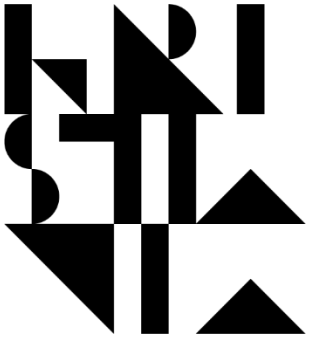
PGR208

Android Programming

Hjemmeeksamen

Høyskolen Kristiania

Kandidat nr. 4, 91, 181



Høst 2024

Innholdsfortegnelse

1 Oversikt over funksjonalitet

Denne rapporten beskriver utviklingen av en Rick and Morty-basert Android-app for hjemmeeksamen. Appen inkluderer skjermer og funksjoner som demonstrerer teknikker fra emnet, samt ekstra funksjonalitet som filtrering, favoritter og show/hide for å oppfylle oppgavekravene.

Funksjonalitet	Beskrivelse
Skjerm 1: Vise API karakterer	Viser en liste med Rick and Morty-karakterer hentet fra API.
Underfunksjoner:	
- Favorittfunksjon (ekstra krav)	Brukeren kan markere karakterer som favoritter og vise de i en liste.
- Filtrering (utover ekstra krav)	Brukeren kan filtrere karakterer etter status eller kjønn.
Skjerm 2: Vise brukers karakterer	Viser karakterer lagret lokalt av brukeren.
Underfunksjoner:	
- Filtrering (ekstra krav)	Brukeren kan filtrere karakterer etter status eller kjønn.
Skjerm 3: Opprett karakter	Gir brukeren mulighet til å lage egne karakterer i lokal database.
Underfunksjoner:	
- Tilbakemelding ved brukerinput (ekstra krav)	Brukeren får en melding om ikke alle tekstfelt er utfylt
- Tilbakemelding ved lagring (utover ekstra krav)	Brukeren får info(navn) om karakterer som er lagt i database
Skjerm 4: Slette karakter	Lar brukeren slette lagrede karakterer fra den lokale databasen.
Underfunksjoner:	
- Show/Hide-funksjon (ekstra krav)	Brukeren kan klikke på et ikon for å utvide/skjule detaljert informasjon.
- Tilbakemelding ved sletting (utover ekstra krav)	Brukeren får en bekreftelse når karakteren er slettet.
Skjerm 5: Vise Favoritter (ekstra krav)	Viser valgte favoritter fra skjerm 1
Underfunksjoner:	
- Fjern favoritter	Brukeren kan fjerne karakterer fra listen over favoritter

Figur 1 Funksjonalitet og beskrivelse

2 Beskrivelser av skjermbilder og hovedteknikker

Kapittelet gir en oversikt over appens skjermer og teknikker, med skjermdump og forklaring av funksjonalitet. Hovedteknikker som ViewModel, Retrofit og Room Database beskrives én gang, der de er mest relevante, for å unngå repetisjon.

2.1 Skjerm 1: Vise API karakterene

Skjerm 1 viser en liste over karakterer hentet fra Rick and Morty API-et, med navn, status, kjønn, art og bilde. Ekstrafunksjonen lar brukeren markere favoritter med bruk av FavouriteCharacterViewModel.

Figur 3 2.1 Skjerm 1: Vise API karakterene

2.1.1 ViewModel

```
class FavouriteCharactersViewModel : ViewModel() {
    private val _favCharacters = MutableStateFlow<List<Character>>(emptyList())
    val favCharacters = _favCharacters.asStateFlow()

    fun addToFavorites(character: Character) {
        viewModelScope.launch {
            try {
                val currentFavorites = _favCharacters.value.toMutableList()
                if (!currentFavorites.contains(character)) {
                    currentFavorites.add(character)
                    _favCharacters.value = currentFavorites
                    Log.d(
                        tag: "FavouriteCharactersViewModel",
                        msg: "${character.name} succesfully added to favourites."
                    )
                }
            } catch (e: Exception) {
                Log.e( tag: "FavouriteCharactersViewModel",
                    msg: "Error. Failed to add: ${character.name} to favourites.", e
```

Figur 2 ViewModel

ViewModel fungerer som en bro mellom databasen og brukergrensesnittet. I dette tilfellet brukes funksjon til å håndtere favorittkarakterene i appen. Dette ViewModel holder styr på hvilke karakterer brukeren har lagt til som favoritter, og oppdaterer brukergrensesnittet.

