Rapport

PGR208 Android Programmering Eksamen - Høst 2024

Kandidat nummer: 40

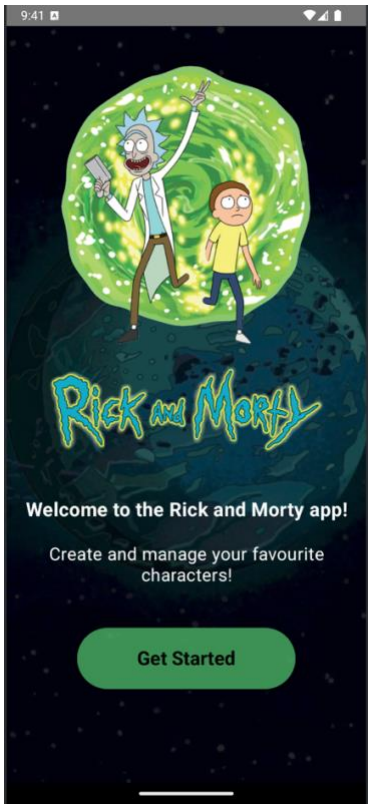
Jeg har utviklet en app ved bruk av Rick and Morty API. Appen inneholder 7 skjermer som lar brukeren lage egen karakter, se disse og utforske karakterer fra showet.

Oversikt over funksjonalitet

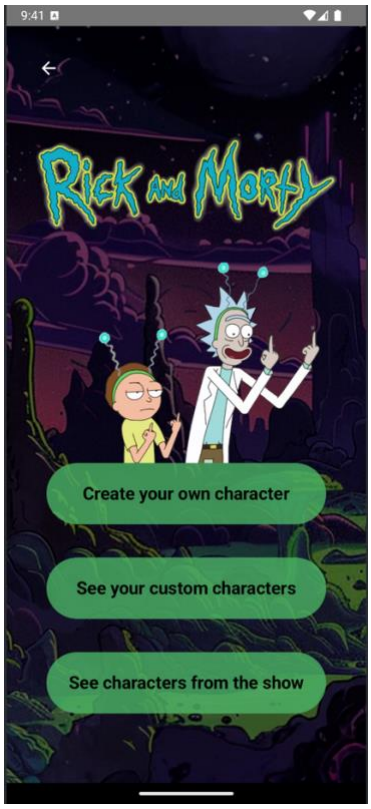
Welcome Screen	Viser en statisk side med en velkomstmelding og en knapp for å starte appen.
Main Page	Gir brukeren alternativer for å se karakterer hentet fra API, lage egne karakterer eller se egne karakterer.
CharacterList	Viser en scrollbar liste over karakterer fra API-et. Man kan trykke på en karakter for å se detaljer.
CharacterDetails	Viser detaljer om den valgte karakteren fra API-et, inkluderer bilde, navn, art, status og opprinnelse.
MakeCharacter	Lar brukeren lage en karakter ved å fylle ut navn, art, kjønn og velge bilde.
MyCharacters	Viser en scrollbar liste over karakterene som brukeren har laget. Man kan trykke på en karakter for å se detaljer.
MyCharacterDetails	Viser detaljer for den valgte karakteren fra MyCharacters. Inkluderer bilde, navn, art og kjønn.

Skjermdump av samtlige skjermer i appen

Appen bruker bakgrunnsbilder av steder fra serien Rick and Morty.



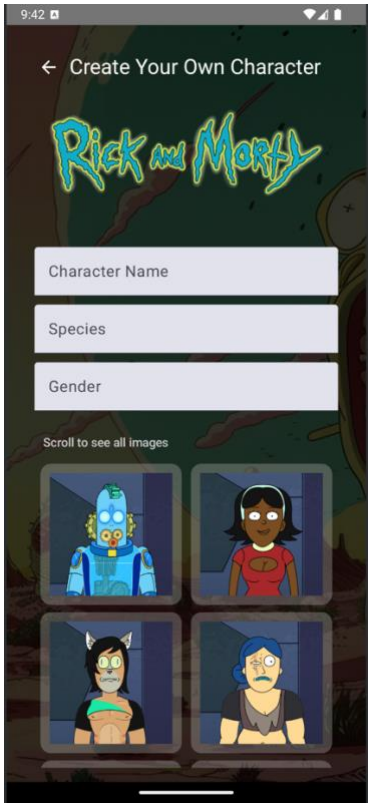
Welcome Screen
Dette er en startside med en velkomstmelding til brukeren. Skjermen inneholder også en knapp som tar brukeren videre til Main Page.



