

29. JULI 2021

# CODE-DOKUMENTATION ASKHSWORMS

*FRAGEN-APP FÜR DIE HOCHSCHULE WORMS*

SALEH CHAABAN  
ISSAM CHAREF

Abgabe bei Herrn Stephan Kurpjuweit

<b>Beschreibung, Zielgruppe und Motivation</b>	<b>1</b>
<b>Allgemeiner Überblick über die App-Struktur</b>	<b>2</b>
<b>MainActivity.kt</b>	<b>4</b>
activity_main.xml	8
slide_layout.xml	9
<b>OnboardingViewPagerAdapter.kt</b>	<b>10</b>
<b>LoginActivity.kt</b>	<b>11</b>
activity_login.xml	15
<b>RegisterActivity.kt</b>	<b>16</b>
activity_register.xml	21
<b>PasswordForgotActivity.kt</b>	<b>22</b>
activity_forgot_password.xml	25
<b>PasswordDoneActivity.kt</b>	<b>26</b>
activity_password_done.xml	27
<b>WelcomeActivity.kt</b>	<b>28</b>
content_main.xml	36
activity_welcome.xml	37
menu_header.xml	38
recycler_view_item.xml	39
<b>NoteActivity.kt</b>	<b>40</b>
activity_notizen.xml	47
fragment_notes.xml	48
add_note.xml	49
activity_content_note.xml	50
delete_note.xml	51
notes_item.xml	52
<b>ForumActivity.kt</b>	<b>53</b>
activity_forum.xml	55
fragment_forum.xml	56

<b>FragenActivity.kt</b>	<b>57</b>
activity_fragen.xml	60
add_frage.xml	61
<b>ProfileActivity.kt</b>	<b>62</b>
activity_profile.xml	68
<b>Code-Qualität – Bedeutsame Namen am Beispiel von MainActivity.kt</b>	<b>69</b>
Gute Variablennamen	69
Gute Methodennamen	71
Gute Klassennamen	73

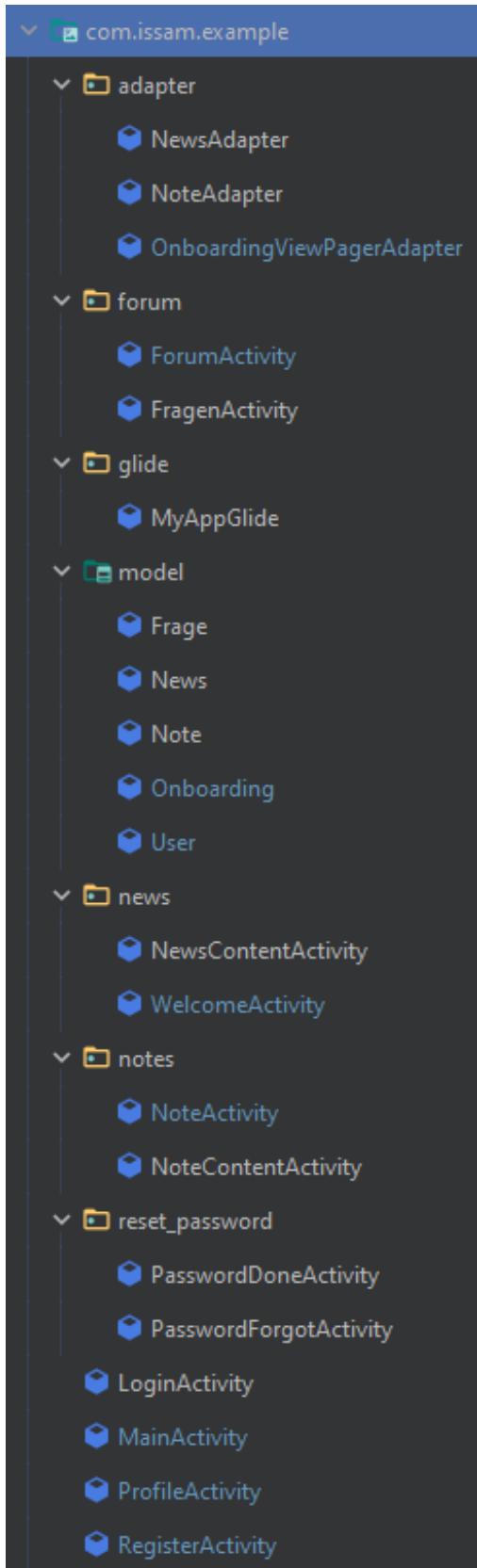
## BESCHREIBUNG, ZIELGRUPPE UND MOTIVATION

Bei der mobilen App *AskHSWorms* handelt es sich um eine Fragen-App für Studierende der Hochschule Worms. Die Studierenden der Hochschule sollen die Möglichkeit erhalten, Fragen zu allen Themengebieten zu stellen, welche im Speziellen die Hochschule betreffen. Darunter fallen Themen im Zusammenhang mit Modulen, Prüfungen, Erfahrungen und Organisation von Auslandssemestern, Informationen zu Kursangeboten der Hochschule, BAföG und noch viel mehr. Das Ziel ist es, eine gemeinsam nutzbare Plattform für alle Studierenden der Hochschule zu entwickeln, unabhängig von Studiengang oder Studiensemester. Da insbesondere in Zeiten der Pandemie der Kontakt zu anderen Kommilitonen beinahe vollständig ausbleibt, ist es das Ziel von *AskHSWorms*, wieder etwas Kontakt zwischen den verschiedenen Fakultäten und Studiensemester zu bringen. Vor allem Erstsemester erleben durch das reine Online-Format einen erschwerten Studienstart, denn sie kennen nur wenige oder keine Personen, an die sie ihre teils wichtigen Fragen richten können oder fühlen sich im Allgemeinen mit dem Übergang von der Schule in das Studium allein gelassen. Durch die vereinfachte Möglichkeit Ansprechpartner aus allen Bereichen zu finden und sofort kontaktieren zu können, soll ihnen somit ein einfacherer Studienstart und insgesamt eine einfachere Studienzeit gelingen.

Zu den möglichen Teilnehmern und damit Anwendern der App gehören neben den Studenten auch die Dozenten, Tutoren und sonstige Mitarbeiter der Hochschule. Damit soll gewährleistet werden, dass zu jedem Fragengebiet Experten einem weiterhelfen können. Die verschiedenen Rollen der Anwender werden in der App optisch gekennzeichnet, um sie eindeutig von den regulären Studierenden unterscheidbar zu machen.

Darüber hinaus erhalten die Anwender in der *AskHSWorms* App die Möglichkeit Notizen zu erstellen, Zugriff auf das LSF, den Webmailer und auf ihren individuellen Stundenplan.

## ALLGEMEINER ÜBERBLICK ÜBER DIE APP-STRUKTUR



Die App ist in unterschiedliche Verzeichnisse strukturiert, wie zum Beispiel einem Verzeichnis *notes* für die Notizenfunktionen oder weitere Verzeichnisse für Hilfs- und Datenklassen. Die wichtigen Aktivitäten für das Einloggen und Registrieren liegen hingegen im selben Verzeichnis wie die Hauptaktivität. Weiterhin befinden sich im Ordner *model* die unterstützenden Klassen, in denen zum Großteil nur die Datenklassen und ihre Konstruktoren für andere Aktivitäten definiert sind. Die Zuständigkeitsbereiche der Aktivitäten sind häufig schon am Namen auszumachen. Trotzdem folgt eine kurze Beschreibung jeder wichtigen Aktivität mit ihrer Hauptaufgabe. Im weiteren Verlauf werden nochmal alle Aktivitäten im Detail aufgegriffen, erläutert und Zusammenhänge dargestellt.

**ForumActivity.** Aktivität für die Forenauswahl für die verschiedenen Studiengänge und Themengebiete. Die Benutzer suchen sich hier zunächst das für sie gewünschte Forum aus. Die Forenübersicht ist über das Kontextmenü, als auch über die Pfeilnavigation erreichbar, nachdem sich ein Benutzer eingeloggt hat.

**FragenActivity.** Aktivität für das jeweilige Forum und den darin enthaltenen Fragenkatalog der Benutzer. Vorab findet über die Forenübersicht zunächst die Auswahl des Forums statt.

**NewsContentActivity.** Aktivität für das Anschauen eines einzelnen Artikels im News-Feed. Durch das Tippen auf eine der gezeigten Nachrichtenartikel, wird sie geöffnet und zeigt den Artikelinhalt und weitere für den Artikel relevante Informationen.

**WelcomeActivity.** Erste Aktivität, welche direkt nach dem erfolgreichen Einloggen eines Benutzers sichtbar wird. Sie implementiert die Übersicht für den News-Feed mit ihren klickbaren Nachrichtenartikeln.

**NoteActivity.** Aktivität für die Notizen-Funktion, auf welche die Benutzer nach dem Login in ihr Profil über das Kontextmenü oder die Pfeilnavigation Zugriff erhalten.

**NoteContentActivity.** Aktivität für das Anschauen einer erzeugten Notiz in der Einzelansicht. Vorab wählt ein Benutzer die entsprechende Notiz in der Notizenlisten aus der Aktivität für die Notizen aus.

**PasswordDoneActivity.** Aktivität, welche gezeigt wird, sobald die E-Mail für das Zurücksetzen des Passworts erfolgreich von Firebase versendet wurde. Sie ist im Wesentlichen eine Erfolgsmeldung in Form einer eigenen Aktivität.

**PasswordForgotActivity.** Aktivität zum Zurücksetzen des Passworts eines Benutzers. Sie ist erreichbar über die Login Aktivität.

**LoginActivity.** Aktivität zum Einloggen in das Benutzerprofil. Erste sichtbare Aktivität, falls das Onboarding bereits abgeschlossen wurde.

**MainActivity.** Erste beim App-Start sichtbare Aktivität und somit die Aktivität, welche das Onboarding implementiert. Sie ist nur sichtbar, sofern das Onboarding noch nicht abgeschlossen wurde.

**ProfileActivity.** Aktivität zum Einsehen und Editieren der Profileinstellungen des Benutzers. Hier können die Benutzer Veränderungen am Profil vornehmen und zum Beispiel ihr Profilbild ändern.

**RegisterActivity.** Aktivität für das Anlegen eines neuen Benutzers auf AskHSWorms.

## MAINACTIVITY.KT

In Kotlin und Java ist für jede Android App die *MainActivity* die erste gestartete Aktivität. Um einen vereinfachten Einstieg in die App *AskHSWorms* zu geben, wird mit den Benutzern zunächst ein Onboarding durchgeführt. Das Onboarding gibt zum Beispiel darüber Auskunft, dass es sich bei der App um ein Projekt im Modul „Entwicklung mobiler Anwendungen“ an der Hochschule Worms handelt, sowie weitere wichtige Hinweise zur vereinfachten Bedienung.

### Der Quellcode:

```
var onboardingViewPagerAdapterRef: OnboardingViewPagerAdapter? = null
var tabLayoutRef: TabLayout? = null
var onboardingViewPager: ViewPager? = null
var nextView: TextView? = null
var currentTab = 0
var sharedpreferencesRef: SharedPreferences? = null
```

Zu Beginn stehen die für die Aktivität wichtigen Variablen. Sie werden bis auf die Variable *currentTab* zunächst mit *null* initialisiert und hinterher mit ihren zugehörigen Views gekoppelt. Um beim Onboarding per Wischgeste zwischen den eingeblendeten Hinweisen zu wechseln, wird zum Beispiel eine Referenz auf *TabLayout* benötigt. Die Hilfsklasse *OnboardingViewPagerAdapter* wird dazu genutzt, um die Hinweise einzublenden beziehungsweise wieder zu entfernen. Die *Shared Preferences* hingegen, sind besonders wichtig, um später das erneute Onboarding auszuschalten. Die Variable *currentTab* dient zur Bestimmung dazu, welcher Hinweis gerade eingeblendet werden soll und wann (= ab welchem Tab) es bei einem erneuten Klick stattdessen zur nächsten Aktivität übergeht.

```
// check if on boarding has already been completed & skip it
if (readFromSharedPreferences()) {
    // go to LoginActivity
    startActivity(Intent(applicationContext , LoginActivity::class.java))
    // terminating current Activity
    finish()
}
```

```
// read flag from sharedPreferences
private fun readFromSharedPreferences(): Boolean {
    sharedpreferencesRef = applicationContext.getSharedPreferences( name: "pref" , Context.MODE_PRIVATE)
    return sharedpreferencesRef!!.getBoolean( key: "isOnboardingCompleted" , defaultValue: false)
}
```

Darauffolgend wird abgefragt, ob das Onboarding bereits abgeschlossen wurde. Sofern dies der Fall ist, wird der Benutzer direkt auf die Login Aktivität weitergeleitet und die aktuelle Aktivität wird beendet. Um zu überprüfen, ob das Onboarding bereits abgeschlossen wurde, werden über die Methode `readFromSharedPreferences()` die *Shared Preferences* gelesen und spezifisch nach dem Wahrheitswert `isOnboardingCompleted` geprüft. Sofern das Flag auf `true` gesetzt wurde, wird das Onboarding beim nächsten App-Start übersprungen.

```
// add data to our model class
val onboardinglest: MutableList<Onboarding> = ArrayList()
onboardinglest.add(
    Onboarding(
        tabTitle = getString(R.string.slide_heading_text1) ,
        tabDescription = getString(R.string.slide_desc_text1) ,
        R.drawable.icon1
    )
)

onboardinglest.add(
    Onboarding(
        tabTitle = getString(R.string.slide_heading_text2) ,
        tabDescription = getString(R.string.slide_desc_text2) ,
        R.drawable.icon2
    )
)

onboardinglest.add(
    Onboarding(
        tabTitle = getString(R.string.slide_heading_text3) ,
        tabDescription = getString(R.string.slide_desc_text3) ,
        R.drawable.icon3
    )
)
```

```
data class Onboarding(var tabTitle: String , var tabDescription: String , var tabImgURL: Int)
```

Die jeweiligen Texte und Titel zu den Hinweisen, welche beim Onboarding gezeigt werden, müssen zunächst über die Methode `add()` der Datenklasse `Onboarding` hinzugefügt und in die Array Liste `Onboarding` geschrieben werden. Anschließend werden über die Methode `setOnboardingViewPagerAdapter()` die in `onboardingList` geschriebenen Hinweise den Benutzern pro Tab präsentiert.

```
val lastTab = onboardingList.size - 1
currentTab = onboardingViewPager!!.currentItem

nextView?.setOnClickListener { it: View!

    when (currentTab) {
        in 0 until lastTab -> onboardingViewPager!!.currentItem = (++currentTab)
        // defines what happens after last tab
        else -> {
            saveInSharedPreferences()
            // go to LoginActivity after clicking 'Los geht's!'
            startActivity(Intent(applicationContext, LoginActivity::class.java))
        }
    }
}
```

```
// set flag in sharedpreferences
private fun saveInSharedPreferences() {

    sharedPreferencesRef = applicationContext.getSharedPreferences(name: "pref", Context.MODE_PRIVATE)
    val sharedPreferencesEditor: SharedPreferences.Editor = sharedPreferencesRef!!.edit()
    sharedPreferencesEditor.putBoolean("isOnboardingCompleted", true)
    sharedPreferencesEditor.apply()
}
```

Zur Navigation auf den nächsten Hinweis, wird alternativ zur Wischgeste auch eine `TextView` mit der Aufschrift „Nächste“ angezeigt. Die `TextView` ist klickbar und navigiert die Benutzer zum darauffolgenden Hinweis. Ist der letzte Hinweis erreicht, wird über die Methode `saveInSharedPreferences()` das Flag `isOnboardingCompleted` in den `Shared Preferences` auf `true` gesetzt, um ein erneutes Durchlaufen des Onboardings zu verhindern. Darauffolgend wird die Aktivität zum Einloggen aufgerufen.

```

// check if tab states has changed
tabLayoutRef!!.addOnTabSelectedListener(object : TabLayout.OnTabSelectedListener {
    // active tab
    override fun onTabSelected(tab: TabLayout.Tab?) {
        // get current tab position
        currentTab = tab!!.position
        // change text at bottom right on the last page
        when (tab.position) {
            lastTab -> nextView!!.text = getString(R.string.los_gehts)
            else -> nextView!!.text = getString(R.string.naechste)
        }
    }

    override fun onTabUnselected(tab: TabLayout.Tab?) {}

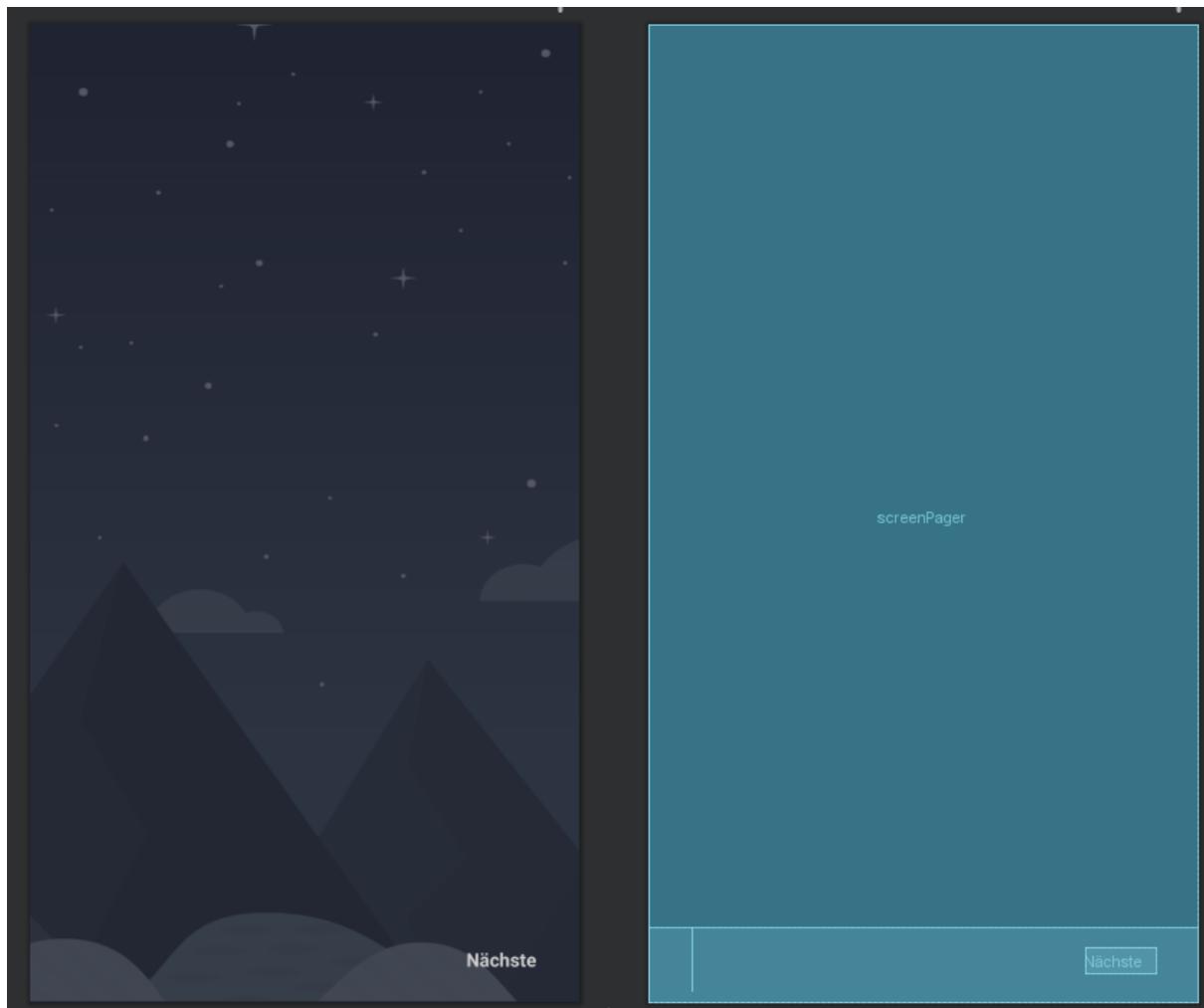
    override fun onTabReselected(tab: TabLayout.Tab?) {}
})
}

```

Zusätzlich wird für den letzten Hinweis, die Beschriftung der *TextView* von „Nächste“ auf „Los geht's!“ abgeändert, um die Benutzer darüber in Kenntnis zu setzen, dass das Onboarding abgeschlossen ist.

