklasse für den Zugriff auf den ML Kit
   Document Scanner.
// Über diese Klasse wird ein Client-Objekt für den Scanner
   erstellt.
import com.google.mlkit.vision.documentscanner.
   GmsDocumentScanning
// Importiert die Klasse, die das Ergebnis eines ML Kit Document
   Scan-Vorgangs repräsentiert.
// Sie enthält Informationen wie URIs zu den gescannten PDF- oder
    JPEG-Dateien.
import com.google.mlkit.vision.documentscanner.
   GmsDocumentScanningResult
// Importiert die Klasse 'File' aus dem Java I/O-Paket.
// Sie wird für Dateioperationen wie das Erstellen, Lesen oder
   Schreiben von Dateien verwendet.
import java.io.File
// Importiert die Klasse 'FileOutputStream' aus dem Java I/O-
   Paket.
// Sie wird verwendet, um Daten in eine Datei zu schreiben.
import java.io.FileOutputStream
// Importiert die Klasse 'InputStream' aus dem Java I/O-Paket.
// Sie repräsentiert einen Eingabestrom von Bytes, z.B. zum Lesen
    von Daten aus einer Datei oder einem Netzwerk-Socket.
import java.io.InputStream
// Importiert die Klasse 'OutputStream' aus dem Java I/O-Paket.
// Sie repräsentiert einen Ausgabestrom von Bytes, z.B. zum
   Schreiben von Daten in eine Datei oder einen Netzwerk-Socket.
import java.io.OutputStream

// Definiert die Haupt-Activity der Anwendung.
// Sie erbt von 'ComponentActivity', um Jetpack Compose nutzen zu
    können.
class MainActivity : ComponentActivity() {
  // Die 'onCreate'-Methode wird aufgerufen, wenn die Activity
     zum ersten Mal erstellt wird.
  // 'savedInstanceState' enthält den zuvor gespeicherten Zustand
      der Activity, falls vorhanden (z.B. nach einer
     Konfigurationsänderung).
  override fun onCreate(savedInstanceState: Bundle?) {
```

```
    // Ruft die 'onCreate'-Methode der Superklasse ('
       ComponentActivity') auf. Dies ist immer erforderlich.
    super.onCreate(savedInstanceState)
    // Definiert den UI-Inhalt der Activity mithilfe von Jetpack
       Compose.
    setContent {
      // Wendet das benutzerdefinierte 'JjscanTheme' auf die
         gesamte UI dieser Activity an.
      // Dies stellt sicher, dass Farben, Typografie usw.
         konsistent sind.
      JjscanTheme {
        // Ruft die Composable-Funktion 'DocumentScannerScreen'
           auf, um die Haupt-UI der App zu erstellen und
           anzuzeigen.
        DocumentScannerScreen()
      }
    }
  }
}

// Definiert eine Composable-Funktion, die den Hauptbildschirm
   der Dokumentenscanner-App darstellt.
// Die Annotation '@Composable' ist erforderlich, damit diese
   Funktion von Compose als UI-Baustein erkannt wird.
@Composable
fun DocumentScannerScreen() {
  // Ruft den aktuellen Android 'Context' ab. 'LocalContext.
     current' ist ein CompositionLocal,
  // das den Kontext innerhalb der Composable-Hierarchie
     bereitstellt.
  val context = LocalContext.current
  // Deklariert eine Zustandsvariable 'message' vom Typ String
     und initialisiert sie mit "Noch kein Scan durchgeführt".
  // 'remember' sorgt dafür, dass der Zustand über
     Neukompositionen hinweg erhalten bleibt.
  // 'mutableStateOf' erstellt ein beobachtbares State-Objekt;
     Änderungen an 'message' lösen eine Neukomposition aus.
  // 'by' ist die Delegat-Syntax, die den direkten Zugriff auf
     den Wert des State-Objekts ermöglicht.
  var message by remember { mutableStateOf("Noch kein Scan
     durchgeführt") }

