Rapport

1. Oversikt over funksjonalitet

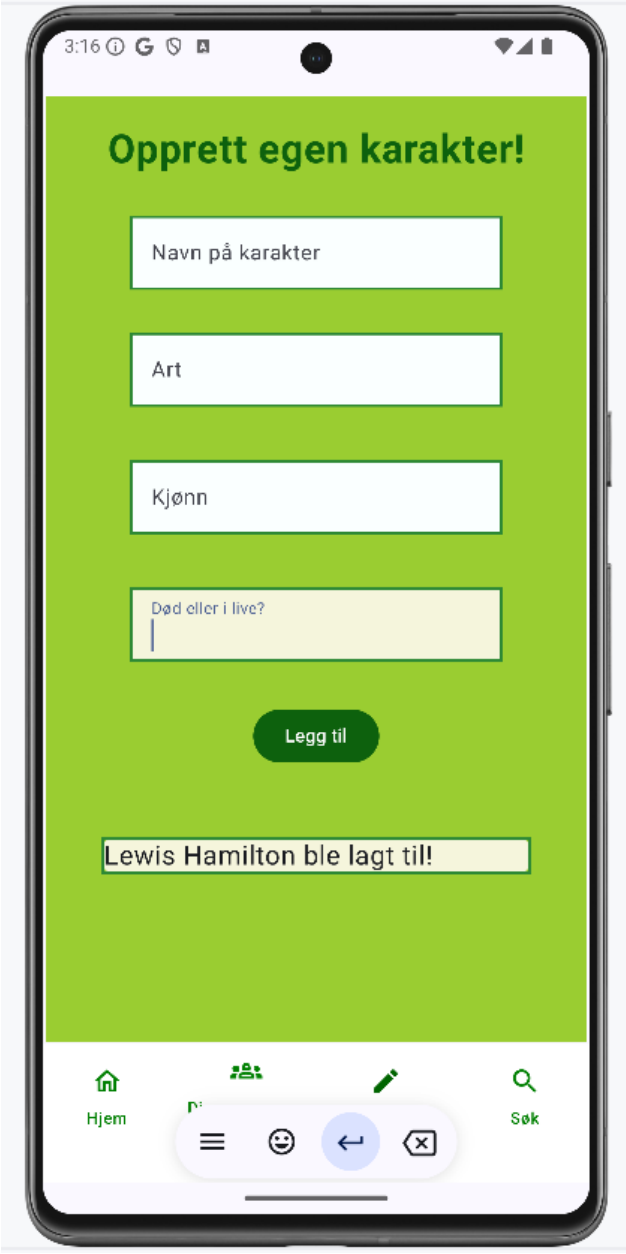
Navigering	Bruker kan trykke mellom skjermer
Opprette objekter	Som lagres i database
Produktboks	Visuell fremvisning av objekter i database og API
Slette objekt	Bruker kan slette egne objekter fra databasen
Søke	Etter navn

2. Skjermdump av samtlige skjermer

Skjermbilde nummer 1 og 2

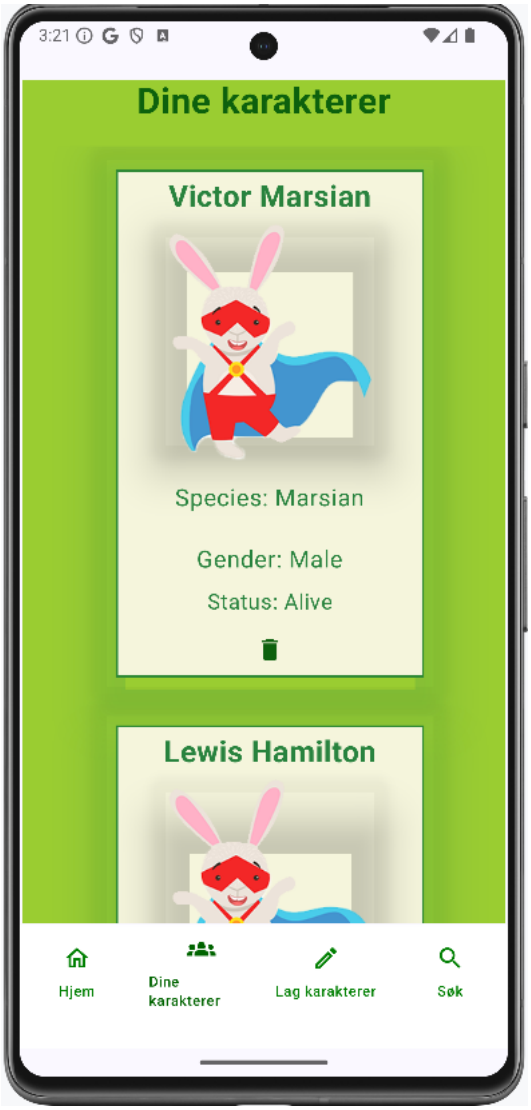
- 1: viser liste over alle karakterene i API
- 2: Bruker kan opprette egen karakter, som blir lagret i database