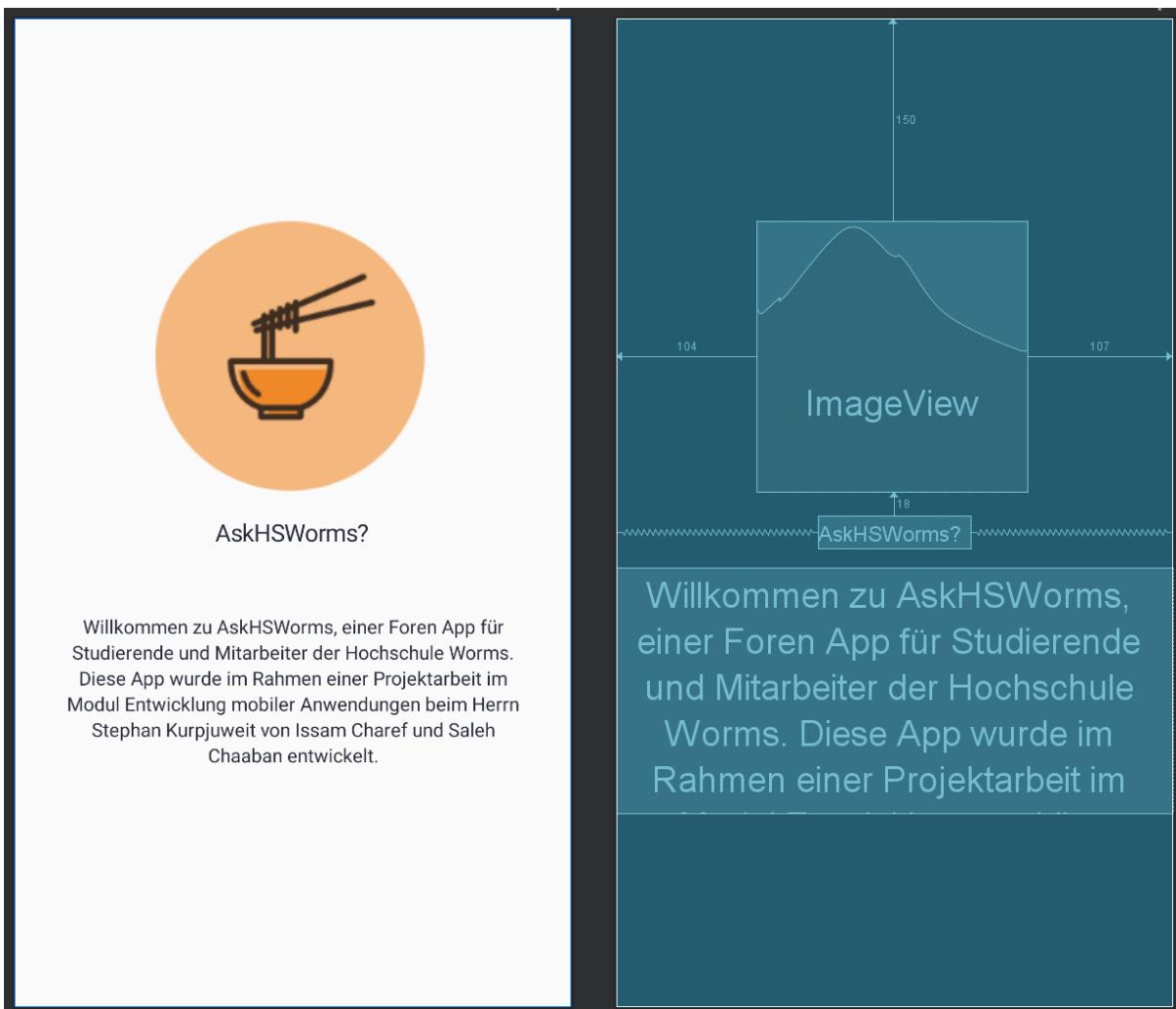
Das gesamte Onboarding wird visuell durch die beiden Layout-Dateien *activity\_main.xml* und *slide\_layout.xml* abgebildet.

## ACTIVITY\_MAIN.XML



Die Datei *activity\_main.xml* enthält den statischen Hintergrund, sowie eine *TextView* zur Navigation.

## SLIDE\_LAYOUT.XML



Ergänzend dazu gibt *slide\_layout.xml* den Aufbau der Hinweise vor. Diese werden daraufhin als zusätzliche Layout Ebene über die *activity\_main.xml* gelegt. Beide Dateien gemeinsam bilden in ihrer Gesamtheit das Layout für das Onboarding.

## ONBOARDINGVIEWPAGERADAPTER.KT

Wie bereits beschrieben, wird für das Einblenden und Entfernen der Onboarding-Hinweise die Datei *OnboardingViewPagerAdapter.kt* genutzt.

```
override fun instantiateItem(container: ViewGroup, position: Int): Any {  
  
    val view = LayoutInflater.from(context).inflate(R.layout.slide_layout, root = null);  
    val imageView: ImageView = view.findViewById(R.id.imageView)  
    val title: TextView = view.findViewById(R.id.slide_heading)  
    val desc: TextView = view.findViewById(R.id.slide_desc)  
  
    imageView.setImageResource(onBoardingList[position].tabImgURL)  
    title.text = onBoardingList[position].tabTitle  
    desc.text = onBoardingList[position].tabDescription  
  
    container.addView(view)  
  
    return view  
}
```

Um einen Hinweis zu erzeugen und in das Layout einzufügen, werden pro Hinweis die Views für das Bild, den Titel und die Beschreibung deklariert und anschließend initialisiert. Den Zugriff auf die Inhalte jeder Komponente erhält die Klasse *MainActivity* über die ihre Liste mit den Daten für das Onboarding. Diese werden dort an den Konstruktor der *OnboardingViewPagerAdapter* Klasse übergeben. Die Textinhalte für jeden Hinweis werden über den Index *position* ermittelt und können entsprechend abgerufen werden. Schlussendlich wird der jeweilige Hinweis noch zum *ViewGroup-Container* hinzugefügt und von der Methode *instantiateItem()* als View zurückgegeben.

```
override fun destroyItem(container: ViewGroup, position: Int, `object`: Any) =  
    container.removeView(`object` as View)
```

Jeder eingeblendete Hinweis ist als Teil einer *ViewGroup* zu sehen und kann wiederum aus dem Elternelement dem *ViewGroup-Container* entfernt werden, womit der Hinweis wieder vom Bildschirm der Benutzer verschwindet.

## LOGINACTIVITY.KT

Um sich in ein Benutzerprofil einzuloggen, wird die Login Aktivität verwendet. Sie stellt die erste richtige Aktivität dar, die ein Benutzer zu sehen bekommt, sofern er das Onboarding abgeschlossen hat. Hierüber sind ferner auch die beiden Aktivitäten zum Registrieren und Zurücksetzen des Passworts erreichbar.

Grundsätzlich wurde für den Authentifizierungs- und Registrierungsprozess die Datenbank aus Google Firebase ausgewählt. Gründe für die Wahl sind vor allem die Benutzerfreundlichkeit, gute API, die Möglichkeit Daten in Echtzeit zu speichern und zu synchronisieren und die gute Kompatibilität zu Android.

### Der Quellcode:

```
override fun onCreate(savedInstanceState: Bundle?) {  
  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_login)  
  
    val btnlogin: Button = findViewById(R.id.btn_login)  
    val btnsignup: ImageView = findViewById(R.id.btnRegistration)  
    val password_reset: TextView = findViewById(R.id.password_reset)  
  
    progressBar = findViewById(R.id.progressBar)  
    auth = FirebaseAuth.getInstance()  
  
    auth = Firebase.auth
```

Zunächst werden die Views, welche zur Weiterleitung auf die genannten Aktivitäten dienen initialisiert. Der Zugriff auf die Datenbank wird über die Variable `auth` realisiert, welche eine Instanz auf die Google Firebase Datenbank hält. Für die Wartezeit beim Abgleich der eingegebenen Daten mit denen aus der Datenbank, wird den Benutzern ein kreisförmiger Ladebalken gezeigt, welcher in diesem Codeabschnitt als `progressBar` initialisiert wird.

```

    btnsignUp.setOnClickListener { it: View!
        startActivity(Intent(applicationContext , RegisterActivity::class.java))
        finish()
    }

    password_reset.setOnClickListener { it: View!
        startActivity(Intent(applicationContext , PasswordForgotActivity::class.java))
        finish()
    }

    btnlogin.setOnClickListener { it: View!
        userLogin()
    }
}

```

In diesem Codeabschnitt wird die Erreichbarkeit zu den anderen genannten Aktivitäten umgesetzt. Die einzelnen Views verfügen jeweils über einen Click Listener, welche mit einer Folgereaktion gekoppelt sind. Für die beiden Views zum Registrieren und Passwort Zurücksetzen, wird die neue zugehörige Aktivität gestartet und die derzeitige beendet. Für die Login Aktivität hingegen, wird die für die Authentifizierung zuständige Methode *userLogin()* ausgelöst, die im Folgenden schrittweise erläutert wird.

```

private fun userLogin() {

    var editTextEmail: EditText = findViewById(R.id.editTextEmail)
    var editTextPassword: EditText = findViewById(R.id.editTextPassword)
    var email = editTextEmail.text.toString()
    var password = editTextPassword.text.toString()
}

```

Zuerst werden die zu prüfenden editierbaren Textfelder mit ihren zugehörigen Views initialisiert. Die eingegebene E-Mail und das Passwort des Benutzers werden über die Variablen *mail* und *password* referenziert und in Form einer String Zeichenkette zwischengespeichert.

```
if (email.isEmpty()) {
    editTextEmail.error = "Das Feld E-Mail ist leer!"
    editTextEmail.requestFocus()

    return
}

if (!Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
    editTextEmail.error = "Ungültiges E-Mail Format!"
    editTextEmail.requestFocus()

    return
}

if (password.isEmpty()) {
    editTextPassword.error = "Das Feld Passwort ist leer!"
    editTextPassword.requestFocus()

    return
}
```

Bevor es jedoch zur Abfrage in der Datenbank kommt, folgt zunächst eine Validierung der Textfelder. Leere Felder und eine ungültige E-Mail sind nicht gestattet und führen beim Absenden der Daten zu einer Fehlermeldung.

```

progressBar.setVisibility(View.VISIBLE)

auth.signInWithEmailAndPassword(email, password)
    .addOnCompleteListener(this) { task ->
        if (task.isSuccessful) {
            val user = auth.currentUser
            if (user.isEmailVerified) {
                val i = Intent(applicationContext, WelcomeActivity::class.java)
                startActivity(i)
            } else {
                user.sendEmailVerification()
                Toast.makeText(
                    baseContext,
                    "Wir haben dir eine E-Mail zur Verifikation gesendet",
                    Toast.LENGTH_LONG
                ).show()
            }
        }

        // Sign in success, update UI with the signed-in user's information
        updateUI(user)
        progressBar.setVisibility(View.GONE)
    } else {
        // If sign in fails, display a message to the user.
        Toast.makeText(
            baseContext,
            "Authentifizierung fehlgeschlagen! Bitte versuche es erneut",
            Toast.LENGTH_SHORT
        )
        .show()
        updateUI(currentUser = null)
        progressBar.setVisibility(View.GONE)
    }
}

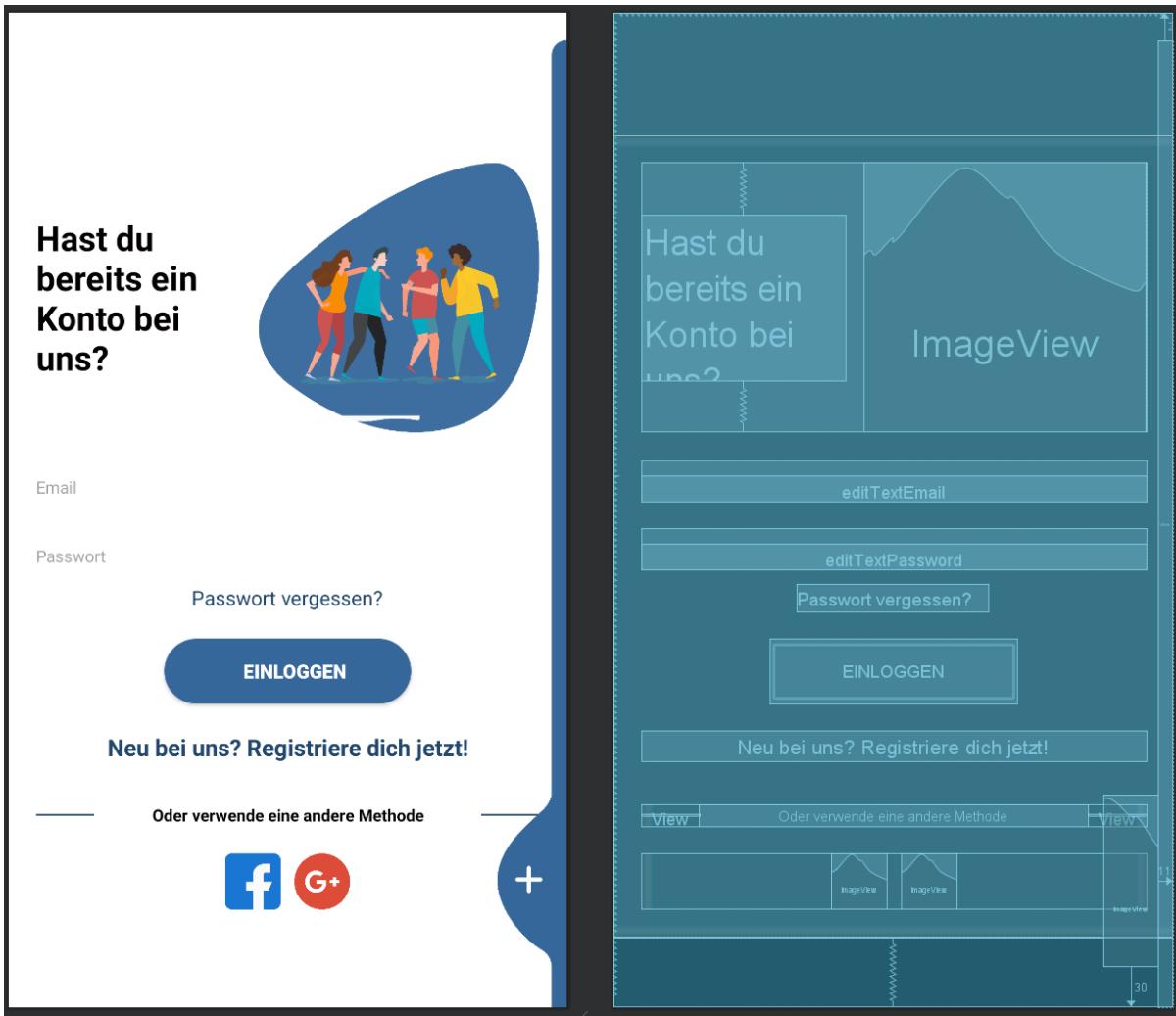
```

An dieser Stelle wird die eigentliche Authentifizierung durchgeführt. Die Inhalte aus den Textfeldern für die E-Mail und das Passwort werden über die Firebase Instanz übermittelt und nach ihrer Korrektheit geprüft. Bei Erfolg wird die Instanz des Benutzers in die Variable *user* gespeist. Daraufhin wird noch geprüft, ob die E-Mail des Benutzers bereits verifiziert wurde. Ist dies nicht der Fall, wird beim erstmaligen Login des Benutzers über Firebase eine E-Mail zur Freischaltung an ihn versendet und eine entsprechende Hinweismeldung als Toast eingeblendet. Wurde die E-Mail im Anschluss bestätigt, gelangt der Benutzer beim erneuten Einloggen in die *WelcomeActivity*. Die Benutzeroberfläche wird im Anschluss noch mit den Daten des Benutzers aktualisiert. Für die Dauer der Aktualisierung wird ein kreisförmiger Ladebalken gezeigt, welcher eine Bedienung der App-Oberfläche verhindert.

Scheitert jedoch die Authentifizierung, wird den Benutzern eine Fehlermeldung präsentiert und die Benutzeroberfläche zurückgesetzt.

Die Login Aktivität wird visuell durch die *activity\_login.xml* repräsentiert.

## ACTIVITY\_LOGIN.XML



Neben den Textfeldern für das Passwort und die E-Mail, sind die bereits genannten Views zum Registrieren und Einloggen erkennbar. Ein Klick auf das „+“ leitet die Benutzer alternativ zur *TextView* für die Registrierung ebenfalls auf die Registrierungs-Aktivität weiter. Die gezeigten Icons zum Login über Facebook und Google sind zunächst nur Platzhalter und veranschaulichen, welche weiteren Umsetzungsmöglichkeiten es bei einer echten Veröffentlichung der App gäbe.

## REGISTERACTIVITY.KT

Bei der *RegisterActivity* handelt es sich wie vom Namen ableitbar um die Aktivität für die Durchführung des Registrierungsprozesses.

### Der Quellcode:

```
lateinit var progressBar: ProgressBar
private lateinit var auth: FirebaseAuth
private val firestoreInstance: FirebaseFirestore by lazy {
    FirebaseFirestore.getInstance()
}
```

Zu Beginn werden die Variablen für die Firebase Datenbank und Firebase Firestore angelegt. Die Initialisierung für die Instanz von Firebase Firestore passiert wegen des *lazy init*s erst beim ersten Zugriff auf die Variable. Für die Dauer des Datenbankzugriffs soll wie bereits in der *LoginActivity* ein Ladebalken angezeigt werden, welcher erstmals nur deklariert wird.

```
override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_register)
    auth = Firebase.auth

    var btnRegister: Button = findViewById(R.id.btnRegister)
    progressBar = findViewById(R.id.progressBar)

    btnRegister.setOnClickListener { it: View!
        signUpUser()
    }

    rjaa.setOnClickListener { it: View!
        startActivity(Intent(applicationContext , LoginActivity::class.java))
        finish()
    }
}
```

Beim Starten der Aktivität wird dann die Variable *FirebaseAuth* mit der Instanz zur Authentifizierung initialisiert, der Registrierungsbutton initialisiert und der Ladebalken mit seiner View verbunden. Zusätzlich wird der Registrierungsbutton mit einem Click

Listener versehen, welcher die Methode `signUpUser()` auslöst, sobald die Benutzer ihre Daten abschicken.

```
// Verification Methode !
private fun signUpUser() {

    val editTextName: EditText = findViewById(R.id.editTextName)
    val editTextEmail: EditText = findViewById(R.id.editTextEmail)
    val editTextPassword: EditText = findViewById(R.id.editPassword)
    val editTextPhone: EditText = findViewById(R.id.editPhone)
```

Damit die eingetragenen Daten der Benutzer zwischengespeichert werden können, müssen zunächst vier Variablen für die editierbaren Textfelder mit ihren Views initialisiert werden.

```
if (editTextName.text.toString().isEmpty()) {
    editTextName.error = "Das Feld Name ist leer!"
    editTextName.requestFocus()

    return
}

if (!Patterns.EMAIL_ADDRESS.matcher(editTextEmail.text.toString()).matches()) {
    editTextEmail.error = "Ungültiges E-Mail Format!"
    editTextEmail.requestFocus()

    return
}

if (editPhone.text.toString().isEmpty()) {
    editPhone.error = "Das Feld Telefon ist leer!"
    editPhone.requestFocus()

    return
}

if (editPassword.text.toString().isEmpty()) {
    editPassword.error = "Das Feld Passwort ist leer!"
    editPassword.requestFocus()

    return
}

if (editPassword.text.toString().length < 6) {
    editPassword.error = "Dein Passwort braucht mindestens 6 Zeichen!"
    editPassword.requestFocus()

    return
}
```

Im gezeigten Codeabschnitt findet die Validierung jedes Textfeldes statt. Besonders interessant ist dabei die Validierung der E-Mail, da die Adresse nach ihrer strukturellen Korrektheit geprüft wird. Für die anderen Felder hingegen, wird geprüft, ob sie beim Abschicken noch leer sind beziehungsweise für das Passwort, ob mindestens sechs Zeichen verwendet wurden. Kommt es dazu, dass ein Feld beim Abschicken leer geblieben ist, wird eine Fehlermeldung angezeigt und der jeweilige Titel des Feldes wird rot gefärbt.

```
auth.createUserWithEmailAndPassword(
    editTextEmail.text.toString() ,
    editTextPassword.text.toString()
)
    .addOnCompleteListener(this) { task ->
```

Nach anschließender Validierung der Felder, wird mittels der Firebase Instanz über die native von Firebase zur Verfügung gestellte Methode *createUserWithEmailAndPassword()* ein Versuch gestartet einen neuen Benutzer mit den angegebenen Daten anzulegen. Die Methode wird zusätzlich mit einem *onCompleteListener* versehen, welcher dann greift, wenn die Ausführung der Aufgabe beendet ist.

```
if (task.isSuccessful) {
    // Sign in success, update UI with the signed-in user's information
    val user = User(
        editTextName.text.toString() ,
        editTextEmail.text.toString() ,
        editTextPhone.text.toString() ,
        editTextPassword.text.toString() ,
        userPower: "" ,
        profilBild: ""
    )
}
```

```
data class User(
    val fullname: String , val email: String , val tel: String , val password: String ,
    val userPower: String , val profilBild: String
)
{
    constructor() : this( fullname: "" ,   email: "" ,   tel: "" ,   password: "" ,   userPower: "" ,   profilBild: "" )
```

MODEL/USER.KT

Wird der erste Schritt erfolgreich ausgeführt, wird ein Benutzer-Objekt mit allen aus den Feldern enthaltenen Daten erstellt. Zur Erzeugung des Objekts wird die Datenklasse *User* verwendet, welche alle für einen Benutzer zugehörigen Daten definiert. Dabei werden die Rolle des Benutzers und das Profilbild zunächst nur als leerer String in das Benutzer-Objekt geschrieben und können ausschließlich hinterher in den Profileinstellungen verändert werden.

```
// Für Database
val currentUserDocRef: DocumentReference = firestoreInstance.collection( collectionPath: "users")
    .document(auth.currentUser.uid)
currentUserDocRef.set(user)

FirebaseDatabase.getInstance().getReference( path: "Users")
    .child(FirebaseAuth.getInstance().getCurrentUser().uid)
    .setValue(user).addOnCompleteListener { it: Task<Void!>
        if (task.isSuccessful) {
            Toast.makeText(
                baseContext ,
                text: "Deine Registrierung war erfolgreich" ,
                Toast.LENGTH_LONG
            ).show()
            progressBar.setVisibility(View.GONE)
        } else {
            Toast.makeText(
                baseContext ,
                text: "Die Registrierung ist fehlgeschlagen! Versuche es erneut" ,
                Toast.LENGTH_LONG
            ).show()
            progressBar.setVisibility(View.GONE)
        }
    }

    startActivity(Intent(applicationContext , LoginActivity::class.java))
    finish()
} else {
```

Vorab sei zu sagen, dass Firebase eine NoSQL Datenbank ist, welche ihre Daten in Dateien mit dem JSON-Format organisiert. Genauer gesagt, werden die Daten in unterschiedlichen Sammlungen von JSON-Dateien gespeichert. Über eine Referenz auf ein Dokument lässt sich dann der Pfad zu dem jeweilig relevanten Eintrag definieren, um Zugriff auf den gewünschten Inhalt zu erhalten.

