Content-Verwertungsmodelle und ihre Umsetzung in mobilen Systemen

AWS Amplify Tutorial

PD Dr.-Ing. habil. Jürgen Nützel
David M. Müller

TU Ilmenau

In dieser Anleitung werden einige Funktionen von AWS Amplify anhand einer Beispielapp gezeigt. AWS Amplify ist eine JavaScript-Bibliothek, die Entwicklern das Erstellen von Full-Stack-Applications ermöglicht. Für die Entwicklung der Applikation wird Flutter verwendet. Flutter ist ein Open-Source-UI Entwicklungskit zur Erstellung von Cross-Platform Apps in der Programmiersprache Dart.

Um die Funktionen von AWS Amplify gut veranschaulichen zu können, ist die Beispielapp recht simpel aufgebaut, enthält dabei aber die wichtigsten von AWS Amplify zur Verfügung gestellten Funktionalitäten. Benutzer können sich eine Liste von Personen aus ihrem Freundes- und Familienkreis erstellen. Zu jeder einzelnen Person kann ein Profilbild, sowie Geschenkideen für den nächsten Geburtstag oder Weihnachten abgespeichert werden. Benutzer können sich außerdem registieren, um mit verschiedenen Geräten auf ihre Geschenkideen zugreifen zu können.

Diese Anleitung dient als Hilfestellung für Teilnehmende der Vorlesung Content-Verwertungsmodelle und ihre Umsetzung in mobilen Systemen, gehalten von PD Dr.-Ing. habil. Jürgen Nützel an der TU Ilmenau. Die Beispielapp dient, wie der Name schon sagt, nur als Beispiel. Von der Entwicklung von Apps mit identischen Funktionalitäten ist abzusehen.

Mit einem Klick auf orangefarbene Wörter gelangt ihr zu Downloadseiten oder zusätzlichen Informationen.

Kapitelübersicht

1. Vorraussetzungen

In diesem Kapitel schaffen wir die Vorraussetzungen um mit der Entwicklung einer App und der Nutzung von Amazon Web Services beginnen zu können.

1.1 Visual Studio Code

In diesem Tutorial wird Visual Studio Code als Editor verwendet und empfohlen. Falls ihr allerdings einen anderen Editor eurer Wahl (z.B. Android Studio) benutzen wollt, könnt ihr diesen Abschnitt überspringen.

Geht zuerst auf die offizielle Downloadseite und ladet euch die entsprechende Version von Visual Studio Code für euer System herunter.

Öffnet Visual Studio Code und klickt auf den Extensions Reiter auf der linken Seite.
Sucht in der Suchzeile nach Dart und nach Flutter und installiert jeweils die aktuelle Version der Plugins.

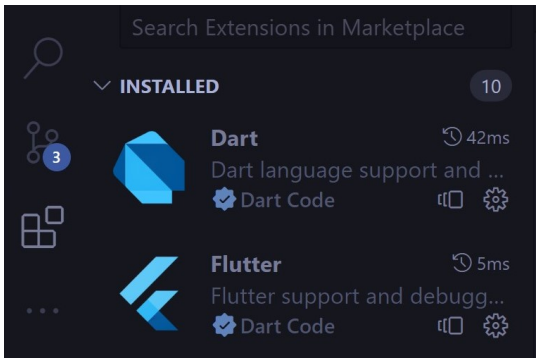


Abbildung 1.1: Flutter und Dart Plugin

1.2 Flutter

Flutter ist ein Framework zur Erstellung von Android-, iOS-, Windows- und Webapps unter Verwendung der Programmiersprache Dart.
Um Flutter zu Installieren, geht ihr auf die offizielle Installationsseite und wählt euer Betriebssystem aus.

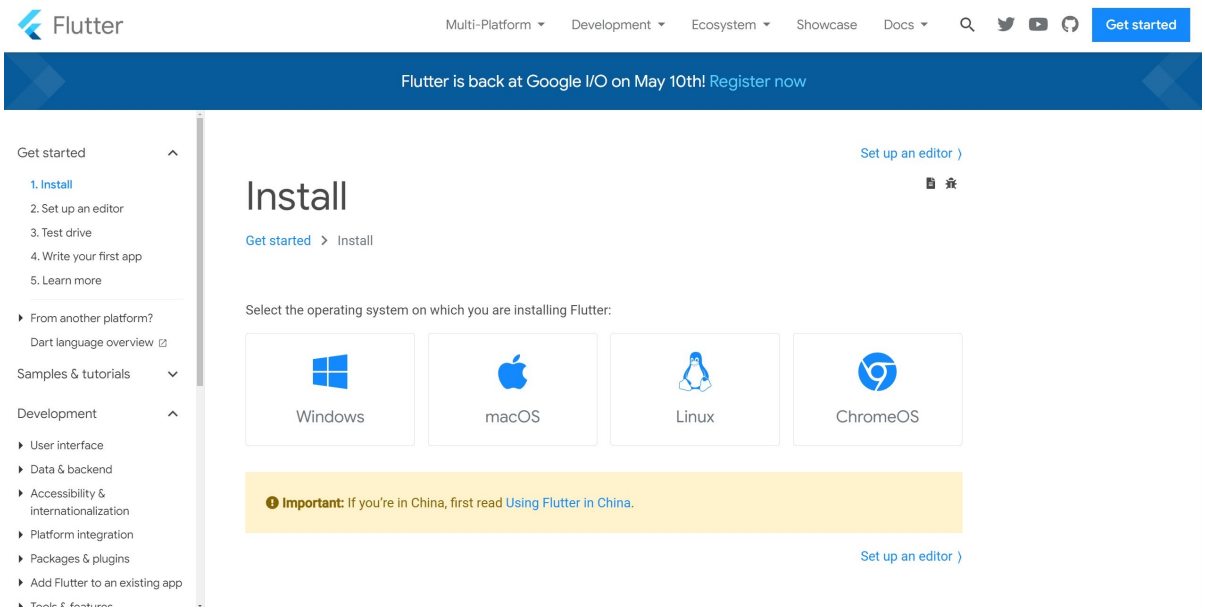


Abbildung 1.2: Installationsanleitung Flutter

Folgt den Anweisungen auf der Seite.

TIPP

Auch wenn ihr Visual Studio Code verwendet, müsst ihr euch, wie in der Installationsanleitung beschrieben, Android Studio herunterladen um Android Apps mit Flutter entwickeln zu können. Für die Entwicklung von iOS Apps, benötigt man einen Mac mit XCode. Falls eure App ebenfalls auf Windows Geräten funktionieren soll, müsst ihr euch zusätzlich Visual Studio (nicht Visual Studio Code) herunterladen.

Wie in der Installationsanleitung beschrieben, könnt ihr mithilfe des Flutter Doctors überprüfen, was euch noch fehlt um mit der Programmierung beginnen zu können. Falls euch die Android SDK Command Line Tools noch fehlen, befolgt die nächsten Schritte:

1. Öffnet Android Studio, klickt auf Customize und auf All settings...:

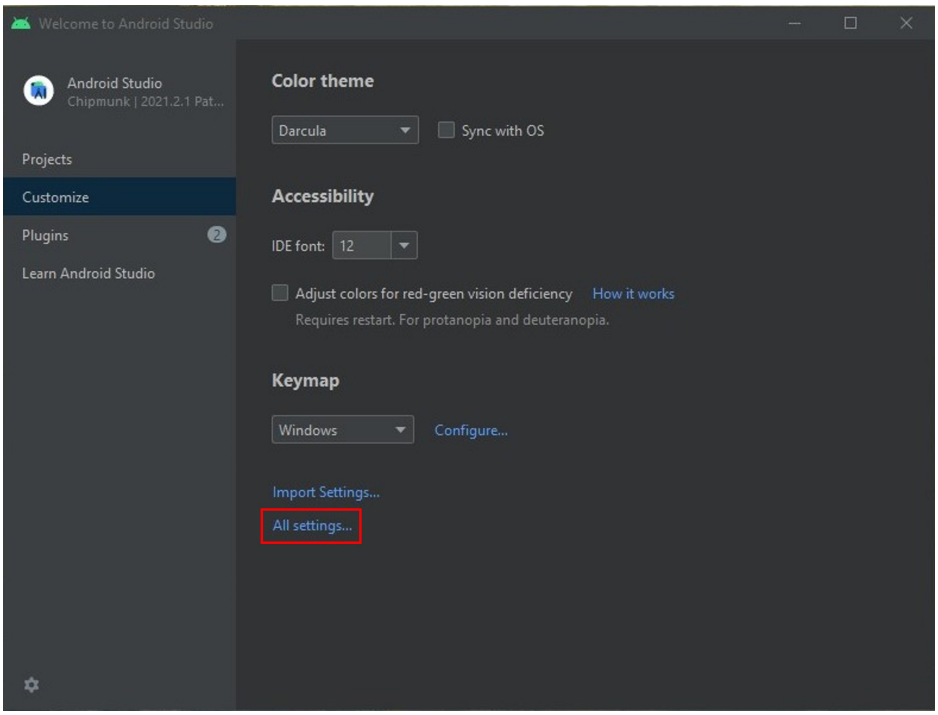


Abbildung 1.3: Android Studio

2. Geht auf den Reiter SDK Tools und setzt das Häckchen an dieser Stelle:

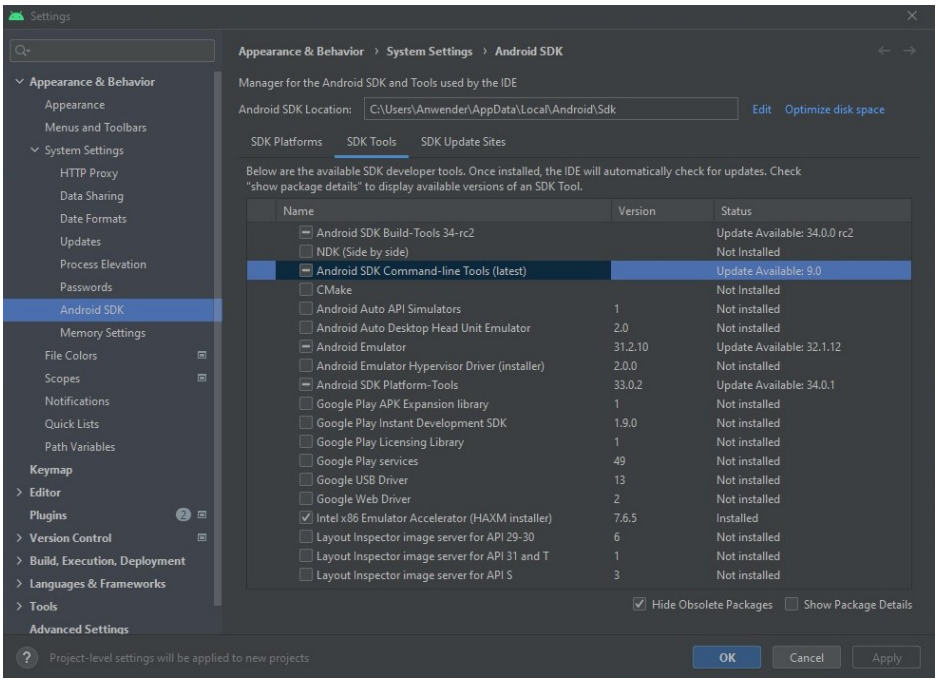


Abbildung 1.4: Android Studio

3. Klickt auf Apply um die Android SDK Command Line Tools zu installieren.

1.3 Amplify CLI

Das Amplify Command Line Interface benötigt ihr um eure App mit den Amazon Cloud Services zu verbinden. Als erstes ist es notwendig sich einen AWS Account einzurichten. Dafür benötigt ihr eine Kreditkarte und müsst einen Cent überweisen. Als neuer Kunde habt ihr 12 Monate lang die Möglichkeit, Amazon Web Services bis zu einer gewissen Grenze kostenlos auszuprobieren. Für die Erstellung einer oder mehrerer Apps im Rahmen dieser Vorlesung werdet Ihr diese Grenze auch nicht überschreiten. Es fallen für euch also keine Kosten an, abgesehen von einem Cent. Genauere Informationen zu den Kosten des Amazon Web Services könnt ihr auf der offiziellen Website nachlesen.

Als nächstes benötigt ihr Node.js. Ladet euch die empfohlene Version herunter. Achtet darauf, dass die Version npm enthält (wie zum Beispiel Version 18.15.0).
Installiert nun das Amplify Command Line Interace mit folgendem Befehl:

```
> npm install -g @aws-amplify/cli
```

Abbildung 1.5: Installation Amplify CLI

TIPP

Falls bei euch ein Eacces Permission Error erscheint versucht wie hier beschrieben vorzugehen.

2. Erste Schritte

2.1 Neues Amplify Projekt

Um ein neues Amplify Projekt zu erstellen muss zuerst ein IAM-User erstellt werden. Dieser übernimmt die Rolle der Schnittstelle zwischen unserer App und AWS. Um einen neuen IAM-User anzulegen, öffnet ein Terminal und führt den Befehl

```
> amplify configure
```

Abbildung 2.1: IAM-User erstellen

aus. Ein Browserfenster öffnet sich und es erscheint die Aufforderung, sich mit seinem AWS Account anzumelden:

```
Follow these steps to set up access to your AWS account:

Sign in to your AWS administrator account:
https://console.aws.amazon.com/
Press Enter to continue
```

Abbildung 2.2: IAM-User erstellen

Nachdem Ihr euch angemeldet habt, drückt auf Enter. Ein neues Menü erscheint:

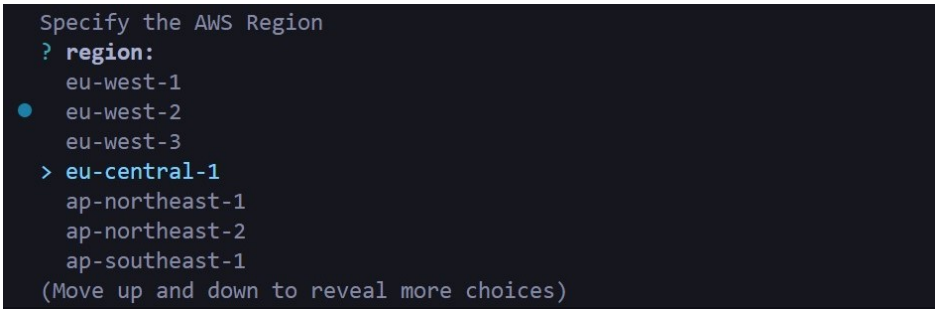


Abbildung 2.3: IAM-User erstellen

Hier wählt Ihr die Server Region aus, in welcher eure Daten gespeichert werden sollen. Als nächstes öffnet sich erneut ein Browserfenster. Hier könnt ihr den Namen für euren IAM-User festlegen. Den Namen könnt ihr frei wählen.

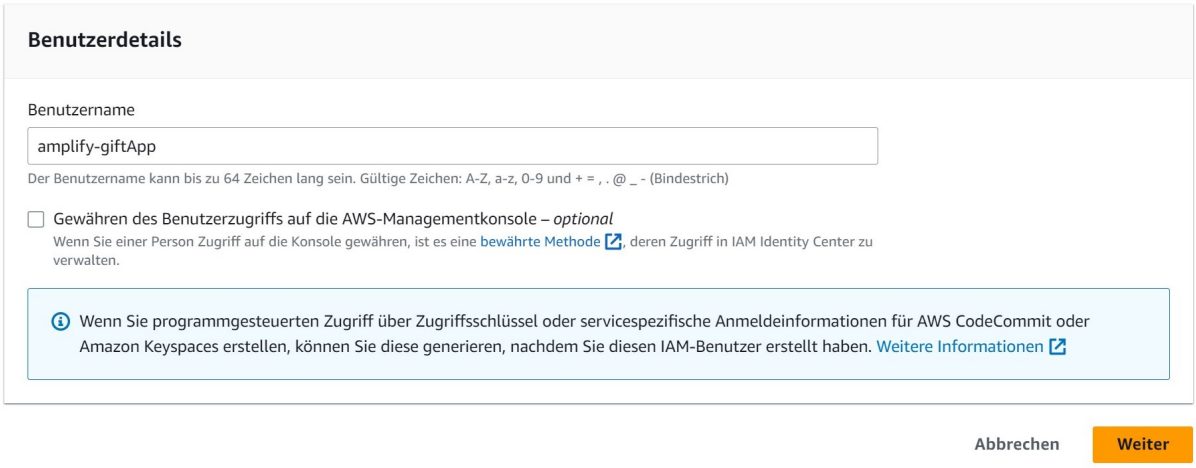


Abbildung 2.4: IAM-User erstellen

Nachdem ihr auf **Weiter** geklickt habt, wählt ihr **Direktes Anfügen von Richtlinien** und **AdministratorAccess-Amplify** aus.

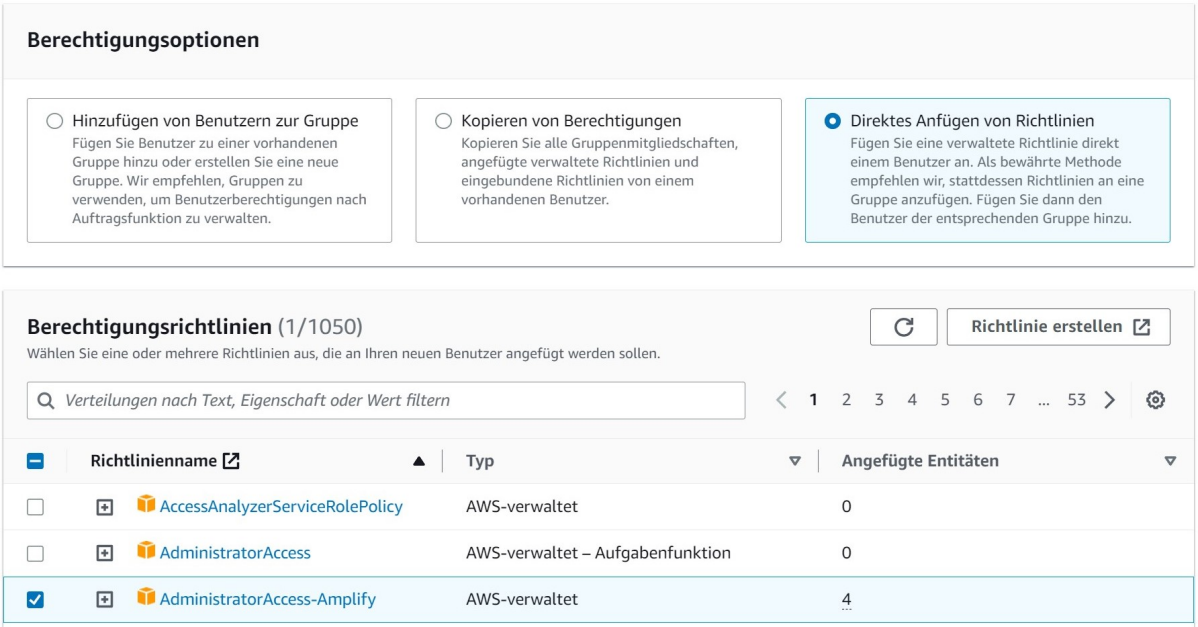


Abbildung 2.5: IAM-User erstellen

Klickt wieder auf **Weiter**. Kontrolliert auf der Übersichtsseite ob alles korrekt ist und erstellt den Benutzer.

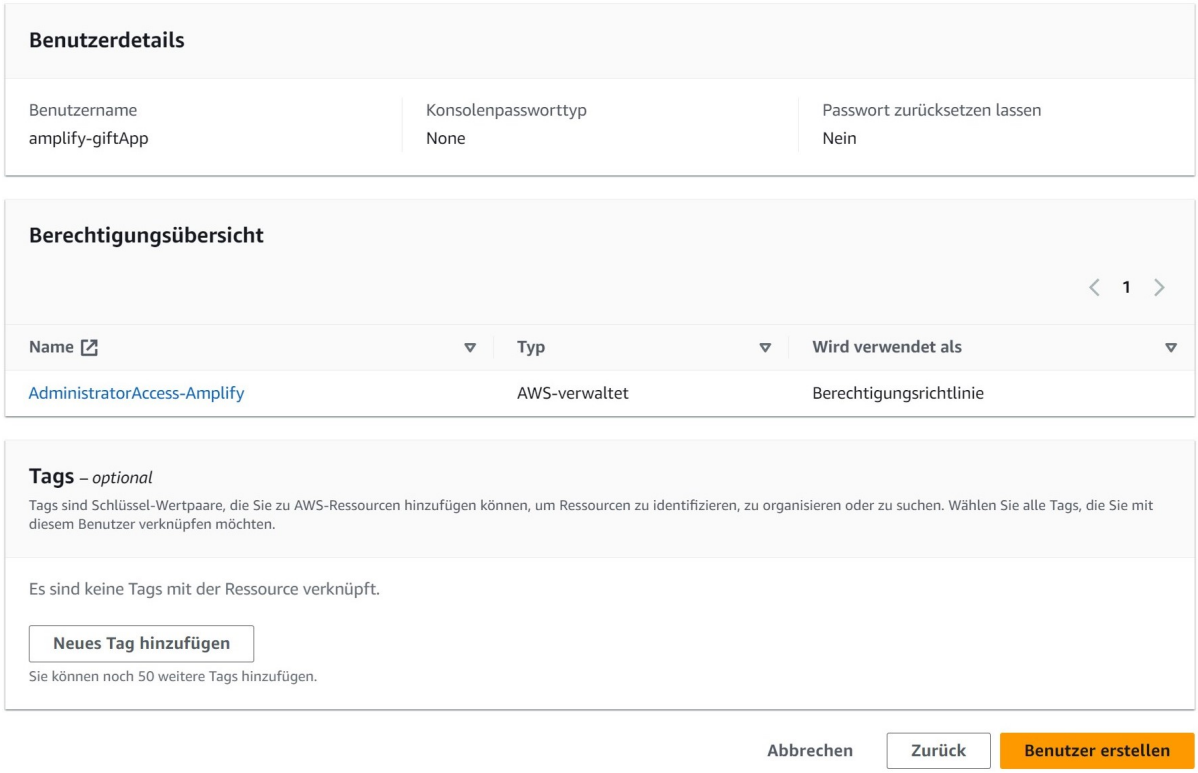


Abbildung 2.6: IAM-User erstellen

Wählt nun den Benutzer aus, den ihr gerade erstellt habt.

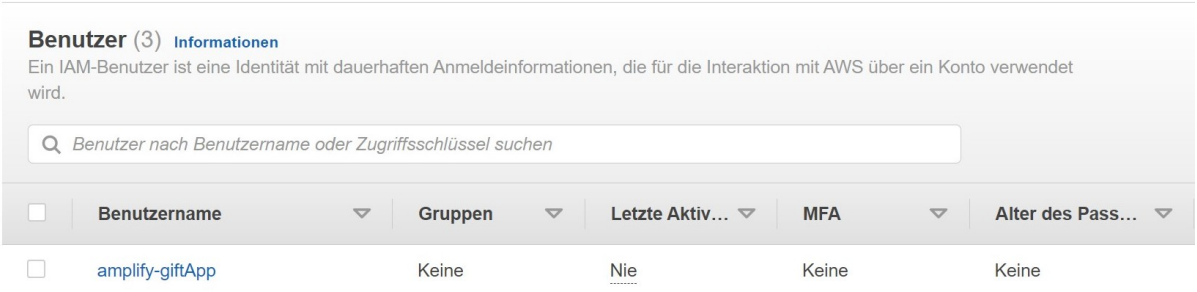


Abbildung 2.7: IAM-User erstellen

Wählt den Reiter **Sicherheitsanmeldeinformationen** aus und klickt auf **Zugriffsschlüssel erstellen**.

