PGR208 Android programmering

Kandidatnummer: 36

1. Oversikt over funksjonalitet du har laget

Navn på funksjonalitet	Beskrivelse av funksjonalitet
Hente data fra API, samt lagre dataen i lokal database.	Ved bruk av Retrofit, kjører jeg en GET forspørsel til *character* endepunktet, mot Rick and Morty API-et. Sammen med en query spørring mot *page*, slik at jeg får tilgang til alle sidene som er tilgjengelige i API-et. Ved bruk av Room og DAOs, lagrer jeg dataen i en lokal database.
Vise liste med karakterer fra database.	Via DAO query metoder, kan jeg skrive egne SQL spørringer for å vise ønskede karakterer for ulike scenarioer i en liste.
Filtrere mellom levende og døde karakterer.	Ved bruk av DAO query metoder, spør jeg om levende og døde karakterer. Via kombinasjonen av kode og *FilterChip* kan man trykke på *Alive* for å se levende karakterer. Trykker man på *Dead* uten å huke av på *Alive*, vil man kunne se både levende og døde karakterer. Er ingen av dem huket av, vil man kunne se alle karakterer, også dem satt til *unknown*.
Kunne laste siden inn på nytt (reload).	Man kan laste inn siden på nytt via en *IconButton* som tar forbehold til filteret beskrevet tidligere.
Se informasjon/data om karakterer.	Trykker man på en karakter, lastes den spesifikke karakter inn basert på id-en den har. Her kan jeg da få ut den dataen jeg ønsker å vise til brukeren.
Lage egen karakter med navn, bilde og art.	Bruker kan lage sin egen karakter, hvor man skriver inn i et *InputField*. Navn er obligatorisk, men bilde og art er valgfritt. Bilde blir satt til et *default* bilde og art til *Unknown*. Skriver man ikke inn et navn, vil man ikke kunne lage en karakter.
Kunne endre navn, bilde og art på karakterer bruker har laget, samt kunne slette dem.	Via DAO *Convenience* metoder, kan jeg bruke *@Update* og *@Delete* til å modifisere eller slette dataen til karakterer som bruker har laget.
Gå tilbake pil	Via en *IconButton* med utseende av en pil, navigerer man tilbake til forrige side ved å bruke *popBackStack*. Ved å

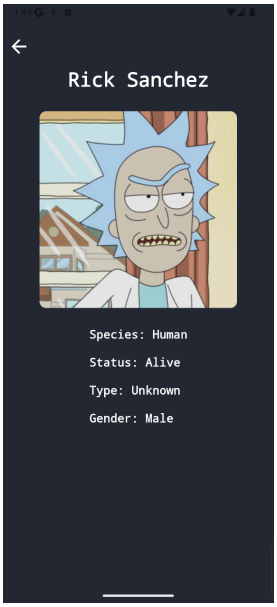
	bruke denne vil man bli tatt tilbake der hvor man sist var, uten å miste tidligere progresjon.
Navigasjons knapper	Via en *ExtendedFloatingActionButton* kan man navigere direkte til de ulike sidene ved å direkte kalle den spesifikke siden fra *MainActivity.*
Loading bar	Under lasting av karakterer fra API-et eller fra databasen, vil bruker se en *CircularProgressIndicator* loading bar.
Vende telefon horisontalt	Ved bruk av *verticalScroll(state = rememberScrollState())* og *LazyVerticalGrid* kan man scrolle på sidene, både om telefonen er holdt vertikalt eller vendt horisontalt.

2. Skjermdump av samtlige skjermer i appen

OBS! Begrepet "default" refererer til originale Rick og Morty karakterer og begrepet "custom" refererer til karakterer bruker har laget.

Skjerm 1 →

- Liste over "default" Rick and Morty karakterer.
- Trykk på karakter for å se informasjon.
- Filtrere mellom levende og døde karakterer.
    - Begge huket av = alle levende og døde.
    - Begge ikke huket av = alle karakterer, samt dem satt til "Unknown".
- Navigere til "My Page" for å se brukerens "custom" karakterer.
- Laste inn karakterer på nytt med "reload" ikonet.