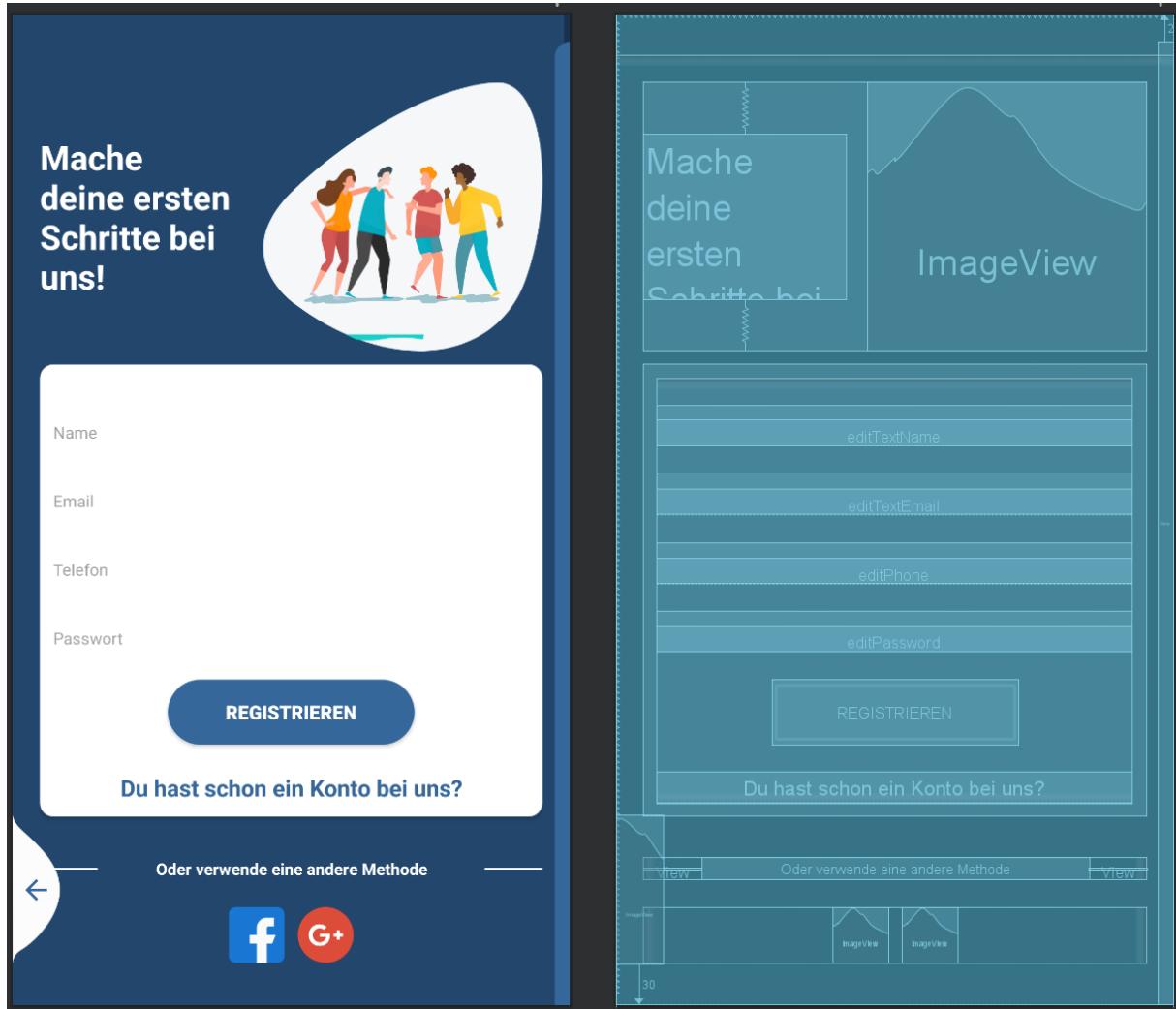
Um folglich das Benutzer-Objekt als Dokument abspeichern zu können, wird zunächst ein neues Referenzobjekt mit dem passenden Pfad und einer einzigartigen Benutzer-UID in Firestore erzeugt und die Inhalte des Benutzerobjekts mit der Methode *set()* gesetzt. Dasselbe in ähnlicher Form geschieht für die Firebase Datenbank mit dem

Zusatz eines *onCompleteListener*, welcher dazu genutzt wird, um zwischen Erfolg und Misserfolg der Operation zu differenzieren. Dementsprechend erhalten die Benutzer bei einer erfolgreichen Übertragung in die Datenbank eine Erfolgsbeziehungsweise bei nicht erfolgreicher Übertragung eine Fehlermeldung als Toast. Eine anschließende Beendigung der aktuellen Aktivität und Weiterleitung zurück in die Aktivität für den Login geschieht in beiden Fällen.

```
    } else {
        // If sign in fails, display a message to the user.
        Toast.makeText(
            baseContext ,
            text: "Die Registrierung ist fehlgeschlagen! Versuche es erneut" ,
            Toast.LENGTH_LONG
        ).show()
        progressBar.setVisibility(View.GONE)
    }
}
```

Sollte bereits zu Beginn bei der Methode *createUserWithEmailAndPassword()* ein Fehler auftreten, erscheint ebenfalls eine Fehlermeldung als Toast. Hier ist der Unterschied, dass die Benutzer im Anschluss in der derzeitigen Aktivität verweilen und somit direkt eine erneute Registrierung durchführen können.

## ACTIVITY\_REGISTER.XML



Zu erkennen ist das Layout für die Aktivität zur Registrierung. Für die Benutzer ist auch hier wieder zu erkennen, dass eine Anmeldung über Google und Facebook möglich wäre. Eine Implementierung dafür erfolgte jedoch in diesem Fall ebenfalls nicht.

## PASSWORDFORGOTACTIVITY.KT

Die Aktivität *PasswordForgotActivity* dient zum Zurücksetzen eines Benutzerpassworts. Wie es üblicherweise der Fall ist, lässt sich die Aktivität über den Login-Bereich über eine *TextView* mit dem Inhalt „Passwort vergessen?“ erreichen.

### Der Quellcode:

```
class PasswordForgotActivity : AppCompatActivity() {

    private lateinit var auth: FirebaseAuth
    lateinit var resetPasswordBtn: Button
    lateinit var btnZurueck: ImageView
    lateinit var emailEditText: EditText
    lateinit var progressBar: ProgressBar
```

Aller Anfang werden wie auch zuvor Variablen für den Zugang zu Firebase, alle relevanten Buttons, das Textfeld zum Eintragen der E-Mail des Benutzers und ein Ladebalken als *lateinits* deklariert.

```
override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_forgot_password)

    resetPasswordBtn = findViewById(R.id.resetPassword)
    btnZurueck = findViewById(R.id.zurick)
    emailEditText = findViewById(R.id.email)
    progressBar = findViewById(R.id.progressBar)

    auth = FirebaseAuth.getInstance()

    btnZurueck.setOnClickListener { it: View!
        startActivity(Intent(applicationContext , LoginActivity::class.java))
        finish()
    }

    resetPasswordBtn.setOnClickListener { it: View!
        newPassword()
    }
}
```

Die Variablen für Buttons und Textfeld und Ladebalken werden nun mit ihren Views gekoppelt. Für den Zugang zu Firebase wird zunächst wieder eine Instanz zur Authentifizierung initialisiert. Falls die Benutzer fälschlicherweise in die Passwort

Vergessen Aktivität gelangen, besteht die Möglichkeit über eine *ImageView* mit einem Pfeil nach links am oberen linken Rand zum Login Bereich zurückzukehren. Dafür wird ein Click Listener benötigt, der die Weiterleitung vornimmt und die aktuelle Aktivität beendet. Ein weiterer Click Listener liegt auf dem Button zum Zurücksetzen des Passworts. Er löst die Methode *resetPassword()* aus, die im weiteren Verlauf erläutert wird.

```
private fun resetPassword() {  
  
    val email = emailEditText.text.toString().trim()  
  
    if (email.isEmpty()) {  
        emailEditText.setError("Das Feld E-Mail ist leer!")  
        emailEditText.requestFocus()  
        return  
    }  
  
    if (!Patterns.EMAIL_ADDRESS.matcher(email).matches()) {  
        emailEditText.error = "Ungültiges E-Mail Format!"  
        emailEditText.requestFocus()  
        return  
    }  
  
    progressBar.setVisibility(View.VISIBLE)
```

In diesem Codeabschnitt wird einerseits überprüft, ob das Feld für die E-Mail nach Betätigen des Zurücksetzen Buttons leer geblieben ist. Andererseits wird geprüft, ob bei der Eingabe eine strukturell gültige E-Mail eingetragen wurde. Ist eines von beidem nicht der Fall, wird eine Fehlermeldung als Toast eingeblendet und der Titel des Feldes hervorgehoben. Wird eine gültige E-Mail eingetragen, aktiviert sich der Ladebalken für die Dauer der nachfolgenden Operation auf die Datenbank.

```
    auth.sendPasswordResetEmail(email).addOnCompleteListener { listener ->
        if (listener.isSuccessful) {
            Toast.makeText(
                applicationContext, text: "E-Mail erfolgreich verschickt",
                Toast.LENGTH_LONG
            ).show()
            progressBar.setVisibility(View.GONE)
            startActivity(Intent(applicationContext, PasswordDoneActivity::class.java))
            finish()
        } else {
            Toast.makeText(
                applicationContext, text: "Es tut uns Leid, etwas ist schiefgelaufen! Probiere es erneut",
                Toast.LENGTH_LONG
            ).show()
            progressBar.setVisibility(View.GONE)
        }
    }
}
```

Firebase implementiert bereits nativ eine Methode `sendPasswordResetEmail()`, welche eine E-Mail mit einem Link zum Zurücksetzen des Passworts an den anfragenden Benutzer versendet. Diese wird beim Klicken auf „Passwort zurücksetzen“ ausgelöst und verfügt außerdem über einen `onCompleteListener`, um den Erfolg der Operation zu überprüfen. Sofern die E-Mail erfolgreich übermittelt wird, erhalten die Benutzer über einen Toast Kenntnis darüber und der Ladebalken verschwindet aus der Sicht des Benutzers. Daraufhin werden die Benutzer im Erfolgsfall auf die Aktivität `PasswordDoneActivity` geführt, welche lediglich eine eigene Erfolgsmeldung für das erfolgreiche Verschicken der E-Mail darstellt und die aktuelle Aktivität beendet.

Im Falle eines Fehlers, wenn die E-Mail nicht verschickt wurde, wird der Benutzer über eine Fehlermeldung als Toast hingewiesen. Der Ladebalken verschwindet auch hier wiederum aus der Sicht des Benutzers.

## ACTIVITY\_FORGOT\_PASSWORD.XML



Der Screenshot zeigt das zugehörige Layout zur *PasswordForgotActivity* Aktivität. Zu sehen sind unter anderem das Textfeld für die E-Mail-Adresse und die Buttons zum Zurückkehren beziehungsweise Zurücksetzen.

## PASSWORDDONEACTIVITY.KT

Wie vorab beschrieben, ist *PasswordDoneActivity* die Aktivität, welche nach einem erfolgreichen Zurücksetzen des Passworts gestartet wird.

### Der Quellcode:

```
class PasswordDoneActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

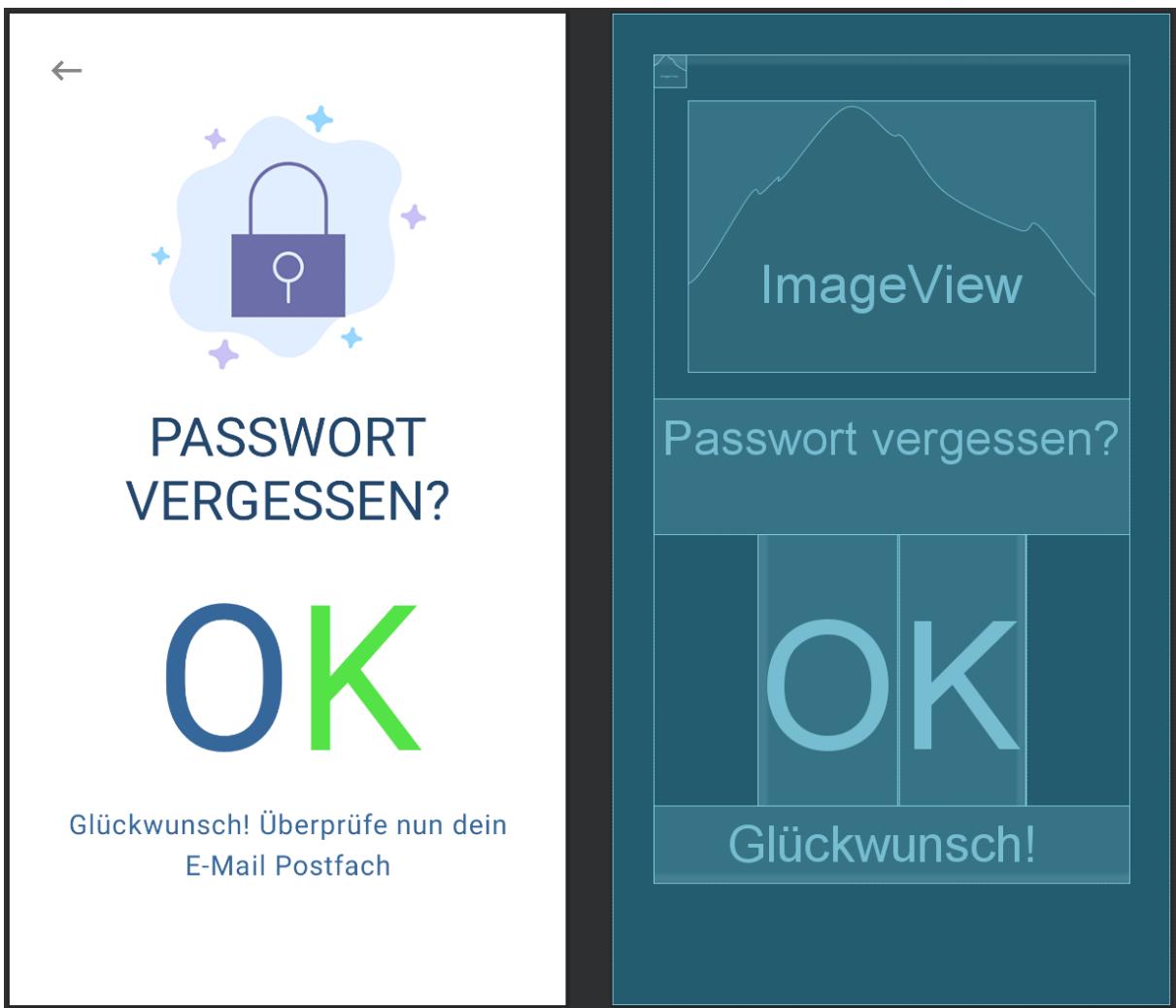
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_password_done)

        val zurueckBtn: ImageView = findViewById(R.id.zurick)

        zurueckBtn.setOnClickListener { it: View!
            startActivity(Intent(applicationContext , LoginActivity::class.java))
            finish()
        }
    }
}
```

Da es sich um eine Erfolgsmeldung handelt, gibt es bis auf eine *ImageView* zur Rückkehr zum Login Bereich keine Interaktionsmöglichkeiten für die Benutzer. Die *ImageView* ist demnach mit einem Click Listener versehen, der lediglich zur *LoginActivity* weiterleitet und die aktuelle beendet.

ACTIVITY\_PASSWORD\_DONE.XML



Der Screenshot bildet das zur Aktivität *PasswordDoneActivity* zugehörige Layout ab. Oben links zu sehen ist die *ImageView* in Form eines nach links gerichtetem Pfeil zur Rückkehr in den Login Bereich.

## WELCOMEACTIVITY.KT

Die *WelcomeActivity* bildet die erste Aktivität, welche nach einem erfolgreichen Einloggen in ein Benutzerprofil erscheint. Die Benutzer sehen hier einen scrollbaren Nachrichtenfeed.

### Der Quellcode:

```
class WelcomeActivity : AppCompatActivity() , NavigationView.OnNavigationItemSelectedListener {  
  
    lateinit var btnRight: ImageView  
    lateinit var btnLeft: ImageView  
    lateinit var userImage: CircleImageView  
    lateinit var nameUser: TextView  
    lateinit var mRecyclerView: RecyclerView  
    private lateinit var newsSection: Section  
  
    private val firestoreInstance: FirebaseFirestore by lazy {  
        FirebaseFirestore.getInstance()  
    }  
  
    // um das Bild in LoadFunction zu bekommen - hier haben wir wir get Root von Storage  
    private val storageInstance: FirebaseStorage by lazy {  
        FirebaseStorage.getInstance()  
    }  
}
```

Zu Beginn werden die Variablen für die Pfeilnavigation, das Profilbild, den Benutzernamen, der Nachrichtensektion und eine *RecyclerView* deklariert. Die *RecyclerView* wird verwendet, um die einzelnen Nachrichteneinträge vertikal und scrollbar darzustellen. Ferner werden jeweils eine Instanz für Firebase Firestore und Firebase Storage per *lazy init* beim ersten Zugriff abgegriffen. Über die Firebase Storage Instanz ist ein Zugriff auf den Firebase Cloud Speicher für zum Beispiel dort abgelegte Mediendateien möglich. Die Firebase Firestore Instanz dient wie üblich zum Zugriff auf die individuellen Daten des eingeloggten Benutzers.

```

override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_welcome)

    userImage = findViewById(R.id.userImage)
    nameUser = findViewById(R.id.nameUser)
    mRecyclerView = findViewById(R.id.mRecyclerView)

    addNewsListener(::initRecyclerView)
}

```

An der Stelle werden die oben genannten Variablen mit ihren Views verbunden. Eine Initialisierung der *RecyclerView* mit ihren Nachrichteneinträgen findet an der Stelle ebenfalls statt. Ein Listener überprüft dabei, ob neue Einträge existieren.

```

// Database get document von Firebase -> Step by id
firestoreInstance.collection( collectionPath: "users")
    .document(FirebaseAuth.getInstance().currentUser?.uid.toString())
    .get()
    .addOnSuccessListener { it: DocumentSnapshot!
        val user = it.toObject(User::class.java) // casting to Object
        nameUser.text = user!!.fullname

        if (user!!.profilBild?.isNotEmpty()) {
            GlideApp.with( activity: this)
                .load(storageInstance.getReference(user.profilBild))
                .into(userImage)
        } else {
            userImage.setImageResource(R.drawable.dozent)
        }
    }
}

```

Über die Firebase Firestore Instanz werden nun die benutzerspezifischen Daten geladen und in der Aktivität abgebildet. Konkret werden der vollständige Name des Benutzers und falls vorhanden auch sein Profilbild in die Aktivität geladen, sofern der *onSuccessListener* feststellt, dass die Benutzerdaten erfolgreich aus der Datenbank geladen werden konnten.

```

btnRight = findViewById(R.id.right)
btnLeft = findViewById(R.id.left)

btnRight.setOnClickListener { it: View!
    startActivity(Intent(applicationContext , NoteActivity::class.java))
}

btnLeft.setOnClickListener { it: View!
    Toast.makeText(
        applicationContext ,
        text: "Du bist bereits auf der ersten Seite!" ,
        Toast.LENGTH_LONG
    ).show()
}

val drawerLayout: DrawerLayout = findViewById(R.id.drawerLayout)
val toolbar1: Toolbar = findViewById(R.id.toolbar1)
val navigationView: NavigationView = findViewById(R.id.nav_view)
setSupportActionBar(toolbar1)

```

Weiterhin werden die Variablen für die Pfeilnavigation initialisiert und jeweils ein Click Listener auf die Navigationspfeile gesetzt. Da der Nachrichtenkanal die erste sichtbare Aktivität nach dem Login ist, soll eine versuchte Navigation nach links über die linke Pfeil-View den Hinweis erzeugen, dass es sich bereits um die erste Aktivität handelt. Hingegen werden die Benutzer bei einer Navigation nach rechts auf die Aktivität für die Notizen weitergeleitet. Zuletzt werden in diesem Codeabschnitt noch das *DrawerLayout*, die Toolbar und das Kontextmenü am oberen linken Rand mit ihren Variablen verbunden.

```

val toggle = ActionBarDrawerToggle(
    activity: this , drawerLayout , toolbar1 , openDrawerContentDescRes: 0 , closeDrawerContentDescRes: 0
)

drawerLayout.addDrawerListener(toggle)
toggle.syncState()
navigationView.setNavigationItemSelectedListener(this)
}