2.1.2 HTTP-kall med Retrofit

Retrofit er et verktøy vi brukte for å hente data fra et web-API på en enkel og effektiv måte. Vi definerte API-et med en GET-metode i et interface, som henter karakterer fra "character"-endepunktet til character repository som kommuniserer med databasen.

```
private val _okHttpClient = OkHttpClient.Builder()
    .addInterceptor(
        HttpLoggingInterceptor().setLevel(
            HttpLoggingInterceptor.Level.BODY
        )
    ).build()

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

Figur 4 Character repository

```
// Hente alle karakterer fra WebAPIet og returnerer "resultat" arrayet
suspend fun getAllCharacters(): List<Character> {
    try {
        val response = _characterService.getAllCharacters()

        if (response.isSuccessful) {
            return response.body()?.results ?: emptyList()
        } else {
            return emptyList()
        }
    } catch (e: Exception) {
        Log.e( tag: "Retrofit service error, can not get characters.", e.toString())
        return emptyList()
    }
}
```

Figur 5 HTTP-kall med Retrofit

2.1.3 LazyColumn

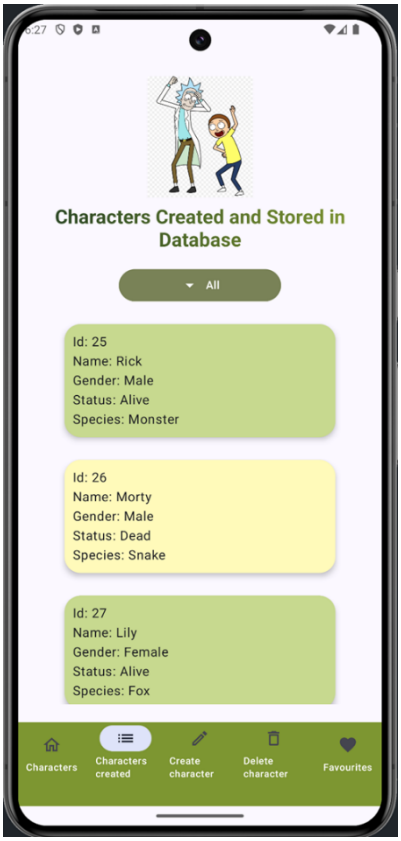
Karakterene som hentes fra API-et, presenteres i appen ved hjelp av en LazyColumn. LazyColumn er optimalisert for store datasett ved kun å laste elementene som er synlige på skjermen. Figur 6 illustrerer hvordan hver karakter vises gjennom en LazyColumn.

```
LazyColumn(
    verticalArrangement = Arrangement.spacedBy(12.dp),
    modifier = Modifier.fillMaxWidth()
) {
    if (characters.isEmpty()) {
        item {
            Text(
                text = "No characters found.",
                fontSize = 16.sp,
                modifier = Modifier.padding(16.dp)
            )
        }
    } else {
        items(characters) { character ->
            CharacterItem(
                character = character,
                modifier = Modifier.padding(8.dp)
            )
        }
    }
```

Figur 6 LazyColumn

2.2 Skjerm 2: Brukers karakterer

Skjerm 2 viser en liste over karakterer lagret i databasen. Hvis listen er tom, informeres bruker. Ekstrafunksjonen lar brukeren filtrere karakterene etter kjønn (Male, Female, Unknown).



Figur 7 Skjerm 2: Brukers karakterer

```
fun filterByGender(gender: String) {
    try {
        _filteredCharacters.value = if (gender == "All") {
            _allCharacters.value
        } else {
            _allCharacters.value.filter { it.gender == gender }
        }
        Log.d(
            tag: "CharacterViewModel", msg: "" +
                    "Filtered characters by gender: $gender"
        )
```

Figur 8 filterByGender

2.2.1 Lokal datalagring

For å lagre karakterene bruker vi Room Database, som gir en strukturert måte å håndtere lokal lagring på. Vi definerte attributter til databasen. Attributtene matcher JSON data fra RickMorty APIet som illustrert i Figur 9.