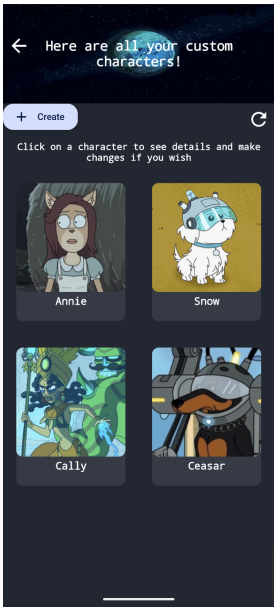


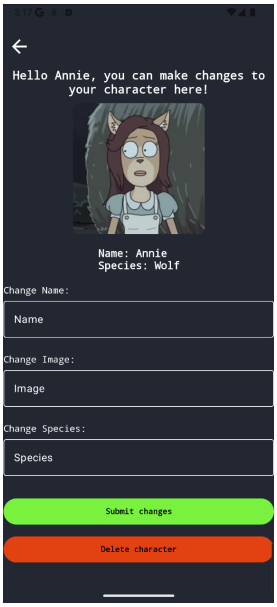
← *Skjerm 2*

- Se detaljert informasjon om valgt karakter.

Skjerm 3 →

- Liste over “custom” karakterer.
- Trykke på en karakter for å se mer informasjon.
- Navigere til “Create” (lag egen karakter).



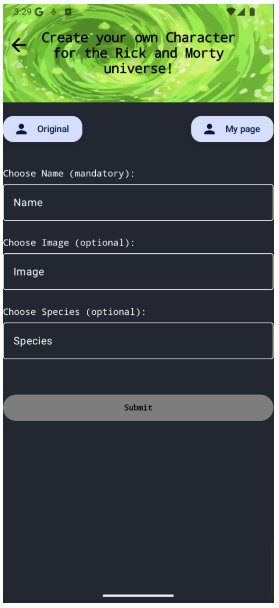


← *Skjerm 4*

- Se “custom” karakterens informasjon.
- Endre navn, bilde og art ved å skrive inn i input feltet samt trykke “Submit changes”.
- Slette “custom” karakteren fra databasen ved å trykke på “Delete character”.

Skjerm 5 →

- Lage ny “custom” karakter ved å velge navn (obligatorisk). Har man ikke navn vil man ikke kunne trykke “Submit”. Knappen blir grønn når man kan fortsette.
- Bilde og art er valgfritt, de vil få tildelt standard verdier om ikke noe er skrevet inn.



3. Beskrivelser og skjermbilder av hovedteknikker brukt

Retrofit for API-kommunikasjon →

Retrofit er et type-sikkert bibliotek for å håndtere API-kall, som forenkler opprettelse, mottak og sending av HTTP-forespørsler og JSON-parsing. Ved å bruke *GsonConverterFactory* konverteres JSON-respons til Kotlin-objekter. Koden setter opp en Retrofit-instans med base-URL og en *OkHttpClient* for nettverksforespørsler og debugging. *RickAndMortyService* brukes for å hente data fra API-et.

```
object RickAndMortyRepository {
    private val _httpClient =
        OkHttpClient.Builder()
            .addInterceptor(
                HttpLoggingInterceptor()
                    .setLevel(HttpLoggingInterceptor.Level.BODY)
            )
            .build()

    private val _retrofit =
        Retrofit.Builder()
            .client(_httpClient)
            .baseUrl("https://rickandmortyapi.com/api/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()

    private val _rickAndMortyService = _retrofit.create(RickAndMortyService::class.java)
```

```
interface RickAndMortyService {
    @GET("character")
    suspend fun getCharacters(@Query("page") page: Int): Response<CharacterListResponse>
}
```

```
data class CharacterListResponse(
    val info: Info,
    val results: List<Character>
)

data class Info(
    val count: Int,
    val pages: Int,
    val next: String?,
    val prev: String?
)
```