```

Ein Klick auf das Symbol zum Öffnen des Kontextmenüs löst in der Methode *actionBarDrawerToggle()* ein Ausfahren beziehungsweise bei einem erneuten Klick ein Einfahren der Navigationsleiste aus. Dafür wird ein Listener für das Navigationssymbol verwendet und zusätzlich der aktuelle Zustand synchronisiert. Ein weiterer Click Listener liegt auf den in der Navigationsleiste enthaltenen Einträgen, um die entsprechenden Weiterleitungen zu den passenden Aktivitäten vorzunehmen.

```

override fun onNavigationItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.home -> {
            startActivity(Intent(applicationContext , WelcomeActivity::class.java))
        }

        R.id.forum -> {
            startActivity(Intent(applicationContext , ForumActivity::class.java))
        }

        R.id.todos -> {
            startActivity(Intent(applicationContext , NoteActivity::class.java))
        }

        R.id.moodle -> {
            val url = "https://moodle.hs-worms.de/moodle/"
            val i = Intent(Intent.ACTION_VIEW)
            i.data = Uri.parse(url)
            startActivity(i)
        }

        R.id.lsf -> {
            val url = "https://lsf.hs-worms.de/"
            val i = Intent(Intent.ACTION_VIEW)
            i.data = Uri.parse(url)
            startActivity(i)
        }

        R.id.webmailer -> {
            val url = "https://webmailer2.hs-worms.de/"
            val i = Intent(Intent.ACTION_VIEW)
            i.data = Uri.parse(url)
            startActivity(i)
        }

        R.id.profile -> {
            startActivity(Intent(applicationContext , ProfileActivity::class.java))
        }

        R.id.abmelden -> {
            FirebaseAuth.getInstance().signOut()
            Toast.makeText(baseContext , text: "Abmeldung läuft" , Toast.LENGTH_SHORT).show()
            startActivity(Intent(applicationContext , LoginActivity::class.java))
        }
    }

    return true
}

```

Jeder Eintrag in der Navigationsleiste wird durch seine jeweilige ID identifiziert. Der Klick eines Benutzers auf einen Eintrag wird in einer *when-Verzweigung* ausgewertet und startet seine zugehörige Aktivität. Für die Einträge Moodle, LSF und Webmailer

wird stattdessen die entsprechende URL im Browser geöffnet. Bei einem Klick auf Abmelden, wird über die Firebase Instanz die Abmeldung eines Benutzers mit einem Hinweis als Toast durchgeführt und er wird zur *LoginActivity* zurückgeleitet.

```
private fun addNewsListener(onListen: (List<Item>) -> Unit): ListenerRegistration {
    return firestoreInstance.collection( collectionPath: "news")
        .addSnapshotListener { querySnapshot , firebaseFirestoreException ->
            if (firebaseFirestoreException != null) {
                return@addSnapshotListener
            }

            val items = mutableListOf<Item>()
            querySnapshot!!.documents.forEach { it: DocumentSnapshot!
                items.add(NewsAdapter(it.toObject(News::class.java)!! , context: this))
            }

            onListen(items)
        }
}

data class News(
    var ContentNews: String ,
    var titleNews: String ,
    var mNewsPhoto: String ,
    var dateNews: String
) {

    constructor() : this( ContentNews: "" , titleNews: "" , mNewsPhoto: "" , dateNews: "" )
}
```

#### MODEL/NEWS.KT

Die Methode *addNewsListener()* konvertiert die Nachrichteneinträge aus Firestore in Objekte der Klasse *News* und bildet diese in der *RecyclerView* ab. Zu Beginn der Aktivität wird eine initialisierte *RecyclerView* an die Methode übergeben, welche dann mit den Nachrichteneinträgen befüllt wird. Konkret wird jedes einzelne Objekt der Klasse *News* an den Konstruktor der Klasse *NewsAdapter* übergeben, welche das Datenobjekt in eine für die *RecyclerView* darstellbare Form umwandelt. Bei einem Fehler in der Methode, wird eine Firebase Firestore Exception ausgelöst.

Bei der Klasse *News* handelt es sich um die zugehörige Datenklasse für die einzelnen Nachrichteneinträge. Sie speichert über den Nachrichteninhalt hinaus alles was zu dem Nachrichteneintrag dazugehört, wie etwa das jeweilige Bild oder das

Verfassungsdatum. Ein *News* Objekt ist das reine Datenobjekt für einen Artikel und muss im Anschluss wie geschildert in eine darstellbare Form gebracht werden.

```
class NewsAdapter(val news: News , val context: Context) : Item() {  
  
    private val storageInstance: FirebaseStorage by lazy {  
        FirebaseStorage.getInstance()  
    }  
  
    override fun bind(viewHolder: GroupieViewHolder , position: Int) {  
        viewHolder.titleNews.text = news.titleNews  
        viewHolder.textNews.text = news.ContentNews  
        viewHolder.dateNews.text = news.dateNews  
  
        if (news.mNewsPhoto.isNotEmpty()) {  
            GlideApp.with(context)  
                .load(storageInstance.getReference(news.mNewsPhoto))  
                .into(viewHolder.imgNews)  
        } else {  
            viewHolder.imgNews.setImageResource(R.drawable.newsbild)  
        }  
    }  
  
    override fun getLayout(): Int {  
        return R.layout.recycler_view_item  
    }  
}
```

ADAPTER/NEWSADAPTER.KT

*NewsAdapter* nimmt dann ein *News* Datenobjekt entgegen und setzt die Inhalte für die Einträge der *RecyclerView* über den *viewHolder* und der zugehörigen Position eines Eintrags als Integer in der überschriebenen Methode *bind()*. Das Titelbild für einen Artikel ist im Cloud Speicher in Firebase Storage abgelegt und muss deshalb zunächst über eine Firebase Storage Instanz geladen und anschließend gesetzt werden. Falls kein zugehöriges Bild existiert, wird ein standardmäßiges Bild gesetzt.

```
@SuppressLint("WrongConstant")
private fun initRecyclerView(item: List<Item>) {
    m RecyclerView.layoutManager = LinearLayoutManager(context, LinearLayout.VERTICAL, false)
    m RecyclerView.apply { this.RecyclerView
        adapter = GroupAdapter<GroupieViewHolder>().apply { this: GroupAdapter<GroupieViewHolder>
            newsSection = Section(item)
            add(newsSection)
            setOnItemClickListener(onItemClick)
        }
    }
}
```

Die Methode `initRecyclerView()` implementiert die Initialisierung der `RecyclerView`. Es wird eine vertikale Darstellung definiert, um die Nachrichteneinträge scrollbar untereinander darzustellen. Neben dem Hinzufügen der Einträge über die Methode `add()` wird für jeden Eintrag ein Click Listener registriert.

```
private val onItemClick = OnItemClickListener { item, view ->
    if (item is NewsAdapter) {
        item.news.titleNews
        item.news.ContentNews
        item.news.mNewsPhoto
        item.news.dateNews

        val intent = Intent(applicationContext, NewsContentActivity::class.java)

        intent.putExtra(name: "title", item.news.titleNews)
        intent.putExtra(name: "contenu", item.news.ContentNews)
        intent.putExtra(name: "img", item.news.mNewsPhoto)
        intent.putExtra(name: "date", item.news.dateNews)
        startActivity(intent)
    }
}
```

Der Click Listener für die einzelnen Einträge der `RecyclerView` überprüft zunächst, ob es sich bei dem Eintrag um einen Nachrichteneintrag handelt. Ist das der Fall, wird die Aktivität `NewsContentActivity` mit den Daten des jeweiligen Eintrags befüllt und anschließend gestartet.

```
class NewsContentActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_news_content)

        val title = intent.getStringExtra( name: "title")
        val content = intent.getStringExtra( name: "contenu")
        val dateNews = intent.getStringExtra( name: "date")

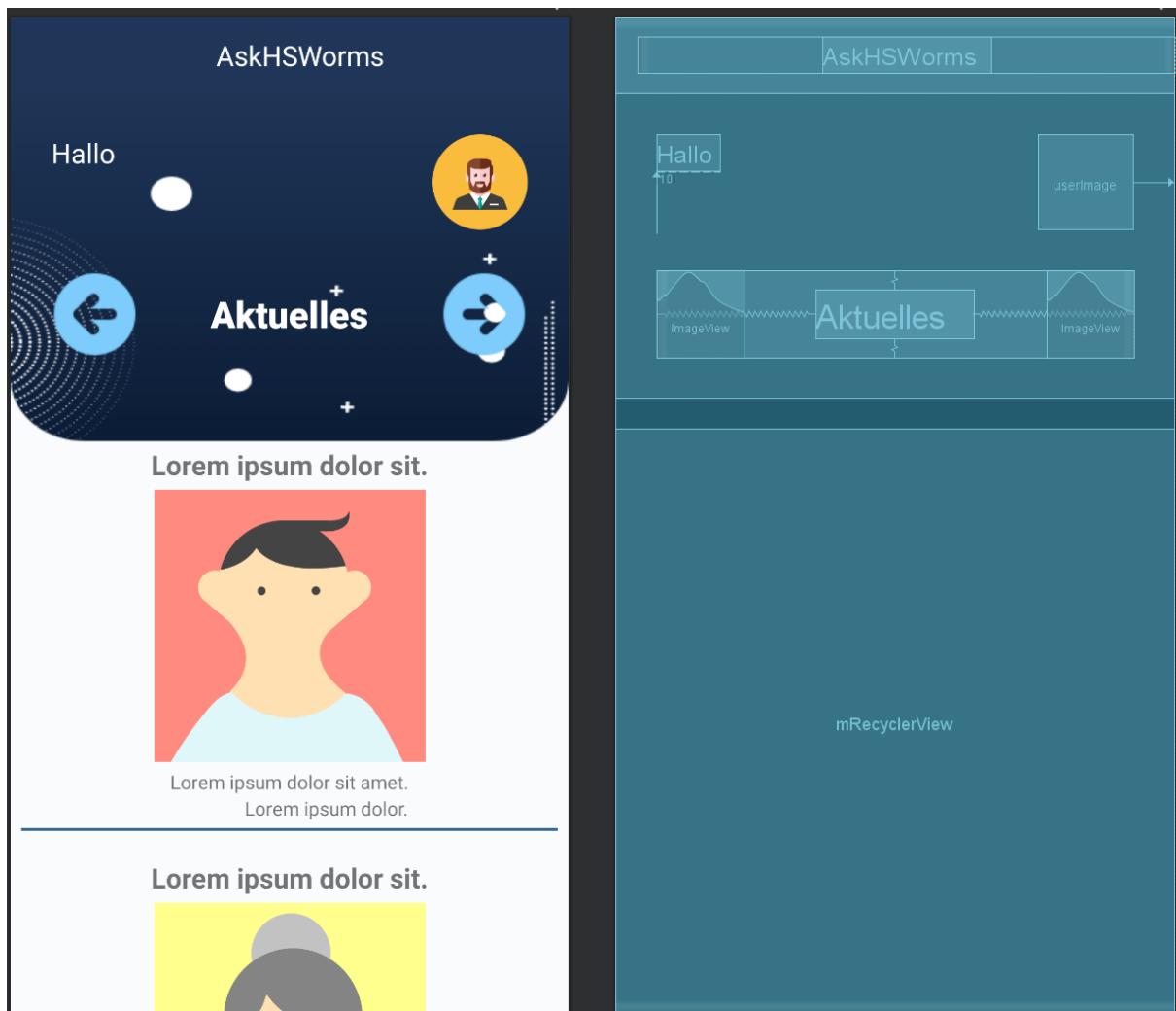
        titleNews.text = title
        contenus.text = content
        date.text = dateNews

        rjaa.setOnClickListener { it: View!
            finish()
        }
    }
}
```

NEWS/NEWSCONTENTACTIVITY.KT

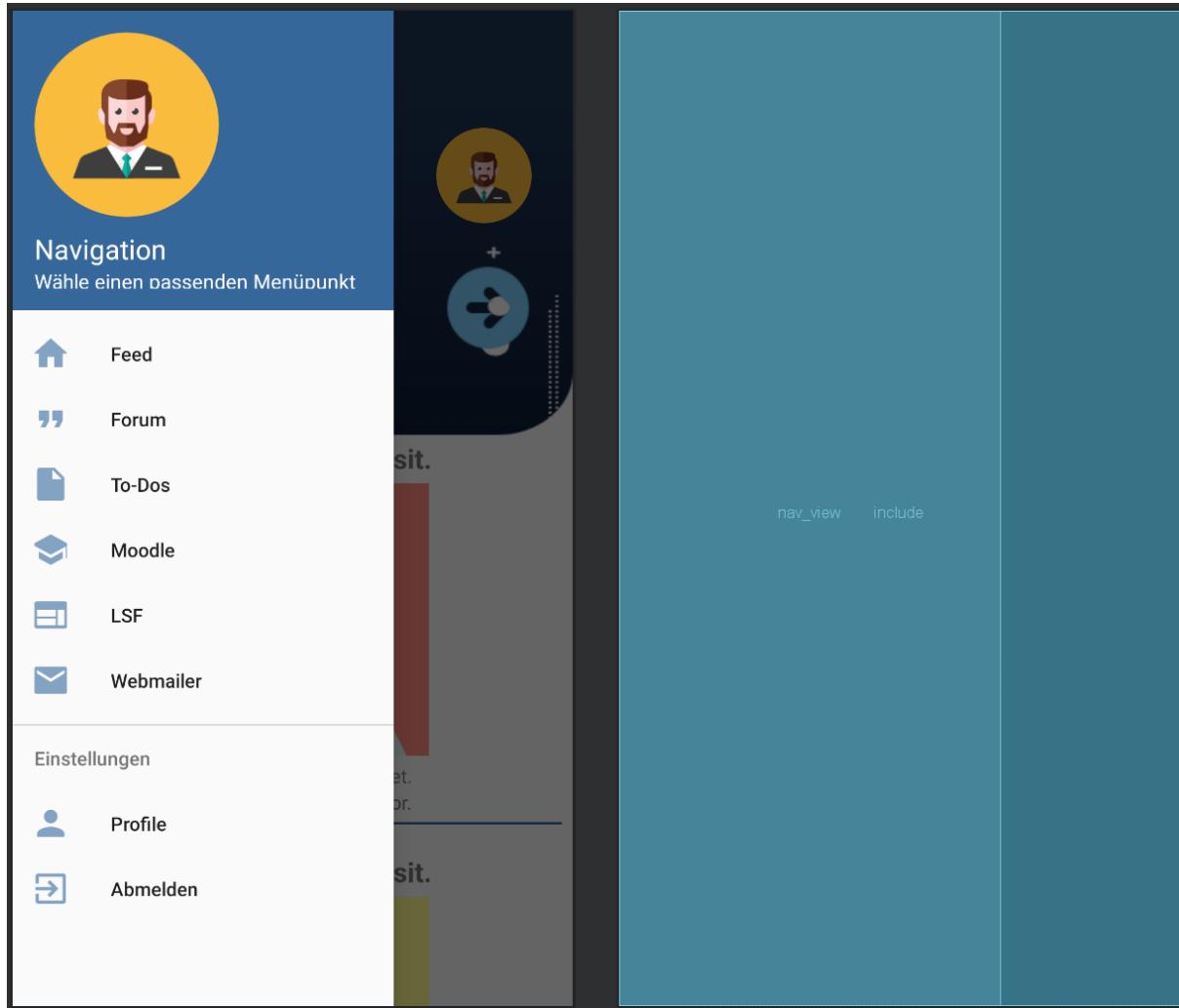
Die Aktivität *NewsContentActivity* bildet dabei die Einzelansicht für jeden Nachrichtenartikel ab, damit die Benutzer eine Nachricht in seiner Gesamtheit in einer eigenen Aktivität ohne weitere Störelemente lesen können. Über den Pfeil zum Zurücknavigieren wird die Aktivität beendet und die Benutzer bekommen dann wieder die vorherige Aktivität zu sehen.

## CONTENT\_MAIN.XML



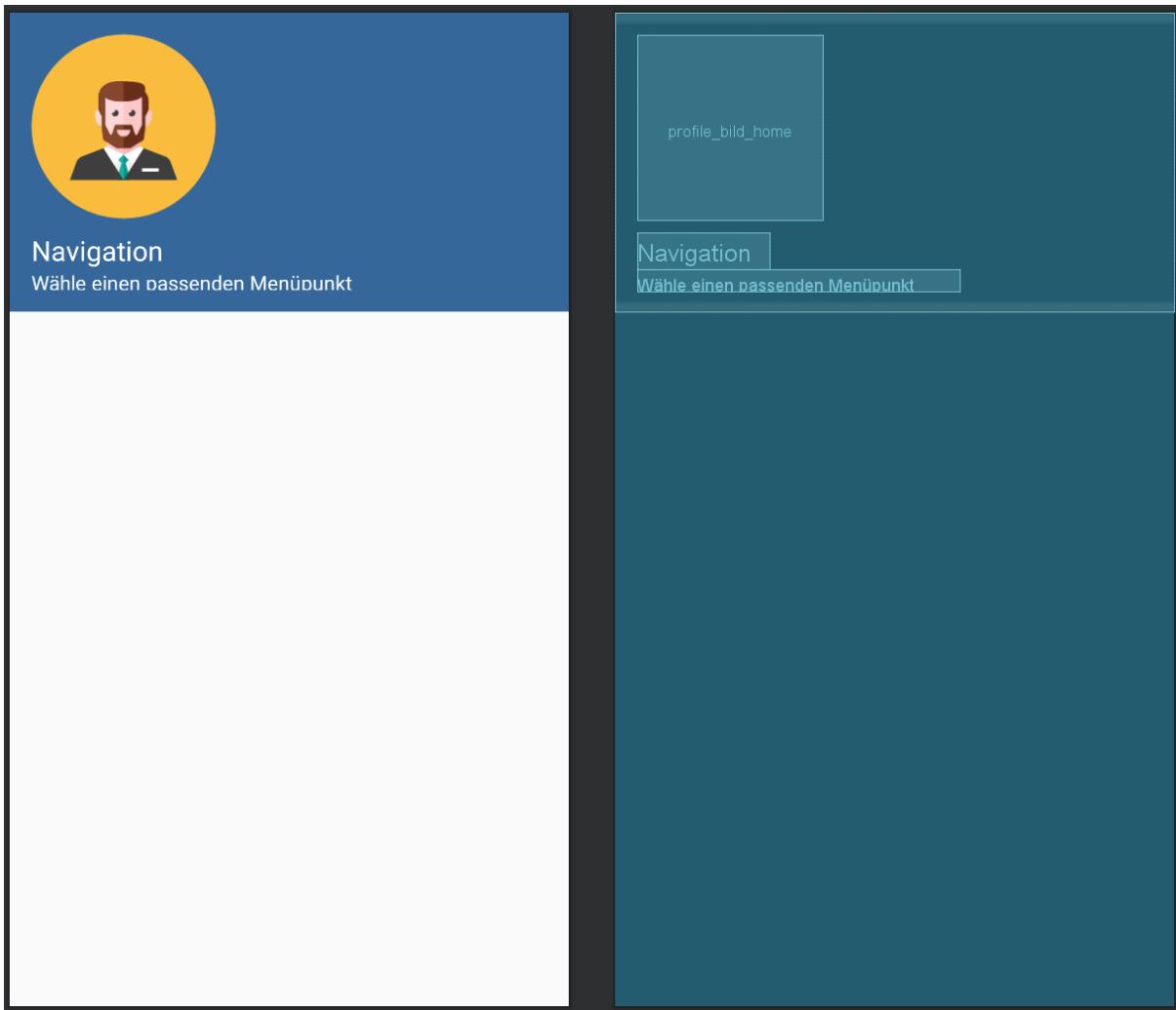
*content\_main.xml* ist das eigentliche Layout für die Aktivität nach dem Login. Zu sehen sind unter anderem die *RecyclerView* mit Platzhaltern für die Nachrichteneinträge, der Titel der Aktivität, die Pfeilnavigation, ein Platzhalter für das Benutzerprofilbild und eine *TextView* mit dem Inhalt „Hallo“, welche während der Laufzeit um den Namen des Benutzers ergänzt wird.