```
@Entity
data class Character(
    @PrimaryKey(autoGenerate = true) // database autogenerer Id
    // har med instansvariabler som matcher JSON webAPIet rickandmortyapi.com/api/character
    val id: Int = 0,
    val name: String,
    val status: String,
    val species: String,
    val gender: String,
    val image: String
)
```

Figur 9 Karakter database

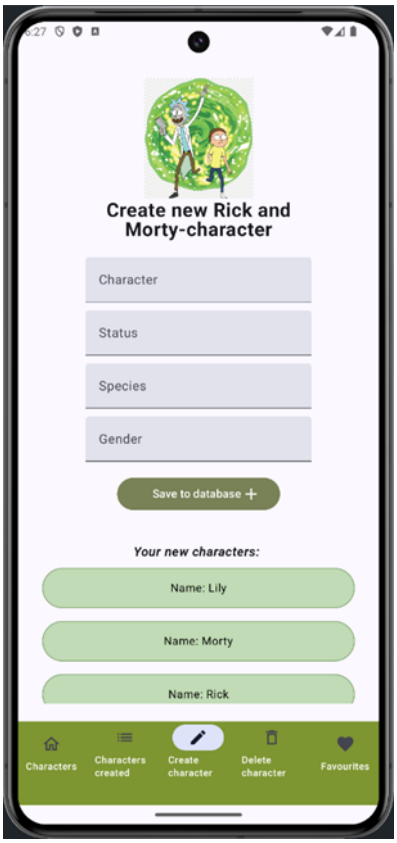
For å hente data fra databasen bruker vi en DAO (Data Access Object) gjennom Interface, som definerer SQL-spørringer for å legge til, hente og slette karakterer (Figur 10).

```
@Dao
interface CharacterDao {
    @Query("SELECT * FROM Character")
    suspend fun getCharacters(): List<Character>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacter(character: Character): Long
    // Long pga ny id blir lagret som karakterer som gis tilbake

    @Delete // Sletting returnerer int (0=dårlig, 1=bra)
    suspend fun deleteCharacter(character: Character): Int
}
```

Figur 10 SQL spørringer

2.3 Skjerm 3: Opprett karakter

Skjerm 3 lar brukeren opprette egne karakterer ved å fylle inn navn, status, art og kjønn. Dataene lagres i databasen ved bruk av Room. Som ekstrafunksjon får brukeren tilbakemelding ved tekstinput om ikke alle felt er utfylt.



Figur 11 Skjerm 3: Opprett karakter

2.3.1 ”by remember” og “mutableStateOf”

For å oppdatere tilstanden til inputfeltene bruker vi “by remember” og “mutableStateOf”. Variablene holder en verdi som kan bli endret over tid gjennom state, som oppdaterer UI ved bruk av mutableStateOf. Remember sørger for at verdiene ikke resettes når Composable kjøres, men blir husket og oppdatert i UI.

```
@Composable
fun CreateCharacterScreen(createCharacterViewModel: CreateCharacterViewModel) {

    val characters = createCharacterViewModel.characters.collectAsState()

    var newCharacterName by remember {
        mutableStateOf("")
    }
    var newCharacterStatus by remember {
        mutableStateOf("")
    }
    var newCharacterSpecies by remember {
        mutableStateOf("")
    }
    var newCharacterGender by remember {
        mutableStateOf("")
    }
```

Figur 12 ”by remember” og “mutableStateOf”

2.3.2 Try catch feilhåndtering

insertCharacter funksjonen i CharacterViewModel bruker coroutines til å utføre database spørringer i bakgrunnen (Dispatchers.IO). Karakterer blir lagt til i database effektivt ved bruk av bakgrunnsoppdateringer asynkron. Ved feil blir det håndtert trygt ved bruk av try catch som viser både feilmelding, og bruker blir informert at handlingen ikke kan utføres.