Ved bruk av @GET-annotasjonen hentes data fra "character"-endepunktet i Rick and Morty API-et. Responsen pakkes inn i et Retrofit-objekt som inneholder *CharacterListResponse* med JSON-dataene. For å hente karakterer fra flere sider brukes @Query-annotasjonen til å sende inn sidetall som parameter.

```
// From Canvas : pgr208-10-lecture-code-finish
private lateinit var _appDatabase: AppDatabase

// From Canvas : pgr208-10-lecture-code-finish
private val _defaultCharacterDao by lazy { _appDatabase.defaultCharacterDao() }
private val _customCharacterDao by lazy { _appDatabase.customCharacterDao() }

// From Canvas : pgr208-10-lecture-code-finish
fun initializeDatabase(context: Context) {
    _appDatabase = Room.databaseBuilder(
        context = context,
        klass = AppDatabase::class.java,
        name = "RickAndMorty-database"
    ).build()
}
```

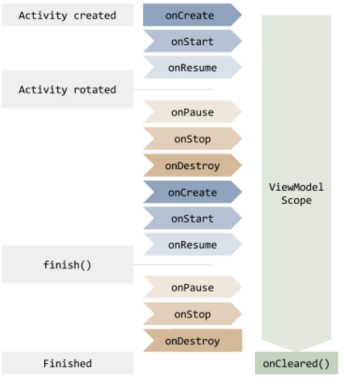
← Room database for lokal lagring

Room er et abstraksjonslag over SQLite for enklere databaseoperasjoner. Databasen initialiseres ved hjelp av en lateinit-variabel, og DAO-er definert for å håndtere dataoperasjoner. Ved å bruke lazy-delegering opprettes DAO-er kun når de trengs, noe som forbedrer ytelsen. Lokal lagring sikrer at appen kan fungere uten

internettforbindelse og gir raskere responstider ved å unngå unødvendige nettverkskall.

ViewModel/MVVM arkitektur →

Appen følger MVVM-arkitektur med tre lag: View (UI), ViewModel (datatilstand og logikk), og Model (datahåndtering). ViewModels sikrer at data og tilstand overlever skjerm rotasjoner og gir en ryddigere separasjon mellom UI og logikk.



The lifecycle of a ViewModel

```
@Composable
fun CharacterContainer(
    character: Character,
    onClick: () -> Unit = {}
){
    Column (
        modifier = Modifier
            .height(250.dp)
            .padding(
                horizontal = 20.dp,
                vertical = 20.dp
            )
            .clip(shape = RoundedCornerShape(10.dp))
            .background(Color.White.copy(alpha = 0.1f))
            .clickable { onClick() },
        horizontalAlignment = Alignment.CenterHorizontally
    ){

        AsyncImage(
            modifier = Modifier
                .fillMaxWidth()
                .clip(RoundedCornerShape(10.dp))
                .background(color = Color.DarkGray),
            model = character.image,
            contentDescription = "Image of ${character.name}"
        )

        Column {
            Text(
                modifier = Modifier
                    .padding(top = 5.dp),
                text = character.name,
                textAlign = TextAlign.Center,
                style =
                TextStyle(
                    fontFamily = FontFamily.Monospace,
                    fontSize = 16.sp,
                    fontWeight = FontWeight.W600,
                    color = Color.White
                )
            )
        }
    }
}
```

← Jetpack Compose

Jetpack Compose er brukt for å lage et moderne UI direkte i Kotlin. Det støtter gjenbrukbare komponenter og automatisk oppdatering av UI når data endres.