  // Erstellt und speichert einen Activity Result Launcher.
  // 'rememberLauncherForActivityResult' sorgt dafür, dass der
     Launcher den Composable-Lifecycle überlebt.
  // 'ActivityResultContracts.StartIntentSenderForResult()' ist
     ein Vertrag, der angibt, dass ein IntentSender gestartet
  // und ein Ergebnis erwartet wird. Dies wird oft für APIs
     verwendet, die einen IntentSender bereitstellen,
  // wie der ML Kit Document Scanner.
  val scannerLauncher = rememberLauncherForActivityResult(
```

```
    ActivityResultContracts.StartIntentSenderForResult()
) { result -> // Dieser Lambda-Ausdruck wird ausgeführt, wenn
    die gestartete Activity ein Ergebnis zurückliefert.
    // 'result' ist ein 'ActivityResult'-Objekt, das den '
       resultCode' und die 'data' (Intent) enthält.
    if (result.resultCode == Activity.RESULT_OK) { // Überprüft,
       ob der Vorgang erfolgreich war.
      // Extrahiert das 'GmsDocumentScanningResult' aus dem
         zurückgegebenen Intent ('result.data').
      val data = GmsDocumentScanningResult.
         fromActivityResultIntent(result.data)

      // Ruft die URI der gescannten PDF-Datei ab, falls
         vorhanden.
      val pdfUri = data?.pdf?.uri
      // Ruft eine Liste der URIs der gescannten Bildseiten ab,
         falls vorhanden.
      // 'map { it.imageUri }' transformiert jede Seite in ihre
         Bild-URI. '?: emptyList()' sorgt dafür,
      // dass eine leere Liste zurückgegeben wird, falls 'data?.
         pages' null ist.
      val imageUris = data?.pages?.map { it.imageUri } ?:
         emptyList()

      // Aktualisiert die 'message'-Zustandsvariable basierend
         auf dem Scan-Ergebnis.
      message = when { // 'when' ist Kotlin's Äquivalent zu einer
          switch-Anweisung.
        pdfUri != null -> { // Fall 1: Eine PDF-URI ist vorhanden
           .
          // Kopiert die PDF-Datei in den "paperless"-Ordner und
             gibt den Pfad der kopierten Datei zurück.
          // Der Dateiname wird dynamisch mit einem Zeitstempel
             generiert.
          val copiedPath = copyToPaperlessFolder(context, pdfUri,
              "scan_${System.currentTimeMillis()}.pdf")
          // Setzt die Nachricht, um den Erfolg und den
             Speicherort der PDF anzuzeigen.
          "PDF kopiert nach: $copiedPath"
        }
        imageUris.isNotEmpty() -> { // Fall 2: Es gibt eine oder
           mehrere Bild-URIs.
          // Kopiert jede Bilddatei in den "paperless"-Ordner.
          // 'mapIndexed' wird verwendet, um einen eindeutigen
             Index für jeden Dateinamen zu generieren.
          val copiedFiles = imageUris.mapIndexed { index, uri ->
            copyToPaperlessFolder(context, uri, "scan_${System.
               currentTimeMillis()}_${index + 1}.jpg")
          }
          // Setzt die Nachricht, um den Erfolg und die
             Speicherorte der Bilder anzuzeigen.
```

```
          "Bilder kopiert nach: $copiedFiles"
        }
        else -> { // Fall 3: Weder PDF noch Bilder wurden
           zurückgegeben.
          // Setzt eine entsprechende Nachricht.
          "Kein Ergebnis erhalten"
        }
      }
    } else { // Wird ausgeführt, wenn 'result.resultCode' nicht '
       Activity.RESULT_OK' ist (z.B. Scan abgebrochen).
      // Setzt eine Nachricht, die den Abbruch des Scans anzeigt.
      message = "Scan abgebrochen"
    }
}

// Erstellt Konfigurationsoptionen für den ML Kit Document
   Scanner.
// 'GmsDocumentScannerOptions.Builder()' startet den Builder
   für die Optionen.
val scannerOptions = GmsDocumentScannerOptions.Builder()
  // Setzt den Scannermodus auf 'SCANNER_MODE_FULL'.
  // Dieser Modus bietet die volle Benutzeroberfläche des
     Scanners (Bearbeiten, Zuschneiden, Filter usw.).
  .setScannerMode(GmsDocumentScannerOptions.SCANNER_MODE_FULL)
  // Definiert die gewünschten Ergebnisformate. Hier werden
     sowohl PDF als auch JPEG angefordert.
  .setResultFormats(
    GmsDocumentScannerOptions.RESULT_FORMAT_PDF,
    GmsDocumentScannerOptions.RESULT_FORMAT_JPEG
  )
  // Setzt ein Limit für die maximale Anzahl der Seiten, die
     gescannt werden können (hier 5).
  .setPageLimit(5)
  // Baut das 'GmsDocumentScannerOptions'-Objekt mit den
     konfigurierten Einstellungen.
  .build()