Main Page
Denne siden gir brukeren tre valg.

- Lag egen karakter
- Se egne karakterer
- Se karakterer hentet fra API

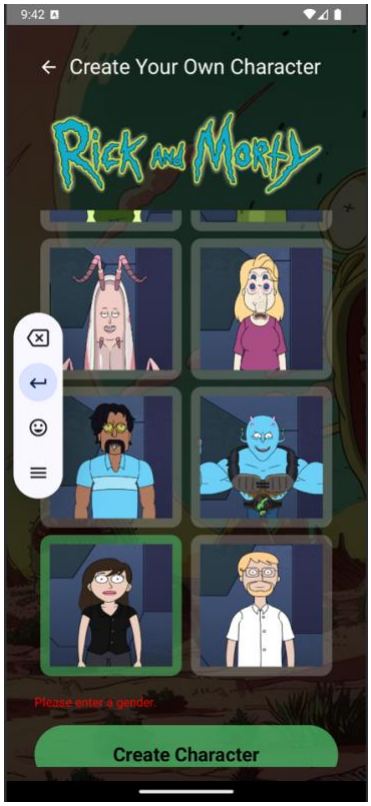
Brukeren kan også navigere tilbake til Welcome Screen.



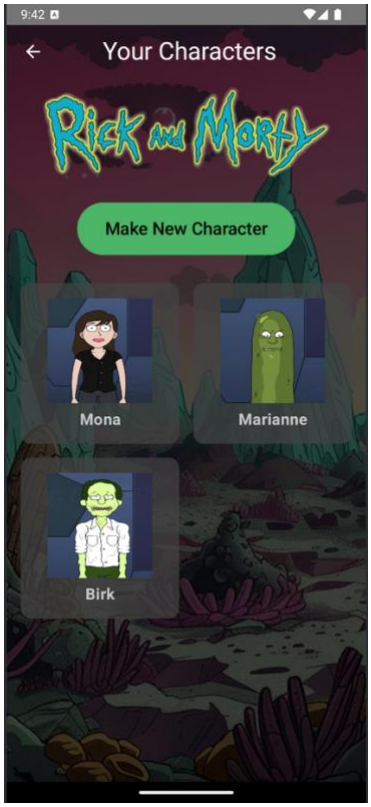
Make Character
Her kan brukeren lage en egen karakter ved å velge:

- Navn
- Art
- Kjønn
- Bilde

Brukeren har mulighet til å velge mellom 12 ulike profilbilder. Disse bildene er designet og hentet fra nettsiden *Go Rick Yourself*.
Brukeren kan også navigere tilbake til enten 'My Characters' eller 'Main Page', avhengig av hvilken av skjermene som var aktiv sist.



Hvis brukeren ikke har fylt ut alle feltene eller valgt et bilde, vil det vises en melding som instruerer brukeren om å fullføre opprettelsen av karakteren.



Characters
Denne skjermen viser karakterene som brukeren har laget. Man kan:

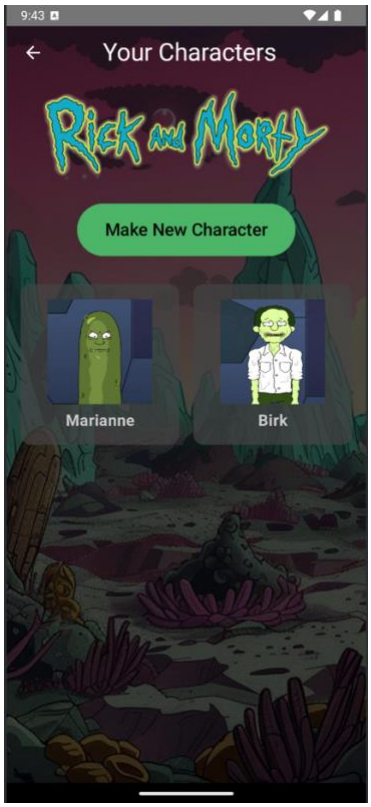
- Se detaljer om karakter.
- Navigere til 'Make Character'.
- Navigere tilbake til 'Main Page'.

Hvis listen over karakterer overskrider skjermens høyde, er den scrollbar.