## ACTIVITY\_WELCOME.XML



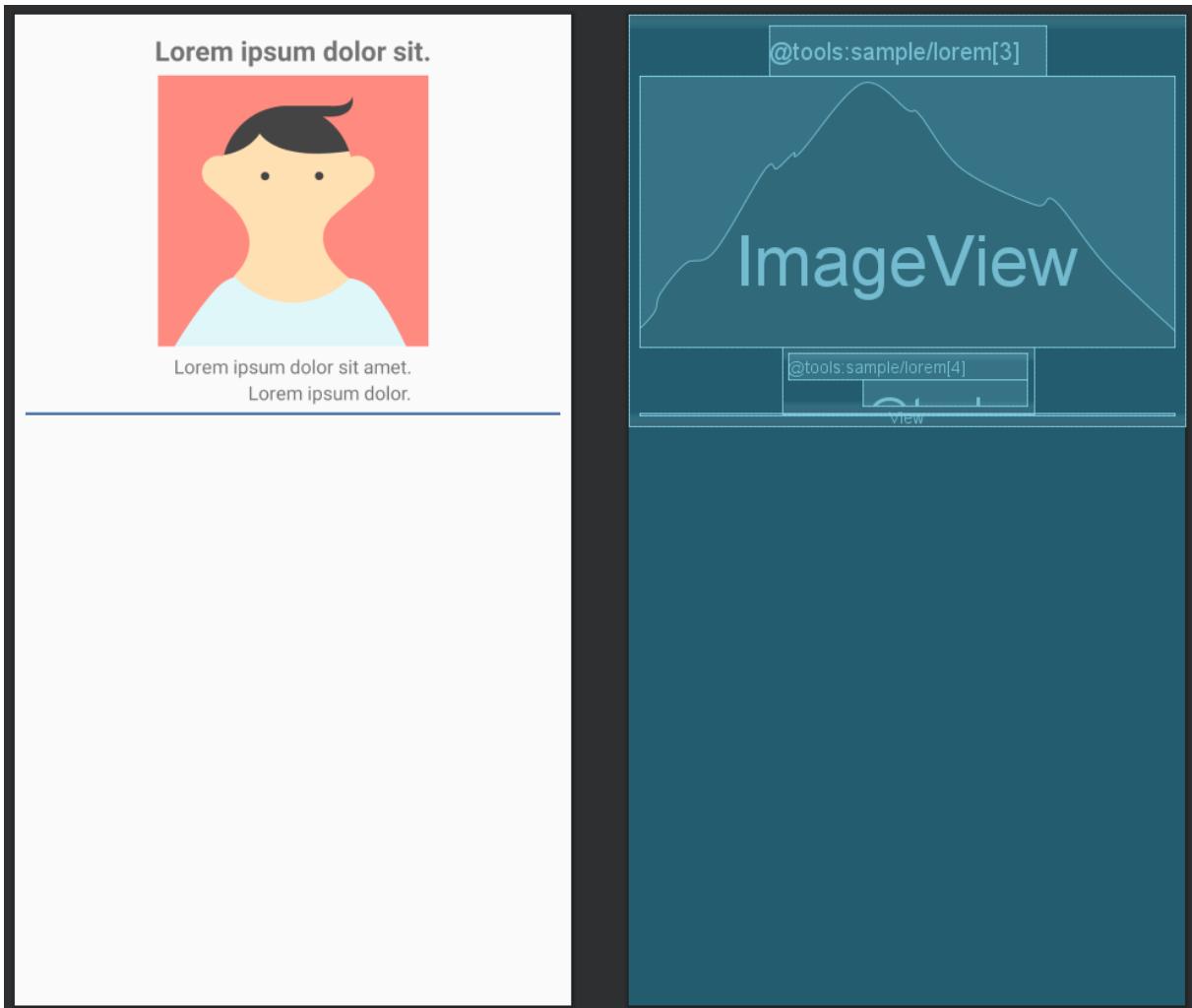
Der Unterschied zur vorherigen Layout-Datei ist lediglich, dass hier die Ansicht mit aufgeklappter Navigationsleiste über dem eigentlichen Layout gezeigt ist. Die Navigationsleiste besteht aus ihrem Kopfbereich, welcher in der Layout-Datei *menu\_header.xml* definiert ist und dem Körperbereich mit den Listeneinträgen.

## MENU\_HEADER.XML



Abgebildet ist an der Stelle die Navigationsleiste mit der Gestaltung des Kopfbereichs und der zunächst leere Körperbereich.

## RECYCLER\_VIEW\_ITEM.XML



Zu sehen ist das Layout für jeden Eintrag einer *RecyclerView*. Die darin gezeigten Inhalte sind lediglich Platzhalter für die während der Laufzeit hinzugefügten Inhalte. Die *RecyclerView* wird neben der *WelcomeActivity* in der Form in weiteren Aktivitäten verwendet.

## NOTEACTIVITY.KT

Ein Benutzer welcher über die Nachrichtenfeed-Aktivität (= *WelcomeActivity*) mittels Pfeilnavigation nach rechts navigiert, landet auf der Aktivität für die Notizenfunktion. Hier können die Benutzer individuelle Notizen erstellen, anschauen, verändern und wieder löschen. Der Aufbau der Aktivität ist im Allgemeinen jedoch ähnlich zu der vorherigen und allen weiteren Aktivitäten, da Elemente wie die Toolbar, die Navigationspfeile, das Benutzerbild, Benutzername und das Kontextmenü sich wiederholen.

### Der Quellcode:

```
class NoteActivity : AppCompatActivity() , NavigationView.OnNavigationItemSelectedListener {

    lateinit var btnRight: ImageView
    lateinit var btnLeft: ImageView
    lateinit var toolbar1: Toolbar
    lateinit var userImage: CircleImageView
    lateinit var drawerLayout: DrawerLayout
    lateinit var navigationView: NavigationView
    lateinit var nameUser: TextView
    lateinit var add: ImageView
    var mNoteList: ArrayList<Note>? = null

    var mRef: DatabaseReference? = null

    private val firestoreInstance: FirebaseFirestore by lazy {
        FirebaseFirestore.getInstance()
    }

    // um das Bild in LoadFunction zu bekommen - hier haben wir wir get Root von Storage
    private val storageInstance: FirebaseStorage by lazy {
        FirebaseStorage.getInstance()
    }
}
```

Alle später verwendeten Elemente werden wie zuvor auch zunächst als *lateinits* deklariert, um sie hinterher mit ihren Views zu verbinden. Hinzu kommt eine *ArrayList* für alle Notizen des Benutzers und eine *ImageView*, welche das Hinzufügen von neuen Notizen ermöglicht. Wie üblich werden Referenzen auf Storage und Datenbank benötigt, um alle benutzerspezifischen Daten zu ermitteln.

```

override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_notizen)

    userImage = findViewById(R.id.userImage)
    nameUser = findViewById(R.id.nameUser)
    add = findViewById(R.id.add)

    val database = FirebaseDatabase.getInstance()
    mRef = database.getReference(path: "Notes")
    mListNote = ArrayList()

    add.setOnClickListener { it: View!
        showDialogAddNote()
    }
}

```

```

private fun showDialogAddNote() {

    val alertDialogBuilder = AlertDialog.Builder(context: this)
    val view = LayoutInflater.inflate(R.layout.add_note , root: null)
    alertDialogBuilder.setView(view)
    val alertDialog = alertDialogBuilder.create()

    alertDialog.show()

    view.save_note.setOnClickListener { it: View!
        val title = view.title_edittext.text.toString()
        val note = view.note_edittext.text.toString()

        if (title.isNotEmpty() && note.isNotEmpty()) {
            val id = mRef!!.push().key
            val myNote = Note(id , title , note , getCurrentDate())

            if (id != null) {
                mRef!!.child(id).setValue(myNote)
            }

            alertDialog.dismiss()
        } else {
            Toast.makeText(context: this , text: "Empty" , Toast.LENGTH_LONG).show()
        }
    }
}

```

Bei der Erstellung der Aktivität, werden anschließend die Views für das Benutzerbild, den Namen und die Hinzufügen *ImageView* mit ihren Variablen gekoppelt, die *ArrayList* initialisiert und eine Datenbank Referenz auf die Notizensektion angelegt. Die *ImageView add* zum Hinzufügen von Notizen wird nun noch über einen Click Listener mit ihrer Methode *showDialogAddNote()* gekoppelt.

Die Methode baut zunächst einen Dialog mit Hilfe der Klasse *AlertDialog* auf und nutzt als Layout die Datei *add\_note.xml*, auf der Notizen hinzugefügt werden können. Der erstellte Dialog wird daraufhin als View gesetzt und über die Methode *show()* angezeigt. Der Button zum Speichern der erstellten Notiz wird im Dialog mit einem Click Listener versehen. Möchte ein Benutzer seine geschriebene Notiz speichern, werden die eingetragenen Inhalte für den Titel und den Notiztext in ihren entsprechenden Variablen zwischengespeichert. Sofern beide Textfelder nicht leer sind, werden die zwischengespeicherten Daten zunächst in ein Objekt der Klasse *Note* umgewandelt und als Ganzes in der Datenbank gesichert.

#### MODEL/NOTE.KT

```
class Note {

    var id: String? = null
    var title: String? = null
    var note: String? = null
    var timestamp: String? = null

    constructor()

    constructor(id: String? , title: String , note: String , timestamp: String?) {
        this.id = id
        this.title = title
        this.note = note
        this.timestamp = timestamp
    }
}
```

Die Klasse *Note* stellt dabei nur eine Hilfsklasse zur Speicherung der Notizeninhalte. Sie speichert darüber hinaus noch einen Zeitstempel für den Zeitpunkt der

Erstellung und eine Notiz-ID. Der Zeitstempel wird bei der Erzeugung des Objekts über die Methode *getCurrentDate()* übergeben.

```
private fun getCurrentDate(): String {

    val calendar = Calendar.getInstance()
    val mdformat = SimpleDateFormat(pattern: "yyyy-MM-dd'T'HH:mm:ss'Z'" , Locale.GERMANY)
    val strDate = mdformat.format(calendar.time)
    return strDate.toString()
}
```

Dabei wird über eine *Calendar* Instanz ein Datumsobjekt erzeugt, welches für den Raum Deutschland das aktuelle Datum und Uhrzeit generiert und als Zeichenkette zurückgibt.

Nach der Übertragung in die Datenbank wird das Dialogfeld geschlossen. Ist jedoch mindestens eines der Textfelder leer, wird lediglich ein Toast mit einer Fehlermeldung erzeugt.

```
note_list_view.onItemClickListener =  
    AdapterView.OnItemClickListener { parent, view, position, id ->  
        val mynote = mNoteList?.get(position)!!  
        val title = mynote.title  
        val note = mynote.note  
        val tarikh = mynote.timestamp  
        val noteIntent = Intent(packageContext: this@NoteActivity, NoteContentActivity::class.java)  
  
        noteIntent.putExtra(name: "Title_Key", title)  
        noteIntent.putExtra(name: "Note_Key", note)  
        noteIntent.putExtra(name: "Time_Key", tarikh)  
        startActivity(noteIntent)  
    }  
}
```

Weitergehend wird in der Methode `onCreate()` ein Click Listener auf die Notizeneinträge geschaltet. Für die angetippte Notiz werden die zugehörigen Daten in Variablen extrahiert und der Aktivität `NoteContentActivity` mitgegeben. Diese wird darauffolgend gestartet und präsentiert vergleichbar zur Aktivität `NewsContentActivity` eine Einzelsicht für den jeweiligen Notizeneintrag in einer neuen Aktivität.

```

note_list_view.onItemLongClickListener =
    AdapterView.OnItemLongClickListener { parent, view, position, id ->
        val alertDialog = AlertDialog.Builder(context: this@NoteActivity)
        val view = layoutInflater.inflate(R.layout.delete_note, root: null)
        val alertDialog = alertDialog.create()

        alertDialog.setView(view)
        alertDialog.show()

        val myNote = mNoteList?.get(position)!!
        val title = myNote.title
        val note = myNote.note

        view.delete_title.setText(title)
        view.delete_note.setText(note)

        view.btnUpdateNote.setOnClickListener { it: View!
            val childRef = mRef?.child(myNote.id.toString())
            val title = view.delete_title.text.toString()
            val note = view.delete_note.text.toString()

            val afterUpdate = Note(myNote.id, title, note, getCurrentDate())
            childRef?.setValue(afterUpdate)
            alertDialog.dismiss()
        }

        view.btnDeleteNote.setOnClickListener { it: View!
            mRef?.child(myNote.id.toString())?.removeValue()

            Toast.makeText(baseContext, text: "Die Notiz wurde gelöscht", Toast.LENGTH_SHORT)
                .show()
            alertDialog.dismiss()
        }
    }

    false ^OnItemLongClickListener
}

```

Ein weiterer Click Listener dient für das Verändern oder Löschen bereits bestehender Notizen und wird durch ein langes Tippen ausgelöst. Statt eine neue Aktivität zu starten, wird wie bei der Notizerstellung ein Dialog mit dem Layout von *delete\_note.xml* erzeugt und eingeblendet. Die extrahierten Daten der Notiz werden wiederholt in Variablen geschrieben und über *setText()* in die Textfelder gefüllt. Die Benutzer können dadurch Veränderungen am bereits bestehenden Text und Titel vornehmen, falls gewünscht. Ein erneutes Speichern der Änderungen geschieht über den „Update“-Button, welcher einen Click Listener implementiert. Ein Klick darauf überschreibt die bereits existierende Notiz in der Datenbank und beendet den Dialog.

Analog dazu löst der „Löschen“-Button eine Löschung des Eintrags aus der Datenbank aus, erzeugt eine Erfolgsmeldung als Toast und beendet den Dialog.

Die verbleibenden Codezeilen innerhalb von `onCreate()` dienen wie gehabt den Funktionalitäten zum dem Laden des Benutzerbildes und -namens, der Erzeugung der Pfeilnavigation, der Toolbox und des Kontextmenüs.

```
override fun onStart() {

    super.onStart()
    mRef?.addValueEventListener(object : ValueEventListener {
        override fun onDataChange(p0: DataSnapshot) {
            mNoteList?.clear()

            for (n in p0!!.children) {
                var note = n.getValue(Note::class.java)
                mNoteList?.add(index: 0 , note!!)
            }

            val noteAdapter = NoteAdapter(applicationContext , mNoteList!!)
            note_list_view.adapter = noteAdapter
        }

        override fun onCancelled(error: DatabaseError) {}
    })
}
```

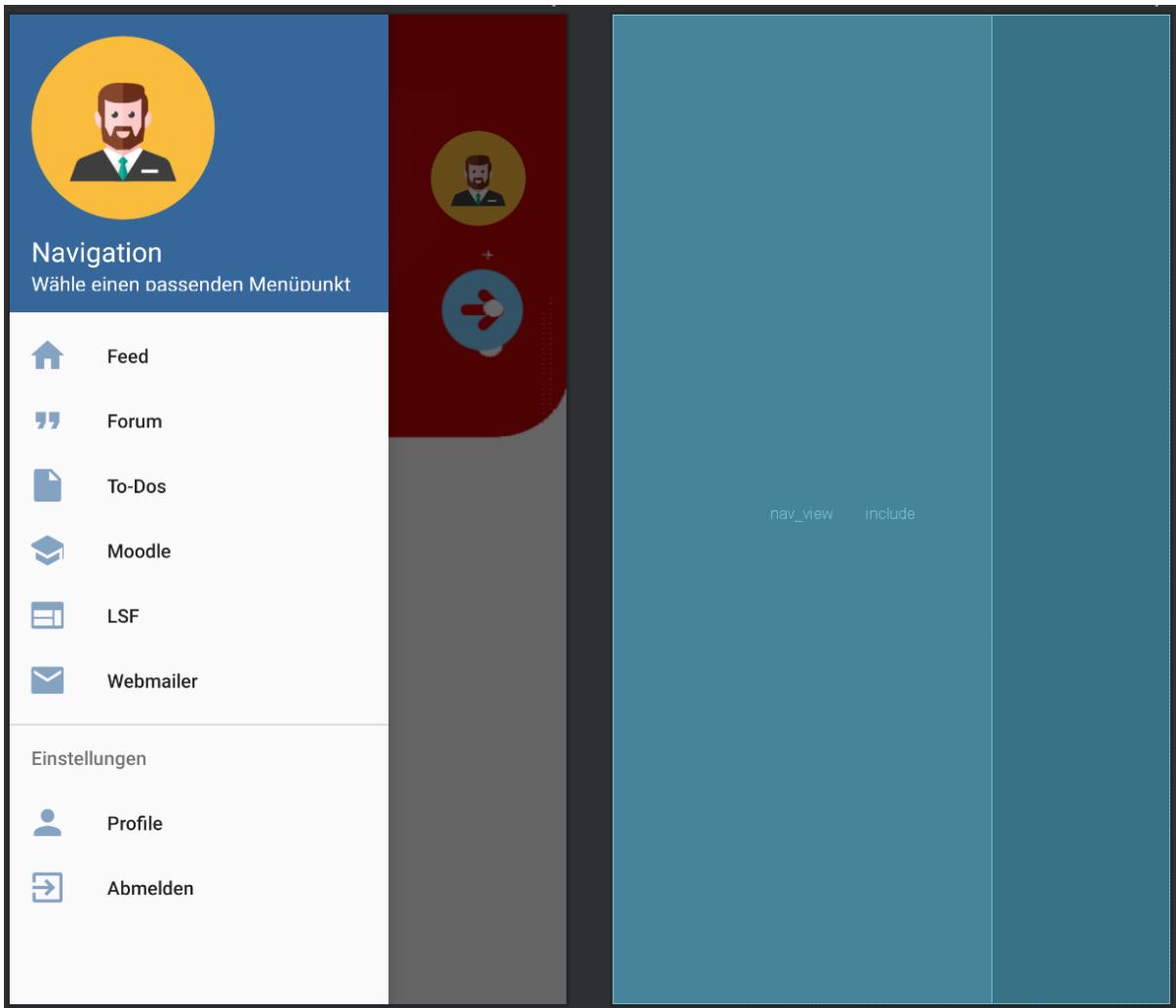
Nach der Initialisierung der Aktivität mit `onCreate()` folgt im Aktivitäten-Lebenszyklus natürlicherweise die `onStart()` Methode, welche an der Stelle überschrieben wurde. Mit Hilfe eines Event Listener, welcher nach Veränderungen der Notizen horcht, werden diese ohne einen Neustart der Aktivität bei Veränderung automatisch aktualisiert. Die veralteten Notizen werden dabei mit der Methode `clear()` aus der `ArrayList` entfernt und die aktualisierten Notizen mit Hilfe einer for-Schleife der nun leeren `ArrayList` wieder hinzugefügt. Zur Darstellung auf dem Bildschirm wird zusätzlich die Helperklasse `NoteAdapter` hinzugezogen, welche den Kontext und die `ArrayList` `mNoteList` übergeben bekommt.

```
class NoteAdapter(context: Context , noteList: ArrayList<Note>) :  
    ArrayAdapter<Note>(context , resource: 0 , noteList) {  
  
    override fun getView(position: Int , convertView: View? , parent: ViewGroup): View {  
  
        val view = LayoutInflator.from(context).inflate(R.layout.notes_items , parent , attachToRoot: false)  
        val note: Note? = getItem(position)  
  
        if (note != null) {  
            view.datum_text.text = note.timestamp.toString()  
            view.titel_note.text = note.title  
            view.contenu_notizen.text = note.note  
        }  
  
        return view  
    }  
}
```

#### ADAPTER/NOTEADAPTER.KT

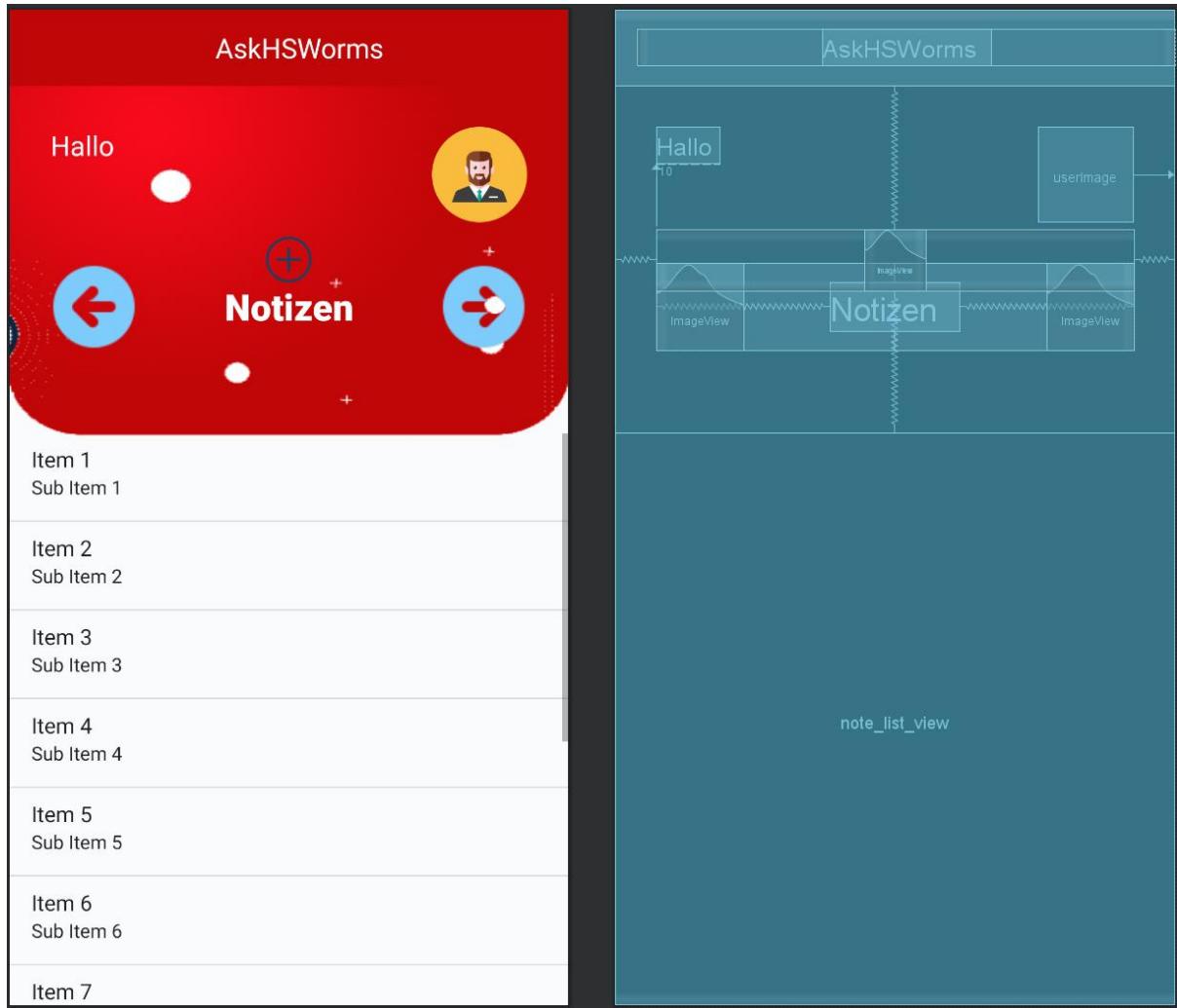
Anhand der Darstellung aus *notes\_items.xml*, erzeugt *NoteAdapter* demzufolge eine neue View pro Notizeintrag. Anschließend werden die Felder Datum, Titel und Inhalt der erzeugten View mit den Daten aus der Notiz befüllt. Zuletzt wird die View zurückgegeben und von der aufrufenden Methode abgebildet.