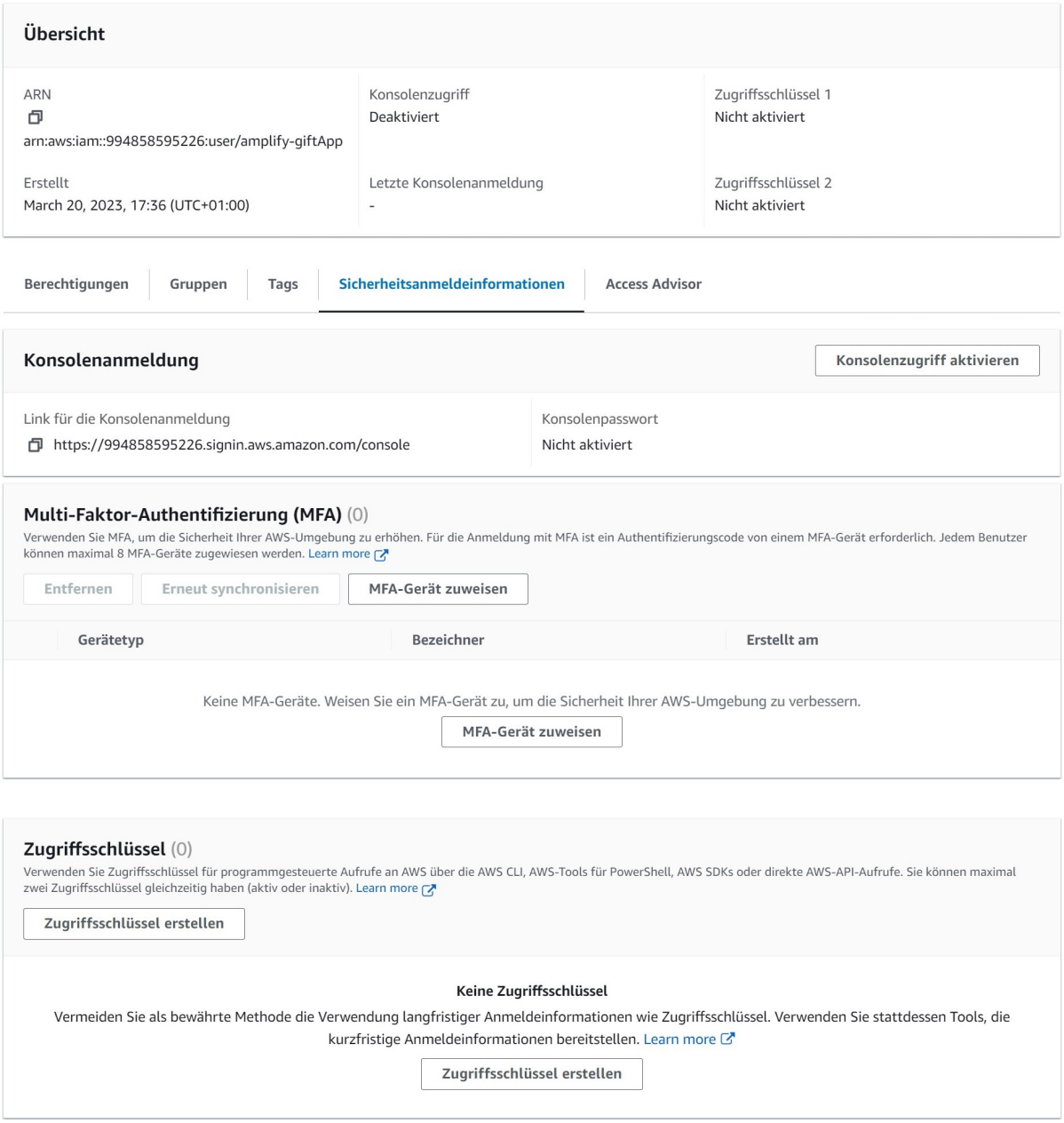


Abbildung 2.8: IAM-User erstellen

Wählt die oberste Option aus.

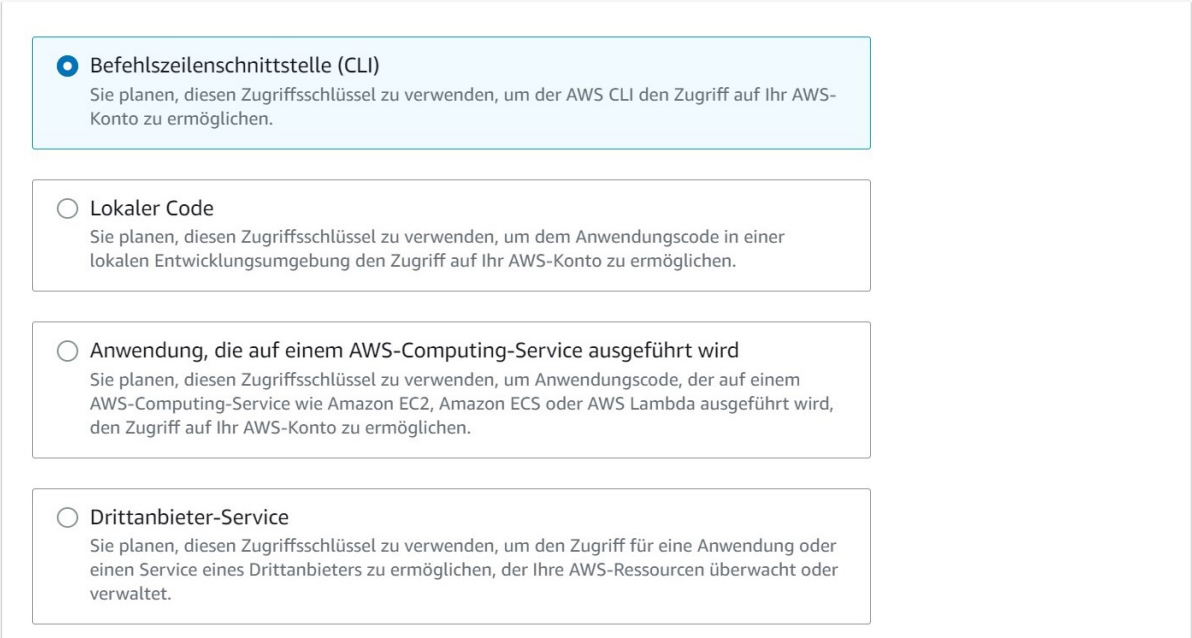


Abbildung 2.9: IAM-User erstellen

Aktzeptiert die Warnung und klickt auf **Weiter**.

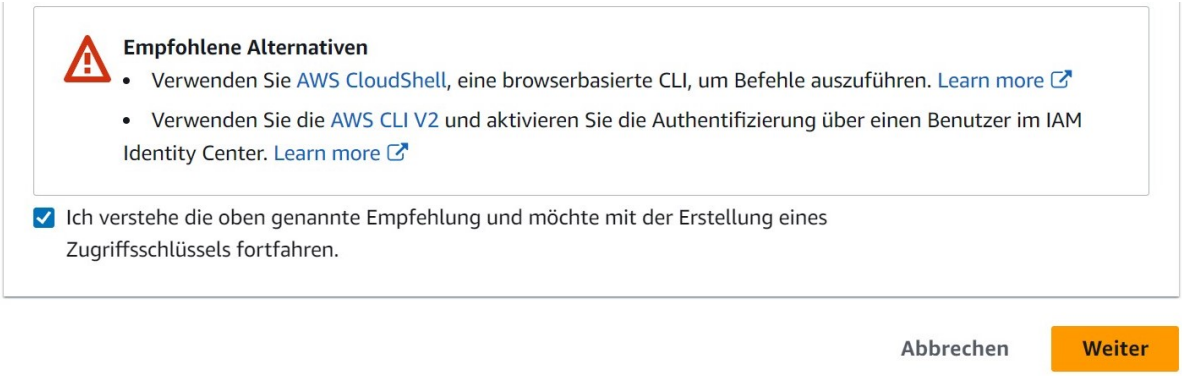


Abbildung 2.10: IAM-User erstellen

Ignoriert die nächste Seite und erstellt euren Zugriffsschlüssel. Nun wird euch euer Zugriffsschlüssel und euer geheimer Zugriffschlüssel angezeigt.

Kehrt wieder in euer Terminal zurück und drückt **Enter** um fortzufahren. Kopiert euren Zugriffsschlüssel und fügt ihn im Terminal ein. Macht das gleiche mit eurem geheimen Zugriffschlüssel. Zum Schluss müsst ihr euch noch einen beliebigen Namen für euer lokales Profil aussuchen. Mit einer erneuten Eingabe von **Enter** wird die Erstellung abgeschlossen.

```
Press Enter to continue

Enter the access key of the newly created user:
? accessKeyId:  ********************
? secretAccessKey:  ****************************************
This would update/create the AWS Profile in your local machine
? Profile Name:  amplify-giftApp

Successfully set up the new user.
```

Abbildung 2.11: IAM-User erstellen

Führt vom Verzeichnis, in dem eure App gespeichert werden soll folgenden Befehl aus (anstelle von *gift_app* tragt ihr natürlich den Namen eurer App ein):

```
> flutter create gift_app --platforms=android,ios
```

Abbildung 2.12: Projekt erstellen

Öffnet nun euer neu erstelltes Projekt in Visual Studio Code. Eure Ordnerstruktur sollte nun wie folgt aussehen:

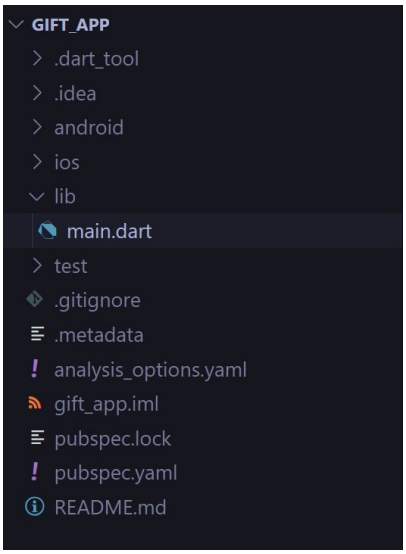


Abbildung 2.13: Ordnerstruktur

Um zu kontrollieren ob alles funktioniert könnt ihr *main.dart* ausführen und einen Emulator auswählen (welcher im Laufe der Installationsanleitung von Flutter in Kapitel 1.2 erstellt wurde). Nach kurzer Wartezeit seht ihr die Flutter Demo App auf eurem Emulator.

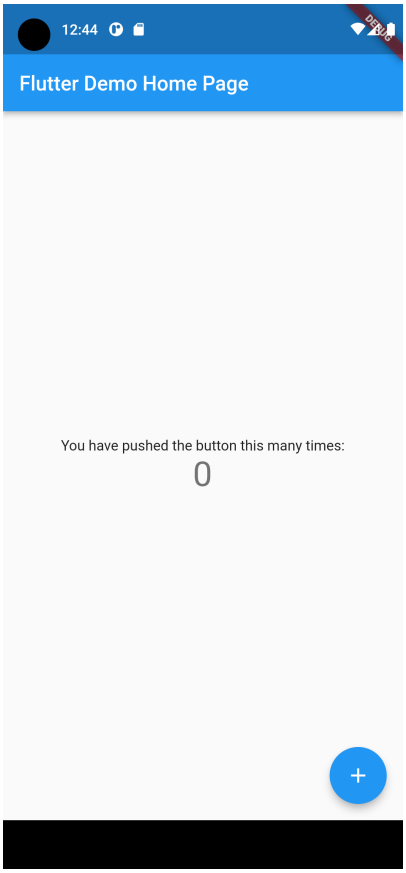


Abbildung 2.14: Demo App

Öffnet die Datei *android/app/build.gradle* und verändert unter android und defaultConfig die minSdkVersion zu **minSdkVersion 21**. Wenn ihr auf einem Mac programmiert öffnet *ios/* und modifiziert das Podfile. Ändert die Zielplattform zu **platform :ios, '13.0'**.

Beendet die Ausführung wieder und klickt auf *pubspec.yaml*. Hier fügt ihr unter **dependencies** und **flutter** eine neue Abhängigkeit hinzu. Speichert die Datei mit **Strg+S** um amplify_flutter herunterzuladen.

```
24  # Dependencies specify other packages that your package needs in order to work.
25  # To automatically upgrade your package dependencies to the latest versions
26  # consider running `flutter pub upgrade --major-versions`. Alternatively,
27  # dependencies can be manually updated by changing the version numbers below to
28  # the latest version available on pub.dev. To see which dependencies have newer
29  # versions available, run `flutter pub outdated`.
30  dependencies:
31    flutter:
32      sdk: flutter
33
34
35    # The following adds the Cupertino Icons font to your application.
36    # Use with the CupertinoIcons class for iOS style icons.
37    cupertino_icons: ^1.0.2
38    amplify_flutter: ^0.6.0
39
40  dev_dependencies:
41    flutter_test:
42      sdk: flutter
```

Abbildung 2.15: *pubspec.yaml*

Führt im Verzeichnis eures Projekts folgenden Befehl aus:

```
> amplify init
```

Abbildung 2.16: Amplify initialisieren

Initialisiert das Projekt mit diesen Einstellungen (als Namen natürlich wieder den Namen eurer App):

```
The following configuration will be applied:

Project information
| Name: giftapp
| Environment: dev
| Default editor: Visual Studio Code
| App type: flutter
| Configuration file location: ./lib/

? Initialize the project with the above configuration? (Y/n)
```

Abbildung 2.17: Amplify initialisieren

Wählt als Authentifizierungsmethode **AWS profile** und wählt euer eben erstelltes Profil aus.

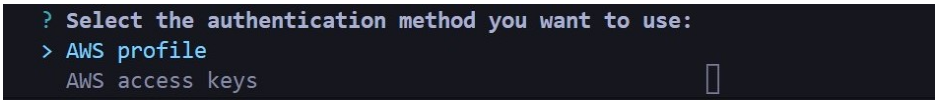


Abbildung 2.18: Amplify initialisieren

Für euch wird nun ein backend environment eingerichtet. Sobald die Einrichtung abgeschlossen ist, könnt ihr mit dem nächsten Kapitel fortfahren.

2.2 Amplify Backend

In diesem Kapitel lernt ihr Amplify Studio kennen. Amplify Studio ist ein Tool, um das Backend eurer App einzurichten und zu verändern. Besucht die AWS-Managementkonsole und meldet euch als Stammbenutzer mit eurem Account an. Auf der Konsole angekommen, klickt in der oberen linken Ecke auf **Services**, dann auf **Alle Services** und auf **AWS Amplify**.

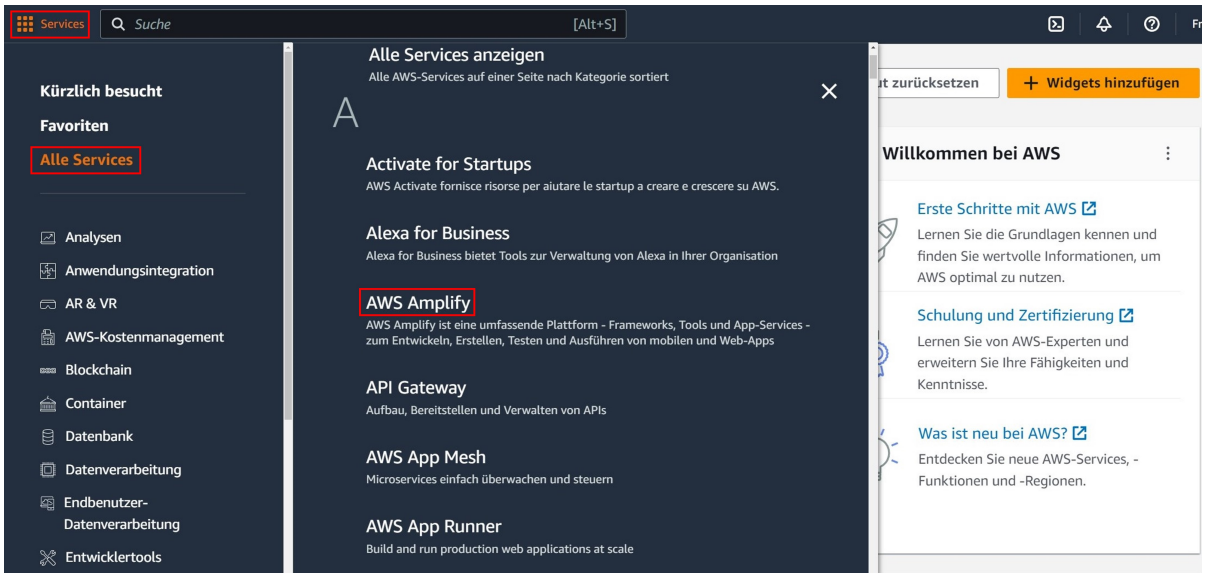


Abbildung 2.19: Amplify Studio starten

Stellt sicher, dass ihr den gleichen Standort ausgewählt habt, den ihr für eure App ausgesucht habt. Euer Projekt sollte euch nun angezeigt werden.

Abbildung 2.20: Amplify Studio starten

Klickt darauf, um mit dem Einrichten von Amplify Studio zu beginnen. Ihr landet auf einer neuen Seite. Wählt **Einrichten von Amplify Studio aus**

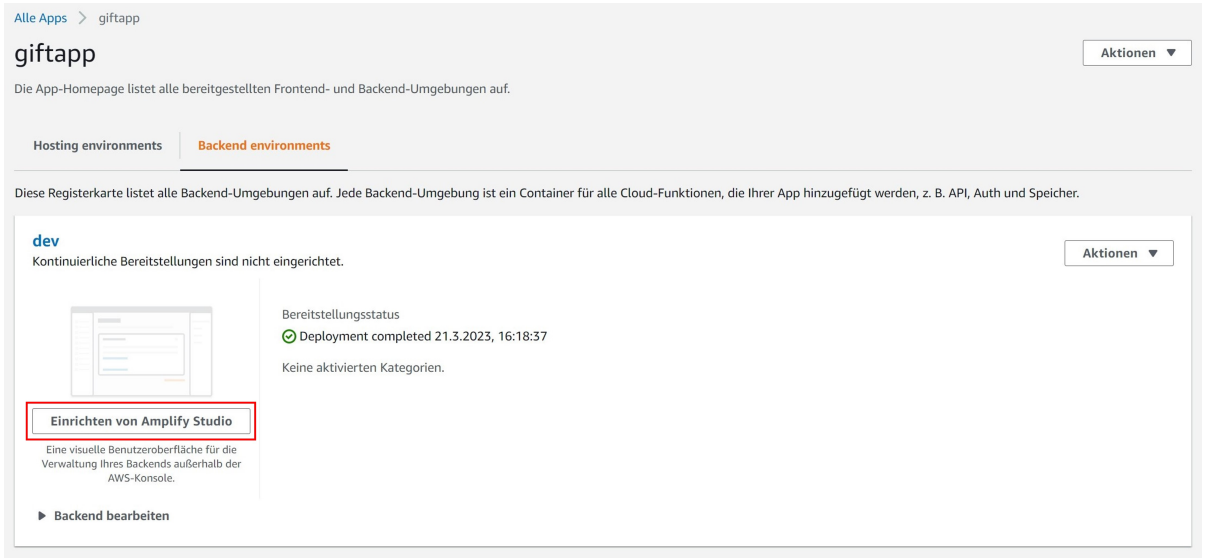


Abbildung 2.21: Amplify Studio starten

und aktiviert den oberen Schalter.

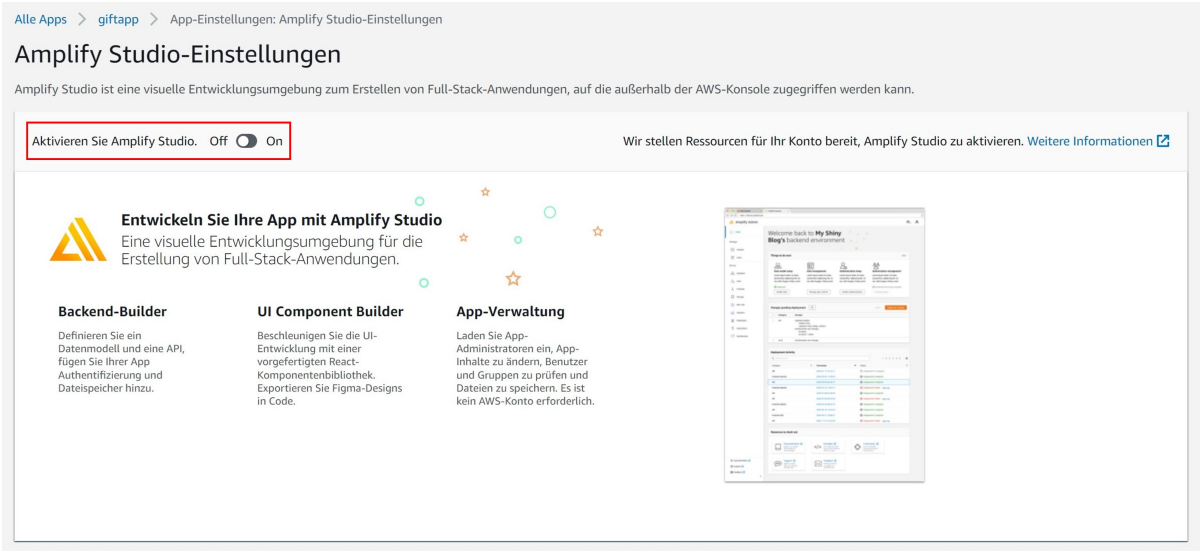


Abbildung 2.22: Amplify Studio starten

Sobald die Aktivierung abgschlossen wurde, geht ihr wieder auf die Übersicht eures Projekts und klickt auf **Studio starten**.

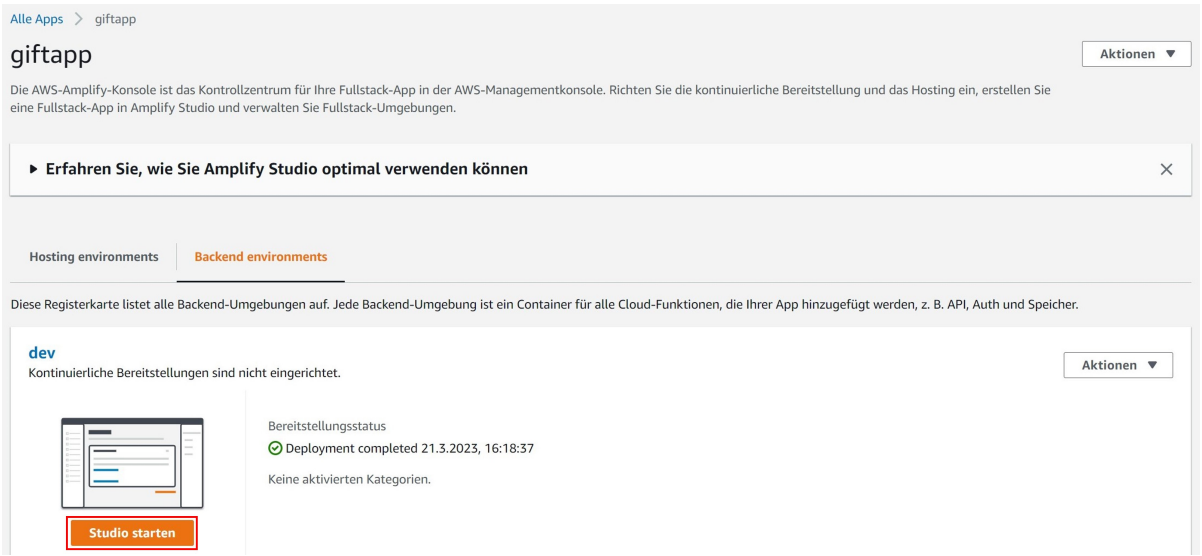


Abbildung 2.23: Amplify Studio starten

Damit ist die Einrichtung aller Komponenten abgschlossen und ihr könnt mit der Erstellung eurer App beginnen. Die verschiedenen Funktionen von Amplify Studio werden nach und nach in den folgenden Kapiteln erläutert.