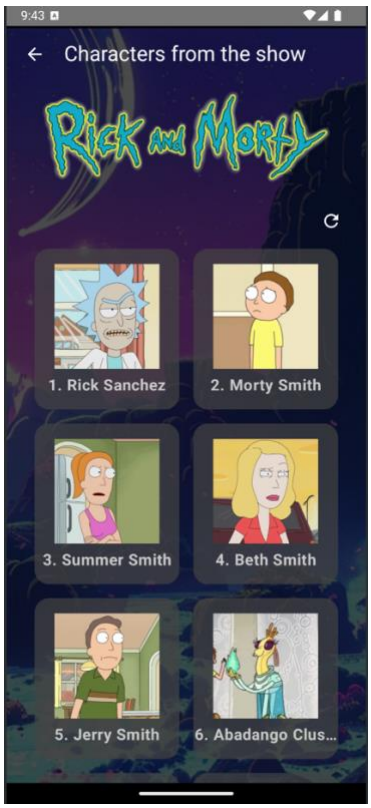
CharacterDetails
Denne skjermen gir brukeren mulighet til å se detaljer om karakterene som er laget, dette inkluderer:

- Profilbilde
- Navn
- Kjønn
- Art

Brukeren kan enten navigere tilbake til 'MY Characters' eller velge å slette karakteren.



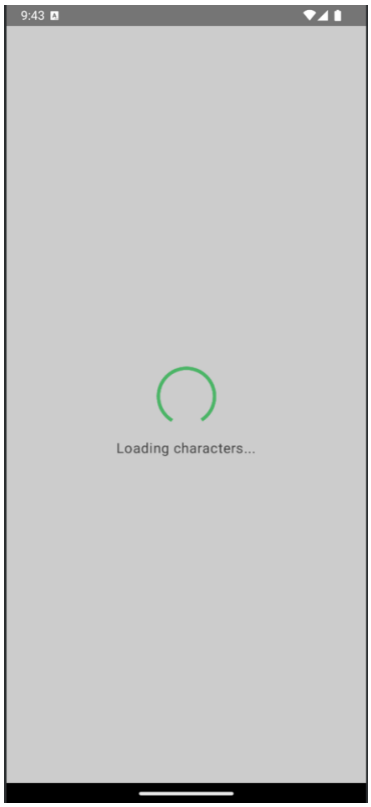
Skjermdump av skjermen etter at karakteren 'Mona' ble slettet.



C**haracters**

Denne skjermen inneholder karakterer fra API-et. Her kan man:

- Navigere seg tilbake til 'Main Page'.
- Trykke på en karakter for å få opp ekstra info.
- Laste inn siden på nytt for oppdatert info.



Når man navigerer til Characters vil det dukke opp en midlertidig loading screen.

CharacterDetails
Denne skjermen inneholder info om valgt karakter:

- Profilbilde
- Navn
- Kjønn
- Rase
- Status
- Opprinnelse

Samme loading screen er implementert her.
Brukeren kan også navigere seg tilbake til 'Character' skjermen.

Beskrivelse av hovedteknikker brukt:

Retrofit for API-kall

Jeg har brukt Retrofit for å kommunisere med Rick and Morty API-et og hente nødvendige data. API-endepunktene er definert i CharacterRepository, som også håndterer logikken mellom nettverkskall og databasen. CharacterService inkluderer @GET-funksjoner for spesifikke API-forespørsler.

```
interface CharacterService {
    @GET("character")
    suspend fun getAllCharacters(): Response<CharacterResponse>
}
```

Room for lokal datalagring

Room brukes som databasebibliotek for å lagre data lokalt, noe som gir offline-støtte og lar brukeren få tilgang til karakterer uten nettverk. AppDatabase inneholder to tabeller: én for API-karakterer (Character) og én for brukerskapte karakterer (UserCharacter).

```
@Database(
    entities = [Character::class, UserCharacter::class],
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun characterDao(): CharacterDao
    abstract fun userCharacterDao(): UserCharacterDao
}
```

For komplekse objekter som Origin har jeg brukt CharacterTypeConverter for å konvertere data til og fra JSON.

```
@TypeConverter
fun fromOrigin(origin: Origin?): String {
    return gson.toJson(origin)
}

@TypeConverter
fun toOrigin(data: String): Origin {
    return gson.fromJson(data, Origin::class.java)
}
```

Jetpack Compose for UI

Jetpack Compose er brukt for å gjøre UI-koden enklere å lese og oppdatere, samt for å bygge responsive og interaktive grensesnitt. Et eksempel er CharacterItem, som er implementert som en modulær og gjenbrukbar komponent for visning av karakterdata.

```
@Composable
fun CharacterItem(
    character: Character,
    onClick: () -> Unit
) {
    Column(
        modifier = Modifier
            .size(width = 170.dp, height = 200.dp)
            .padding(8.dp)
            .background(Color.DarkGray.copy(alpha = 0.7f), shape = RoundedCornerShape
            .clickable { onClick() },
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        AsyncImage(
            model = character.image,
            contentDescription = "Image of ${character.name}",
            modifier = Modifier.size(120.dp),
            contentScale = ContentScale.Crop
        )
        Spacer(modifier = Modifier.height(8.dp))
        Text(
            text = character.name,
            style = MaterialTheme.typography.bodyLarge,
            maxLines = 1,
            overflow = TextOverflow.Ellipsis
        )
    }
}
```