## ACTIVITY\_NOTIZEN.XML



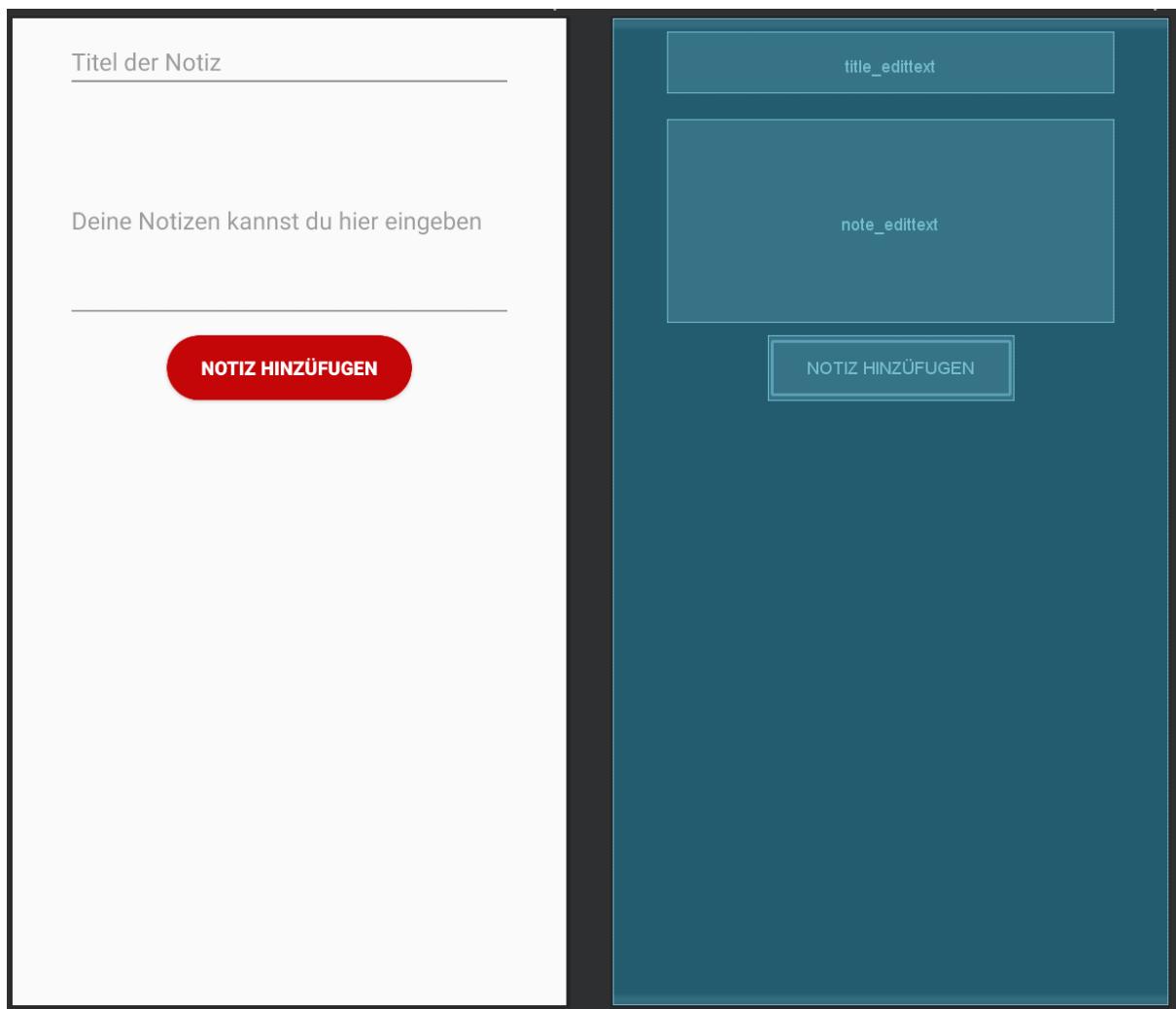
Abgebildet ist die Notizen Aktivität mit geöffneter Navigationsleiste.

## FRAGMENT\_NOTES.XML



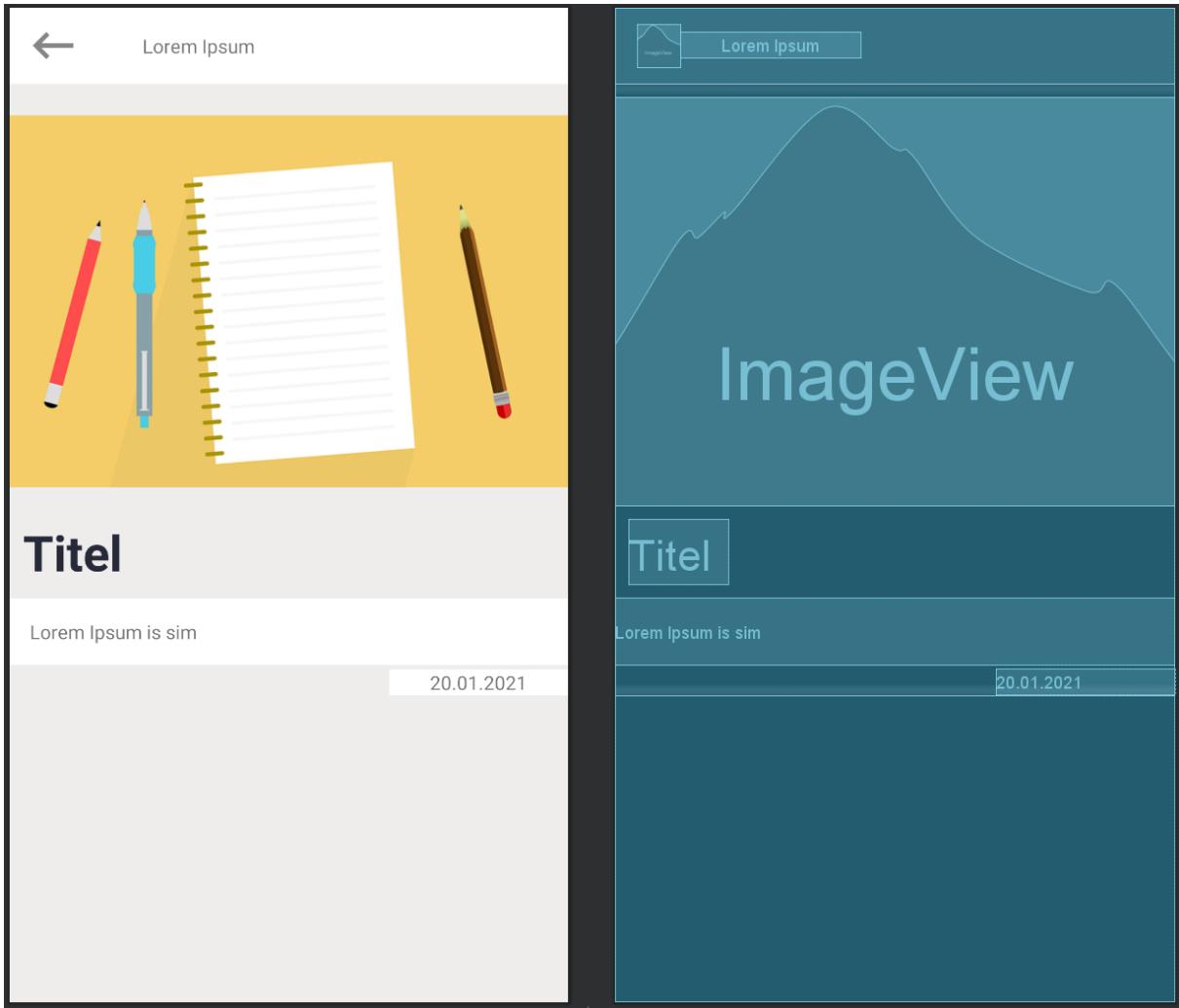
Zu sehen ist das eigentliche Layout für die Notizen Aktivität ohne Navigationsleiste mit einigen Beispielelementen, welche später durch die selbsterstellten Notizen der Benutzer ersetzt werden. Mit der *ImageView* „+“ können neue Notizen hinzugefügt werden. Alle weiteren Bedienelemente sind die standardmäßig vorhandenen.

## ADD\_NOTE.XML



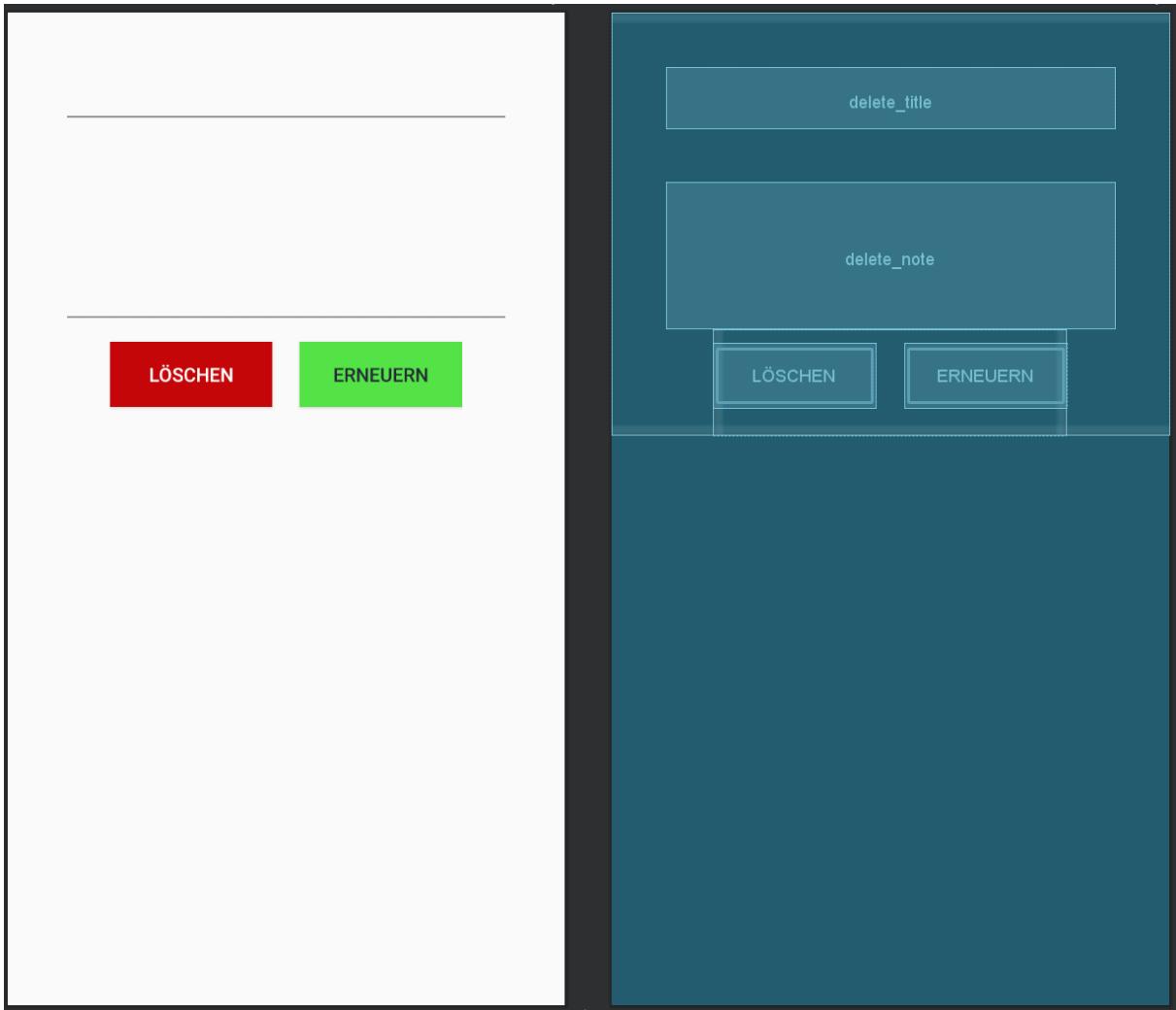
Entscheidet sich ein Benutzer dazu eine neue Notiz zu erstellen, bekommt er das gezeigte Dialogfenster zu sehen. Ausfüllbar sind ein Titelfeld für die Notizbeschreibung und ein Textfeld für den Notizinhalt. Das Speichern der Notiz geschieht über den „Notiz hinzufügen“-Button, welcher das Dialogfenster im Anschluss beendet.

## ACTIVITY\_CONTENT\_NOTE.XML



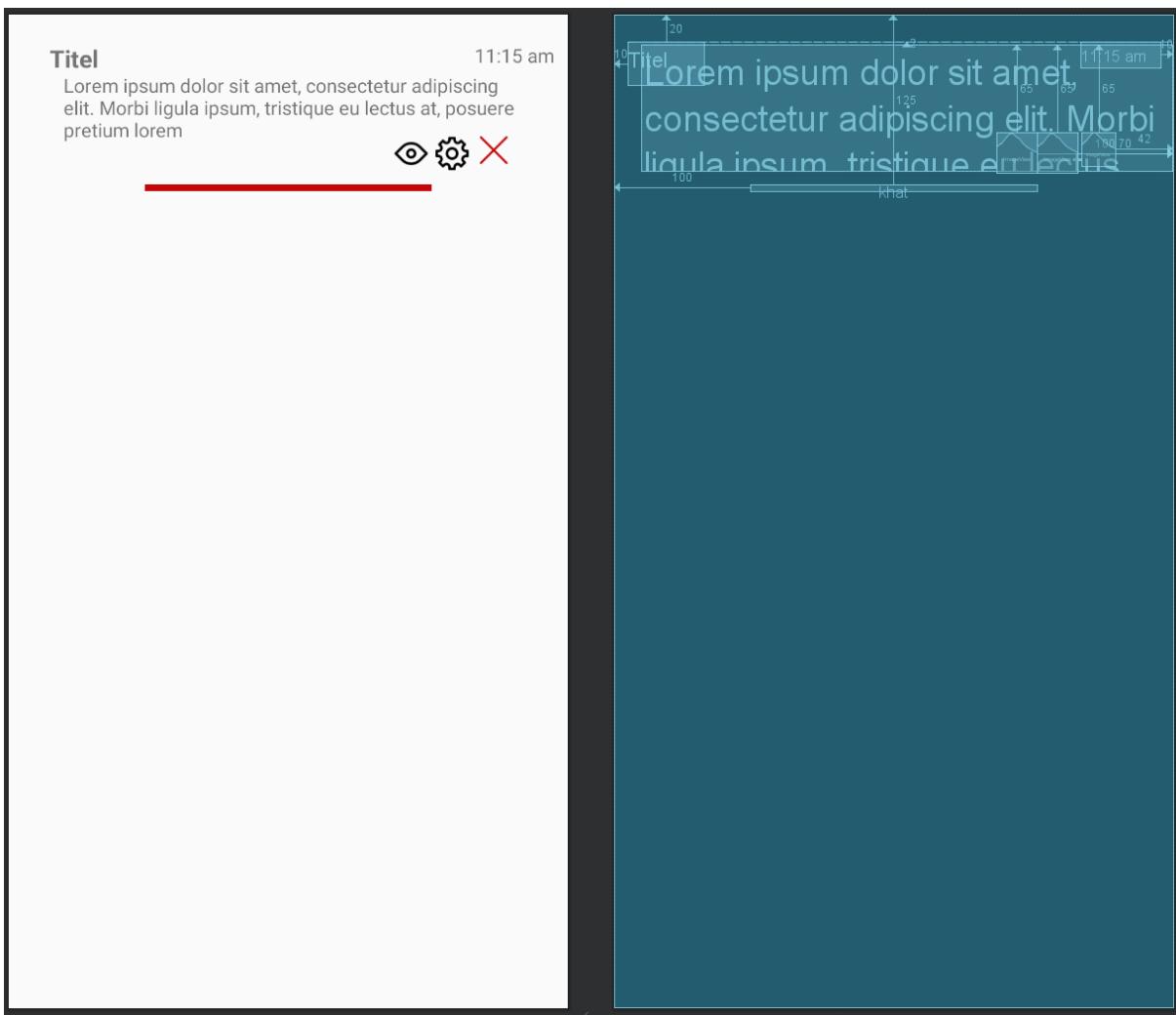
Beim Tippen auf eine Notiz in *NoteActivity*, wird eine Einzelansicht geöffnet, in der die jeweilige Notiz im Detail überblickt werden kann. Das Layout korrespondiert mit der Aktivität *NoteContentActivity*.

## DELETE\_NOTE.XML



An der Stelle kann im Vergleich zu *add\_note.xml* eine Notiz erneuert oder gelöscht werden. Ein langes Tippen auf eine Notiz in *NoteActivity* bildet das gezeigte Layout in einem eigenen Dialogfenster ab. Die Buttons „Löschen“ und „Update“ lösen intern ihre Methoden aus und beenden das Dialogfenster im Anschluss.

## NOTES\_ITEM.XML



Über die Klasse *NoteAdapter* werden die Views für die einzelnen Notizen mit dem gezeigten Layout erzeugt. Die *NoteActivity* bildet die View anschließend als Aufrufer in der Notizenliste ab und präsentiert sie den Benutzern als klickbarer Eintrag.

## FORUMACTIVITY.KT

Die *ForumActivity* ist die Aktivität für das Forum von AskHSWorms. An der Stelle können die Benutzer zunächst ein Thema oder Studiengang auswählen und anschließend dort ihre Fragen stellen. Jeder Student kann unabhängig von seinem eigenen Studiengang oder seiner Benutzerrolle auf den Fragenkatalog eines unterschiedlichen Studiengangs Zugreifen. Eine beispielsweise Implementierung wurde lediglich für den Studiengang Informatik umgesetzt. Alle weiteren gezeigten Studiengänge sind zu Veranschaulichungszwecken gedacht und demonstrieren was bei einer echten Veröffentlichung möglich wäre.

