Innholdsfortegnelse

Funksjonalitet

Funksjonalitet	Beskrivelse
Skjerm #1	
Vise karakterer	Statisk side. Oversikt over alle karakterer i Rick&Morty universet.
Skjerm #2	
Vise karakterer	Statisk side med oversikt over alle karakterer brukeren selv har laget.
Skjerm #3	
Inputfunksjon	Fem tekstfelt som tar imot input fra bruker. De har hint om hva som kan legges inn i de ulike feltene. Knapp for å lagre karakteren.
Skjerm #4	
Søkefunksjon	Tekstfelt som tar imot fritekst fra bruker. Hint til bruker om hva man kan skrive inn. «Search»-knapp for å få opp liste over treff.

Hovedteknikker

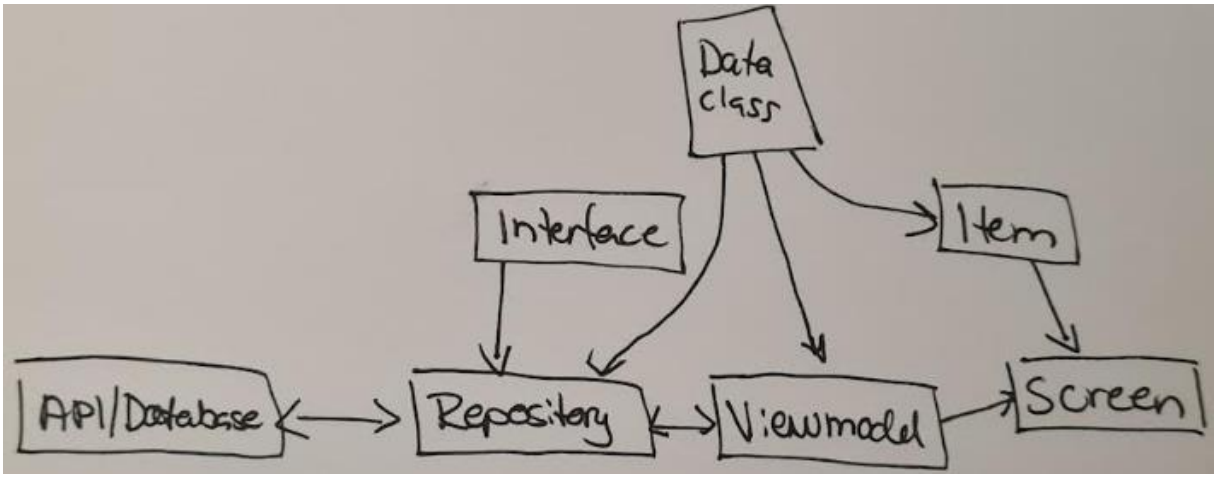
Room-databasen initieres i MainActivity når applikasjonen kjøres. Alle viewmodels blir opprettet og initiert i MainActivity. Her blir AppNavigation-funksjonen kalt på, hvor den tar inn alle viewmodels som parameter. AppNavigation har all navigasjon i appen og bruker Compose Navigation-biblioteket som gjør at man kan velge på en meny nederst på skjermen hvilken

skjerm man ønsker å se.

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    bottomBar = {
        NavigationBar(
            containerColor = Color( red: 128,  green: 203,  blue: 196,  alpha: 255)
        ) {
            NavigationBarItem(selected = currentScreen == 0,
                onClick = { currentScreen = 0
                          navController.navigate(Characters)},
                icon = {
                    if (currentScreen == 0){
                        Icon(imageVector = Icons.Filled.AccountBox,
                            contentDescription = null)
                    } else {
                        Icon(imageVector = Icons.Outlined.AccountBox,
                            contentDescription = null)
                    }
                },
                label = {
                    Text(text = "Characters")
                },
                colors = bottomBarTheme
            )   // end navigationBarItem
```

Figur 1: Del av koden i AppNavigation

Viewmodels er en del av MVVM-arkitekturen, en måte å organisere koden sin på, og er bindeleddet mellom kommunikasjon med lokal database eller Web API og grensesnittet. Det kobler seg til et repository, et objekt som gjør litt forskjellige ting ettersom det er lokal database eller Web API det gjør kall mot.



Figur 2: Illustrasjon av MVVM-arkitektur

For lokal database brukes Room, et bibliotek som tar i bruk SQLite-database som er på Android-enheter. Interfacet DAO, Data Access Object, som inneholder SQL-spørringer mot SQLite-databasen brukes for å lese og lage karakterer. Det er en del av CRUD, Create, Read, Update og Delete, operasjoner man kan gjøre mot databaser. I denne besvarelsen har jeg med Read og Create. Repository har ansvar for å definere og opprette databasen samt at den har funksjoner som gjør kall mot databasen, getAllCharacters og createNewCharacter.