// Ruft einen Client für den ML Kit Document Scanner ab, unter
   Verwendung der zuvor definierten Optionen.
val scanner = GmsDocumentScanning.getClient(scannerOptions)

// Definiert eine 'Surface', die als Root-Container für den
   Bildschirm dient.
// 'Surface' ist ein Composable, das typischerweise für
   Hintergründe und die Anwendung von Material Design-
   Oberflächenstilen verwendet wird.
Surface(
  // Der 'Modifier' wird verwendet, um das Aussehen und
     Verhalten der Surface anzupassen.
  // 'Modifier.fillMaxSize()' bewirkt, dass die Surface den
     gesamten verfügbaren Platz einnimmt.
```

```
    modifier = Modifier.fillMaxSize(),
    // Setzt die Hintergrundfarbe der Surface auf die
       Hintergrundfarbe des aktuellen Material-Themes.
    color = MaterialTheme.colorScheme.background
  ) {
    // Definiert eine 'Column', die ihre Kind-Elemente vertikal
       untereinander anordnet.
    Column(
      // Der 'Modifier' für die Column.
      modifier = Modifier
        .fillMaxSize() // Füllt den gesamten verfügbaren Platz
           aus.
        .padding(32.dp), // Fügt ein Padding von 32dp auf allen
           Seiten hinzu.
      // Richtet die Kind-Elemente horizontal in der Mitte der
         Column aus.
      horizontalAlignment = Alignment.CenterHorizontally,
      // Definiert die vertikale Anordnung der Kind-Elemente
         innerhalb der Column.
      // 'Arrangement.SpaceBetween' platziert das erste Element
         am Anfang, das letzte am Ende
      // und verteilt den restlichen Platz gleichmäßig zwischen
         den Elementen.
      verticalArrangement = Arrangement.SpaceBetween
    ) {

      // Fügt einen leeren vertikalen Abstand von 64dp hinzu.
      // Dient als Abstandshalter, um Elemente nach unten zu
         verschieben.
      Spacer(modifier = Modifier.height(64.dp))

      // Zeigt den App-Titel "jjscan" als Text an.
      Text(
        text = "jjscan", // Der anzuzeigende Text.
        // Verwendet den 'displayLarge'-Typografiestil aus dem
           aktuellen Material-Theme.
        style = MaterialTheme.typography.displayLarge,
        // Setzt die Textfarbe auf die 'onBackground'-Farbe des
           aktuellen Material-Themes
        // (eine Farbe, die gut auf der Hintergrundfarbe lesbar
           ist).
        color = MaterialTheme.colorScheme.onBackground,
      )

      // Fügt einen weiteren leeren vertikalen Abstand von 32dp
         hinzu.
      Spacer(modifier = Modifier.height(32.dp))

      // Definiert einen 'IconButton', einen Button, der
         typischerweise ein Icon anzeigt und bei Klick eine Aktion
          ausführt.
```