3. Authentifizierung Teil 1

In diesem Kapitel fügen wir unserer App Ihre ersten Funktionalitäten hinzu. Einem Benutzer wird es möglich sein, sich zu registrieren und sich anzumelden, um auf den Startbildschirm unserer App zu gelangen.

3.1 Authentifizierungsregeln erstellen

Wir beginnen mit dem Erstellen von Authentifizierungsregeln unseres Backends in Amplify Studio. Dafür öffnen wir Amplify Studio und klicken auf den Reiter **Authentication**.

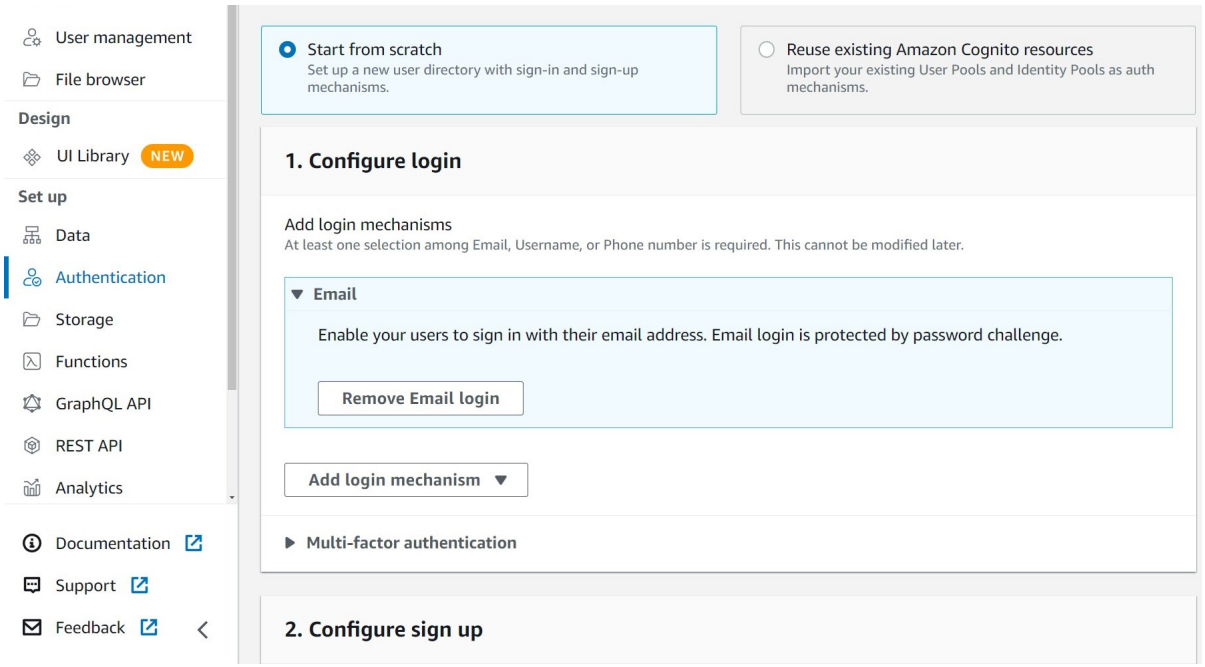


Abbildung 3.1: Amplify Studio

Als erstes Modifizieren wir den Log In Mechanismus. Hier könnt Ihr auswählen, welche Angaben ein Benutzer eurer App machen muss, um sich mit seinem Account anzumelden.
Im Rahmen dieses Tutorial werden wir einen Benutzernamen zum anmelden benutzen, ihr könnt aber natürlich verwenden was ihr möchtet.

Desweiteren haben wir die Möglichkeit Multifaktor-Authentifizierung anzulegen. Im Rahmen dieses Tutorials werden wir allerdings darauf verzichten, da jede von Amazon gesendete SMS kostenpflichtig ist und in diesem Tutorial keine Kosten entstehen sollen. Unsere Einstellungen für den Log In Mechanismus sehen also wie folgt aus:

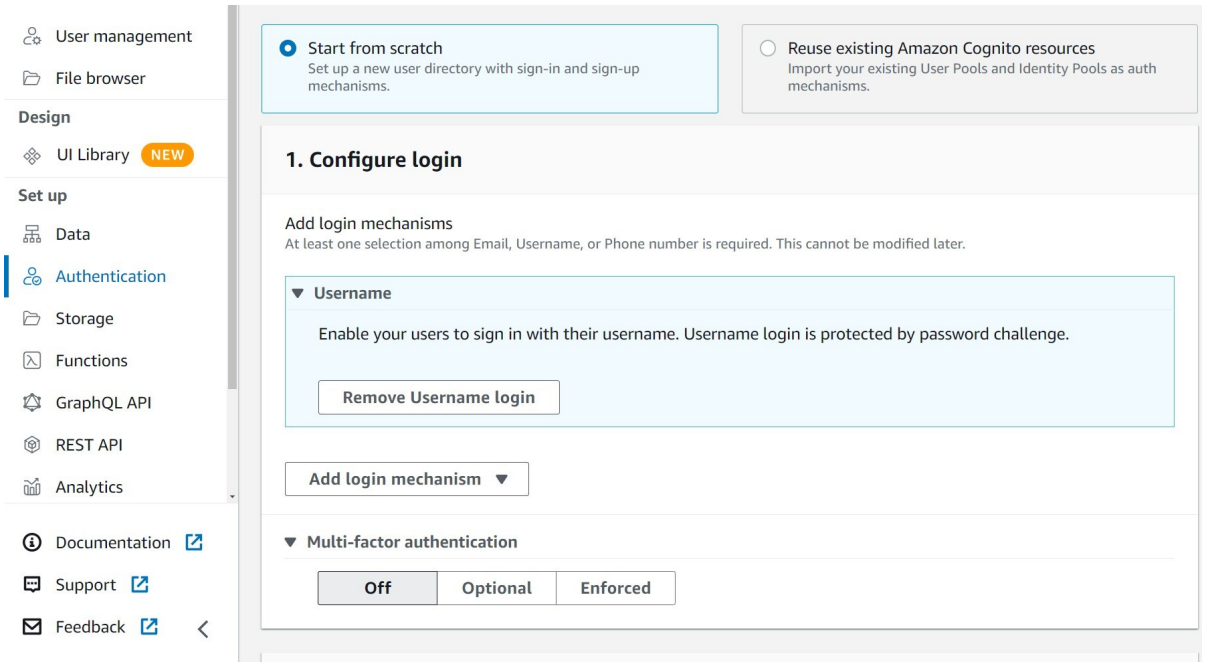
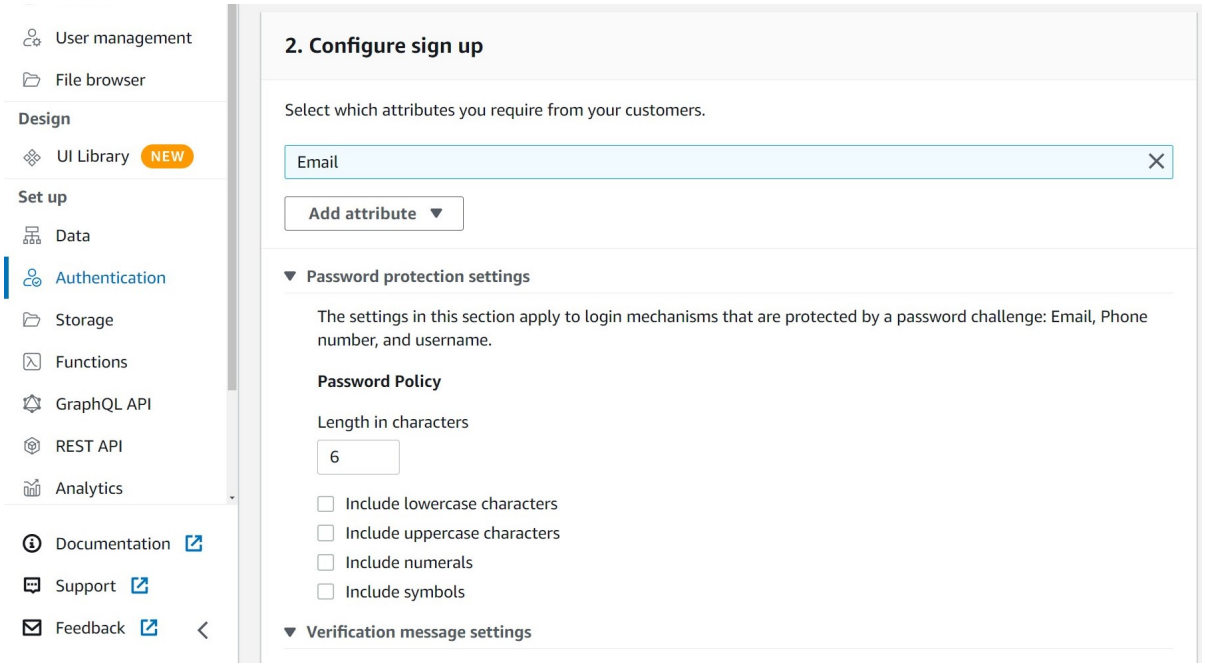


Abbildung 3.2: Amplify Studio

Als nächstes Modifizieren wir den Sign Up Mechanismus. Hier können wir als erstes einstellen, was ein neuer Benutzer angeben muss, um sich in der App zu registrieren. Desweiteren könnt ihr Regeln für das Passwort der Nutzer festlegen und den Text eurer Verifizierungsemail bzw. SMS festlegen. Die Einstellungen der Beispiel-App sehen wie folgt aus:



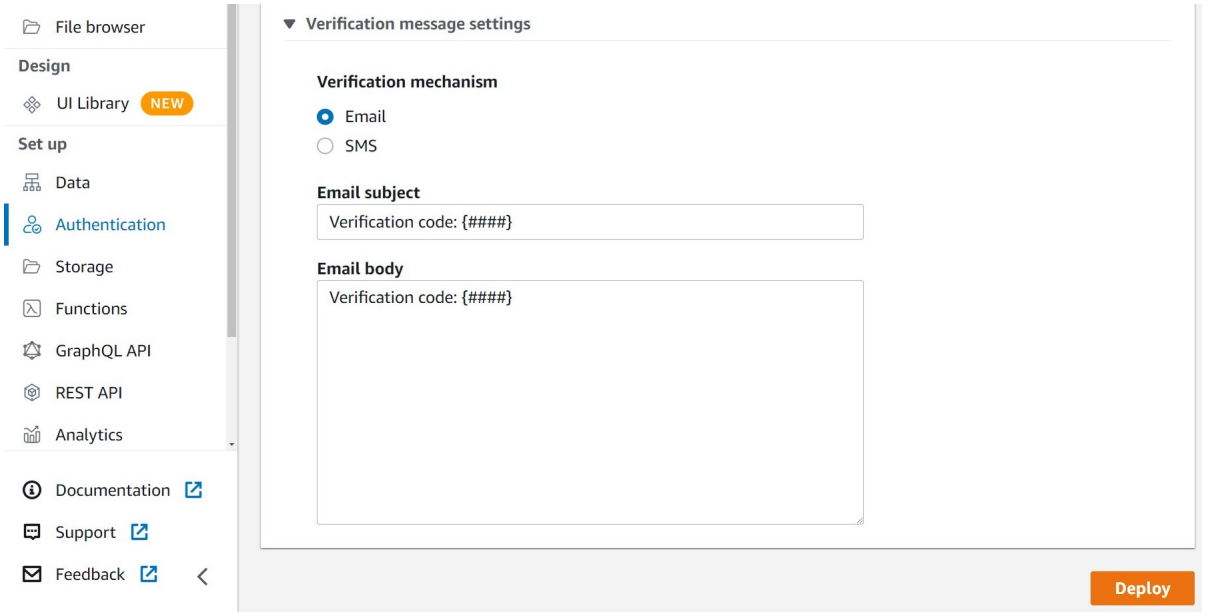


Abbildung 3.3: Amplify Studio

Mit einem Klick auf **Deploy** werden eure Einstellungen gespeichert. Nach kurzer Wartezeit werdet ihr aufgefordert die Veränderungen mit eurer App zu Synchronisieren. Kopiert dafür den bei euch angezeigten Befehl und führt ihn im Verzeichnis eures Projekts aus. Anschließend klickt ihr auf **Done**.

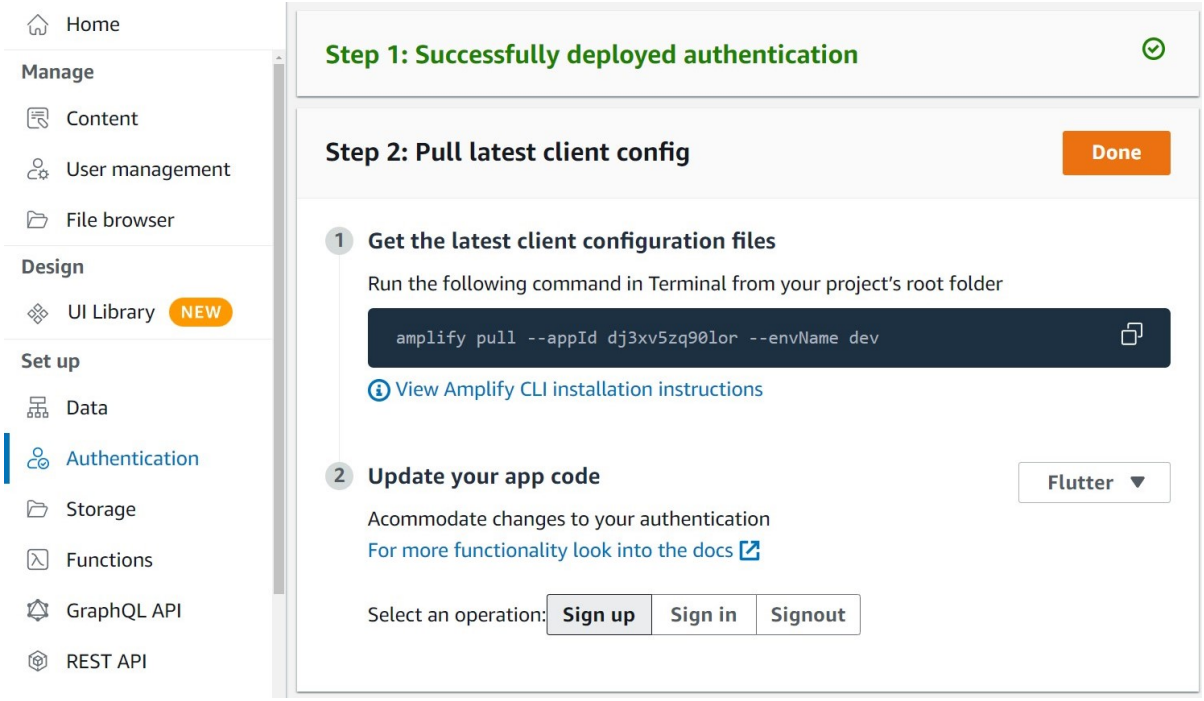


Abbildung 3.4: Amplify Studio

3.2 Amplify Authenticator

In diesem Abschnitt lernt ihr den Amplify Authenticator und seine Funktionen kennen. Dieser ermöglicht es euch einen vorgefertigten Authentifizierungsdialog zu verwenden und diesen beliebig anzupassen.

Als erstes gehen wir zurück nach Visual Studio Code und fügen 2 neue Abhängigkeiten in *pubspec.yaml* ein und installieren sie, indem wir die Datei speichern:

```
24  # Dependencies specify other packages that your package needs in order to work.
25  # To automatically upgrade your package dependencies to the latest versions
26  # consider running `flutter pub upgrade --major-versions`. Alternatively,
27  # dependencies can be manually updated by changing the version numbers below to
28  # the latest version available on pub.dev. To see which dependencies have newer
29  # versions available, run `flutter pub outdated`.
30  dependencies:
31    flutter:
32      sdk: flutter
33
34
35    # The following adds the Cupertino Icons font to your application.
36    # Use with the CupertinoIcons class for iOS style icons.
37    cupertino_icons: ^1.0.2
38    amplify_flutter: ^0.6.0
39    amplify_auth_cognito: ^0.6.0
40    amplify_authenticator: ^0.2.0
```

Abbildung 3.5: pubspec.yaml

TIPP

Immer wenn wir *pubspec.yaml* verändern muss **pub get** ausgeführt werden, um die eingefügten Abhängigkeiten zu installieren. Dies ist zum Beispiel durch Speichern der Datei möglich.

Als nächstes öffnen wir *main.dart*, löschen den kompletten Code in der Datei und ersetzen ihn wie folgt:

```dart
import 'package:amplify_auth_cognito/amplify_auth_cognito.dart';
import 'package:amplify_authenticator/amplify_authenticator.dart';
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'amplifyconfiguration.dart';

Run | Debug | Profile
void main() {
  runApp(const GiftApp());
}

class GiftApp extends StatefulWidget {
  const GiftApp({Key? key}) : super(key: key);

  @override
  State<GiftApp> createState() => _GiftAppState();
}

class _GiftAppState extends State<GiftApp> {
  @override
  void initState() {
    super.initState();
    _configureAmplify();
  }

  void _configureAmplify() async {
    try {
      final authPlugin = AmplifyAuthCognito();
      await Amplify.addPlugin(authPlugin);
      await Amplify.configure(amplifyconfig);
    } on Exception catch (e) {
      safePrint(e);
    }
  }

  @override
  Widget build(BuildContext context) {
    return Authenticator(
      child: MaterialApp(
        builder: Authenticator.builder(),
        home: const Scaffold(
          body: Center(
            child: Text('Du hast dich erfolgreich eingeloggt!'),
          ), // Center
        ), // Scaffold
      ), // MaterialApp
    ); // Authenticator
  }
}
```

Abbildung 3.6: *main.dart*

Die Funktion *_configureAmplify()* wird bei jedem Neustart der App ausgeführt.
Wir fügen dass Amplify Authentifizierungs-Plugin hinzu und konfigurieren Amplify mit unseren Einstellungen aus *amplifyconfiguration.dart*.

In der *build()* Methode benutzen wir den Authenticator von Amplify. Ohne weitere Konfigurationen übernimmt dieser unsere Backend-Einstellungen aus Amplify Studio.

Führt den neuen Code aus und ihr habt bereits einen funktionierenden Authentifizierungsprozess. Wenn Ihr diesem Tutorial folgt, wartet bis Kapitel 4 mit dem Anlegen eines Benutzers:

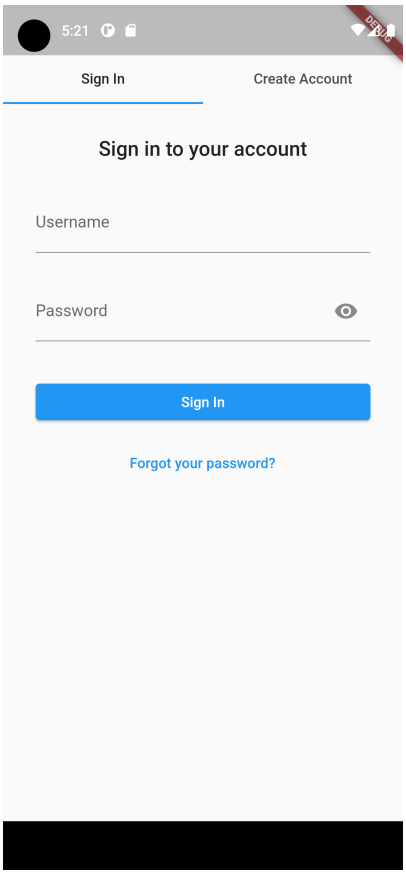


Abbildung 3.7: Sign In Screen

TIPP

Wenn ihr euren Code verändert und die Datei speichert, werden eure änderungen direkt im Emulator sichtbar. Wenn ihr allerdings Veränderungen in *main()* vornehmt, müsst ihr die Ausführung beenden und neu starten. Ebenso, kann es bei unerklärlichen Fehlermeldungen immer helfen, den Code nochmal neu auszuführen.

3.3 Authenticator Anpassen

Als nächstes passen wir den Sign In und Create Account Bildschirm nach unseren Vorstellungen an.

3.3.1 Felder Hinzufügen

Im Amplify Studio ist es nicht mehr möglich zusätzliche Felder auf dem Registrierungsbildschirm hinzuzufügen, allerdings können wir dies manuell mit dem Amplify Authenticator tun. In diesem Beispiel wollen wir, dass sich der Benutzer neben seinem Benutzernamen, der einzigartig sein muss, noch einen beliebigen Nickname aussuchen kann. Dafür fügen wir unserem *Authenticator()* eine angepasste *SignUpForm()* hinzu:

```
37      return Authenticator(
38        signUpForm: SignUpForm.custom(
39          fields: [
40            SignUpFormField.username(),
41            SignUpFormField.email(required: true),
42            SignUpFormField.custom(
43              required: true,
44              validator: (value) {
45                if (value == null || value.isEmpty) {
46                  return 'Du musst einen Nickname angeben';
47                }
48                return null;
49              },
50              title: 'Nickname',
51              attributeKey: CognitoUserAttributeKey.nickname,
52            ),
53            SignUpFormField.password(),
54            SignUpFormField.passwordConfirmation(),
55          ],
56        ), // SignUpForm.custom
```

Abbildung 3.8: *main.dart*

Wenn wir eine custom form benutzen, müssen wir alle Felder die wir brauchen manuell hinzufügen. Außerdem legen wir zusätslich ein Nickname Feld an. Dieses Feld muss vom Benutzer ausgefüllt werden. *validator* gibt eine Nachricht an den Benutzer aus, wenn er dass Feld nicht ausgefüllt hat. Hier lassen sich auch noch andere Regeln festlegen. Als Attributsschlüssel wählen wir das korrespondierende Feld im Cognito-User-Pool von Amazon aus. Falls dieses nicht existiert können wir hier ebenfalls ein selbst erstelltes hinzufügen:

```
51              attributeKey: const CognitoUserAttributeKey.custom('neuesFeld'),
```

Abbildung 3.9: *Beispiel*

Unser neuer Registrierungsbildschirm sieht nun so aus:

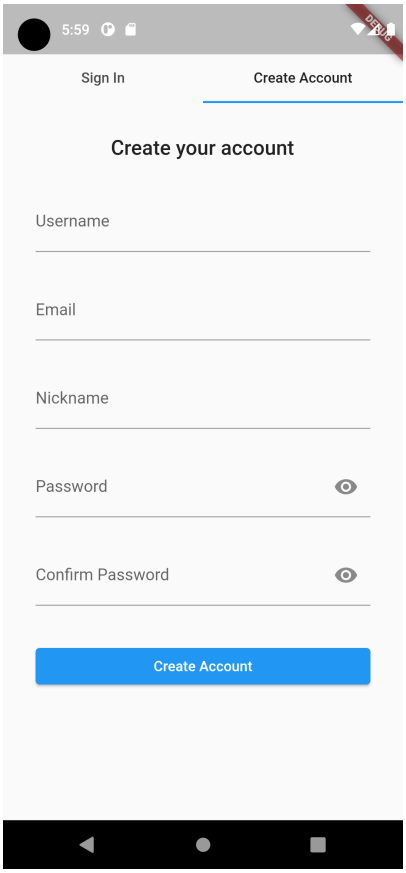


Abbildung 3.10: Registrierungsbildschirm

Gleichermaßen lässt sich die Sign In Form anpassen.