```
// Definerer og oppretter databasen
fun initializeDatabase(context: Context) {
    _rickandmortyDatabase = Room.databaseBuilder(
        context,
        RickandmortyDatabase::class.java,
        name: "rickandmorty-database"
    ).build()
}

// Henter du alle karakterer som er lagret i databasen
suspend fun getAllCharacters() : List<Rickandmorty> {
    try{
        return _rickandmortyDao.getAllCreatedCharacters()
    }catch (e: SQLException){
        //TODO Skriv ut melding til bruker
        Log.d( tag: "Feil i databasen", e.toString())
        return emptyList()
    } catch (e: Exception){
        //TODO Skriv ut melding til bruker
        Log.d( tag: "Annen feil", e.toString())
        return emptyList()
    }
}
```

Figur 3: Room-database + funksjon for å hente ut alle karakterer.

Retrofit2-teknikken brukes for å gjøre kall mot Web API. Interfacet har metoder der man legger inn hvilke endpoints man ønsker å hente informasjon fra. Interfacet blir brukt av repository til å utføre kall mot Web API'et. Der er det også metoder som oppretter en http-klient som gjør det mulig å koble seg opp mot Web API'et samt funksjoner for selve kallet, fetchAllCharacters henter ut alle karakterene og getCharacterByName henter ut navn som brukeren har søkt

på. Disse blir brukt i sine respektive viewmodels som sender informasjonen til screens.

Screen-filene har en Composable-funksjon som annoteres med @Composable. Disse kan gjenbrukes flere steder i koden. Et eksempel på det er HeaderComposable som har «banneret» som går igjen på alle sider. Består av en kolonne med bilde, tekst og en bord. Ved å legge en String-variabel som parameter kan jeg enkelt kalle på Composable på samtlige skjermer og kun legge inn egendefinert tekst som overskrift for den aktuelle siden som variabel for så å sende den inn som parameter i funksjonen. For det innholdet som hentes fra API/database som skal lastes inn på nettsiden har jeg laget egne Composables i egne filer også, «Item»-filer, for grensesnittet som skal være for hver ting som hentes frem til appen.

```
Column (modifier = Modifier
    .fillMaxWidth()
    .background(
        Color( red: 137,  green: 191,  blue: 200),
    )
){
    LazyColumn{
        items(characters.value) { character ->
            CharacterItem(character = character)
        }
    }
}
```

Figur 4: Henter data fra Item-fil til å printe ut i screen.

Composables for skjermene tar i bruk LaunchedEffect, som kjører koden i viewmodels, hvor man får tak i informasjon fra API'et/databasen og legger det til grensesnittet via LazyColumn, som i dette tilfellet er det beste valget (fremfor Column) fordi det gjør at ikke alt som er hentet fra API'et rendres/lastes inn med en gang, men etter hvert som brukeren scroller nedover i appen. Dette er en fordel for at ikke hele API'et skal lastes inn på en gang, noe som kan gjøre

appen treg når så mye data blir lastet inn på en gang.

```
@Composable
fun ShowCharactersScreen(showCharacterViewModel: ShowCharactersViewModel){