### Der Quellcode:

```
class ForumActivity : AppCompatActivity() , NavigationView.OnNavigationItemSelectedListener {

    lateinit var btnRight: ImageView
    lateinit var btnLeft: ImageView
    lateinit var toolbar1: Toolbar
    lateinit var userImage: CircleImageView
    lateinit var drawerLayout: DrawerLayout
    lateinit var navigationView: NavigationView
    lateinit var nameUser: TextView

    private val firestoreInstance: FirebaseFirestore by lazy {
        FirebaseFirestore.getInstance()
    }

    // um das Bild in LoadFunction zu bekommen - hier haben wir wir get Root von Storage
    private val storageInstance: FirebaseStorage by lazy {
        FirebaseStorage.getInstance()
    }

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_forum)

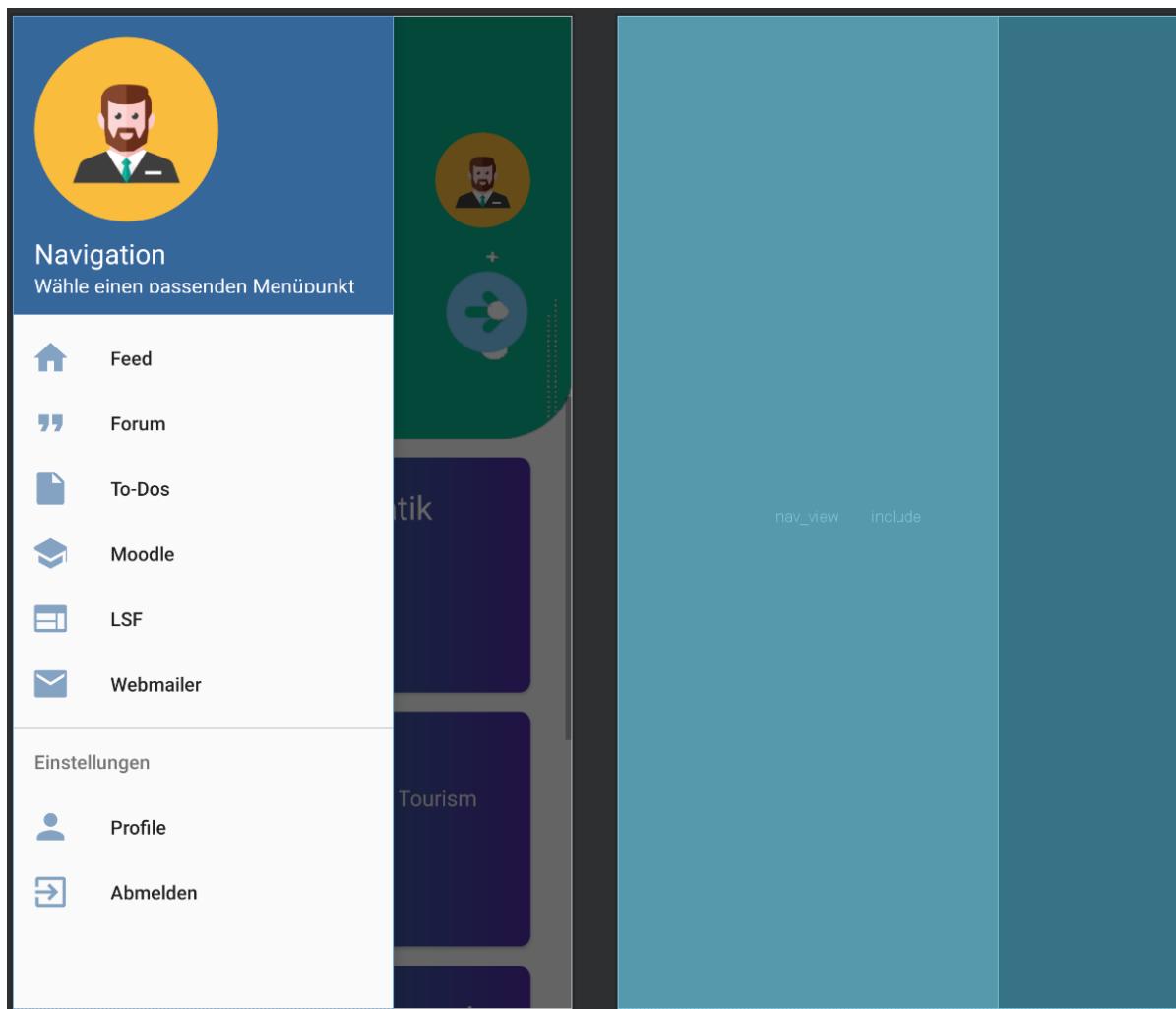
        userImage = findViewById(R.id.userImage)
        nameUser = findViewById(R.id.nameUser)

        informatik.setOnClickListener { it: View!
            startActivity(Intent(applicationContext , FragenActivity::class.java))
        }
    }
}
```

Zu Beginn werden standardmäßig alle üblichen Variablen, etwa für die Pfeilnavigation, das Benutzerbild, den Zugriff auf Firestore und den Cloud Speicher als *lateinit*s

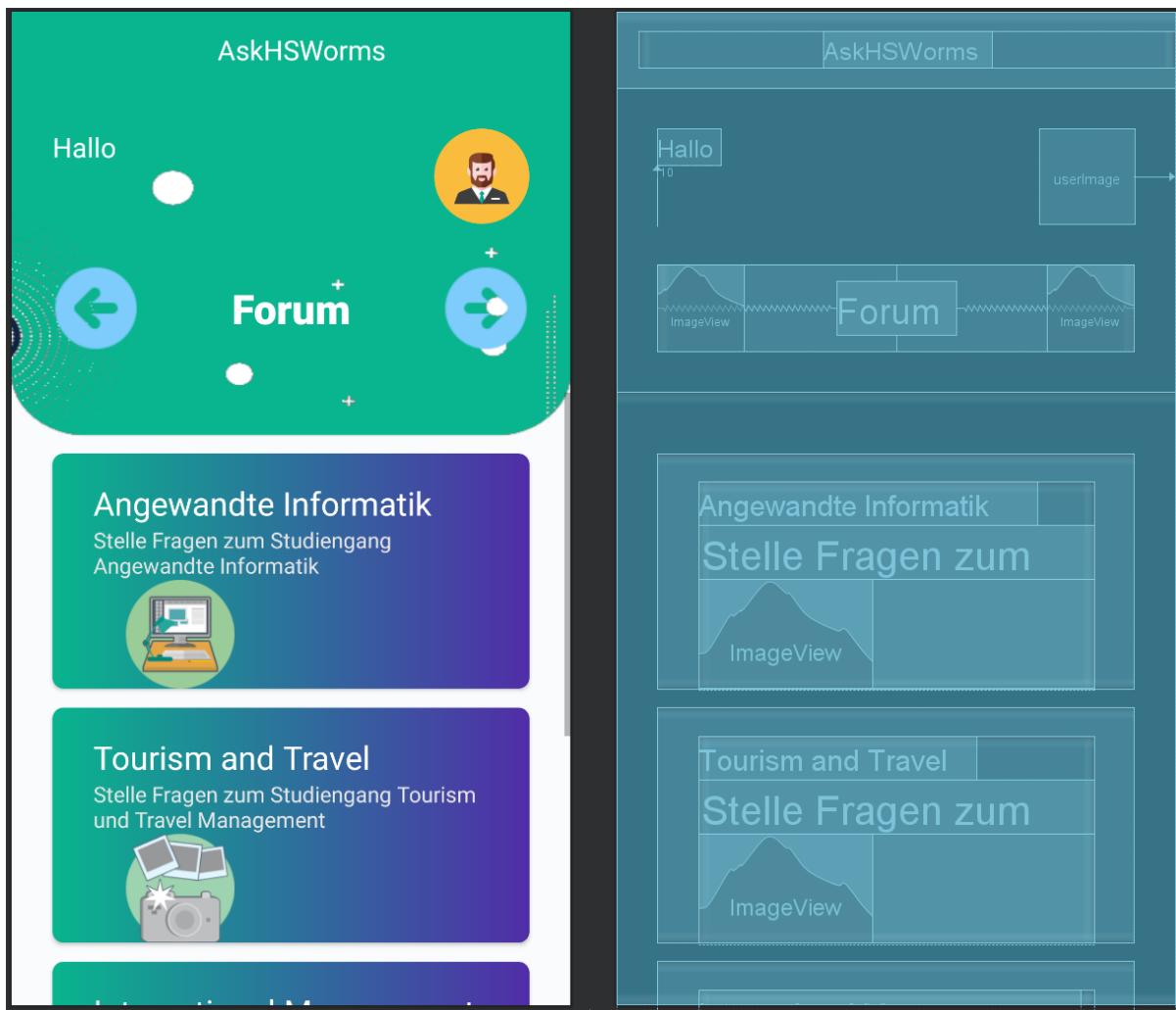
deklariert. Im Anschluss werden diese entweder initialisiert oder mit ihren Views verbunden. Das für die Übersicht verwendete Layout geht aus der Datei *activity\_forum.xml* hervor. Das Label mit der enthaltenen *Text*- und *ImageView* für den Studiengang Informatik wird als derzeit einzig aktives Forum mit einem Click Listener registriert. Dieser startet die Aktivität *FragenActivity* mit dem jeweiligen Fragenkatalog des ausgewählten Studiengangs oder Themas. Die übrigen Codezeilen in der Datei *ForumActivity.kt* dienen wie üblich den Funktionalitäten zum Anzeigen des Benutzernamens und Benutzerbildes, der Pfeilnavigation und dem Kontextmenü.

## ACTIVITY\_FORUM.XML



Zu sehen ist die Layout Datei für die Aktivität `ForumActivity`. Das eigentliche Layout `fragment_forum.xml` wird im XML-Quellcode eingebunden und an dieser Stelle von der Navigationsleiste im geöffneten Zustand überdeckt.

## FRAGMENT\_FORUM.XML



Abgebildet ist das Haupt-Layout für die Foren-Aktivität, welche von *activity\_forum.xml* eingebunden wird. Zu erkennen sind die einzelnen Forenabschnitte und jeweils eine zugehörige Beschreibung für den Abschnitt. Die Einträge befinden sich dabei in einer scrollbaren Liste und leiten die Benutzer auf ihren individuellen Fragenkatalog weiter. Die übrigen erkennbaren Elemente sind standardmäßig implementiert.

## FRAGENACTIVITY.KT

Im Kontrast zu *ActivityForum* ist hier nicht die Übersicht aller Foren, sondern die Einzelansicht jedes Forums mit seinem individuell von den Benutzern zusammengestellten Fragenkatalog im Mittelpunkt. Benutzer dürfen hier neue Fragen stellen oder bereits bestehende Fragen beantworten.

### Der Quellcode:

```
class FragenActivity : AppCompatActivity() {

    private lateinit var auth: FirebaseAuth
    var mRef: DatabaseReference? = null
    var mFragenList: ArrayList<Frage>? = null

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_fragen)

        auth = Firebase.auth
        val database = FirebaseDatabase.getInstance()
        mRef = database.getReference( path: "fragen")
        mFragenList = ArrayList()

        add_new_frage.setOnClickListener { it: View!
            showDialogAddFrage()
        }

        rjaa.setOnClickListener { it: View!
            finish()
        }
    }
}
```

Für die Authentifizierung mit Firebase und dem Abspeichern der Fragen in der Datenbank über eine Referenz darauf, wird jeweils eine Variable reserviert. Zusätzlich wird eine *ArrayList* deklariert, welche im späteren Verlauf die Fragen der Benutzer als Objekte der Klasse *Frage* zwischenspeichern soll.

Bei der Initialisierung der Aktivität werden die genannten Variablen initialisiert und eine Referenz auf die Sektion „fragen“ in der Datenbank geschaltet. Außerdem wird ein

Click Listener für die *ImageView add\_new\_frage* registriert, welche die Methode *showDialogAddFrage()* zum Hinzufügen einer Frage auslöst.

```
private fun showDialogAddFrage() {  
  
    val alertBuilder = AlertDialog.Builder( context: this)  
    val view = layoutInflater.inflate(R.layout.add_frage, root: null)  
    alertBuilder.setView(view)  
    val alertDialog = alertBuilder.create()  
    alertDialog.show()  
}
```

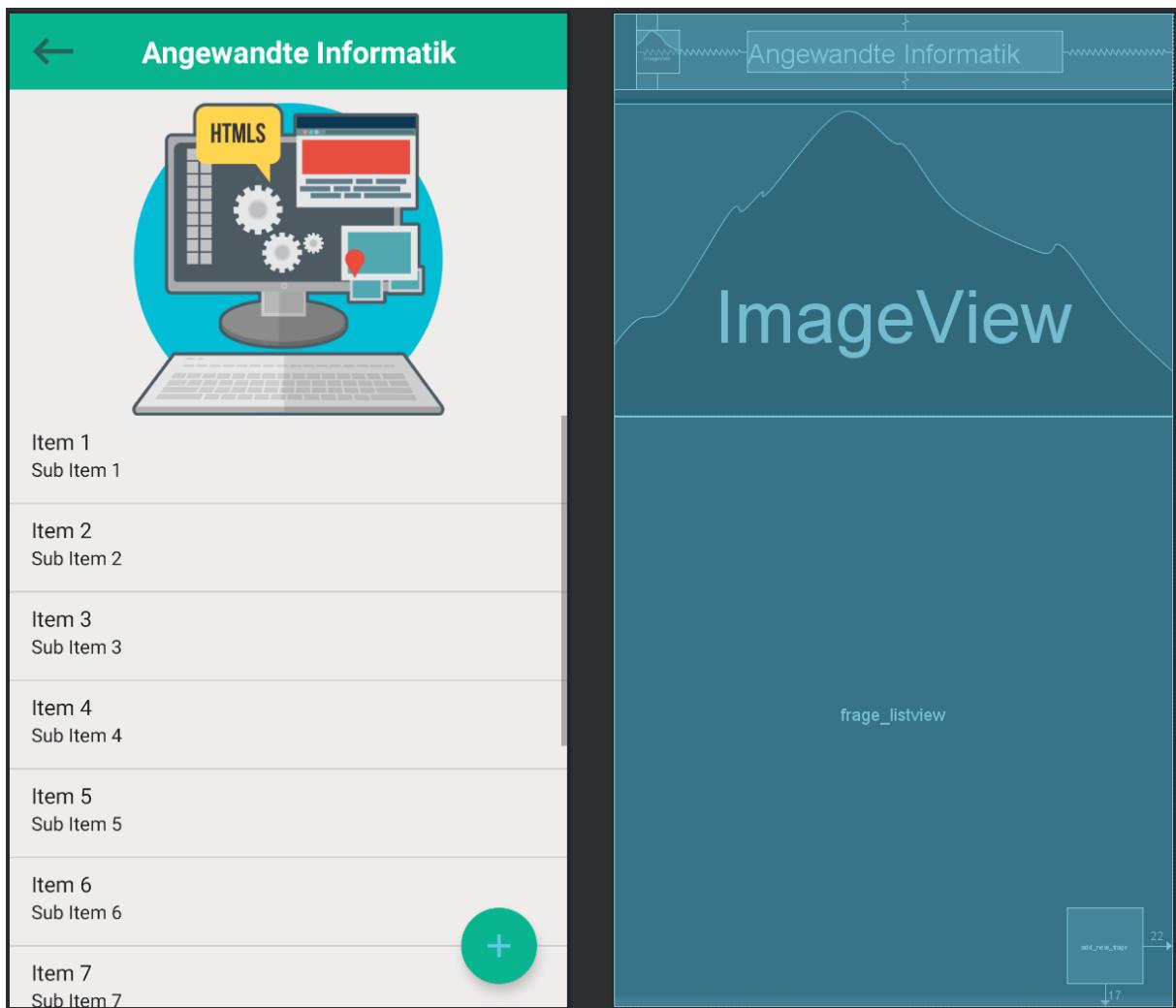
Die Methode beginnt damit einen Dialog mit Hilfe der Klasse *AlertDialog* anzulegen. Für das Layout des Dialoges wird die Datei *add\_frage.xml* verwendet. Nach der Erzeugung des Dialoges mit der Methode *create()* wird der dieser daraufhin über die Methode *show()* eingeblendet.

```
view.btnSaveFrage.setOnClickListener { it: View!  
    val title = view.fragentitel_input.text.toString()  
    val frage = view.fragencotenu_input.text.toString()  
  
    if (title.isNotEmpty() && frage.isNotEmpty()) {  
        val db = FirebaseFirestore.getInstance()  
        val itemfrage: MutableMap<String , Any> = HashMap()  
  
        itemfrage["title"] = title  
        itemfrage["contenu"] = frage  
        itemfrage["studienfand"] = "informatik"  
        itemfrage["time"] = getCurrentDate()  
        itemfrage["id_user"] = auth.currentUser.uid  
  
        db.collection( collectionPath: "fragen")  
            .add(itemfrage)  
            .addOnSuccessListener { it: DocumentReference!  
                alertDialog.dismiss()  
                Toast.makeText( context: this , text: "Frage hinzugefügt" , Toast.LENGTH_LONG).show()  
            }  
            .addOnFailureListener { it: Exception  
                Toast.makeText( context: this , text: "Fehler beim Hinzufügen!" , Toast.LENGTH_LONG).show()  
            }  
  
    } else {  
        Toast.makeText(  
            context: this , text: "Fülle bitte zunächst beide Felder aus!" ,  
            Toast.LENGTH_LONG  
        ).show()  
    }  
}
```

Zur Speicherung der Frage und seinem Titel dient der Button „Stellen“ am unteren Dialogrand. Dieser verfügt über einen Click Listener und überprüft als erstes, ob die

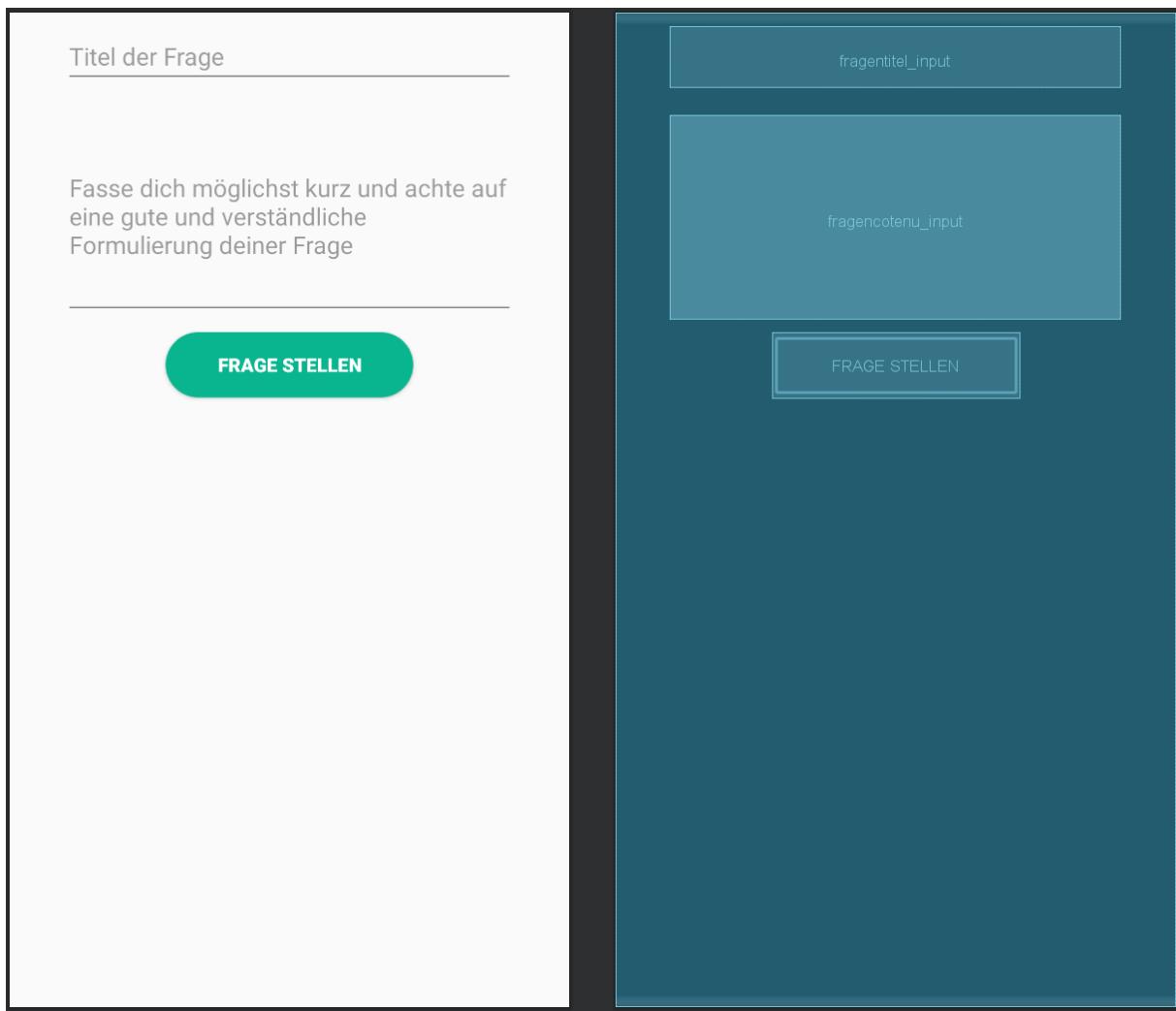
beiden Textfelder ausgefüllt wurden. Sofern dies der Fall ist, wird eine Instanz für die Datenbank geholt und die geschriebenen Inhalte in eine *HashMap* geschrieben. Darüber hinaus wird der Studiengang, ein Zeitstempel für den Zeitpunkt der Fragestellung und die UID des Fragestellers darin gespeichert. Die Frage wird nun über die Referenz auf die Datenbank mit der Methode *add()* hinzugefügt. Der darauf horchende *onSuccessListener* überprüft, ob die Frage erfolgreich gespeichert wurde. Bei Erfolg wird der Dialog beendet und eine Erfolgsmeldung als Toast gezeigt. Bei einem Fehler, wird alleinig eine Fehlermeldung als Toast eingeblendet.

## ACTIVITY\_FRAGEN.XML



Die Aktivität für den Fragenkatalog enthält die Bezeichnung des Forenabschnitts als Titel in der Toolbar und eine *ImageView* zur Rückwärtsnavigation in die Forenübersicht. Die Benutzer fügen über die *ImageView* mit dem „+“ Symbol Fragen hinzu, womit sie dann wie dargestellt auf dem Bildschirm angezeigt werden. Die einzelnen Listeneinträge sind zunächst Platzhalter für die von den Benutzern hinzugefügten Fragen.

## ADD\_FRAGE.XML



Abgebildet ist der Dialog für das Hinzufügen einer Frage über die „+“ ImageView in *activity\_fragen.xml*. Nachdem die Benutzer einen Titel und eine Fragestellung formuliert haben, veröffentlichen sie ihre Frage über den „Frage stellen“ Button.

## PROFILEACTIVITY.KT

Einstellungen am eigenen Profil können in der Aktivität ProfileActivity vorgenommen werden. Dort lassen sich der Name, das Passwort, die E-Mail, die Telefonnummer und das Benutzerbild verändern. Darüber hinaus bekommen die Benutzer eine Übersicht zu der Anzahl an gestellten Fragen und Antworten. Dieses Feature wurde jedoch nicht implementiert und zeigt nur was im späteren Verlauf noch möglich wäre. Die Aktivität für die Profileinstellungen ist einzig über das Kontextmenü unter dem vorletzten Eintrag „Profile“ erreichbar.