3.3.2 Design anpassen

Als nächstes ändern wir das Aussehen unseres Anmeldungs- und Registrierungsbildschirms. Dafür legen wir uns zuerst 2 neue Ordner an:

- **lib/assets**
- **lib/widgets**

In den Ordner **assets** legen wir eine Bilddatei, die wir später als Logo in unserer App verwenden. Im Ordner **widgets** ertellen wir die Datei *auth_widgets.dart*. Unsere neue Ordnerstruktur sieht wie folgt aus:

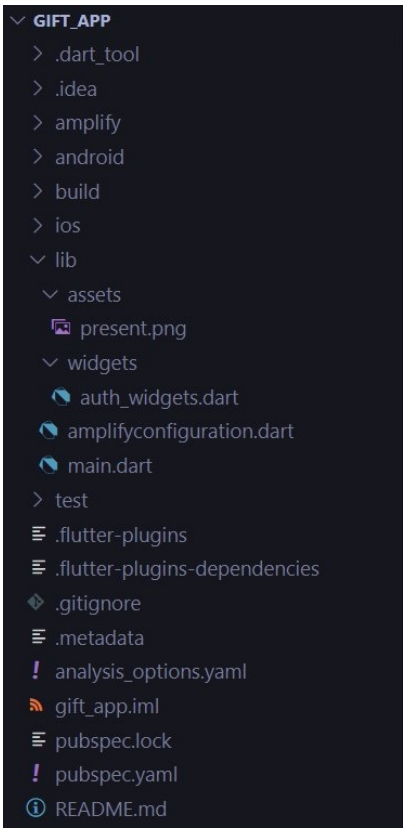


Abbildung 3.11: Ordnerstruktur

Damit wir die Bilddatei auch verwenden können, müssen wir eine neue Zeile in *pubspec.yaml* hinzufügen:

```
56  # The following section is specific to Flutter packages.
57  flutter:
58
59    # The following line ensures that the Material Icons font is
60    # included with your application, so that you can use the icons in
61    # the material Icons class.
62    uses-material-design: true
63
64    # To add assets to your application, add an assets section, like this:
65    assets:
66      - lib/assets/present.png
67
```

Abbildung 3.12: *pubspec.yaml*

In *main.dart* fügen wir zuerst einen neue Import hinzu:

```
1  import 'package:amplify_auth_cognito/amplify_auth_cognito.dart';
2  import 'package:amplify_authenticator/amplify_authenticator.dart';
3  import 'package:amplify_flutter/amplify_flutter.dart';
4  import 'package:flutter/material.dart';
5  import 'package:gift_app/widgets/auth_widgets.dart';
6  import 'amplifyconfiguration.dart';
```

Abbildung 3.13: *main.dart*

Um nun das Aussehen des Anmeldungs- und Registrierungsbildschirms zu verändern, ändern wir die Struktur von *Authenticator()*. Wir fügen *authenticatorBuilder* hinzu. Damit ist es möglich, jeden Schritt im Authentifizierungsprozess beliebig anzupassen. In unserem Beispiel verändern wir nur Sign In und Sign Up. Aus Gründen der Übersichtlichkeit lagern wir den neuen Code der beiden Bildschirme nach *auth_widgets.dart* aus, indem wir dort 2 Widgets (*SignInWidget* und *SignUpWidget*) erstellen. Außerdem verändern wir mithilfe von *theme* die Helligkeit unserer App in den Darkmode und das Farbthema zu einem Gelbton:

```
38      return Authenticator(
39        authenticatorBuilder: (BuildContext context, AuthenticatorState state) {
40          switch (state.currentStep) {
41            case AuthenticatorStep.signIn:
42              return SignInWidget(state: state);
43            case AuthenticatorStep.signUp:
44              return SignUpWidget(state: state);
45            default:
46              return null;
47          }
48        },
49        child: MaterialApp(
50          builder: Authenticator.builder(),
51          theme: ThemeData(
52            brightness: Brightness.dark,
53            primarySwatch: Colors.amber,
54          ), // ThemeData
```

Abbildung 3.14: *main.dart*

In *auth_widgets.dart* definieren wir das Aussehen unserer beiden Widgets. Dies sind nur exemplarische Beispiele, eurer Kreativität sind hier keine Grenzen gesetzt. Wichtig ist nur, dass ihr wieder die Amplify Forms *SignInForm()* und *SignUpForm()* verwendet. Außerdem legen wir jeweils einen neuen Button an um zwischen den beiden Bildschirmen zu wechseln. Hierbei ist es wichtig, nicht zu vergessen den Zustand des *AuthenticatorSteps* ebenfalls zu wechseln:

```dart
import 'package:amplify_auth_cognito/amplify_auth_cognito.dart';
import 'package:amplify_authenticator/amplify_authenticator.dart';
import 'package:flutter/material.dart';

class SignInWidget extends StatelessWidget {
  final AuthenticatorState state;
  const SignInWidget({super.key, required this.state});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Padding(
        padding: const EdgeInsets.only(left: 16, right: 16, top: 48, bottom: 16),
        child: SingleChildScrollView(
          child: Column(
            children: [
              const SizedBox(
                height: 50,
              ), // SizedBox
              // App Logo
              Center(
                child: Image.asset(
                  "lib/assets/present.png",
                  scale: 1.5,
                ), // Image.asset
              ), // Center
              const SizedBox(
                height: 30,
              ), // SizedBox
              // Amplify Sign In Form
              SignInForm(),
            ],
          ), // Column
        ), // SingleChildScrollView
      ), // Padding
```

Abbildung 3.15: *auth_widgets.dart*

```dart
      // Buttom um den Benutzer zum Registrierungsbildschirm zu bringen
      persistentFooterButtons: [
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text('Noch keinen Account?'),
            TextButton(
              onPressed: () => state.changeStep(
                AuthenticatorStep.signUp,
              ),
              child: const Text('Registrieren'),
            ), // TextButton
          ],
        ), // Row
      ],
    ); // Scaffold
  }
}

class SignUpWidget extends StatelessWidget {
  final AuthenticatorState state;
  const SignUpWidget({super.key, required this.state});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Padding(
        padding: const EdgeInsets.only(left: 16, right: 16, top: 48, bottom: 16),
        child: SingleChildScrollView(
          child: Column(
            children: [
              const SizedBox(
                height: 50,
              ), // SizedBox
              // App Logo
              Center(
                child: Image.asset(
                  "lib/assets/present.png",
                  scale: 1.5,
                ), // Image.asset
              ), // Center
              const SizedBox(
                height: 30,
              ), // SizedBox
```

Abbildung 3.16: *auth_widgets.dart*

```dart
                // Unsere angepasste Sign Up Form
                SignUpForm.custom(
                  fields: [
                    SignUpFormField.username(),
                    SignUpFormField.email(required: true),
                    SignUpFormField.custom(
                      required: true,
                      validator: (value) {
                        if (value == null || value.isEmpty) {
                          return "Bitte gib einen Nickname ein";
                        }
                        return null;
                      },
                      title: 'Nickname',
                      attributeKey: CognitoUserAttributeKey.nickname,
                    ),
                    SignUpFormField.password(),
                    SignUpFormField.passwordConfirmation(),
                  ],
                ), // SignUpForm.custom
```

```dart
      ), // Padding
      // Buttom um den Benutzer zum Anmeldebildschirm zu bringen
      persistentFooterButtons: [
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text('Du hast bereits einen Account?'),
            TextButton(
              onPressed: () => state.changeStep(
                AuthenticatorStep.signIn,
              ),
              child: const Text('Einloggen'),
            ), // TextButton
          ],
        ), // Row
      ],
    ); // Scaffold
  }
}
```

Abbildung 3.17: *auth_widgets.dart*

So sehen unsere Veränderungen aus:

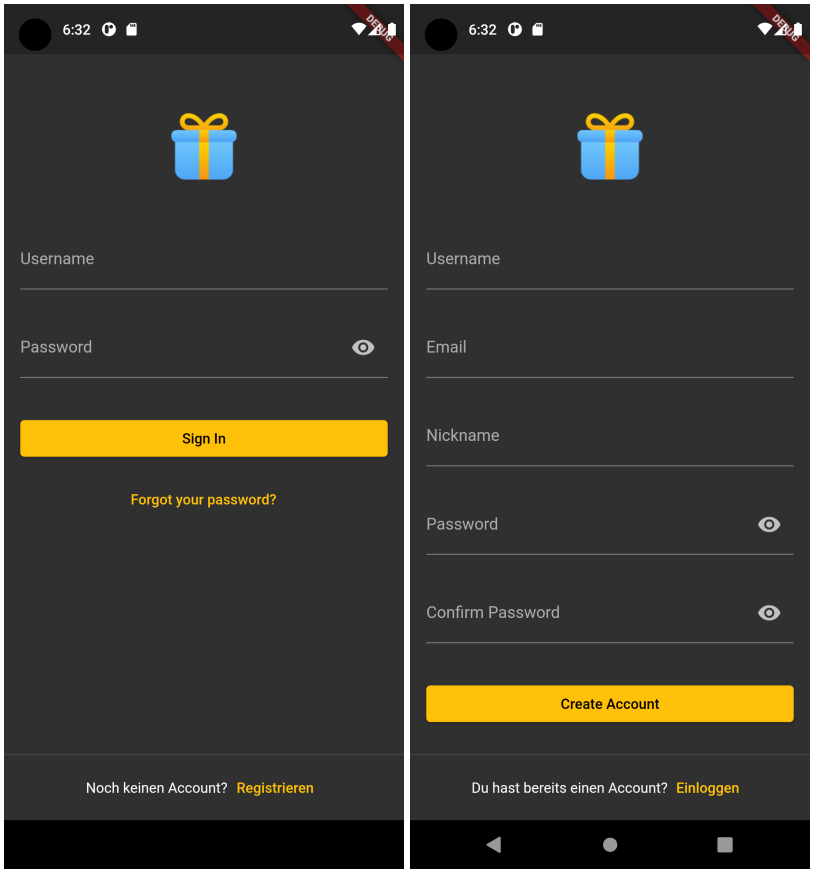


Abbildung 3.18: Anmelde- und Registrierungsbildschirm

3.3.3 Sprache ändern

Zum Schluss ändern wir noch die Sprache der Felder auf Deutsch. Dafür gibt es leider keine einfache Funktion, sondern ist nur etwas umständlich möglich. Um die Beschriftungen der einzelnen Felder verändern zu können, müssen wir Internationalization (oder kurz: I18n) verwenden. Damit ist es normalerweise möglich, den Benutzer je nach Standort eine andere Sprache in der App zu präsentieren. Wir verwenden I18n, um den Text der einzelnen Felder des Authenticators anzupassen.

Dafür müssen wir zuerst wieder Abhängigkeiten in *pubspec.yaml* hinzufügen:

```
30  dependencies:
31    flutter:
32      sdk: flutter
33    flutter_localizations:
34      sdk: flutter
35
36
37    # The following adds the Cupertino Icons font to your application.
38    # Use with the CupertinoIcons class for iOS style icons.
39    cupertino_icons: ^1.0.2
40    amplify_flutter: ^0.6.0
41    amplify_auth_cognito: ^0.6.0
42    amplify_authenticator: ^0.2.0
43    intl: ^0.17.0
```

Abbildung 3.19: pubspec.yaml

Außerdem müssen wir das *generate* Flag auf true setzen um die automatische Erstellung von Code zu ermöglichen.

```
59  # The following section is specific to Flutter packages.
60  flutter:
61
62    # The following line ensures that the Material Icons font is
63    # included with your application, so that you can use the icons in
64    # the material Icons class.
65    uses-material-design: true
66    generate: true
```

Abbildung 3.20: pubspec.yaml

Als nächstes erstellen wir die Datei *l10n.yaml* (l10n steht für Localization), *lib/localized_resolver.dart* und den Ordner **l10n** mit der Datei *amplify_de.arb*. Unsere neue Ordnerstruktur sieht wie folgt aus:

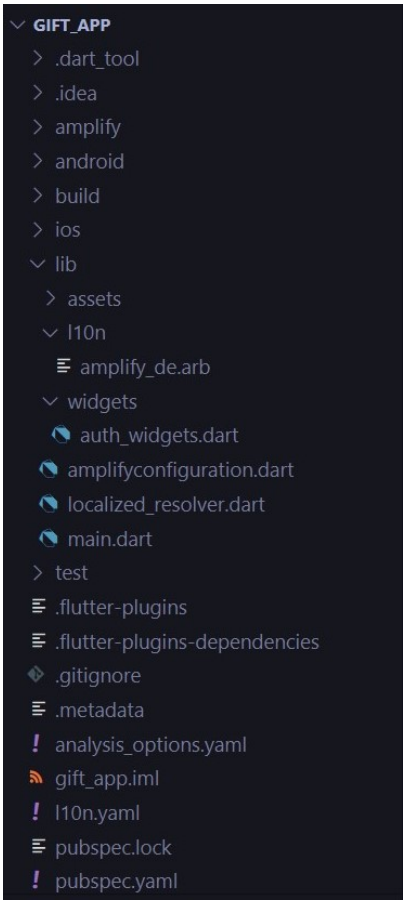


Abbildung 3.21: Ordnerstruktur

l10.yaml konfiguriert die Localization. In unserem Fall liegen die Input Files in *lib/l10n*, das Template wird bereitgestellt von *amplify_de.arb* und der generierte Code soll in *app_localizations.dart* gespeichert werden. Daher fügen wir folgenden Code ein:

```
1  arb-dir: lib/l10n
2  template-arb-file: amplify_de.arb
3  output-localization-file: app_localizations.dart
```

Abbildung 3.22: *l10.yaml*

Zunächst wollen wir den Text auf dem Anmeldebildschirm verändern. Dort haben wir 2 Buttons und 2 Felder. Außerdem wird ein Hinweistext angezeigt, wenn wir auf eines der Felder drücken. Im amplify_authenticator Repository auf Github gibt es unter lib/src/l10n/src die vollständigen Listen für die Bezeichnungen der verschiedenen Felder, Buttons, Titel, Nachrichten und Länderbezeichnungen des Authenticators. Für unseren Sign In Bildschirm brauchen wir die Bezeichnungen der beiden Buttons und Felder sowie der Hinweisnachricht. Diese tragen wir in *amplify_de.arb* mit den entsprechenden Übersetzungen ein. Um den Hinweistext nicht anzeigen zu lassen können wir einfach einen leeren String angeben:

```
{
  "signin": "Einloggen",
  "username": "Nutzername",
  "password": "Passwort",
  "forgotPassword": "Passwort vergessen?",
  "promptFill": ""
}
```

Abbildung 3.23: *amplify_de.arb*

Als nächstes kümmern wir uns um *localized_resolver.dart*. Ein Resolver übernimmt die Aufgabe, die von uns gerade ausgewählten neuen Bezeichnungen an den entsprechenden Stellen anzeigen zu lassen. Zunächst nehmen wir nötige Imports vor.

```
import 'package:amplify_authenticator/amplify_authenticator.dart';
import 'package:flutter/material.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart';
```

Abbildung 3.24: *localized_resolver.dart*

TIPP

Falls das Paket flutter_gen nicht gefunden wird, speichert nochmal alle .yaml Files oder öffnet Visual Studio Code neu.

Danach erstellen wir einen *CustomButtonResolver* der die Funktionen des *ButtonResolvers* vom Amplify Authenticator erweitert. Hier ändern wir die Texte der beiden Buttons. Dafür überschreiben wir die *signIn()* und *forgotPassword()* Funktion des *ButtonResolvers* mit unseren in *amplify_de.arb* festgelegten Strings.

```dart
class CustomButtonResolver extends ButtonResolver {
  const CustomButtonResolver();

  @override
  String signIn(BuildContext context) {
    return AppLocalizations.of(context)!.signin;
  }

  @override
  String forgotPassword(BuildContext context) {
    return AppLocalizations.of(context)!.forgotPassword;
  }
}
```

Abbildung 3.25: *localized_resolver.dart*

TIPP

In Visual Studio Code könnt ihr mit einem Rechtsklick auf **ButtonResolver** und einem Klick auf **Go to definition** die Implementierung von ButtonResolver anschauen. Hier findet ihr alle Funktionen die ihr überschreiben könnt.

Das gleiche Prinzip wenden wir auf die beiden Felder Username und Passwort an. Diesmal überschreiben wir Funktionen des *InputResolvers*:

```dart
class CustomInputResolver extends InputResolver {
  const CustomInputResolver();

  @override
  String title(BuildContext context, InputField field) {
    switch (field) {
      case InputField.username:
        return AppLocalizations.of(context)!.username;
      case InputField.password:
        return AppLocalizations.of(context)!.password;
      default:
        return super.title(context, field);
    }
  }

  @override
  String hint(BuildContext context, InputField field) {
    return AppLocalizations.of(context)!.promptFill;
  }
}
```

Abbildung 3.26: *localized_resolver.dart*

Zum Schluss müssen wir noch unsere beiden *CustomResolver* dem *AuthStringResolver* hinzufügen:

```dart
const stringResolver = AuthStringResolver(
  buttons: CustomButtonResolver(),
  inputs: CustomInputResolver(),
); // AuthStringResolver
```

Abbildung 3.27: *localized_resolver.dart*

TIPP

Wenn ihr in Visual Studio Code mit der Maus über AuthStringResolver geht, seht ihr, welche Resolver ihr noch überschreiben könnt.

Nun müssen wir nur noch *main.dart* etwas anpassen. Zuerst fügen wir 3 neue Imports hinzu:

```dart
import 'package:amplify_auth_cognito/amplify_auth_cognito.dart';
import 'package:amplify_authenticator/amplify_authenticator.dart';
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/widgets/auth_widgets.dart';
import 'package:flutter_localizations/flutter_localizations.dart';
import 'package:gift_app/localized_resolver.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart';
import 'amplifyconfiguration.dart';
```

Abbildung 3.28: *main.dart*

Danach fügen wir dem *Authenticator* unseren *stringResolver* hinzu:

```dart
@override
Widget build(BuildContext context) {
  return Authenticator(
    stringResolver: stringResolver,
    authenticatorBuilder: (BuildContext context, AuthenticatorState state) {
      switch (state.currentStep) {
        case AuthenticatorStep.signIn:
          return SignInWidget(state: state);
        case AuthenticatorStep.signUp:
          return SignUpWidget(state: state);
        default:
          return null;
      }
    },
```

Abbildung 3.29: *main.dart*

Zum Schluss müssen wir noch noch 2 Ergänzungen in *MaterialApp()* machen, damit die Lokalisierung in unserer App funktioniert:

```dart
      child: MaterialApp(
        localizationsDelegates: const [
          GlobalMaterialLocalizations.delegate,
          GlobalWidgetsLocalizations.delegate,
          GlobalCupertinoLocalizations.delegate,
          AppLocalizations.delegate,
        ],
        supportedLocales: const [
          Locale('de'),
        ],
        builder: Authenticator.builder(),
```

Abbildung 3.30: *main.dart*

Wenn wir unseren Code neu ausführen haben wir einen Anmeldebildschirm auf Deutsch:

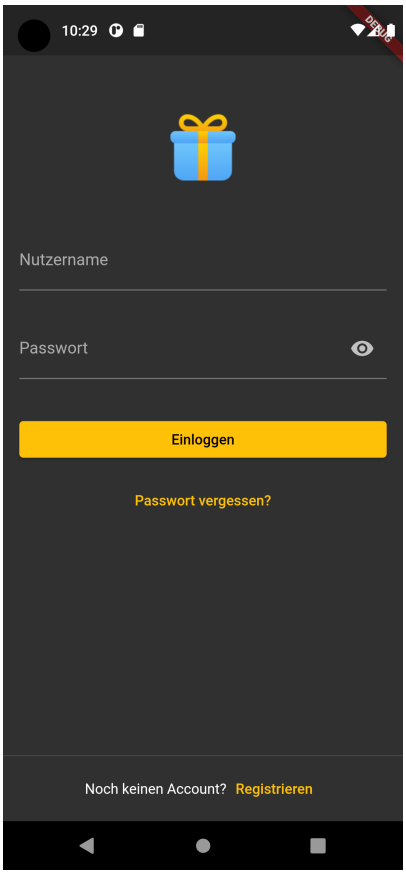


Abbildung 3.31: Anmeldebildschirm

Nach dem gleichen Prinzip lassen sich die restlichen Authentifizierungsbildschirme anpassen. Den aktuellen Code findet Ihr hier.

4. Authentifizierung Teil 2

In diesem Kapitel registrieren wir einen Benutzer und finden ihn im Amplify Studio. Außerdem, fügen wir eine Log Out Funktion hinzu und nutzen die Attribute der Benutzer in unserer App.

4.1 Verwaltung in Amplify Studio

Startet eure App und Registriert einen neuen Benutzer. Ihr erhaltet eine Email um euch zu verifizieren. Gebt den Verifizierungscode in eurer App ein. Nun seht ihr unseren bisher sehr leeren Home Screen. Bevor wir daran etwas ändern starten wir nochmal Amplify Studio. Hier befindet sich unter dem Reiter **User management** der gerade angelegte Benutzer:

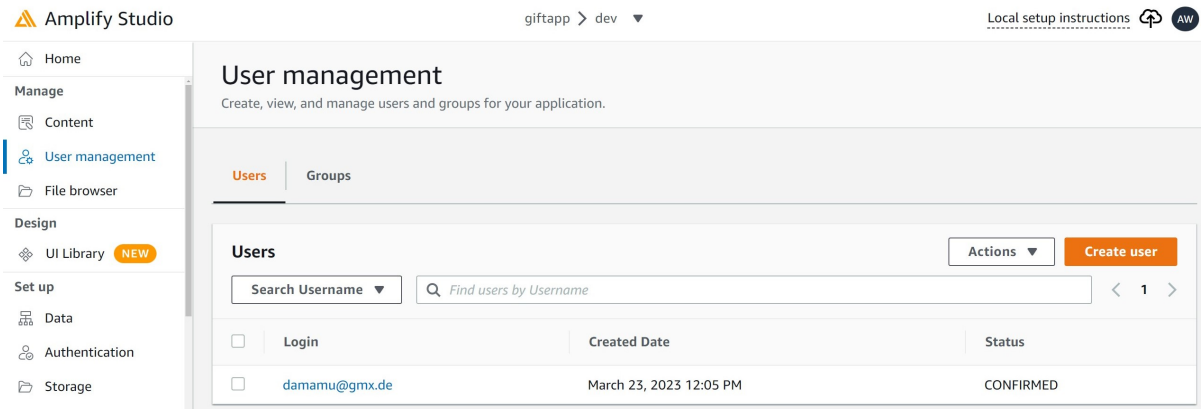


Abbildung 4.1: Amplify Studio

Außerdem könnt ihr hier auch manuell neue Benutzer erstellen oder eure existierenden Benutzer verwalten.

Mit einem Klick auf den Benutzer landet ihr auf seinem Profil. Hier seht ihr Dinge wie den Benutzernamen und den Verifizierungsstatus. Desweiteren könnt ihr diesen speziellen Benutzer einzeln bearbeiten.