    // LaunchedEffect-metode som kjører viewmodel
    LaunchedEffect(Unit) {
        showCharacterViewModel.loadCharacters()
    }
```

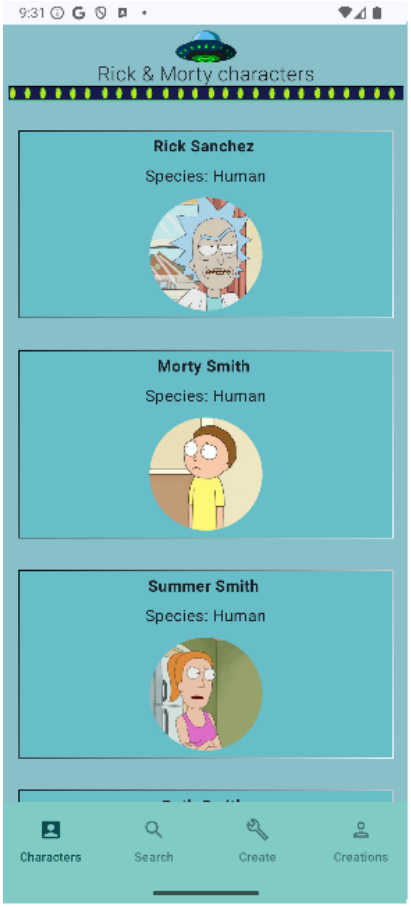
Figur 5:LaunchedEffect kjører viewmodels-koden.

En teknikk i Jetpack Compose som tar vare på endringer i variablers verdi underveis kalles remember. Man sier at variablene har en state. I stedet for å sette nye verdier variablene og oppdatere hele appen, så blir bare variablene oppdatert i appen. Det gjør programmet mer smidig og effektiv.

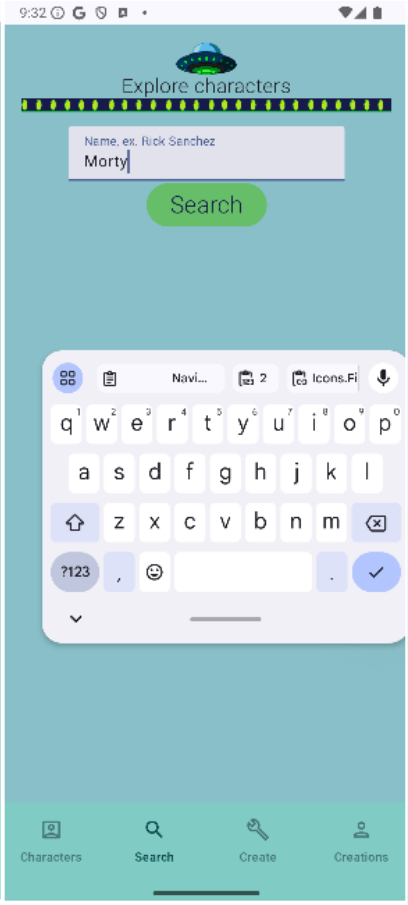
Kvalitet og struktur

For å skape god prosjektstruktur har jeg brukt packages – mapper. Basert på MVVM har jeg fordelt dataklasser i en egen pakke, fordelt filer relatert til room og retrofit i hver sine for å gjøre det enklere å orientere seg i hvilke filer som tilhører hvilke teknikker. Navngiving av variabler og klasser har jeg forsøkt å gjøre så beskrivende som mulig, men må