```
fun insertCharacter(character: Character) {
    viewModelScope.launch(Dispatchers.IO) {
        try {
            val newCharacterId = CharacterRepository.insertCharacter(character)
            if (newCharacterId != -1L) {
                val newCharacter = character.copy(id = newCharacterId.toInt())
                Log.d(
                    tag: "CreateCharacterViewModel",
                    msg: "Successfully inserted: ${newCharacter.name}"
                )
                _characters.value += newCharacter
            } else {
                Log.e( // Error message
                    tag: "CreateCharacterViewModel",
                    msg: "Could not insert character: ${character.name} in the database."
                )
            }
        } catch (e: Exception) {
            Log.e(
                tag: "CreateCharacterViewModel",
                msg: "Error. Failed to insert character to database ${e.message} ",
                e
```

Figur 13 Try catch feilhåndtering

2.4 Skjerm 4: Slette karakter

Denne skjermen lar brukeren slette lagrede karakterer fra databasen (Figur 12). Karakterene vises i en liste, og brukeren kan trykke på søppelbøtte-ikonet for å slette den.

Ekstrafunksjonen som gjør at bruker kan trykke på navnet for å få resten av data til karakteren ved bruk av true false og “!” som modifiserer variablenes verdi.

Funksjonen kaller på Repository sin "suspend fun deleteCharacter" som sletter karakteren fra databasen, lager ny liste ved hjelp av filter og oppdateres i bakgrunnen med viewModelScope.

Figur 14 Skjerm 4: Slette karakter

```
fun deleteCharacter(character: Character) {
    viewModelScope.launch(Dispatchers.IO) {
        try {
            val characterDeleted =
                CharacterRepository.deleteCharacter(character) // Sletter fra Db
            if (characterDeleted == 1) { // Hvis slettingen var suksessfull
                val currentList = _characters.value // Lagrer midlertidig listen
                val afterDeleteList = currentList.filter { it != character }
                _characters.value =
                    afterDeleteList // Oppdaterer ny liste filtrert uten den slettede.
                // Setter fra state og grensesnitt oppdateres.
                Log.d( tag: "DeleteCharacterViewModel", msg: "Successfully deleted: ${character.name}.")
            } else {
                Log.e(
                    tag: "DeleteCharacterViewModel",
                    msg: "Failed to delete: ${character.name} from database."
                )
            }
        } catch (e: Exception) {
            Log.e(
                tag: "DeleteCharacterViewModel",
                msg: "Error. Failed to delete: ${character.name} ${e.message}.",
                e
```

Figur 15 deleteCharacter

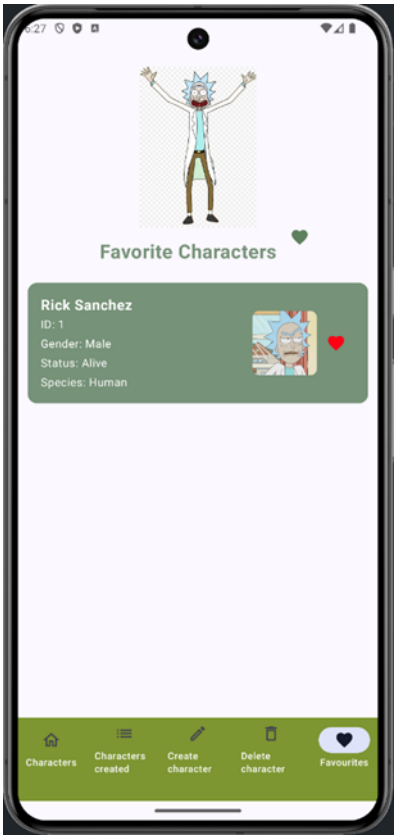
2.4.1 onDelete

onDelete er en lambdafunksjon som verken returnerer verdi eller tar imot input og kan gjenbrukes. DeleteCharacterScreen gjenbruker CharacterItem, for å unngå duplikat og for å vise til UI når bruker klikker på karakternavn for hele dataen om karakterene.

```
// Delete icon
if (onDelete != null) {
    IconButton(onClick = { onDelete() }) {
        Row {
            Icon(
                imageVector = Icons.Filled.Delete,
                contentDescription = "Delete icon"
            )
        }
    }
}
```

Figur 16 onDelete

2.5 Skjerm 5: Vise favoritter



Figur 17 Skjerm 5: Vise favoritter

Denne skjermen er laget for å vise en liste over karakterer som brukeren har markert som favoritter fra skjerm 1. Favorittene lagres med en markør i databasen som identifiserer dem, og brukeren kan også fjerne en karakter fra favorittlisten ved å trykke på et ikon. Når dette gjøres, oppdateres UI og endringen lagres direkte i databasen.

Når bruker klikker på søppel-ikonet, kalles removeFromFavourites-funksjonen i ViewModel som fjerner karakter.