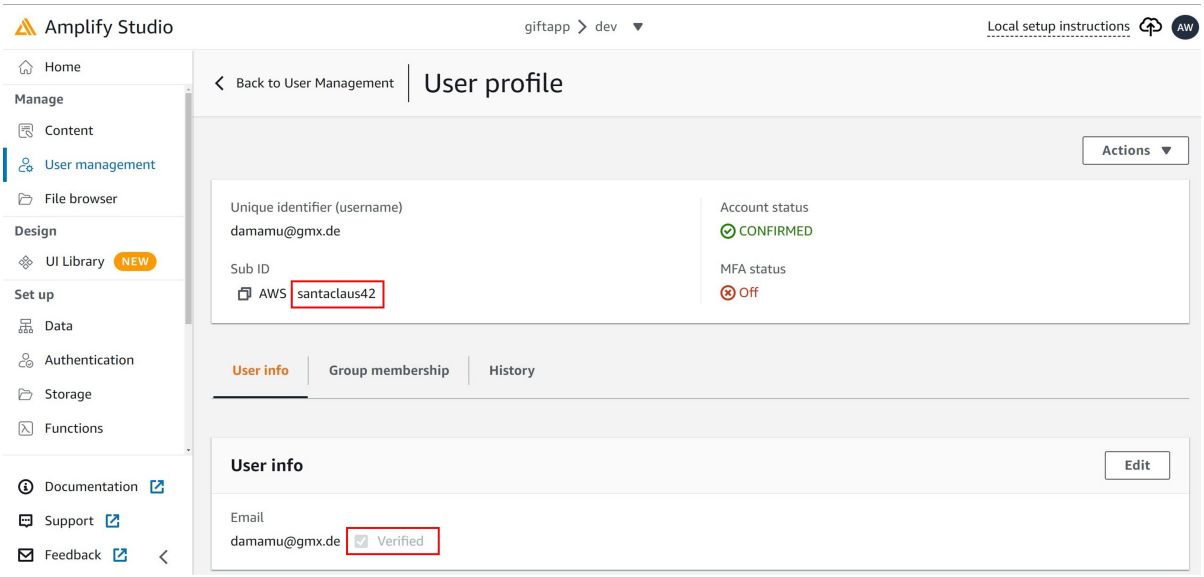


Abbildung 4.2: Amplify Studio

4.2 Log Out

Nun sind wir zwar mit einem Benutzer eingeloggt, haben aber noch keine Möglichkeit uns wieder abzumelden. Wir kehren wieder zu Visual Studio Code zurück und legen uns zuerst 2 neue Ordner und 2 neue Dateien an:

- *lib/views/home.dart*
- *lib/constants/routes.dart*

Unsere neue Ordnerstruktur sieht wie folgt aus:

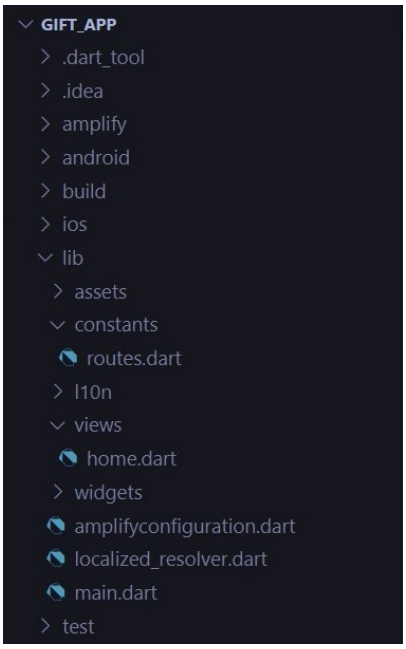


Abbildung 4.3: Ordnerstruktur

In *home.dart* definieren wir das Aussehen und die Funktionalitäten unseres Home Screens. Wir beginnen mit einem Titel und einem Icon um den Nutzer auszuloggen:

```dart
import 'package:flutter/material.dart';

class HomeView extends StatefulWidget {
  const HomeView({super.key});

  @override
  State<HomeView> createState() => _HomeViewState();
}

class _HomeViewState extends State<HomeView> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Home'),
        centerTitle: true,
        actions: [
          IconButton(
            onPressed: () {},
            icon: const Icon(Icons.logout),
          ), // IconButton
        ],
      ), // AppBar
    ); // Scaffold
  }
}
```

Abbildung 4.4: *home.dart*

Danach legen wir uns in *routes.dart* einen String für unseren Home Screen an:

```dart
const homeRoute = "/home";
```

Abbildung 4.5: *routes.dart*

In *main.dart* fügen wir die Route hinzu und legen unseren Home Screen als Startbildschirm fest:

```
66            theme: ThemeData(
67              brightness: Brightness.dark,
68              primarySwatch: Colors.amber,
69            ), // ThemeData
70            routes: {
71              homeRoute: (context) => const HomeView(),
72            },
73            home: const HomeView(),
74          ), // MaterialApp
75        ); // Authenticator
76      }
77    }
```

Abbildung 4.6: *main.dart*

Nun sieht unser Startbildschirm wie folgt aus:

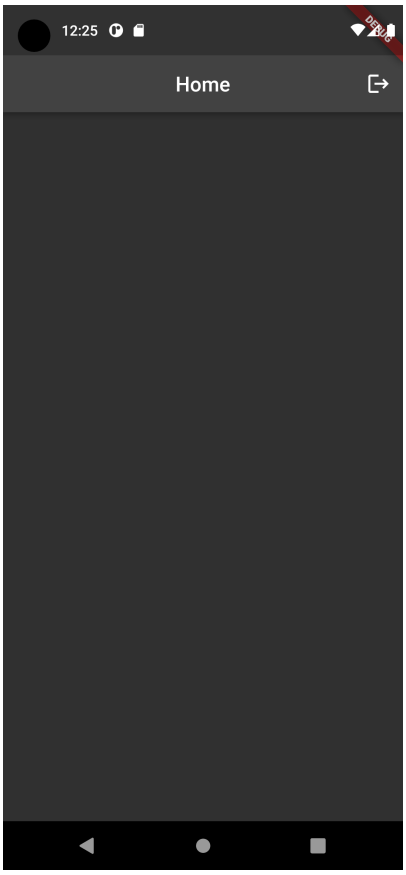


Abbildung 4.7: Startbildschirm

Allerdings erfüllt unser Log Out Button noch keine Funktion. Dafür öffnen wir erneut *home.dart* und fügen einen neuen Import hinzu:

```dart
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
```

Abbildung 4.8: *home.dart*

Danach fügen wir die Funktion *signOutCurrentUser()* hinzu. Diese benutzt die von Amplify bereitgestellte Funktion *signOut()* um den Nutzer auszuloggen. Wichtig sind hier die Schlüsselwörter async und await, da das Ausloggen des Nutzers eine kurze Zeit in Anspruch nimmt.

```dart
class _HomeViewState extends State<HomeView> {
  Future<void> signOutCurrentUser() async {
    try {
      await Amplify.Auth.signOut();
    } on AuthException catch (e) {
      safePrint(e.message);
    }
  }
```

Abbildung 4.9: *home.dart*

Nun fügen wir die Funktion noch unserem Button hinzu:

```dart
            IconButton(
              onPressed: signOutCurrentUser,
              icon: Icon(Icons.logout),
            ), // IconButton
```

Abbildung 4.10: *home.dart*

Jetzt können wir uns An- und Abmelden.

4.3 Benutzerattribute verwenden

Wir können außerdem die Attribute des Nutzers wie Benutzername, Emailadresse usw. in unserer App verwenden. In diesem Beispiel soll der Nickname auf dem Home Screen in der oberen linken Ecke angezeigt werden. Dafür fügen wir *home.dart* eine neue Variable und Funktion hinzu:

```
11  class _HomeViewState extends State<HomeView> {
12    late String nickname;
13
14    Future<void> getNickname() async {
15      try {
16        final attributes = await Amplify.Auth.fetchUserAttributes();
17        for (final element in attributes) {
18          if (element.userAttributeKey == CognitoUserAttributeKey.nickname) {
19            nickname = element.value;
20          }
21        }
22      } on AuthException catch (e) {
23        safePrint(e);
24      }
25    }
```

Abbildung 4.11: *home.dart*

Durch die Amplify Funktion *fetchUserAttributes()* erhalten wie eine Liste mit allen Benutzerattributen. Durch den *userAttributKey* können wir uns dann das gesuchte Attribut herausfiltern.

Unsere *build()* Funktion müssen wir ebenfalls ein bisschen anpassen. Da wir mit dem Anzeigen des Home Screens warten müssen, bis der Nickname gelesen wurde, benötigen wir einen FuturBuilder. Dieser sorgt dafür, dass unser Scaffold erst angezeigt wird, wenn die Funktion *getNickname()* ausgeführt wurde:

```dart
@override
Widget build(BuildContext context) {
  return FutureBuilder(
    builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {
      if (snapshot.connectionState == ConnectionState.done) {
        // Bisheriger Code
        return Scaffold(
          appBar: AppBar(
            leading: Center(
              child: Text(nickname),
            ), // Center
            title: const Text('Home'),
            centerTitle: true,
            actions: [
              IconButton(
                onPressed: signOutCurrentUser,
                icon: const Icon(Icons.logout),
              ), // IconButton
            ],
          ), // AppBar
        ); // Scaffold
      }
      // Ende Bisheriger Code
      return const Center(
        child: CircularProgressIndicator(),
      ); // Center
    },
    future: getNickname(),
  ); // FutureBuilder
}
}
```

Abbildung 4.12: *home.dart*

In der linken oberen Ecke wird nun der Nickname angezeigt:

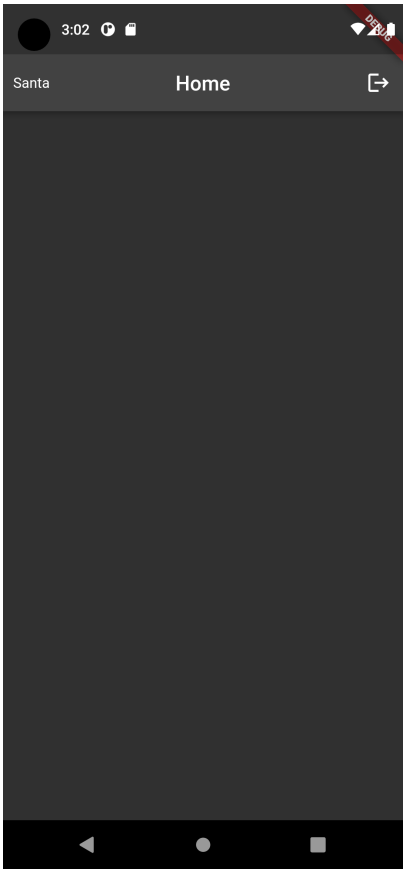


Abbildung 4.13: Startbildschirm

Den aktuellen Code findet Ihr hier.

5. AWS Datastore

In diesem Kapitel nutzen wir Amplify Studio um Datenrelationen zu erstellen, diese in unserer App zu verwenden und Operationen auf diesen Relationen auszuführen.

5.1 Datenmodell erstellen

Wir öffnen Amplify Studio und gehen auf den Reiter **Data**:

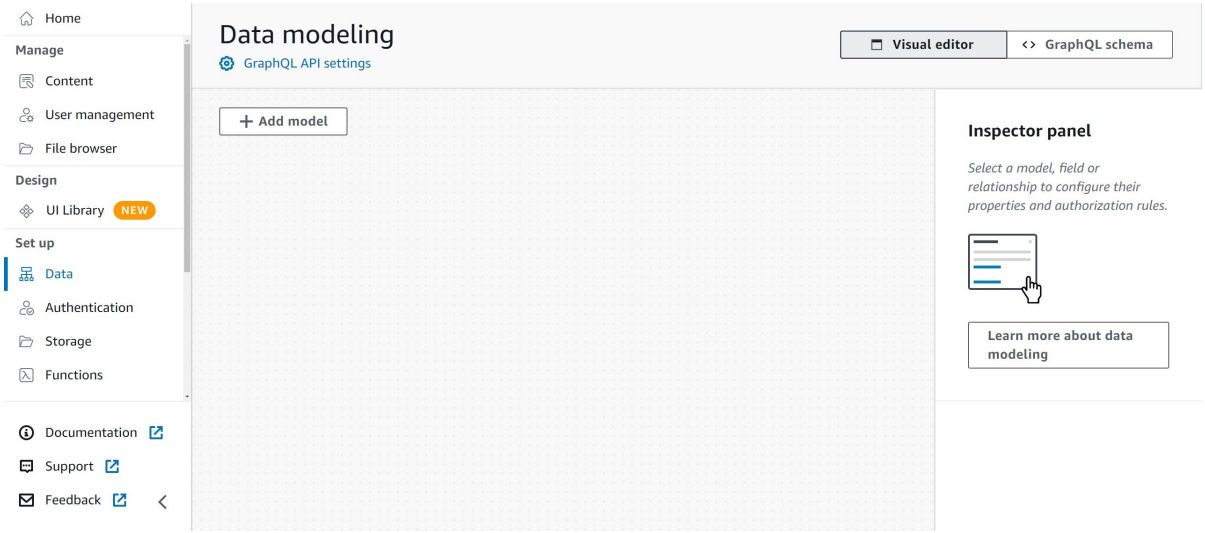


Abbildung 5.1: Amplify Studio

In diesem Bereich können wir Models für die Daten erstellen, die unsere App verwenden soll. Durch den Klick auf **Add model** erscheint ein neues Schema, dass wir nach belieben ausfüllen können. Für unser Beispiel fügen wir ein Model für Personen hinzu, die wir beschenken wollen:

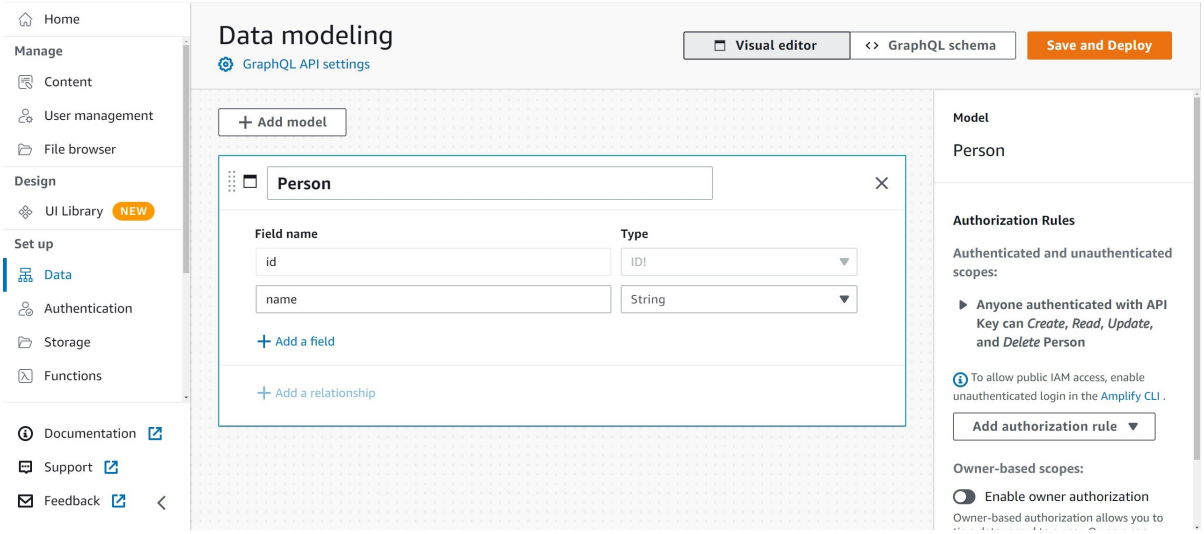


Abbildung 5.2: Amplify Studio

Eine Person hat eine ID, welche in jedem Model benötigt wird und einen Namen vom Typ String. Indem wir ein Feld des Models auswählen, können wir seine Eigenschaften ändern. Name soll bei uns ein requiered Feld sein:

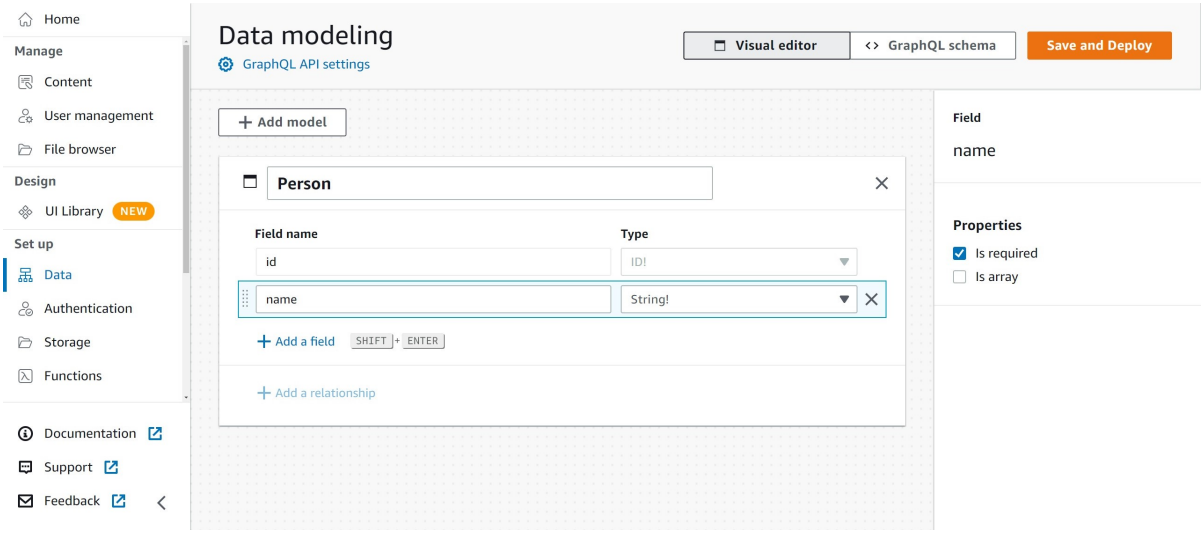


Abbildung 5.3: Amplify Studio

Als nächstes fügen wir eine zweite Relation für unsere Geschenke hinzu. Diese haben ebenfalls einen Namen vom Typ String und zusätzlich einen Bool Wert, um Geschenke später abhaken zu können:

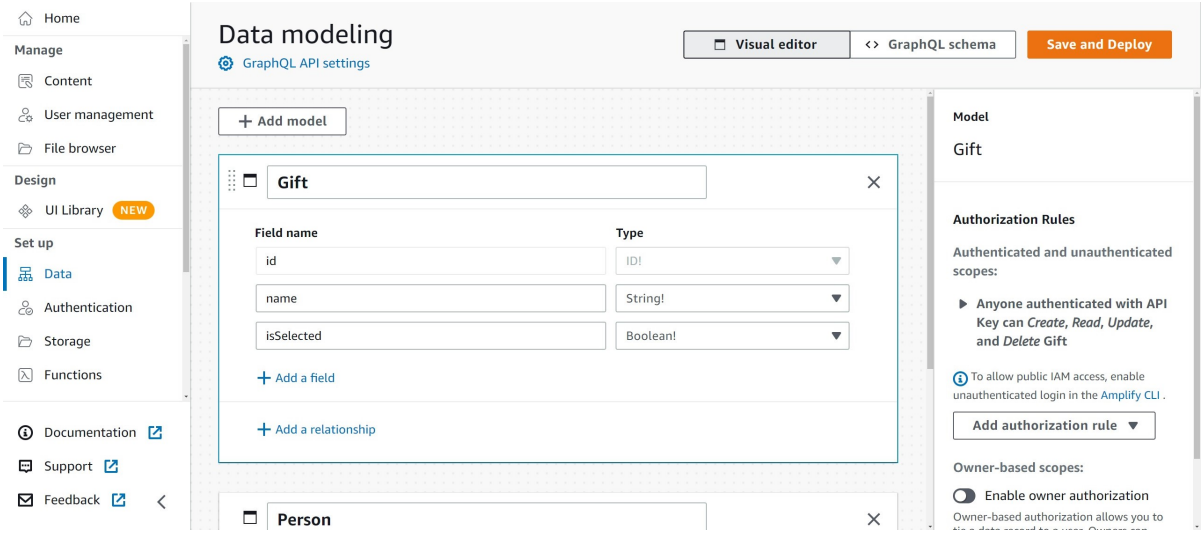


Abbildung 5.4: Amplify Studio

Nun fügen wir eine Beziehung zwischen den beiden Models hinzu. Dafür klicken wir wir auf **Add a relationship**:

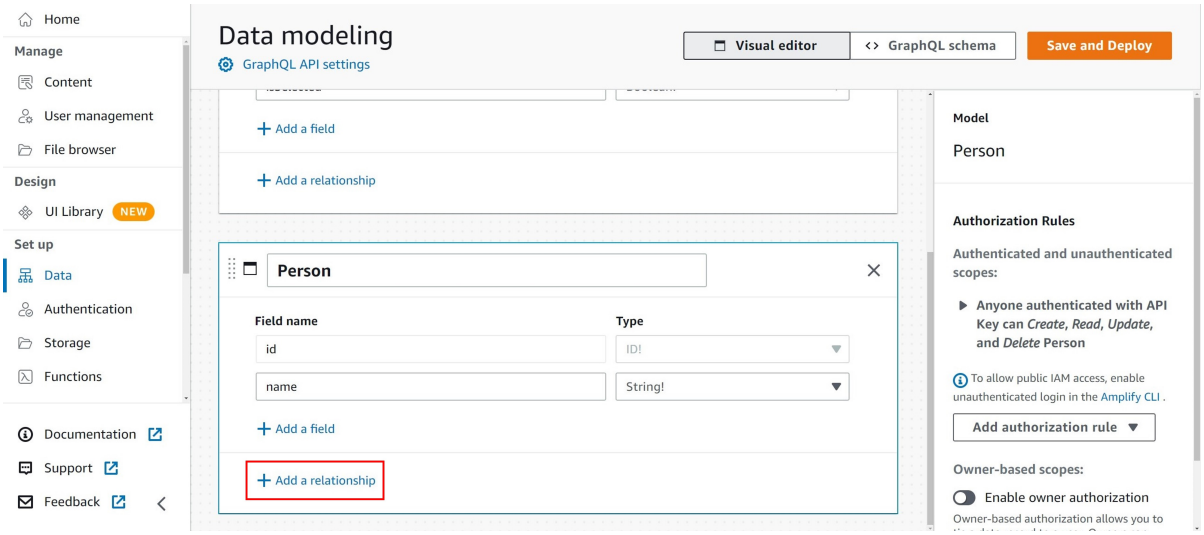


Abbildung 5.5: Amplify Studio

Ein neues Fenster öffnet sich, in dem wir ein Model und die Art der Beziehung auswählen können. Der Beziehungsname muss einen andere Namen haben als die Models selbst, da es in Flutter sonst zu Problemen kommt. Schreibt ihn am besten klein oder ändert ihn.