innrømme at det var en liten utfordring, spesielt klassenavn. Har forsøkt å holde meg til substantiv + verb, men ser at «Rickandmorty»-dataklassen mangler klarhet i navnet sitt.

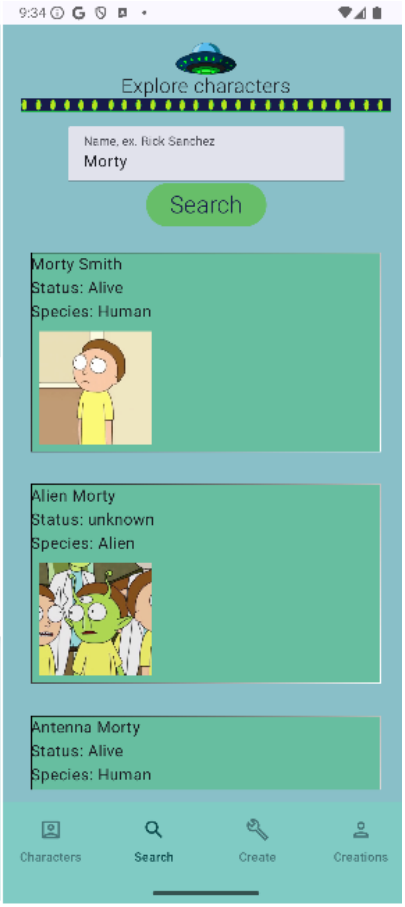
Skjermbilder



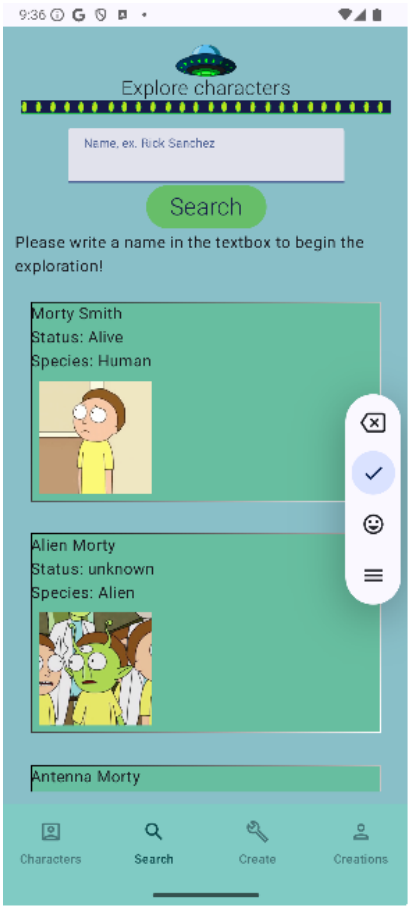
Skjerm 1. Statisk. Bruker kan scrolle for å vise alle karakterene tilgjengelig i API'et



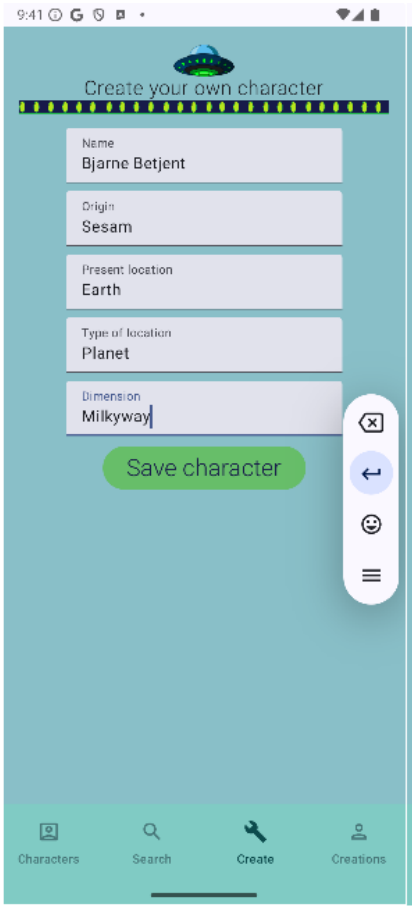
Skjerm 2: Bruker kan søke i API'et etter karakterer.



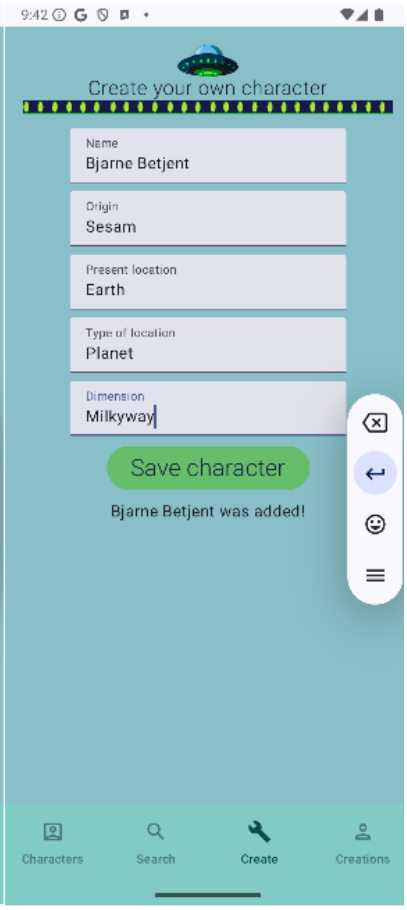
Resultat av søket



Skjerm 2: Tilbakemelding hvis bruker trykker søk uten å skrive noe.



Skjerm 3: Bruker lager egen karakter, trykker "Save Character når ferdig



Skjerm 3: Bruker får tilbakemelding når ny karakter er lagt til

Skjerm 4: Bruker kan få se sine egne karakterer.

Kilde til ufo-bildet ufo_4778062_640.png: https://pixabay.com/vectors/ufo-alien-ship-spaceship-alien-4778062/ ufoborder.png har jeg laget selv med Microsoft Paint.