```
fun removeFromFavorites(character: Character) {
    viewModelScope.launch {
        try {
            val currentFavorites = _favCharacters.value.toMutableList()
            if (currentFavorites.remove(character)) {
                _favCharacters.value = currentFavorites
            } else {
                Log.e(
                    tag: "FavouriteCharactersViewModel",
                    msg: "Error. Failed to remove: ${character.name} from favourites."
                )
            }
        } catch (e: Exception) {
            Log.e( tag: "FavouriteCharactersViewModel",
                msg: "Could not remove character ${character.name} from favourite ${e.message}.", e
            )
```

Figur 18 removeFromFavourites-

OnClick fra ShowAllCharacterScreen kaller på ViewModel sin funksjon i figur 18 som utføres ved klikk.

```
IconButton(
    onClick = {
        if (isFavorite) {
            favouriteCharactersViewModel.removeFromFavorites(character)
        } else {
            favouriteCharactersViewModel.addToFavorites(character)
        }
    }
```

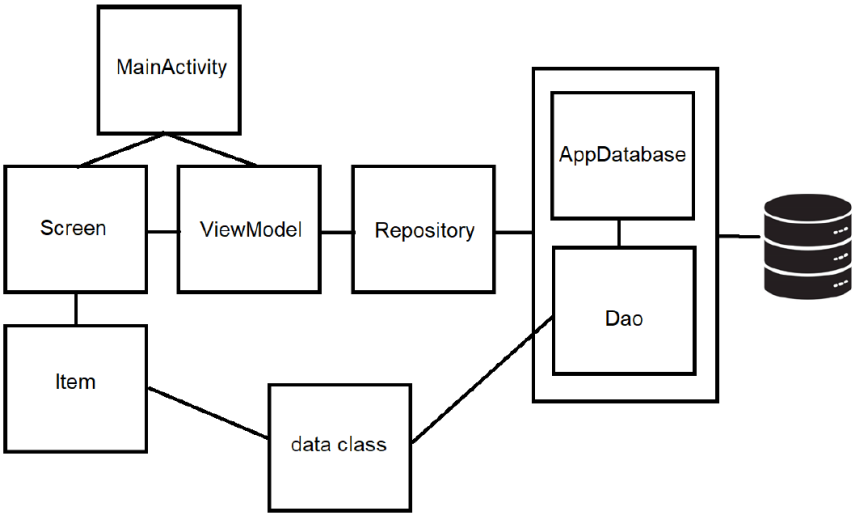
Figur 19 onClick

AsyncImage viser bilde av karakterene fra den angitte WebAPI-url. Det foregår asynkront for å ikke blokkere UI ved treg app eller nettverksfeil, men lastes ned i bakgrunnen istedet.

```
AsyncImage(
    model = favCharacter.image,  // Bildets URL
    contentDescription = "Image of ${favCharacter.name}",
    modifier = Modifier
        .size(80.dp) // Sett ønsket størrelse på bildet
        .clip(RoundedCornerShape(8.dp))  // Rund hjørnene
)
```

Figur 20 AsyncImage

3 Kvalitet og struktur



Figur 21MVVM-arkitektur

Vi har brukt MVVM-arkitekturen for å sørge for god prosjektstruktur og ansvarsfordeling. Som diagrammet viser, har vi delt prosjektet inn i klare komponenter: ViewModel håndterer logikken og kommuniserer med Repository, som igjen kobler til DAO og databasen. Skjermene (Screens) er ansvarlige for brukergrensesnittet.

Navngivning er viktig for lesbarhet. For eksempel beskriver DeleteCharacterViewModel tydelig at den håndterer slettelogikk, og funksjonen deleteCard() viser eksplisitt hva den gjør.

Når det gjelder samarbeid, delte vi arbeidet slik at én person fokuserte på API-integrasjon, en annen på databasen, og en tredje på frontend. Vi samarbeidet om større funksjoner, som sletting og favoritter, og delte koden gjennom jevnlige møter og kodegjennomganger. Det fungerte godt, men vi kunne forbedret planleggingen for å unngå små misforståelser om hvordan delene skulle integreres.

Denne oppgaven har vært utfordrende og lærerik. Vi føler at vi har vokst som utviklere og har en mer helhetlig forståelse av app utvikling. Dette har gitt mersmak og motivert oss for å satse videre på dette i arbeidslivet.