```
    IconButton(
      // Die Lambda-Funktion, die ausgeführt wird, wenn der
         Button geklickt wird.
      onClick = {
        // Ruft die 'getStartScanIntent'-Methode des Scanners
           auf, um einen IntentSender für den Start des
           Scanvorgangs zu erhalten.
        // 'context as Activity' ist erforderlich, da diese
           Methode einen Activity-Context erwartet.
        scanner.getStartScanIntent(context as Activity)
          // Wird ausgeführt, wenn der IntentSender erfolgreich
              abgerufen wurde.
          .addOnSuccessListener { intentSender ->
            // Startet den Scanvorgang unter Verwendung des
               zuvor erstellten 'scannerLauncher'.
            // 'IntentSenderRequest.Builder(intentSender).build
               ()' erstellt die Anfrage für den Launcher.
            scannerLauncher.launch(IntentSenderRequest.Builder(
               intentSender).build())
          }
          // Wird ausgeführt, wenn beim Abrufen des
             IntentSenders ein Fehler auftritt.
          .addOnFailureListener { e ->
            // Aktualisiert die 'message'-Zustandsvariable mit
               der Fehlermeldung.
            message = "Fehler: ${e.message}"
          }
      },
      // Der 'Modifier' für den IconButton.
      // 'Modifier.size(250.dp)' setzt die Breite und Höhe des
         Buttons auf 250dp.
      modifier = Modifier.size(250.dp)
    ) {
      // Zeigt ein Bild innerhalb des IconButton an.
      Image(
        // Lädt das Bild aus den App-Ressourcen. 'R.mipmap.
           ic_launcher_foreground' verweist
        // auf eine Bilddatei im 'mipmap'-Ordner (
           typischerweise das App-Icon).
        painter = painterResource(R.mipmap.
           ic_launcher_foreground),
        // Eine textuelle Beschreibung des Bildes, wichtig für
           Barrierefreiheit (z.B. für Screenreader).
        contentDescription = "JJScan starten",
        // Der 'Modifier' für das Bild.
        // 'Modifier.fillMaxSize()' bewirkt, dass das Bild den
           gesamten Platz innerhalb des IconButton einnimmt.
        modifier= Modifier.fillMaxSize()
      )
    }
```

```
      // Zeigt den aktuellen Wert der 'message'-Zustandsvariable
         als Text an.
      Text(
        text = message, // Der anzuzeigende Text (z.B. "Noch kein
            Scan durchgeführt", Erfolgs- oder Fehlermeldung).
        // Verwendet den 'bodyLarge'-Typografiestil aus dem
           aktuellen Material-Theme.
        style = MaterialTheme.typography.bodyLarge,
        // Setzt die Textfarbe auf die 'onBackground'-Farbe des
           aktuellen Material-Themes.
        color = MaterialTheme.colorScheme.onBackground
      )

      // Fügt einen leeren vertikalen Abstand von 24dp hinzu.
      Spacer(modifier = Modifier.height(24.dp))

    }
  }
}

/**
 * Kopiert eine Datei von der gegebenen Quell-Uri ('sourceUri')
    in den Ordner "Documents/paperless"
 * im öffentlichen externen Speicher des Geräts.
 * Gibt den absoluten Pfad der neu erstellten Zieldatei zurück.
 *
 * @param context Der Android-Context, wird benötigt, um auf den
    ContentResolver und das Dateisystem zuzugreifen.
 * @param sourceUri Die Uri der Quelldatei, die kopiert werden
    soll.
 * @param fileName Der gewünschte Name für die Zieldatei.
 * @return Der absolute Pfad der kopierten Datei als String.
 */
fun copyToPaperlessFolder(
  context: android.content.Context, // Der Android-
     Anwendungskontext.
  sourceUri: Uri, // Die URI der Quelldatei.
  fileName: String // Der gewünschte Dateiname für die Kopie.
): String { // Der Rückgabetyp ist der absolute Pfad der
   kopierten Datei.
  // Ruft den ContentResolver ab. Der ContentResolver ermöglicht
     den Zugriff auf Inhaltsanbieter (Content Providers),
  // die Daten (wie Dateien) kapseln und bereitstellen.
  val resolver = context.contentResolver

  // Definiert das Zielverzeichnis: Documents/paperless im
     öffentlichen externen Speicher.
  // 'Environment.getExternalStoragePublicDirectory(Environment.
     DIRECTORY_DOCUMENTS)' gibt den Pfad zum öffentlichen
     Dokumentenverzeichnis zurück.
  val paperlessDir = File(
```

```
    Environment.getExternalStoragePublicDirectory(Environment.
        DIRECTORY_DOCUMENTS),
    "paperless" // Der Name des Unterordners.
  )
  // Überprüft, ob das Zielverzeichnis "paperless" existiert.
  if (!paperlessDir.exists()) {
    // Wenn das Verzeichnis nicht existiert, wird es erstellt (
        inklusive aller übergeordneten Verzeichnisse, falls nötig).
    paperlessDir.mkdirs()
  }
  // Erstellt ein 'File'-Objekt, das die Zieldatei im "paperless
      "-Verzeichnis repräsentiert.
  val targetFile = File(paperlessDir, fileName)