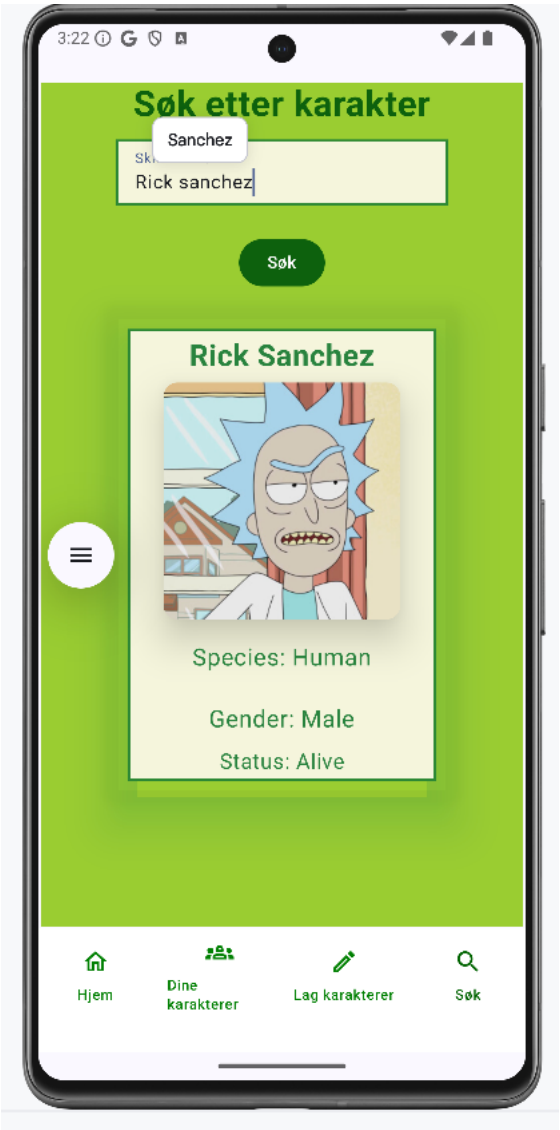

3:12
Rick & Morty Characters
Rick Sanchez
Species: Human
Gender: Male
Status: Alive
Morty Smith
Hjem
Dine karakterer
Lag karakterer
Søk


3:16
Opprett egen karakter!
Navn på karakter
Art
Kjønn
Død eller i live?
Legg til
Lewis Hamilton ble lagt til!
Hjem
Søk

Skjermbilde nummer 3 og 4:

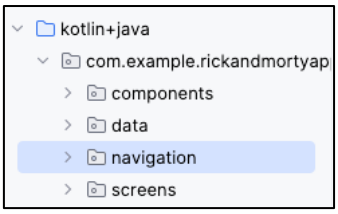
- 3: Viser liste over karakterene brukeren har lagd selv, som er lagret i databasen
- 4: Bruker kan søke på karakterene i API med navn

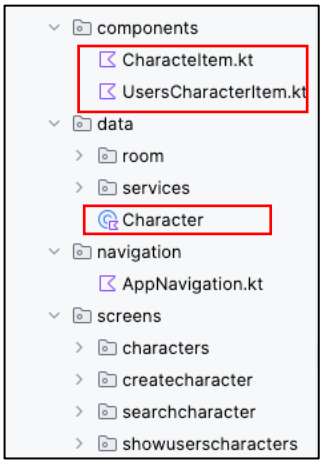




3. Beskrivelser og skjermbilder av hovedteknikker

Mappestruktur som fordeler mapper og filer etter ansvarsområde. Inne i components ligger filer som definerer hvordan karakterene skal vises i UI. Disse filene baserer seg på vilkår definert i dataklassen «Character», som igjen definerer Entity og Primarykey til databasen.