Coroutines for asynkron programmering →

Coroutines håndterer ikke-blokkerende operasjoner, som API-kall og databaseoperasjoner, ved bruk av *suspend*-funksjoner og *Dispatchers.IO*. Dette tillater effektiv utnyttelse av ressursene.

```
suspend fun getDefaultCharacters(): List<Character> {
```

```
fun addCharacter() {
    viewModelScope.launch(Dispatchers.IO) {
```

```
private val _defaultCharacterListViewModel: DefaultCharacterListViewModel by viewModels()
private val _defaultCharacterDetailsViewModel: DefaultCharacterDetailsViewModel by viewModels()
```

```
val navController = rememberNavController()
NavHost(
    navController = navController,
    startDestination = DefaultCharacterList
) {
    composable <DefaultCharacterList> {
        DefaultCharacterListScreen(
            viewModel = _defaultCharacterListViewModel,
            onCustomClick = {navController.navigate(CustomCharacterList)},
            onCharacterClick = { characterId -> navController.navigate(DefaultCharacterDetails(characterId)) }
        )
    }
```

← Navigation mellom skjermer

NavController og NavHost gir type-sikker navigasjon mellom skjermer. Ved å bruke composable-ruter og *viewModels()* for hver skjerm, håndteres tilstand og data effektivt og korrekt.

Serialization →

Kotlins serialisering brukes for å konvertere objekter til og fra JSON. Dette forenkler lagring, henting og overføring av data i navigasjonsargumenter.

```
@Serializable
object DefaultCharacterList

@Serializable
data class DefaultCharacterDetails(
    val characterId: Int
)

@Serializable
object CustomCharacterList

@Serializable
object CreateCharacter

@Serializable
data class CustomCharacterEdit(
    val characterId: Int
)
```

4. Kvalitet og struktur

Navn	Beskrivelse
Mappestruktur	**“data” package:** Håndterer data fra API-et og databasen.

	“room” package: Inneholder DAO-definisjoner og databaseoppsett. **“navigation” package:** Logikken for navigasjon mellom skjermer. **”screen” package:** Ansvarlig for UI, logikk og tilstand, med underpakker for hver skjerm. **“components” package:** Gjenbrukbare komponenter for flere deler av appen. Resultatet er en tydelig mappestruktur som sikrer en effektiv arbeidsflyt og oversikt over hele kodebasen. Hver package har da et klart ansvarsområde.
Navngivning: klasser	Navnekonvensjon: “PascalCase”. Klasser har navn som reflekterer sitt ansvarsområde, f.eks. *CreateCharacterViewModel* som holder på logikk rundt opprettelse av karakterer.
Navngivning: variabler	Navnekonvensjon: “camelCase”. Variabler har i noen tilfeller tydelig beskrivende navn, mens andre har fått mer generelle navn. I *DefaultCharacterListViewModel* har jeg en beskrivende variabel, *isAliveFilterSelected*. Jeg har “is” med i starten da dette er en boolean verdi samt “AliveFilterSelected” da dette beskriver spesifikt hvilken verdi den holder på. Andre ganger er variabel navnene enklere, i *DefaultCharacterListViewModel* har jeg *_characters, hvor* “_” signaliserer at det er en privat variabel, samt en “s” på slutten for å vise at jeg arbeider med flere enn en karakter. Denne variabelen viser til at jeg har en privat liste med flere karakterer.
Navngivning: funksjoner	Navnekonvensjon: “camelCase”. Funksjoner er navngitt nøyaktig til hva funksjonen gjør, samt med klasse navnet i tankene. Det gjør det mulig å ha spesifikke navn som *loadCharacters* i stedet for *loadCustomCharacters*. Ved å se på klasse navnet *CustomCharacterEditViewModel* vet man at de underliggende funksjonene omhandler custom karakterer.
Leselighet	Jeg har lagt fokus på at koden skal være lett leselig og at navnene er selvforklarende, samt tydelig viser hvilken hensikt de har i forhold til sin plassering i koden.

Komponenter	Bruk av komponenter kutter ned på mengden av koden, dette gjør det også lettere for meg å endre stylingen i fremtiden globalt i applikasjonen.
Språk	Jeg bruker engelsk som språk, slik at koden er leselig av alle, men også fordi språket jeg programmerer i, bruker engelske ord. Det gir mest mening for meg å holde alt av kode i samme språk.