### Der Quellcode:

```
class ProfileActivity : AppCompatActivity() {

    private lateinit var _userName: String
    private lateinit var _userEmail: String
    private lateinit var _userTel: String
    private lateinit var _userPassword: String
    private lateinit var _userPower: String
    private lateinit var _userBild: String

    lateinit var user: FirebaseUser
    lateinit var reference: DatabaseReference
    lateinit var userID: String
    lateinit var bild_change: CircleImageView

    private val storageInstance: FirebaseStorage by lazy {
        FirebaseStorage.getInstance()
    }

    private val currentUserStorage: StorageReference
        get() = storageInstance.reference.child(FirebaseAuth.getInstance().currentUser.uid)

    private val firestoreInstance: FirebaseFirestore by lazy {
        FirebaseFirestore.getInstance()
    }

    private val currentUserDocRef: DocumentReference
        get() = firestoreInstance.document( documentPath: "users/${FirebaseAuth.getInstance().currentUser.uid}" )
```

Zunächst werden alle Variablen als *lateinit* deklariert, welche hinterher die Benutzerdaten zwischenspeichern und auf dem Bildschirm darstellen. Anschließend folgen die notwendigen Variablen für Firebase beziehungsweise alles was mit der Datenbank zusammenhängt. Dazu gehören Variablen für eine Firebase

Benutzerinstanz, für eine Datenbankreferenz und eine Referenz auf den Cloud Speicher. Diese sind ebenfalls als *lateinit* oder *lazy init* deklariert. All das dient dazu, um die textuellen Daten des Benutzers und das Profilbild abzufragen.

```
override fun onCreate(savedInstanceState: Bundle?) {  
  
    super.onCreate(savedInstanceState)  
    setContentView(com.issam.example.R.layout.activity_profile)  
  
    user = FirebaseAuth.getInstance().currentUser  
    reference = FirebaseDatabase.getInstance().getReference( path: "Users")  
    userID = user.uid  
  
    Toast.makeText(baseContext , userID , Toast.LENGTH_LONG).show()  
  
    lateinit var input_name: EditText  
    lateinit var input_email: EditText  
    lateinit var input_tel: EditText  
    lateinit var input_password: EditText  
    lateinit var btn_update: Button  
  
    bild_change = findViewById(com.issam.example.R.id.bild_change)  
    input_name = findViewById(com.issam.example.R.id.input_name)  
    input_email = findViewById(com.issam.example.R.id.input_email)  
    input_tel = findViewById(com.issam.example.R.id.input_tel)  
    input_password = findViewById(com.issam.example.R.id.input_password)  
    btn_update = findViewById(com.issam.example.R.id.btn_update)
```

Zu Beginn der Initialisierung wird das korrespondierende Layout *activity\_profile.xml* eingebettet. Für einen Zugriff auf die benutzerspezifischen Daten, wird der Firebase Benutzer mit dem aktuell angemeldeten Benutzer initialisiert und eine Referenz für die Sektion der Benutzer in der Datenbank geholt. Über die Benutzerinstanz wird die Benutzer-UID in der Variable *userID* zwischengespeichert und als Toast eingeblendet. Für die editierbaren Textfelder zur Darstellung und Veränderung der Benutzerdaten werden Variablen deklariert und direkt im Anschluss mit ihren Views zusammengebunden. Damit können Änderungen der Textfelder in den Variablen zwischengespeichert und später an die Datenbank als Updates übermittelt werden. Selbes gilt für den Button zum Aktualisieren der Daten.

```

getUserInfo { user ->
    userName = user.fullname
    userEmail = user.email
    userTel = user.tel
    userPassword = user.password
    userPower = user.userPower
    userBild = user.userPower

    if (user.profilBild.isNotEmpty()) {
        GlideApp.with( activity: this@ProfileActivity)
            .load(storageInstance.getReference(user.profilBild))
            .placeholder(R.drawable.picture_frame)
            .into(bild_change)
    }
}

```

Bereits zu Beginn werden die Textfelder mit den aktuellen Daten der Benutzer vorgefüllt. Das Füllen der Textfelder mit den Benutzerdaten geschieht über die Firebase Benutzerinstanz `user`. Falls der Benutzer ein Profilbild besitzt, wird das Bild mit Hilfe der Klasse `MyAppGlide` über die Storage Instanz aus dem Cloud Speicher geladen und in die bereits existierende `ImageView` eingefügt.

```

val ordersRef = reference.child(userID)

val valueEventListener = object : ValueEventListener {
    override fun onDataChange(dataSnapshot: DataSnapshot) {

        val fullname = dataSnapshot.child( path: "fullname").getValue(String::class.java)
        val email = dataSnapshot.child( path: "email").getValue(String::class.java)
        val tel = dataSnapshot.child( path: "tel").getValue(String::class.java)
        val password = dataSnapshot.child( path: "password").getValue(String::class.java)

        input_name.setText(fullname)
        input_email.setText(email)
        input_tel.setText(tel)
        input_password.setText(password)
        full_name.text = fullname
    }

    override fun onCancelled(databaseError: DatabaseError) {}
}

ordersRef.addValueEventListener(valueEventListener)

```

Ein *valueEventListener* registriert eine Veränderung der Benutzerdaten in Firebase über eine benutzerspezifische Referenz und fügt die aktualisierten Daten ohne einen Neustart der Aktivität in die Textfelder ein.

```
btn_update.setOnClickListener { it: View!  
    val input_email_new: Editable? = input_email.text  
    val input_name_new: Editable? = input_name.text  
    val input_password_new: Editable? = input_password.text  
    val input_tel_new: Editable? = input_tel.text  
  
    var afterUpdate = User(  
        input_name_new.toString() , input_email_new.toString() , input_tel_new.toString() ,  
        input_password_new.toString() , userPower: "" , profilBild: ""  
    )  
  
    ordersRef?.setValue(afterUpdate)  
    Toast.makeText(baseContext , text: "Profildaten aktualisiert" , Toast.LENGTH_LONG).show()  
}
```

Auf den Button zum Aktualisieren der Daten wird ein Click Listener geschaltet, welcher die gesetzten Daten zunächst in Variablen zwischenspeichert und in ein Benutzer-Objekt umwandelt. Daraufhin wird das Benutzer-Objekt als neuer Datensatz in die Datenbank übertragen und überschreibt den bereits existierenden Datensatz. Zuletzt wird eine Erfolgsmeldung als Toast eingeblendet.

```
bild_change.setOnClickListener { it: View!  
    val intentImage = Intent().apply { this: Intent  
        type = "image/*"  
        action = Intent.ACTION_GET_CONTENT  
        putExtra(Intent.EXTRA_MIME_TYPES , arrayOf("image/jpeg" , "image/png"))  
    }  
  
    startActivityForResult(  
        Intent.createChooser(  
            intentImage ,  
            title: "Wähle ein Profilbild aus"  
        ) , requestCode: 2  
    )  
}
```

Das Hochladen eines eigenen Profilbildes geschieht über den internen Smartphone-Speicher. Dabei werden als erstes die zum Hochladen erlaubten Dateitypen PNG und JPEG spezifiziert und anschließend der Datei-Explorer als eigene Aktivität gestartet. Ein Tippen auf die *CircleImageView* über der Aufschrift „Bild ändern“ löst die beschriebene Funktionalität aus.

```

override fun onActivityResult(requestCode: Int , resultCode: Int , data: Intent?) {

    super.onActivityResult(requestCode , resultCode , data)

    if (requestCode == 2 && resultCode == Activity.RESULT_OK && data != null && data.data != null) {
        bild_change.setImageURI(data.data)

        val selectedImagePath = data.data
        val selectedImageBmp =
            MediaStore.Images.Media.getBitmap(this.contentResolver , selectedImagePath)
        val outputStream = ByteArrayOutputStream()

        selectedImageBmp.compress(Bitmap.CompressFormat.JPEG , quality: 20 , outputStream)

        val selectedImageBytes = outputStream.toByteArray()
        uploadProfileBild(selectedImageBytes) { path ->
            val userFiledMap = mutableMapOf<String , Any>()
            userFiledMap["email"] = userEmail
            userFiledMap["fullname"] = userName
            userFiledMap["password"] = userPassword
            userFiledMap["tel"] = userTel
            userFiledMap["userPower"] = userPower
            userFiledMap["profilBild"] = path
            currentUserDocRef.update(userFiledMap)
        }
    }
}

```

Sofern ein gültiges Bild ausgewählt wurde und die Operation fehlerfrei verlaufen ist, erhält die *CircleImageView* den Dateipfad des ausgewählten Bildes. Weiterführend ist es wichtig das Bild in der Datenbank abzuspeichern. Im echten Einsatz wäre unsere Anwendung darauf ausgelegt viele Benutzer zu haben, daher werden die Bilder vor der Übertragung in ein komprimiertes Byte-Array umgewandelt, um möglichst wenig Speicherplatz zu beanspruchen. Die eigentliche Übertragung wird von der Methode *uploadProfileBild()* durchgeführt, welche das komprimierte Byte-Array als Parameter übergeben bekommt.

```

private fun uploadProfileBild(
    selectedImageBytes: ByteArray ,
    onSuccess: (imagePath: String) -> Unit
) {

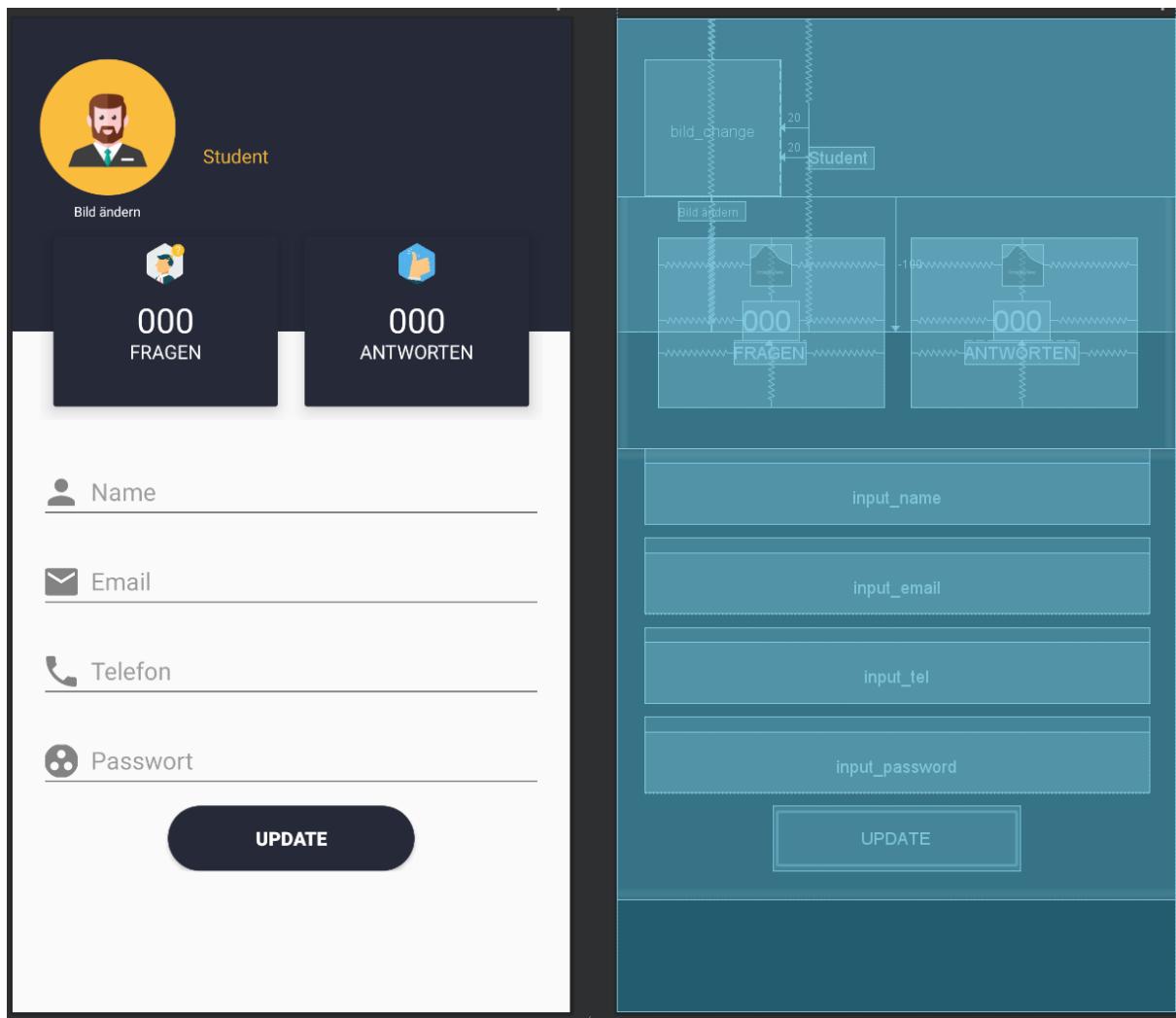
    val ref =
        currentUserStorage.child( pathString: "ProfileBild/${UUID.randomUUIDFromBytes(selectedImageBytes)}")

    ref.putBytes(selectedImageBytes).addOnCompleteListener { it: Task<UploadTask.TaskSnapshot>
        if (it.isSuccessful) {
            onSuccess(ref.path)
        } else {
            Toast.makeText(
                context: this , text: "Error: ${UUID.randomUUIDFromBytes(selectedImageBytes)} " ,
                Toast.LENGTH_LONG
            ).show()
        }
    }
}

```

Wieder ist eine Referenz auf den Cloud Speicher maßgeblich, um den Speicherort des Bildes festzulegen. Dafür dient die Sektion „ProfileBild“. Hinzukommend wird eine UUID aus der Bytekette des Bildes generiert, um eine nachträgliche Identifizierung anhand dieser zu ermöglichen. Die Bytekette wird durch die Methode `putBytes()` am spezifizierten Ort abgelegt. Ein darauf agierender `onCompleteListener` überprüft zuletzt noch den Erfolg der Operation und gibt ggf. eine Fehlermeldung mit der UUID des Bildes als Toast aus.

## ACTIVITY\_PROFILE.XML



Gezeigt ist das Layout für die Profileinstellungen. Zu erkennen sind unter anderem mehrere Textfelder zum Anzeigen und Eintragen der Benutzerdaten, sowie ein „Update“-Button zum Speichern dieser. Im Kopfbereich befinden sich außerdem der Name des Benutzers, seine Benutzerrolle und falls vorhanden ein Profilbild direkt über der *TextView* mit der Aufschrift „Bild ändern“. Eine Statistik für die Anzahl an gestellten beziehungsweise beantworteten Fragen, wäre sofern implementiert darunter in den beiden Badges zu sehen.

## CODE-QUALITÄT – BEDEUTSAME NAMEN AM BEISPIEL VON MAINACTIVITY.KT

Speziell für die Klasse *MainActivity*, welche das Onboarding implementiert, wurde auf eine bedeutsame Namensgebung der Variablen, Methoden und verwendeten Klassen geachtet. Eine sinnvolle Benennung ist vor allem dann wichtig, wenn es sich um ein größeres Projekt handelt, an dem mehrere Personen beteiligt sind. Ein Beispiel dafür wäre etwa ein Open-Source-Projekte, an dem häufig viele verschiedene Menschen aus gänzlich verschiedenen Ländern beteiligt sind. Die Code-Lesbarkeit wird um ein Vielfaches gesteigert und insgesamt ist ein produktiveres Arbeiten möglich. Zusätzlich wird es Personen von außerhalb erleichtert einem Projekt beizutreten, da bei einer guten Benennung Muster und Zusammenhänge einfacher zu erkennen sind. Im Folgenden werden Beispiele für eine gute Namensgebung für den Onboarding-Mechanismus der Klasse *MainActivity* aufgezeigt.

### GUTE VARIABLENNAMEN

```
var nextView: TextView? = null  
var currentTab = 0  
var sharedPreferencesRef: SharedPreferences? = null
```

**nextView.** Der Variablenname deutet darauf hin, dass es sich um eine klickbare View mit der Aufschrift „next“ handelt. Diese befindet sich auf jedem Tab am unteren rechten Rand.

**currentTab.** Der Name deutet auf eine aktuelle Position im *TabLayout* hin und meint den aktuell eingeblendeten Hinweis beim Onboarding.

**sharedPreferencesRef.** Die Variable nimmt Bezug auf die *Shared Preferences*, welcher ein für Android-Entwickler geläufiger Begriff ist. Es ist anzunehmen, dass die Variable im späteren Verlauf dazu genutzt wird Änderungen an den *Shared Preferences* vorzunehmen.

```
val onboardignList: MutableList<Onboarding> = ArrayList()
```

**onboardingList.** Der Name verdeutlicht, dass es sich bei der Variablen um ein Listenobjekt handelt, welches im Zusammenhang mit dem Onboarding steht. Das Onboarding zeigt den Benutzern Hinweise zur App. Darum ist anzunehmen, dass die Liste alle eingeblendeten Hinweise enthält.

```
val lastTab = onboardignList.size - 1
```

**lastTab.** Der Name ist ein Indikator für den letzten Tab im Onboarding. Ein Entwickler könnte annehmen, dass die Variable definiert wurde, um ein abweichendes Verhalten zu allen vorherigen Tabs zu definieren und/oder dass sie als Begrenzer für eine Schleife oder ähnliches verwendet wird.

```
val sharedpreferencesEditor: SharedPreferences.Editor = sharedpreferencesRef!!.edit()
```

**sharedPreferencesEditor.** Der Variablenname deutet darauf hin, dass der Editor für die *Shared Preferences* zur Verwendung geöffnet wird und sie vermutlich eine Referenz darauf hält. Ein Programmierer würde vermuten, dass über diese Referenz Veränderungen an den *Shared Preferences* bewirkt werden.

```
(var tabTitle: String , var tabDescription: String , var tabImgURL: Int)
```

MODEL/ONBOARDING.KT

**tabTitle. tabDescription. tabImgURL.** Sind die Parameter des Konstruktors der Datenklasse *Onboarding* und stehen in der Reihenfolge für

- Den Titel des jeweiligen Onboarding Tabs
- Den jeweilig eingeblendeten Text pro Tab
- Und einem Bild, welches sich pro Tab unterscheidet

In diesem Fall ist die Bedeutung der Variablen sehr intuitiv und klar.

## GUTE METHODENNAMEN

```
private fun setOnboardingViewPagerAdapter(onboardingList: List<Onboarding>) {
```

**setOnboardingViewPagerAdapter()**. Ein Programmierer erfährt anhand des Präfixes „set“, dass es sich bei der Methode um eine Setter-Methode handelt. Aus dem Namen lässt sich ableiten, dass die Methode dazu genutzt wird, um die Tabs samt ihren Inhalten erstmalig mit Hilfe des OnboardingViewPagerAdapter zu initialisieren.

```
private fun saveInSharedPreferences() {
```

**saveInSharedPreferences()**. Der Methodenname deutet darauf hin, dass Änderungen an den *Shared Preferences* gespeichert werden sollen. Was anhand des Namens nicht ersichtlich ist, ist dass lediglich das gesetzte Flag *isOnboardingCompleted* verändert und gespeichert wird.

```
private fun readFromSharedPreferences(): Boolean {
```

**readFromSharedPreferences()**. Hier liegt der umgekehrte Fall vor, dass aus den *Shared Preferences* gelesen wird. Wie im vorherigen Fall ist hier nicht absehbar, dass lediglich das Flag *isOnboardingCompleted* gelesen wird.

```
tabLayoutRef!!.addOnTabSelectedListener(object : TabLayout.OnTabSelectedListener {
```

```
    override fun onTabUnselected(tab: TabLayout.Tab?) {}
```

```
    override fun onTabReselected(tab: TabLayout.Tab?) {}
```

```
        onboardingList.add(
```

Die vorliegenden Methoden stehen bereits nativ zur Verfügung oder werden von den eingebetteten Bibliotheken zur Verfügung gestellt. Sie stellen ebenso Beispiele für eine gute Namensgebung dar.

**`addOnTabSelectedListener()`**. Das Wort Listener ist ein Begriff, welcher Programmierern geläufig ist. Der Name deutet darauf hin, dass Veränderungen am damit verknüpften Objekt beobachtet werden und bei einer bestimmten Aktion des Objekts eine vom Programmierer definierte Reaktion ausgelöst wird. In diesem Fall wird der nach der geschweiften Klammer befindliche Codeabschnitt (= Reaktion) erst ausgeführt, wenn ein Tab aktiv wird (= Aktion).

**`onTabUnselected()/onTabReselected()`**. Analog dazu ließen sich an der Stelle von der Standardimplementierung abweichende Reaktionen für ein Verlassen eines Tabs beziehungsweise der Wiederauswahl eines bereits besuchten Tabs definieren. Die Methodennamen repräsentieren ihre Aktion passend und sind gut gewählt.

## GUTE KLASSENNAMEN

```
data class Onboarding(var tabTitle: String , var tabDescription: String , var tabImgURL: Int)
```

MODEL/ONBOARDING.KT

**Onboarding.** Der Klassename Onboarding verrät auf den ersten Blick nur, dass die Klasse im Zusammenhang mit dem Onboarding-Mechanismus der App steht, jedoch nicht welche Rolle die Klasse einnimmt. Wenn die Klasse allerdings im Kontext betrachtet wird, ergibt sich ihre Bedeutung. Denn bei der Klasse handelt es sich um eine Datenklasse im Ordner *model*, welcher wiederum nur weitere Datenklassen enthält. Daraus und aus ihren Parametern ergibt sich, dass die Klasse als Datenobjekt für die Inhalte des Onboardings dient. Aus diesem Grund ist die Namenswahl für die Klasse an der Stelle gerechtfertigt.

```
class OnboardingViewPagerAdapter(  
    private var context: Context ,  
    private var onBoardingList: List<Onboarding>  
) : PagerAdapter() {
```

ADAPTER/ONBOARDINGVIEWPAGERADAPTER.KT

**OnboardingViewPagerAdapter.** Es ist erkennbar, dass es sich um eine Hilfsklasse für den *ViewPager-Mechanismus* in Android handelt, mit Hilfe dessen der Benutzer beim Onboarding zwischen den Seiten (= Tabs) nach links oder rechts wechseln kann. Die Bedeutung wäre in diesem Fall für einen Android-Entwickler einfacher zu verstehen als für eine Person, welche den Begriff *ViewPager* nicht kennt.