Navigation Component for navigering
Jeg har brukt NavHost og composable for å lettere kunne navigere mellom skjermer.
Eksempel:

```
NavHost(
    navController = navController,
    startDestination = Welcome
) {
    composable<Welcome> {
        WelcomeScreen(
            onStartClick = { navController.navigate(Main) }
        )
    }

    composable<CharacterDetails> { backStackEntry ->
        val details = backStackEntry.toRoute<CharacterDetails>()
        CharacterDetailsScreen(
            characterId = details.characterId,
            onBackButtonClick = { navController.popBackStack() }
        )
    }
}
```

Kvalitet og struktur:

Mappestrukturen til prosjektet er organisert i følgende mapper:

- Data: Inneholder all datalogikk (Room-tabeller, DAO's og API-relatert kode.)
- Screens: Mapper for hver skjerm med tilhørende elementer og ViewModels.
- Andre deler: MainActivity og res/drawable (inneholder bakgrunnsbilder, logo og profilbilder for brukere).

Referanser

LazyVerticalGrid

Under utviklingen av MyCharacterListScreen og CharacterListScreen møtte jeg utfordringer med å dele karakterer i to kolonner. Jeg fant løsningen ved å bruke **LazyVerticalGrid** i Jetpack Compose, som er beskrevet på nettsiden *LazyVerticalGrid - Jetpack Compose Playground*, og lot meg inspirere til implementasjonen.
Eksempelkoden på nettsiden:

```
  colums = GridCells.Fixed(2)
```

Koden min:

```
LazyVerticalGrid(
                columns = GridCells.Fixed(2),
                modifier = Modifier
                    .fillMaxSize()
                    .fillMaxHeight()
                    .padding(horizontal = 8.dp)
            )
```

Chunked

For å dele bildelisten i grupper for visning i flere rader, brukte jeg funksjonen **chunked**, beskrevet i dokumentasjonen for Kotlin. Denne funksjonen delte bildelisten i grupper på to bilder per rad, noe som løste utfordringen effektivt.
Eksempelkoden på nettsiden:

```
val words = "one two three four five six seven eight nine ten".split(' ')
val chunks = words.chunked(3)

println(chunks) // [[one, two, three], [four, five, six], [seven, eight, nine], [te
```

Ved å tilpasse dette prinsippet implementerte jeg følgende kode for å vise bilder i rader med to kolonner:

```
Column(
    modifier = Modifier
        .align(Alignment.CenterHorizontally)
) {
    viewModel.availableImages.chunked(2)
        .forEach { rowImages ->
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceEvenly
            ) {
                rowImages.forEach { imageRes ->
                    val isSelected = imageRes == selectedImage
                    val backgroundColor = if (isSelected) Color(0xFF4CB567),
```

Referanse liste:

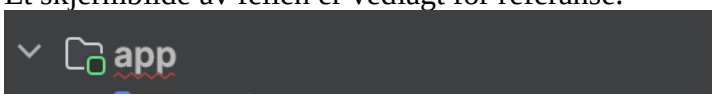
- *LazyVerticalGrid - Jetpack Compose Playground*, (2022), foso.github.io/Jetpack-Compose-Playground/foundation/lazyverticalgrid/.
- *chunked - Kotlin Programming Language*. (2024), https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/chunked.html
- *Rick and Morty Avatar Maker – Go Rick Yourself, Adult Swim*, https://www.gorickyourself.com/

Uforventet feil

I *app*-mappen vises en rød strek som indikerer en feil. Jeg har forsøkt følgende tiltak uten å løse problemet:

- **Clean Project**
- **Rebuild Project**
- **Invalidate Caches and Restart**
- Konsultert ChatGPT
- Søk på nettet

Et skjermbilde av feilen er vedlagt for referanse.



Avslutning

Gjennom dette prosjektet har jeg fått en dypere forståelse av Android-utvikling, inkludert Jetpack Compose, Room og Retrofit. Dette har vert en verdifull lærings opplevelse som har styrket mine ferdigheter innen mobilutvikling.