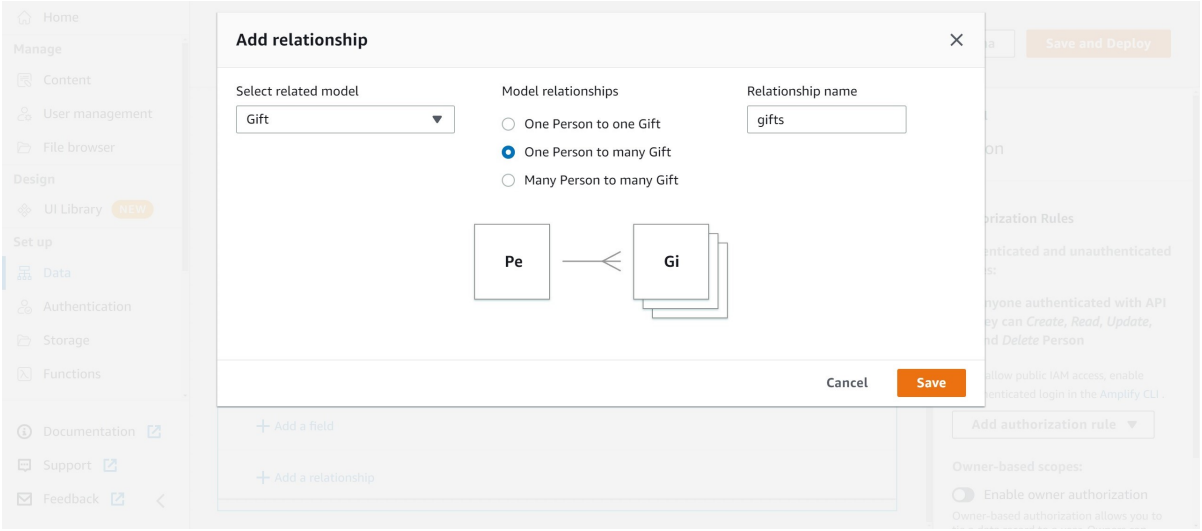


Abbildung 5.6: Amplify Studio

Das gleiche machen wir auch mit unserem zweiten Model:

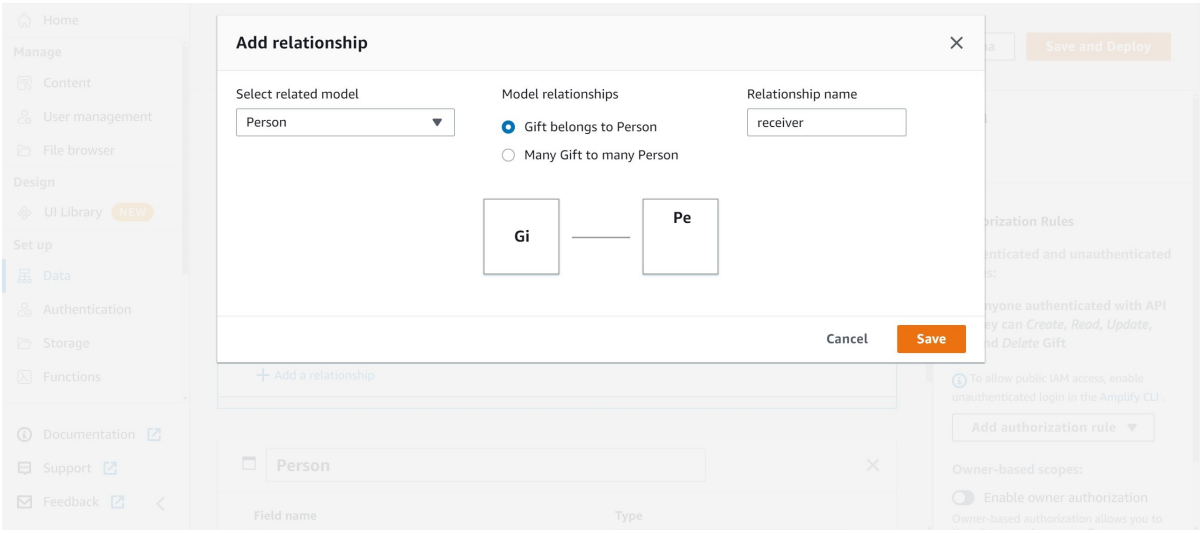


Abbildung 5.7: Amplify Studio

Neben der Ansicht im Editor können die Models auch als GraphQL Schema betrachtet werden. Um die Models in unserer App verwenden zu können klicken wir auf **Save and Deploy**:

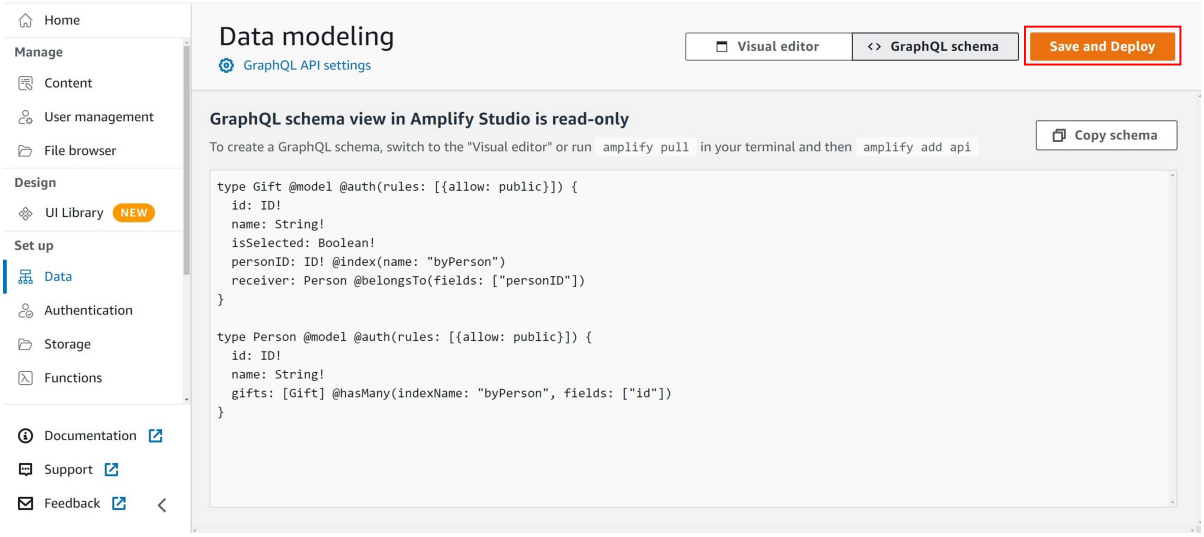


Abbildung 5.8: Amplify Studio

Nach kurzer Wartezeit ist die Bereitstellung abgeschlossen. Nun müsst ihr nur noch den euch angezeigten Befehl im Verzeichnes eures Projekts ausführen.

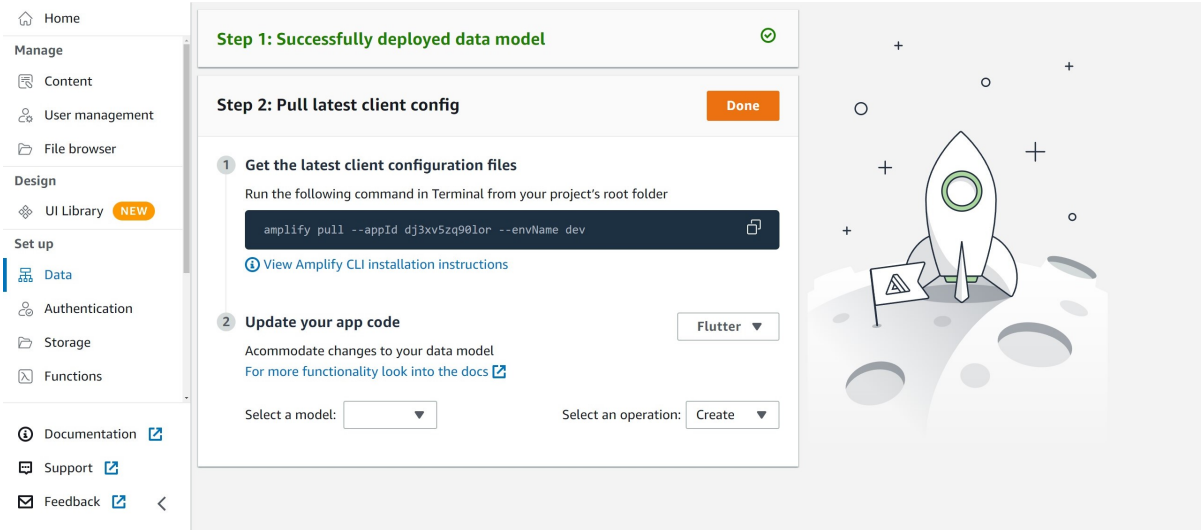


Abbildung 5.9: Amplify Studio

5.2 Datenmodell in der App verwenden

Wenn wir uns unserer Ordnerstruktur in Visual Studio Code anschauen, stellen wir fest, dass 3 neue Dateien erstellt wurden sind:

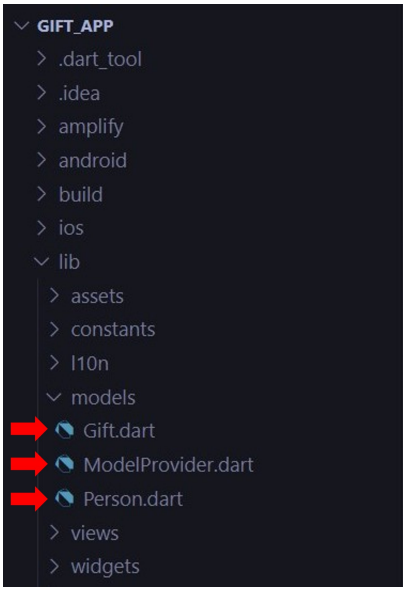


Abbildung 5.10: Ordnerstruktur

In *Gift.dart* und *Person.dart* befindet sich der an Flutter angepasste Code unserer beiden Models. In *ModelProvider.dart* befindet sich, wie der Name schon sagt, ein Model Provider, durch den wir die beiden Models verwenden können.

Um den Amplify Datastore verwenden zu können, müssen wir zuerst neue Abhängigkeiten in *pubspec.yaml* hinzufügen:

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter


  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  amplify_flutter: ^0.6.0
  amplify_auth_cognito: ^0.6.0
  amplify_authenticator: ^0.2.0
  intl: ^0.17.
  amplify_datastore: ^0.6.0
  amplify_api: ^0.6.0

```

Abbildung 5.11: *pubspec.yaml*

Falls ihr in eurer App nur einen lokalen Speicher ohne Cloud Service verwenden wollt, müsst ihr amplify_api nicht hinzufügen.

Als nächstes müssen wir noch eine Kleinigkeit in den Android-Einstellungen verändern, um Java8 features, die im Datastore Plugin verwendet werden zu unterstützen. Dafür öffnet ihr *android/app/build.gradle* und fügt 2 Zeilen Code hinzu:

```
android {
    compileSdkVersion flutter.compileSdkVersion
    ndkVersion flutter.ndkVersion

    compileOptions {
        // fügt diese Zeile hinzu
        coreLibraryDesugaringEnabled true

        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
```

Abbildung 5.12: *android/app/build.gradle*

```
dependencies {
    //fügt diese Zeile hinzu
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'

    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
}
```

Abbildung 5.13: *android/app/build.gradle*

Abschließend müssen wir noch unserer *configureAmplify()* Funktion in *main.dart* anpassen. Dazu fügen wir zuerst 3 neue Imports hinzu:

```dart
import 'package:amplify_auth_cognito/amplify_auth_cognito.dart';
import 'package:amplify_authenticator/amplify_authenticator.dart';
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/views/home.dart';
import 'package:gift_app/widgets/auth_widgets.dart';
import 'package:flutter_localizations/flutter_localizations.dart';
import 'package:gift_app/localized_resolver.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart';
import 'amplifyconfiguration.dart';
import 'constants/routes.dart';
import 'package:amplify_api/amplify_api.dart';
import 'package:amplify_datastore/amplify_datastore.dart';
import 'package:gift_app/models/ModelProvider.dart';
```

Abbildung 5.14: *main.dart*

und fügen die neuen Plugins unserer Funktion hinzu:

```dart
void _configureAmplify() async {
  try {
    final authPlugin = AmplifyAuthCognito();
    final apiPlugin = AmplifyAPI();
    final datastorePlugin = AmplifyDataStore(modelProvider: ModelProvider.instance);
    await Amplify.addPlugins([authPlugin, apiPlugin, datastorePlugin]);
    await Amplify.configure(amplifyconfig);
  } on Exception catch (e) {
    safePrint(e);
  }
}
```

Abbildung 5.15: *main.dart*

5.3 Operationen ausführen

In diesem Kapitel verwenden wir die verschiedenen auf dem Datastore ausführbaren Operationen. Doch vorher müssen wir ein bisschen Vorarbeit leisten. Wir benötigen 3 neue Screens,

- zum Erstellen neuer zu beschenkender Personen (*views/new_person.dart*)
- zum Erstellen neuer Geschenke (*views/new_gift.dart*)
- eine Liste aller einer Person zugeordneten Geschenke (*views/gift_list.dart*)

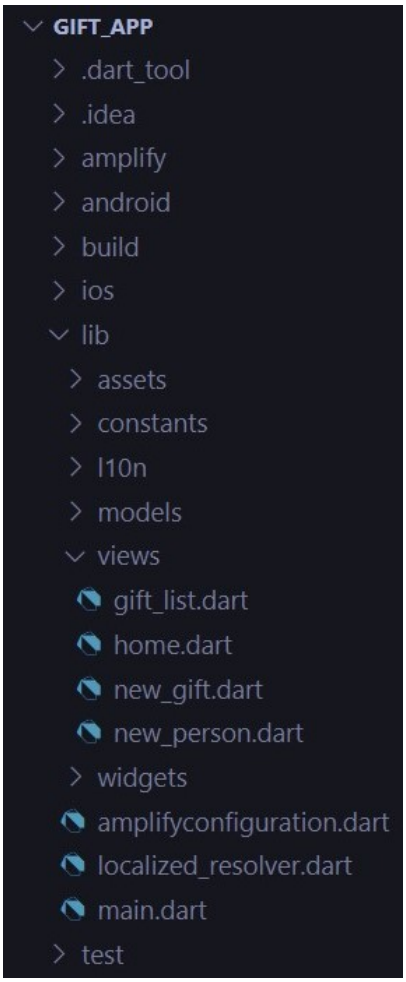


Abbildung 5.16: Ordnerstruktur

Um von unserem Home Screen auf die neuen Screens zu gelangen müssen wir unsere Routen aktualisiern:

```
1  const homeRoute = "/home";
2  const newPersonRoute = "/home/newPerson";
3  const giftListRoute = "/home/giftList";
4  const newGiftRoute = "/home/newGift";
```

Abbildung 5.17: *routes.dart*

```
4  import 'package:flutter/material.dart';
5  import 'package:gift_app/views/gift_list.dart';
6  import 'package:gift_app/views/home.dart';
7  import 'package:gift_app/views/new_gift.dart';
8  import 'package:gift_app/views/new_person.dart';
9  import 'package:gift_app/widgets/auth_widgets.dart';
```

Abbildung 5.18: *main.dart*

```
78          routes: {
79            homeRoute: (context) => const HomeView(),
80            newPersonRoute: (context) => const NewPersonView(),
81            newGiftRoute: (context) => const NewGiftView(),
82            giftListRoute: (context) => const GiftListView(),
83          },
```

Abbildung 5.19: *main.dart*

Danach fügen wir unserem Home Screen einen Button hinzu, um auf den New Person Screen zu gelangen:

```
54              ), // AppBar
55              floatingActionButton: FloatingActionButton(
56                onPressed: () {
57                  Navigator.of(context).pushNamed(newPersonRoute);
58                },
+ 59              child: const Icon(Icons.add),
60              ), // FloatingActionButton
```

Abbildung 5.20: *home.dart*

In der Datei *new_person.dart* erstellen wir unsere *NewPersonView*. Diese besitzt einen kurzen Text, ein Textfeld, mit einem TextEditingController um den eingegeben Text verwenden zu können, und einen Button:

```dart
import 'package:flutter/material.dart';

class NewPersonView extends StatefulWidget {
  const NewPersonView({super.key});

  @override
  State<NewPersonView> createState() => _NewPersonViewState();
}

class _NewPersonViewState extends State<NewPersonView> {
  late TextEditingController _nameController;

  @override
  void initState() {
    _nameController = TextEditingController();
    super.initState();
  }

  @override
  void dispose() {
    _nameController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Person Hinzufügen'),
        centerTitle: true,
      ), // AppBar
      body: Column(
        children: [
          const SizedBox(
            height: 40,
          ), // SizedBox
```

Abbildung 5.21: *new_person.dart*

```
            _description(),
            const SizedBox(
              height: 40,
            ), // SizedBox
            _textField(),
            const SizedBox(
              height: 200,
            ), // SizedBox
            _button(),
          ],
        ), // Column
      ); // Scaffold
    }

    Widget _description() {
      return const Center(
        child: Text(
          "Trage eine neue Person ein, die du beschenken möchtest!",
          style: TextStyle(fontSize: 20),
          textAlign: TextAlign.center,
        ), // Text
      ); // Center
    }

    Widget _textField() {
      return Container(
        margin: const EdgeInsetsDirectional.all(8),
        child: TextField(
          controller: _nameController,
          decoration: const InputDecoration(
            hintText: 'Name',
            border: OutlineInputBorder(),
          ), // InputDecoration
        ), // TextField
      ); // Container
    }

    Widget _button() {
      return ElevatedButton(
        onPressed: () {},
        child: const Text('Hinzufügen'),
      ); // ElevatedButton
    }
  }
```

Abbildung 5.22: *new_person.dart*

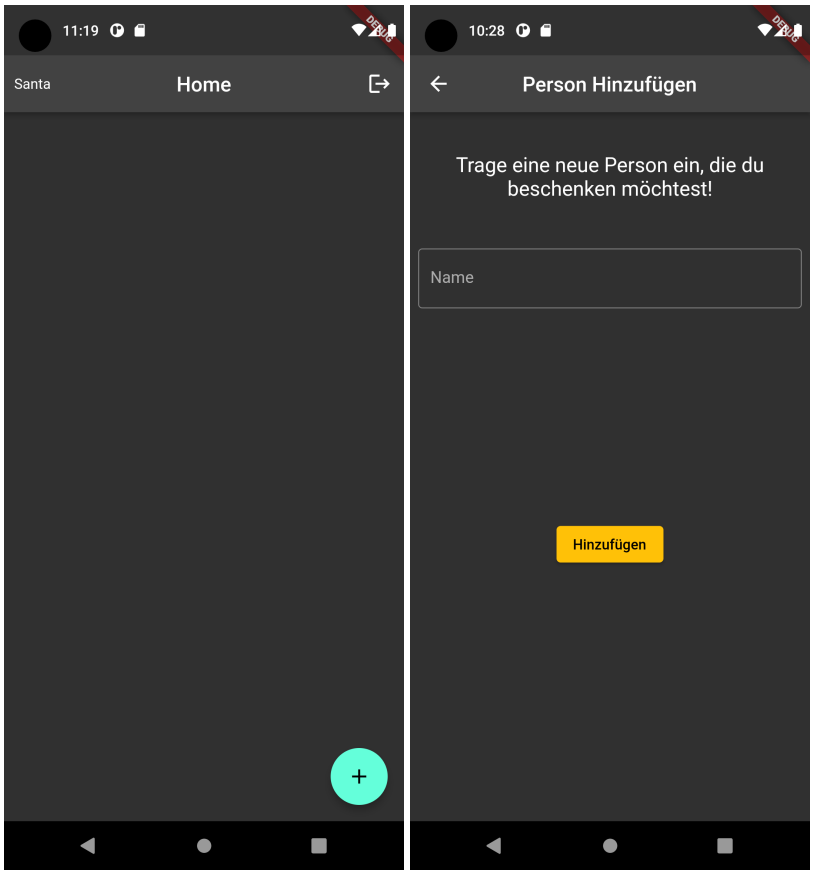


Abbildung 5.23: Person Hinzufügen

Nun können wir uns um das abspeichern einer neuer Person in unserer Datenbank kümmern. Dafür importieren wir zuerst 2 neue Pakete:

```dart
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/models/ModelProvider.dart';
```

Abbildung 5.24: *new_person.dart*

Danach fügen wir *NewPersonView* die Funktion *saveNewPerson()* hinzu. Diese erstellt ein neues Objekt Person und speichert dieses dann mithilfe der Amplify Funktion *save()* in der Datenbank:

```
15    Future<void> saveNewPerson(String name) async {
16      final newPerson = Person(
17        name: name,
18      );
19      try {
20        await Amplify.DataStore.save(newPerson);
21      } on DataStoreException catch (e) {
22        safePrint(e);
23      }
24    }
```

Abbildung 5.25: *new_person.dart*

Nun fügen wir unserem Button die neue Funktion hinzu. Wenn das Textfeld nicht leer ist, speichern wir die Person und und gehen zurück zum Home Screen:

```
87    Widget _button() {
88      return ElevatedButton(
89        onPressed: () async {
90          if (_nameController.text.isNotEmpty) {
91            await saveNewPerson(_nameController.text);
92          }
93          if (mounted) {
94            Navigator.of(context).pop();
95          }
96        },
97        child: const Text('Hinzufügen'),
98      ); // ElevatedButton
99    }
100 }
```

Abbildung 5.26: *new_person.dart*

Hier werden uns die neuen Personen allerdings noch nicht angezeigt. Um das zu verändern fügen wir *home.dart* zuerst 2 neue Imports hinzu:

```
1  import 'package:amplify_flutter/amplify_flutter.dart';
2  import 'package:flutter/material.dart';
3  import 'package:gift_app/constants/routes.dart';
4  import 'package:gift_app/models/ModelProvider.dart';
5  import 'dart:async';
```

Abbildung 5.27: *home.dart*

Wir benötigen eine Liste von allen Personen die wir anzeigen wollen und eine StreamSubscription. Zweitere ermöglicht es uns, an die Inhalte der Datenbank aus Personen zu gelangen und bei Veränderungen sofort unsere lokale Liste von Personen zu aktualisieren:

```
14 class _HomeViewState extends State<HomeView> {
15   late String nickname;
16   List<Person> _persons = [];
17   late StreamSubscription<QuerySnapshot<Person>> _subscription;
```

Abbildung 5.28: *home.dart*

Unsere StreamSubscription aktualisiert unsere Liste von Personen wenn es eine Veränderung in der Datenbank gibt:

```
40   @override
41   void initState() {
42     _subscription = Amplify.DataStore.observeQuery(Person.classType).listen((
43       QuerySnapshot<Person> snapshot,
44     ) {
45       setState(() {
46         _persons = snapshot.items;
47       });
48     });
49     super.initState();
50   }
51
52   @override
53   void dispose() {
54     _subscription.cancel();
55     super.dispose();
56   }
```

Abbildung 5.29: *home.dart*

Nun erstellen wir noch ein neues Widget *personList()*, um unserer Liste _persons auf dem Bildschirm anzuzeigen:

```
94    Widget _personList() {
95      return Column(
96        children: [
97          Expanded(
98            child: ListView.builder(
99              itemCount: _persons.length,
100             itemBuilder: ((context, index) {
101               return ListTile(
102                 title: Text(_persons[index].name),
103                 trailing: IconButton(
104                   onPressed: () {},
105                   icon: const Icon(Icons.delete),
106                 ), // IconButton
107                 shape: const RoundedRectangleBorder(
108                   side: BorderSide(
109                     color: Colors.black38,
110                     width: 1,
111                   ), // BorderSide
112                 ), // RoundedRectangleBorder
113                 onTap: () {},
114               ); // ListTile
115             }),
116           ), // ListView.builder
117         ), // Expanded
118       ],
119     ); // Column
120   }
121 }
```

Abbildung 5.30: *home.dart*

und fügen es unserem Scaffold hinzu:

```
76              ), // AppBar
77              body: _personList(),
```

Abbildung 5.31: *home.dart*

Neu angelegte Personen werden uns nun auf dem Home Screen angezeigt:

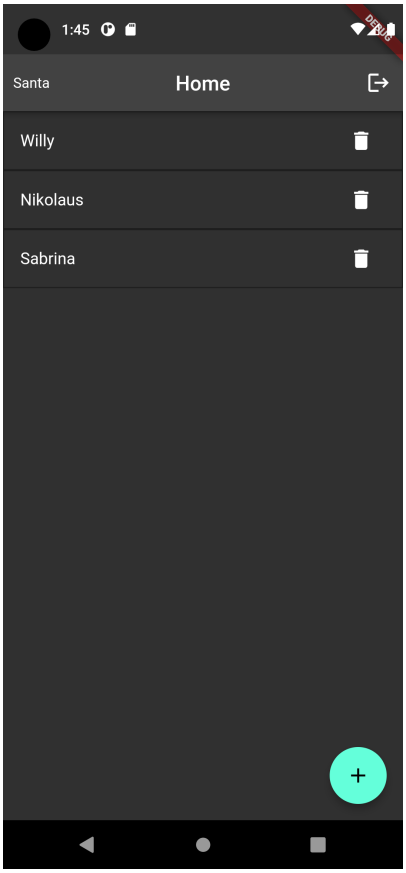


Abbildung 5.32: Startbildschirm

Jetzt können wir zwar Personen erstellen, allerdings noch nicht löschen. Dafür legen wir uns die neue Methode *deletePerson()* an. Diese nutzt die Amplify Funktion *delete()*:

```
40  Future<void> deletePerson(Person person) async {
41    try {
42      await Amplify.DataStore.delete(person);
43    } catch (e) {
44      safePrint(e);
45    }
46  }
```

Abbildung 5.33: *home.dart*

Diese Funktion fügen wir unserem IconButton hinter jedem Namen hinzu und übergeben ihr die aktuelle Person:

```
111                 trailing: IconButton(
112                   onPressed: () {
113                     deletePerson(_persons[index]);
114                   },
115                   icon: const Icon(Icons.delete),
116                 ), // IconButton
```

Abbildung 5.34: *home.dart*

Als nächstes wollen wir jeder Person Geschenke zuordnen können. Mit einem Klick auf eine Person soll sich ein neuer Screen öffnen, auf der alle dieser Person zugeordneten Geschenke angezeigt werden. Dafür verwenden wir wieder die *pushNamed()* Funktion. Wir fügen diese unseren ListTiles hinzu und übergeben ihr die geklickte Person:

```
123               onTap: () {
124                 Navigator.of(context).pushNamed(giftListRoute, arguments: _persons[index]);
125               },
```

Abbildung 5.35: *home.dart*

Da wir die aktuelle Person übergeben wollen müssen wir in *main.dart* unsere MaterialApp anpassen. Wir entfernen die Route zu GiftListView und fügen dafür ein neues Feld onGenerateRoute hinzu. Dort legen wir fest, dass wenn GiftListView aufgerufen wird, eine Person als Argument übergeben wird:

```
78      routes: {
79        homeRoute: (context) => const HomeView(),
80        newPersonRoute: (context) => const NewPersonView(),
81        newGiftRoute: (context) => const NewGiftView(),
82      },
83      onGenerateRoute: (settings) {
84        if (settings.name == giftListRoute) {
85          final args = settings.arguments as Person;
86          return MaterialPageRoute(
87            builder: (context) {
88              return GiftListView(
89                person: args,
90              ); // GiftListView
91            },
92          ); // MaterialPageRoute
93        }
94        return null;
95      },
```

Abbildung 5.36: *main.dart*

In *gift_list.dart* fügen wir nun den Code für die GiftListView ein. Anders als in den anderen Views ist diesmal, dass wir eine Person als Argument bekommen und diese in _person speichern. Der Name dieser Person wird dann als Titel in der Appbar angezeigt:

```
import 'dart:async';
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/constants/routes.dart';
import 'package:gift_app/models/ModelProvider.dart';

class GiftListView extends StatefulWidget {
  final Person person;
  const GiftListView({super.key, required this.person});

  @override
  State<GiftListView> createState() => _GiftListViewState();
}

class _GiftListViewState extends State<GiftListView> {
  late final Person _person = widget.person;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(_person.name),
        centerTitle: true,
        actions: [
          IconButton(
            onPressed: () {},
            icon: const Icon(Icons.add),
          ), // IconButton
        ],
      ), // AppBar
    ); // Scaffold
  }
```

Abbildung 5.37: *gift_list.dart*

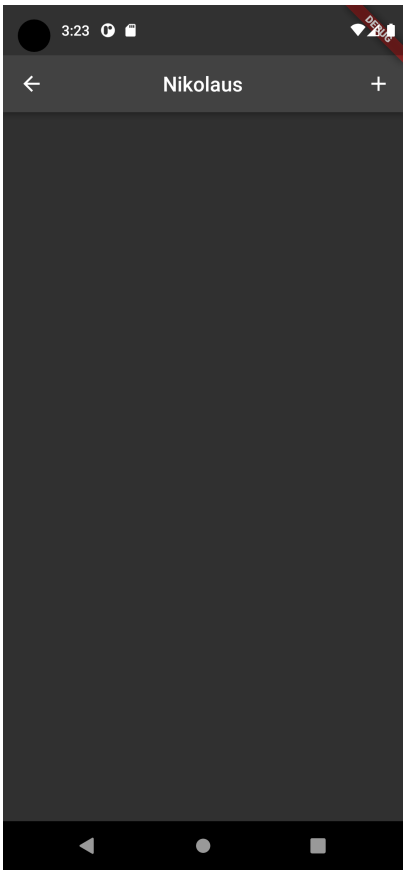


Abbildung 5.38: Eine neue Person

Um die Geschenke dieser Person anzuzeigen gehen wir genauso vor wie in HomeView mit der Liste der Personen. Wir legen uns zuerst wieder eine Liste aus Geschenken und eine StreamSubscription an:

```
15  class _GiftListViewState extends State<GiftListView> {
16    late final Person _person = widget.person;
17    List<Gift> _presents = [];
18    late StreamSubscription<QuerySnapshot<Gift>> _subscription;
19
```

Abbildung 5.39: *gift_list.dart*

Diesmal fügen wir beim befüllen unserer lokalen Geschenkeliste noch hinzu, dass die Geschenke der aktuellen Person hinzugefügt werden sollen. Das erreichen wir mit der *eq()* Funktion:

```
20    @override
21    void initState() {
22      _subscription = Amplify.DataStore.observeQuery(
23        Gift.classType,
24        where: Gift.RECEIVER.eq(_person.id),
25      ).listen((QuerySnapshot<Gift> snapshot) {
26        setState(() {
27          _presents = snapshot.items;
28        });
29      });
30      super.initState();
31    }
32
33    @override
34    void dispose() {
35      _subscription.cancel();
36      super.dispose();
37    }
```

Abbildung 5.40: *gift_list.dart*

Danach erstellen wir wieder ein Listen Widget, diesmal mit Checkboxen, die abgehakt werden, falls isSelected des Geschenks den Wert *true* besitzt:

```
56    Widget _giftList() {
57      return Column(
58        children: [
59          Expanded(
60            child: ListView.builder(
61              itemCount: _presents.length,
62              itemBuilder: (context, index) {
63                return CheckboxListTile(
64                  title: Text(_presents[index].name),
65                  value: _presents[index].isSelected,
66                  onChanged: (val) {},
67                ); // CheckboxListTile
68              },
69            ), // ListView.builder
70          ), // Expanded
71        ],
72      ); // Column
73    }
74  }
```

Abbildung 5.41: *gift_list.dart*

und fügen es der *build()* Funktion hinzu:

```
51          ), // AppBar
52          body: _giftList(),
53        ); // Scaffold
```

Abbildung 5.42: *gift_list.dart*

Bisher können wir allerdings noch keine neuen Geschenke erstellen. Dafür fügen wir zuerst unserem Add Button die Funktion hinzu, auf einen neuen Screen weiterzuleiten. Außerdem übergeben wir wieder die aktuelle Person:

```
46              IconButton(
47                onPressed: () {
48                  Navigator.of(context).pushNamed(newGiftRoute, arguments: _person);
49                },
+ 50              icon: const Icon(Icons.add),
51              ), // IconButton
```

Abbildung 5.43: *gift_list.dart*

Danach fügen wir in *new_gift.dart* folgenden Code ein (ähnlich zu *new_person.dart*):

```dart
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/models/ModelProvider.dart';

class NewGiftView extends StatefulWidget {
  const NewGiftView({super.key});

  @override
  State<NewGiftView> createState() => _NewGiftViewState();
}

class _NewGiftViewState extends State<NewGiftView> {
  late TextEditingController _nameController;

  @override
  void initState() {
    _nameController = TextEditingController();
    super.initState();
  }

  @override
  void dispose() {
    _nameController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Geschenk Hinzufügen'),
        centerTitle: true,
      ), // AppBar
      body: Column(
        children: [
          const SizedBox(
            height: 40,
          ), // SizedBox
```

Abbildung 5.44: *new_gift.dart*

```
            _icon(),
            const SizedBox(
              height: 40,
            ), // SizedBox
            _description(),
            const SizedBox(
              height: 40,
            ), // SizedBox
            _textField(),
            const SizedBox(
              height: 50,
            ), // SizedBox
            _button(),
          ],
        ), // Column
      ); // Scaffold
    }

    Widget _icon() {
      return Center(
        child: Image.asset(
          "lib/assets/present.png",
          scale: 1.5,
        ), // Image.asset
      ); // Center
    }

    Widget _description() {
      return const Center(
        child: Text(
          "Trage eine neue Geschenkidee ein!",
          style: TextStyle(fontSize: 20),
          textAlign: TextAlign.center,
        ), // Text
      ); // Center
    }
```

Abbildung 5.45: *new_gift.dart*

```dart
  Widget _textField() {
    return Container(
      margin: const EdgeInsetsDirectional.all(8),
      child: TextField(
        controller: _nameController,
        decoration: const InputDecoration(
          hintText: 'Geschenk',
          border: OutlineInputBorder(),
        ), // InputDecoration
      ), // TextField
    ); // Container
  }

  Widget _button() {
    return ElevatedButton(
      onPressed: () {},
      child: const Text('Hinzufügen'),
    ); // ElevatedButton
  }
}
```

Abbildung 5.46: *new_gift.dart*

Der neue Screen sieht nun wie folgt aus:

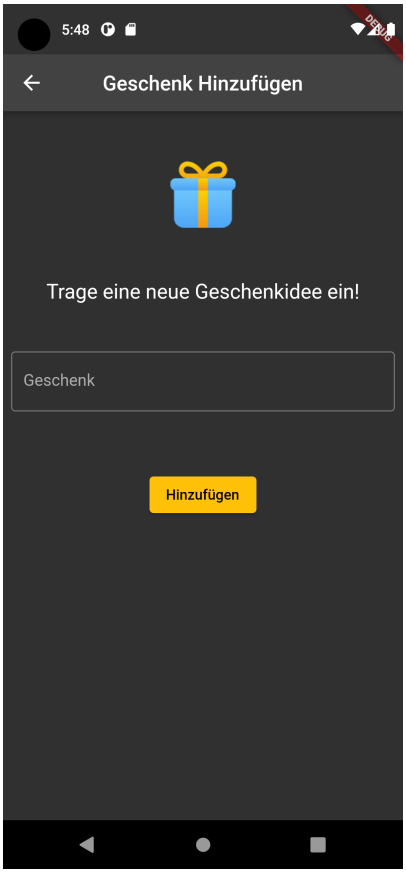


Abbildung 5.47: Neues Geschenk erstellen

Da wir wieder wissen müssen, zu welcher Person wir Geschenke hinzufügen, benötigen wir eine Variable _person:

```
12  class _NewGiftViewState extends State<NewGiftView> {
13    late TextEditingController _nameController;
14    late final Person _person;
15
```

Abbildung 5.48: *new_gift.dart*

Als nächstes fügen wir die Funktion *saveGift()* hinzu, um Geschenke speichern zu können. Wir erstellen ein Objekt vom Typ Geschenk mit Name, isSelected und einem Objekt vom Typ Person (unsere aktuelle Person). Dann nutzen wir wieder die Amplify Funktion *save()* um das neue Geschenk zu speichern:

```
16    Future<void> saveGift(String name) async {
17      final newGift = Gift(
18        name: name,
19        isSelected: false,
20        receiver: _person,
21      );
22      try {
23        await Amplify.DataStore.save(newGift);
24      } catch (e) {
25        safePrint('Fehler beim speichern: $e');
26      }
27    }
```

Abbildung 5.49: *new_gift.dart*

Da wir die Funktion *saveGift()* erst in unserer *build()* Funktion aufrufen, reicht es hier, anders als in *gift_list.dart*, die Variable _person erst in der *build()* Funktion zu belegen. Daher können wir folgenden, weniger umständlichen Code benutzen:

```
41    @override
42    Widget build(BuildContext context) {
43      _person = ModalRoute.of(context)!.settings.arguments as Person;
44      return Scaffold(
```

Abbildung 5.50: *new_gift.dart*

Nun fügen wir noch die neue Funktion unserem Button hinzu:

```
104   Widget _button() {
105     return ElevatedButton(
106       onPressed: () async {
107         if (_nameController.text.isNotEmpty) {
108           saveGift(_nameController.text);
109         }
110         if (mounted) {
111           Navigator.of(context).pop();
112         }
113       },
114       child: const Text('Hinzufügen'),
115     ); // ElevatedButton
116   }
117 }
```

new_gift.dart

Jetzt können wir Geschenke zu bestimmten Personen erstellen:

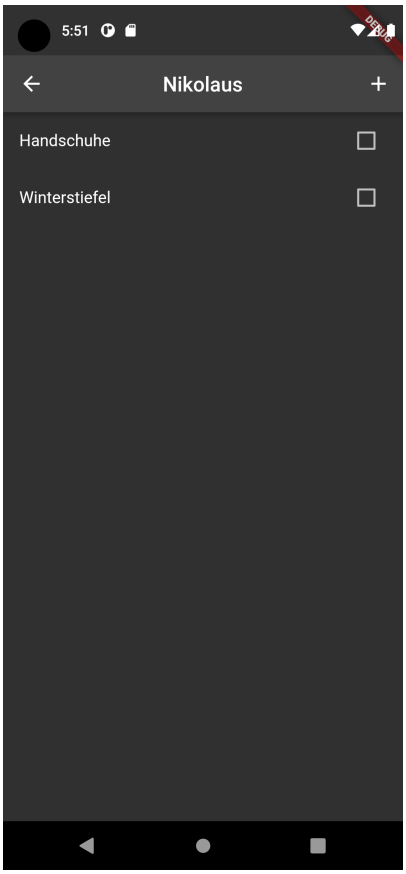


Abbildung 5.51: Geschenkeliste

Als nächstes wollen wir Geschenke abhaken können. Dafür müssen wir den isSelected Wert bei einem Klick auf das Kästchen verändern. Dafür erstellen wir die Funktion *changeIsSelected()* in *gift_list.dart*. Hier nutzen wir zuerst die Amplify Methode *query()* um Anhand der ID das aktuelle Geschenk in giftWithId zu speichern. Da *query()* theoretisch mehrere Objekte zurückgeben kann, versichern wir uns mit giftWithId.first, dass wir nur ein Element verändern. Danach erstellen wir ein neues Geschenk, welches eine Kopie von dem alten ist nur mit einem veränderten isSelected Wert und speichern es mit *save()* ab.

```
20    Future<void> changeIsSelected(id) async {
21      try {
22        final giftWithId = await Amplify.DataStore.query(
23          Gift.classType,
24          where: Gift.ID.eq(id),
25        );
26        final oldGift = giftWithId.first;
27        final newGift = oldGift.copyWith(isSelected: !oldGift.isSelected);
28        await Amplify.DataStore.save(newGift);
29      } catch (e) {
30        safePrint(e);
31      }
32    }
```

Abbildung 5.52: *gift_list.dart*

TIPP

Wenn ein Objekt mit gleicher ID wie ein bereits vorhandenes gespeichert wird, wird das alte durch das neue ersetzt

Nun müssen wir die neue Funktion nur noch unserem CheckboxListTile hinzufügen:

```
79                  return CheckboxListTile(
80                    title: Text(_presents[index].name),
81                    value: _presents[index].isSelected,
82                    onChanged: (val) {
83                      setState(() {
84                        changeIsSelected(_presents[index].id);
85                      });
86                    },
87                  ); // CheckboxListTile
```

Abbildung 5.53: *gift_list.dart*

Jetzt können wir unsere Geschenke abhaken:

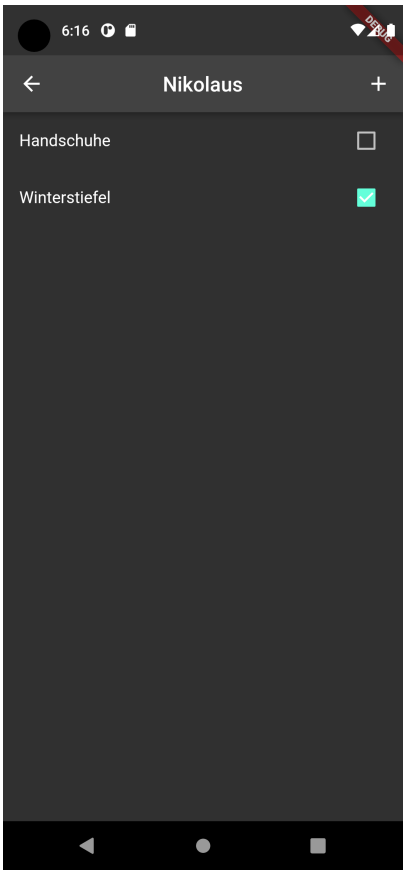


Abbildung 5.54: Geschenkeliste

Im Amplify Studio können wir unter dem Reiter **Content** unsere neuen Objekte ebenfalls sehen und bearbeiten:

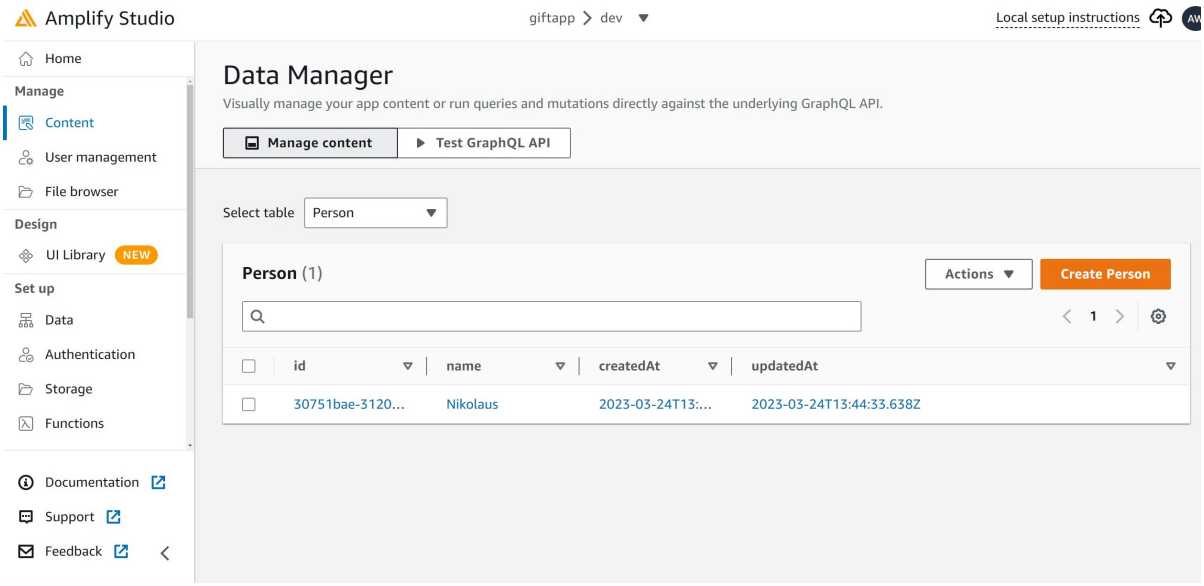


Abbildung 5.55: Amplify Studio

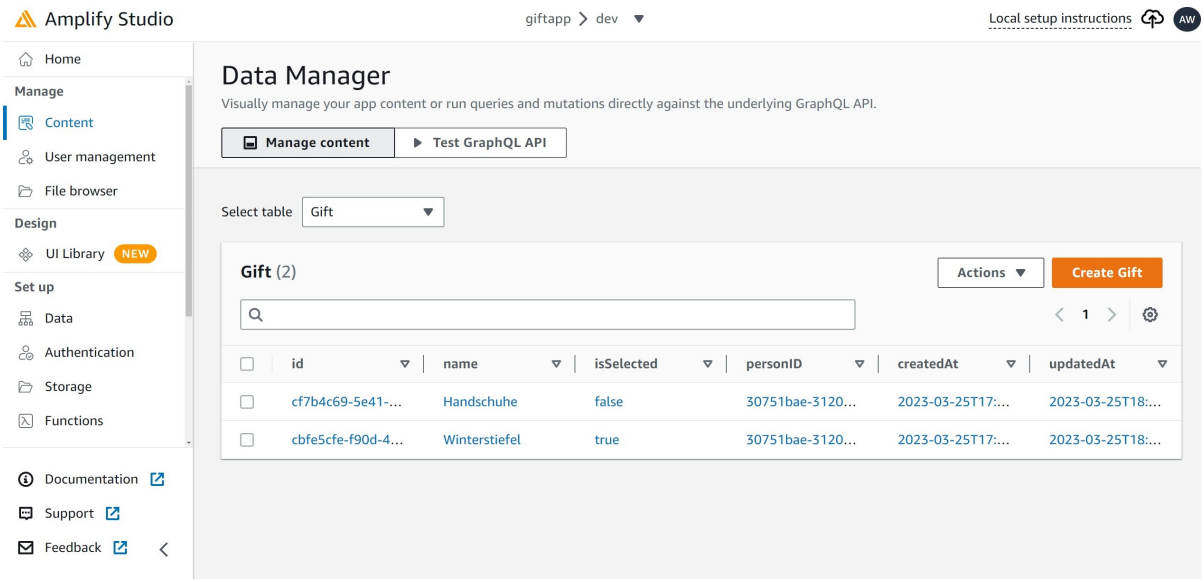


Abbildung 5.56: Amplify Studio

Beim Löschen einer Person werden ihre zugeordneten Geschenke ebenfalls gelöscht.

5.4 Zugriffsrechte

Momentan kann jeder Benutzer der sich anmeldet alle Personen und Geschenke sehen die in der Datenbank gespeichert sind, also auch die von anderen Nutzern. Um das zu verändern gehen wir in Amplify Studio wieder auf den Reiter **Data**. Wenn wir auf ein Model klicken werden uns auf der rechten Seite Optionen angezeigt die Zugriffsrechte zu verändern:

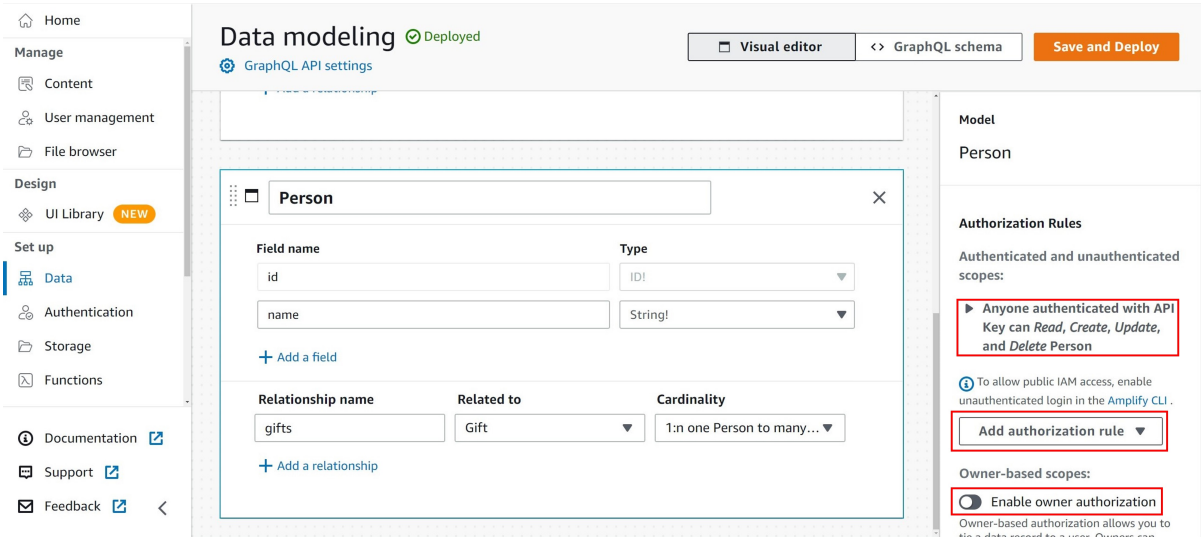


Abbildung 5.57: Amplify Studio

Jeder Nutzer kann momentan lesen, erstellen, updaten und löschen. Mit dem Button **Add authorization rule** könnt ihr verschiedenste Regeln aufstellen, je nachdem was ihr für eure App benötigt. In unserem Beispiel soll jeder Nutzer nur seine eigenen Daten lesen und verändern können. Deshalb aktivieren wir **owner authorization**. Außerdem löschen wir die default Authentifizierungsregel:

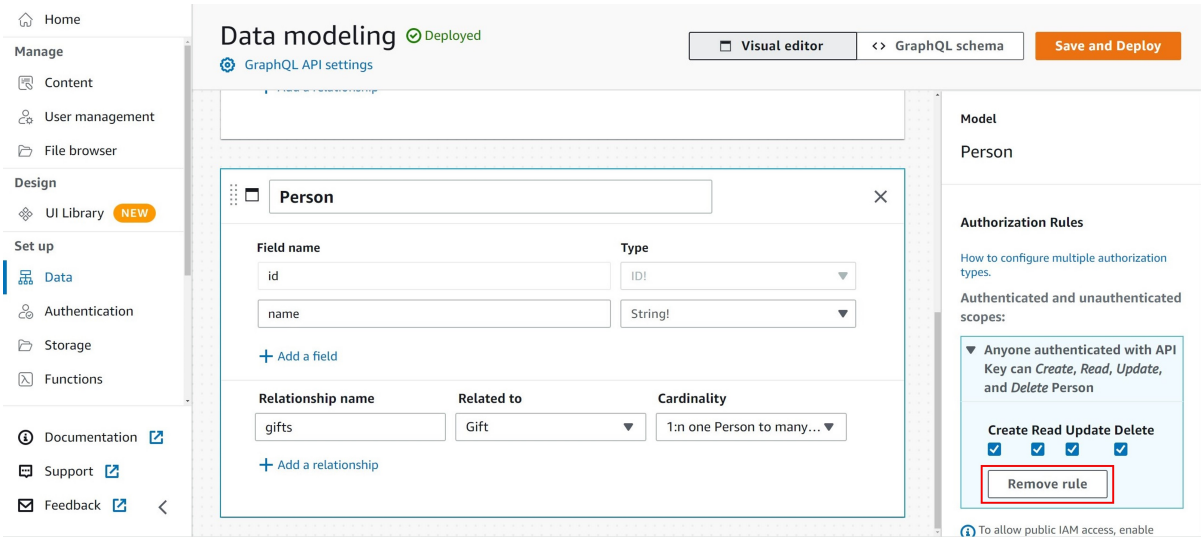


Abbildung 5.58: Amplify Studio

Als nächstes klicken wir auf **GraphQL API settings** und ändern den **Default authorization mode** auf **Cognito User Pool**:

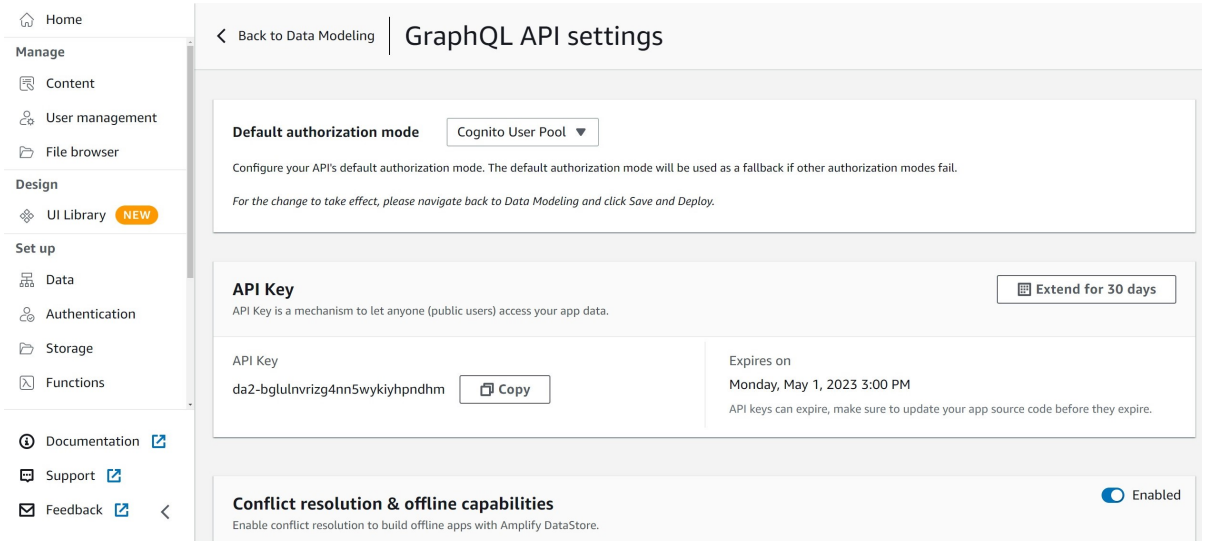


Abbildung 5.59: Amplify Studio

Das gleiche machen wir für unser zweites Model. Immer wenn ihr eine Veränderung an euren Models vornehmt, müsst ihr auf **Save and Deploy** klicken und den danach angezeigten Befehl in eurem Projektverzeichnis ausführen.

Den aktuellen Code findet Ihr hier.

6. AWS Storage

In diesem Kapitel erstellen wir einen S3 Bucket und nutzen diesen, um Bilder in der Cloud zu speichern und diese in unserer App zu verwenden.

6.1 S3 Bucket erstellen

Zuerst öffnen wie Amplify Studio und den Reiter **Storage**. Hier erstellen wir einen S3 Bucket in dem unsere User Dateien hochladen, lesen und löschen können.

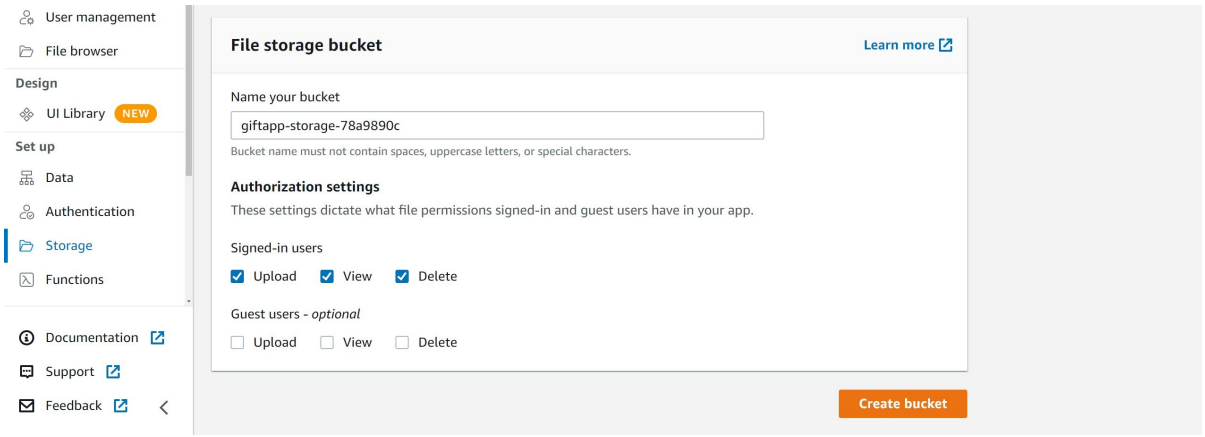


Abbildung 6.1: Amplify Studio

Nachdem die Einrichtung abgeschlossen wurde führt ihr wieder den euch angezeigten Befehl im Verzeichnis eures Projekts aus und klickt auf **Done**:

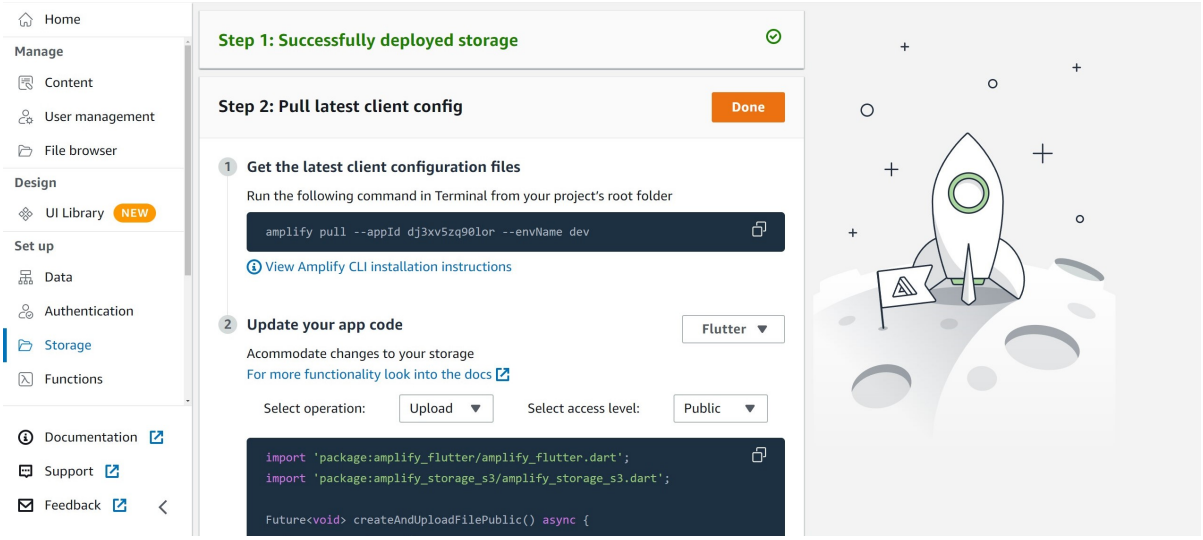


Abbildung 6.2: Amplify Studio

Unter dem Reiter **Strorage** könnt ihr euch über einen Link euren Speicher anschauen und bearbeiten.

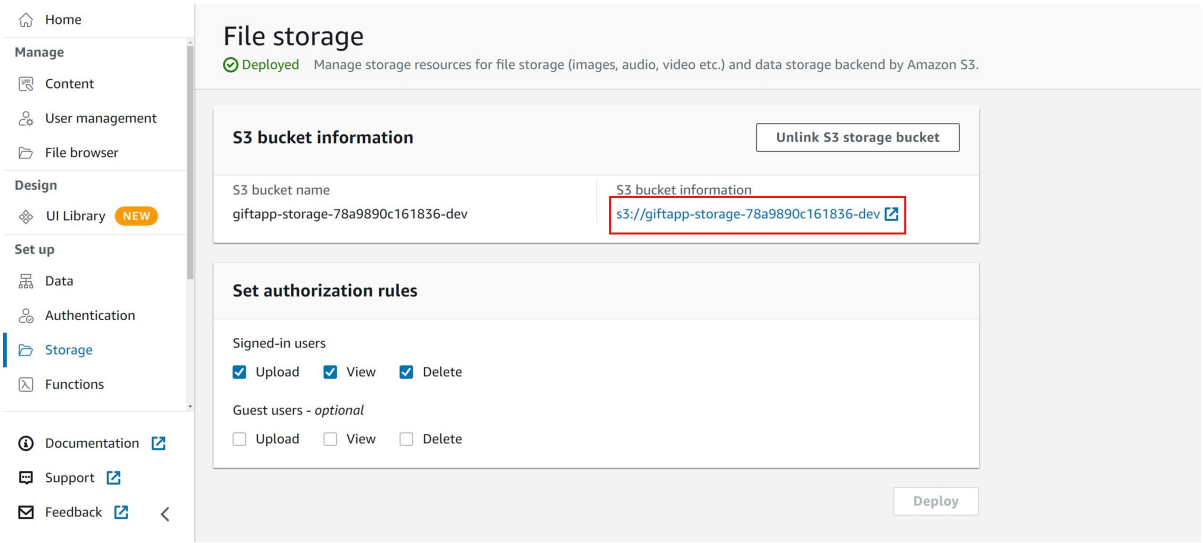


Abbildung 6.3: Amplify Studio

6.2 S3 Bucket in App einbinden

Um den neu angelegten Datenspeicher in unserer App nutzen zu können, fügen wir *pubspec.yaml* amplify_storage_s3 hinzu. Um Bilder aus unserer Galerie hochladen zu können, fügen wir außerdem image_picker und uuid hinzu.

```
39   cupertino_icons: ^1.0.2
40   amplify_flutter: ^0.6.0
41   amplify_auth_cognito: ^0.6.0
42   amplify_authenticator: ^0.2.0
43   intl: ^0.17.0
44   amplify_datastore: ^0.6.0
45   amplify_api: ^0.6.0
46   amplify_storage_s3: ^0.6.0
47   image_picker: ^0.8.0
48   uuid: ^3.0.6
49
```

Abbildung 6.4: *pubspec.yaml*

Danach fügen wir *main.dart* einen neuen Import hinzu:

```
18  import 'package:amplify_storage_s3/amplify_storage_s3.dart';
```

Abbildung 6.5: *main.dart*

Und ergänzen das neue Plugin in *_configureAmplify()*:

```
38  void _configureAmplify() async {
39    try {
40      final authPlugin = AmplifyAuthCognito();
41      final apiPlugin = AmplifyAPI();
42      final datastorePlugin = AmplifyDataStore(modelProvider: ModelProvider.instance);
43      final storagePlugin = AmplifyStorageS3();
44      await Amplify.addPlugins([authPlugin, apiPlugin, datastorePlugin, storagePlugin]);
45      await Amplify.configure(amplifyconfig);
46    } on Exception catch (e) {
47      safePrint(e);
48    }
49  }
```

Abbildung 6.6: *main.dart*

6.3 Dateien hochladen

Um den Personen Profilbilder zuordnen zu können, müssen wir zuerst unser Model **Person** im Amplify Studio anpassen. Wir fügen ein neues Feld **pictureKey** hinzu, speichern das Model und fügen es wieder der App hinzu.

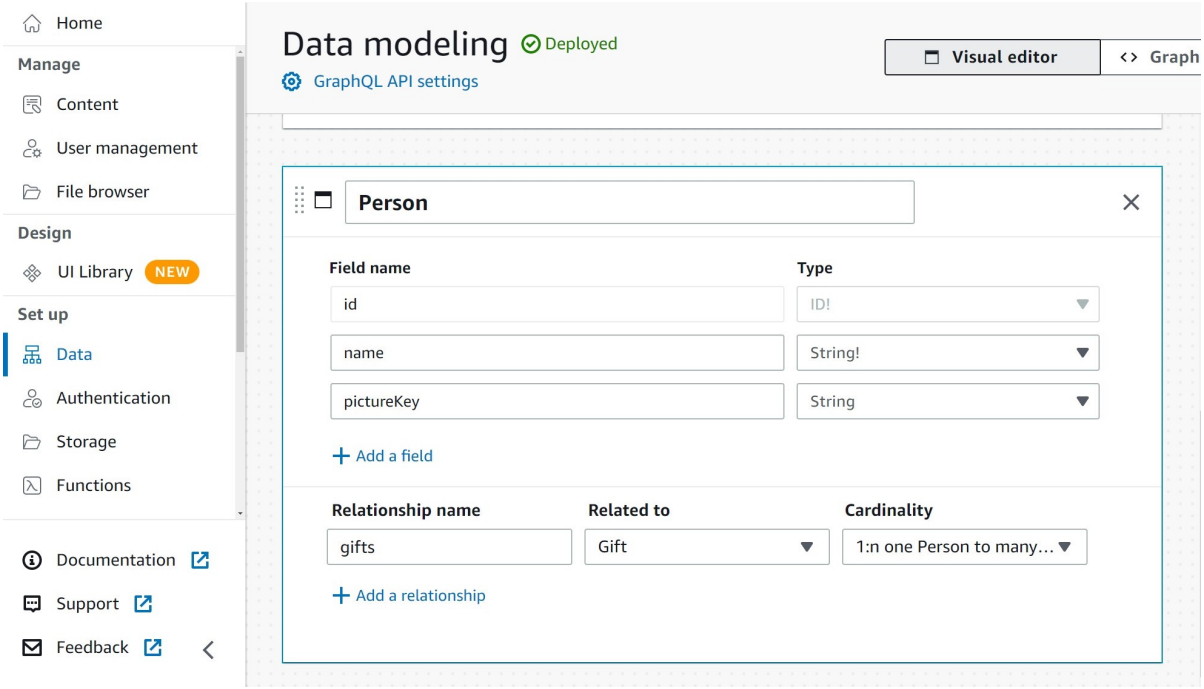


Abbildung 6.7: Amplify Studio

Als nächstes passen wir unsere NewPersonView so an, dass wir beim erstellen einer Person auch ein Bild hinzufügen können. Dafür fügen wir *new_person.dart* zuerst neue Imports hinzu:

```dart
import 'package:amplify_flutter/amplify_flutter.dart';
import 'package:flutter/material.dart';
import 'package:gift_app/models/ModelProvider.dart';
import 'package:image_picker/image_picker.dart';
import 'dart:io';
import 'package:uuid/uuid.dart';
```

Abbildung 6.8: *new_person.dart*

Wir erstellen einen ImagePicker(), um Bilder aus der Galerie auswählen zu können und einen **_key**, den wir unseren neu erstellten Personen hinzufügen können. Durch den Keys wissen wir später, welches Bild zu welcher person gehört.

```
15  class _NewPersonViewState extends State<NewPersonView> {
16    late TextEditingController _nameController;
17    final picker = ImagePicker();
18    String _key = "";
```

Abbildung 6.9: *new_person.dart*

saveNewPerson() passen wir dementsprechend an:

```
20    Future<void> saveNewPerson(String name) async {
21      final newPerson = Person(
22        name: name,
23        pictureKey: _key,
24      );
25      try {
26        await Amplify.DataStore.save(newPerson);
27      } on DataStoreException catch (e) {
28        safePrint(e);
29      }
30    }
```

Abbildung 6.10: *new_person.dart*

Wir erstellen außerdem eine neue Funktion *uploadImage()*. Hier nutzen wir zuerst den ImagePicker() um ein Bild aus der Galerie auszuwählen. Danach erstellen wir uns mithilfe von uuid eine einzigartigen Schlüssel für dieses Bild, um es anschließend mit der Amplify Funktion *uploadFile()* in unseren Speicher zu laden.

```dart
Future<void> uploadImage() async {
  final pickedFile = await picker.pickImage(
    source: ImageSource.gallery,
  );
  if (pickedFile == null) {
    safePrint('Kein Bild ausgewählt');
    return;
  }
  final uniqueKey = const Uuid().v1();
  final key = pickedFile.name + uniqueKey;
  final file = File(pickedFile.path);
  try {
    final UploadFileResult result = await Amplify.Storage.uploadFile(
      local: file,
      key: key,
      onProgress: (progress) {
        safePrint('Abschnitt fertig: ${progress.getFractionCompleted()}');
      },
    );
    _key = key;
    safePrint('Bild wurde hochgeladen: ${result.key}');
  } on StorageException catch (e) {
    safePrint(e);
  }
}
```

Abbildung 6.11: *new_person.dart*

Danach erstellen wir uns einen neuen Button mit der neuen Funktion und fügen diesen unserem Scaffold hinzu

```dart
Widget _imageButton() {
  return ElevatedButton(
    onPressed: () {
      uploadImage();
    },
    child: const Text('Bild hinzufügen'),
  ); // ElevatedButton
}
```

Abbildung 6.12: *new_person.dart*

```
89            ), // SizedBox
90            _imageButton(),
91            _button(),
```

Abbildung 6.13: *new_person.dart*

Nun kann jeder neu erstellten Person ein Bild hinzugefügt werden:

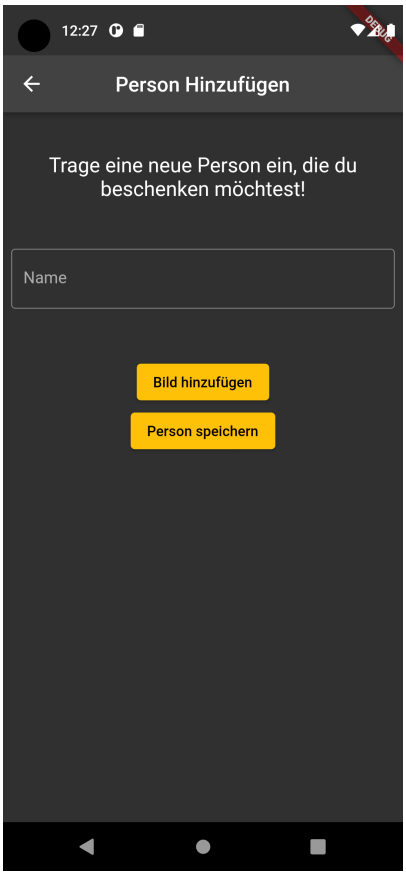


Abbildung 6.14: Person Hinzufügen

6.4 Dateien benutzen

Um die hochgeladenen Bilder in der App zu verwenden, nutzen wir ihre URL. GiftListView fügen wir deshalb einen neuen String **_url** hinzu:

```
15  class _GiftListViewState extends State<GiftListView> {
16    late final Person _person = widget.person;
17    List<Gift> _presents = [];
18    late StreamSubscription<QuerySnapshot<Gift>> _subscription;
19    String _url = "";
20
```

Abbildung 6.15: *gift_list.dart*

In der Funktion *getUrl()* nutzen wir den **pictureKey** der aktuellen Person um mithilfe der Amplify Funktion *getUrl()* die URL des Bildes zu erhalten:

```
35    Future<void> getUrl(Person person) async {
36      final key = person.pictureKey;
37      if (key != null) {
38        _url = (await Amplify.Storage.getUrl(key: key)).url;
39      }
40    }
41
```

Abbildung 6.16: *gift_list.dart*

Nun erstellen wir noch ein neues Widget mit einem Network Image, dem wir die aktuelle URL übergeben:

```
115   Widget _profilePic() {
116     return CircleAvatar(
117       radius: 60,
118       backgroundImage: NetworkImage(
119         _url,
120       ), // NetworkImage
121     ); // CircleAvatar
122   }
123 }
```

Abbildung 6.17: *gift_list.dart*

Das neue Widget fügen wir anschließend unserem Scaffold hinzu. Wir nutzen wie in *home.dart* einen FutureBuilder um den Bildschirm erst anzuzeigen, wenn die URL des Bildes geladen wurde:

```
76        body: FutureBuilder(
77          builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {
78            return Column(
79              children: [
80                const SizedBox(
81                  height: 20,
82                ), // SizedBox
83                _profilePic(),
84                const SizedBox(
85                  height: 20,
86                ), // SizedBox
87                _giftList(),
88              ],
89            ); // Column
90          },
91          future: getUrl(_person),
92        ), // FutureBuilder
93      ); // Scaffold
94    }
```

Abbildung 6.18: *gift_list.dart*

Auf der Detailseite der einzelnen Personen wird nun das Profilbild angezeigt:

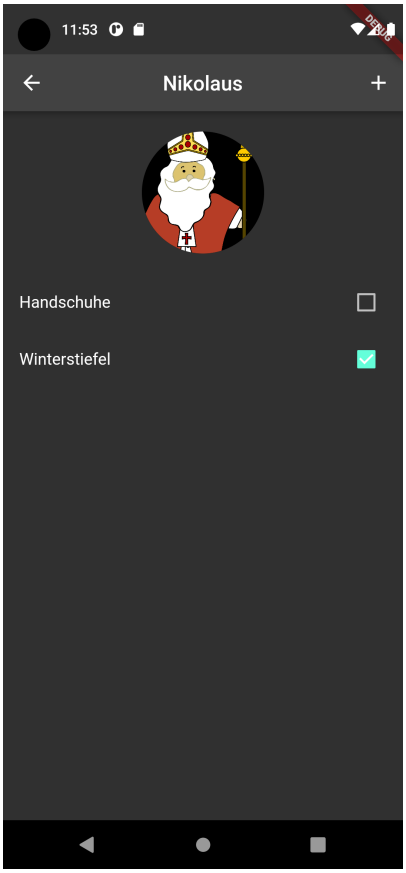


Abbildung 6.19: Geschenkeliste

Als letztes müssen wir noch die *deletePerson()* Funktion in *home.dart* anpassen, damit beim Löschen einer Person auch das zugeordnete Bild gelöscht wird. Dafür nutzen wir die Amplify Funktion remove():

```
40  Future<void> deletePerson(Person person) async {
41    try {
42      if (person.pictureKey != null) {
43        await Amplify.Storage.remove(key: person.pictureKey!);
44      }
45      await Amplify.DataStore.delete(person);
46    } catch (e) {
47      safePrint(e);
48    }
49  }
```

Abbildung 6.20: *home.dart*

Den aktuellen Code findet Ihr hier.