  // Öffnet einen InputStream zur Quelldatei über den
      ContentResolver.
  // 'resolver.openInputStream(sourceUri)' gibt einen InputStream
       zurück, oder null, wenn die URI nicht geöffnet werden kann.
  // '.use' ist eine Kotlin-Standardfunktion, die sicherstellt,
      dass der Stream nach der Verwendung automatisch geschlossen
      wird (auch im Fehlerfall).
  resolver.openInputStream(sourceUri)?.use { input -> // 'input'
      ist der geöffnete InputStream.
    // Erstellt einen FileOutputStream zur Zieldatei.
    // '.use' stellt auch hier sicher, dass der Stream
        automatisch geschlossen wird.
    FileOutputStream(targetFile).use { output -> // 'output' ist
        der geöffnete FileOutputStream.
      // Ruft die Hilfsfunktion 'copyStream' auf, um die Daten
          vom InputStream zum OutputStream zu kopieren.
      copyStream(input, output)
    }
  }

  // Gibt den absoluten Pfad der Zieldatei zurück.
  return targetFile.absolutePath
}

/**
 * Hilfsfunktion zum Kopieren von Daten von einem InputStream zu
     einem OutputStream.
 *
 * @param input Der InputStream, aus dem gelesen wird.
 * @param output Der OutputStream, in den geschrieben wird.
 */
fun copyStream(input: InputStream, output: OutputStream) {
  // Erstellt einen Puffer (Byte-Array) der Größe 4096 Bytes (4KB
      ).
  // Dieser Puffer wird verwendet, um Daten blockweise zu lesen
      und zu schreiben, was effizienter ist als byte-weises
      Kopieren.
```

```
    val buffer = ByteArray(4096)
    // Variable, um die Anzahl der gelesenen Bytes bei jedem
       Lesevorgang zu speichern.
    var bytesRead: Int
    // Eine Schleife, die so lange läuft, wie 'input.read(buffer)'
       nicht -1 zurückgibt.
    // 'input.read(buffer)' liest bis zu 'buffer.size' Bytes aus
       dem InputStream in den Puffer
    // und gibt die Anzahl der tatsächlich gelesenen Bytes zurück,
       oder -1, wenn das Ende des Streams erreicht ist.
    // '.also { bytesRead = it }' weist das Ergebnis von 'input.
       read(buffer)' der Variablen 'bytesRead' zu.
    while (input.read(buffer).also { bytesRead = it } != -1) {
      // Schreibt 'bytesRead' Bytes aus dem Puffer ('buffer') in
         den OutputStream ('output'),
      // beginnend beim Offset 0 im Puffer.
      output.write(buffer, 0, bytesRead)
    }
}
```

Literaturverzeichnis

[1] J. J. Buchholz. (2025) jjscan. Hochschule Bremen. [Online]. Available: https://m-server.fk5.hs-bremen.de/jjscan/jjscan.html

[2] Paperless-ngx-Community. (2025) Paperless-ngx. [Online]. Available: https://docs.paperless-ngx.com/

[3] Wikipedia. (2025) Paperless-ngx. [Online]. Available: https://de.wikipedia.org/wiki/Paperless-ngx

[4] The Syncthing Foundation. (2025) Syncthing. [Online]. Available: https://syncthing.net

[5] Wikipedia. (2025) Syncthing. [Online]. Available: https://de.wikipedia.org/wiki/Syncthing

[6] Google. (2025) Document Scanner (ML Kit). [Online]. Available: https://developers.google.com/ml-kit/vision/doc-scanner

[7] Wikipedia. (2025) Unixzeit. [Online]. Available: https://de.wikipedia.org/wiki/Unixzeit

[8] Google. (2025, September) Gemini in Android Studio. [Online]. Available: https://developer.android.com/studio/gemini/overview

[9] Github Copilot. (2024). [Online]. Available: https://github.com/features/copilot