Database - Room

Interface DAO – Data Access Object

Definerer bruken av databasen og inneholder spørringene. Ettersom jeg kun har en tabell, har jeg kun en DAO, men ville hatt flere dersom jeg hadde hatt flere tabeller. Dette for å dele opp koden til å håndtere færrest mulig ting.

```
@Dao
interface CharacterDao {

    // Leser
    @Query("SELECT * FROM Character")
    suspend fun getCharacters(): List<Character>

    // Legger til
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacter(character: Character): Long

    // Sletter fra databasen/room
    @Query("DELETE FROM Character")
    suspend fun deleteCharacters()
}
```

Oppretter databasen og håndterer spørringene i DAO. Hver tabell ville hatt sin egen DAO, men her er kun én. Her starter koden som konfigurerer databasen, men den ferdigstilles i Repository.

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

Repository for database. Databasen blir ferdig konfigurert her og initialiseres. Denne funksjonen blir hentet inn i mainActivity, for å initialisere databasen før skjermene blir hentet inn via AppNavigation.

Videre har den de metodene som asynkront håndterer spørringene fra DAO.

```
object CharacterRepository {

    // AppDatabase
    // RickAndMortyDatabase
    private lateinit var _rickandmortydatabase : RickAndMortyDatabase
    private val _characterDao by lazy { _rickandmortydatabase.characterDao() }

    // 1. ferdigstille konfigurering av database
    fun initializeDatabase(context: Context){
        _rickandmortydatabase = Room.databaseBuilder(
            context = context,
            klass = RickAndMortyDatabase::class.java,
            name = "rickandmorty-database"
        ).build()
        Log.d( tag: "Databasen initialiseres her", context.toString())
    }
    // 2. Repository-metoder
    suspend fun getCharacters() : List<Character>{
        try {
            return _characterDao.getCharacters()
        } catch (e: SQLException){
            Log.d( tag: "Databasefeil", e.toString())
            return emptyList()
        } catch (e: Exception){
            Log.d( tag: "Annen feil", e.toString())
            return emptyList()
        }
```

```
    suspend fun insertCharacter( character: Character) : Long {
        try {
            return _characterDao.insertCharacter(character)
        } catch (e: SQLException){
            return -1L
        }
    }

    suspend fun deleteCharacters(){
        try {
            return _characterDao.deleteCharacters()
        } catch(e: Exception){
            return
        }
    }
}
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    CharacterRepository.initializeDatabase(applicationContext)
    enableEdgeToEdge()
    setContent {
        RickAndMortyAppTheme {
```

API

Hente data fra API. Repository er et object som tar vare på informasjonen – som et depot.

Her er fire punkter/oppgaver:

1. HTTP-client – selve oppkoblingen til API, 2. Retrofitt – har base-url for hvor dataene skal hentes, 3. Service-objekt – som henter informasjonen, 4. funksjoner som gjør bruk av service. I de tre første punktene er private variabler og bruker derfor understrek i variabelnavnet, i henhold til konvensjon i Kotlin.

Det siste punktet er funksjoner som skal brukes asynkront, for å vente på at oppkoblingen har fungert. Disse funksjonene er de som responderer på http-kallet og sender informasjonen fra api´et videre i applikasjonen.

```
object CharacterRepository {

    //Oppkobling til API
    // 1. HTTP-Client

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

    // 2. Retrofitt- objekt

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

    // 3. Service - objekt

        private val _characterService = _retrofit.create(CharacterService::class.java)

    // 4. Funksjon som gjør bruk av service
```

Data class CharacterList er der for å håndtere arrayet «results» som er i API´et.

```
data class CharacterList(
    val results: List<Character>
)
```

Screen og viewmodel

ViewModel er slik jeg forstår det, som et bindeledd mellom «depoet» og UI. Den henter inn dataene via _characterRepository og oppretter en liste med karakterer, som i utgangspunktet er tom, men som fylles med innhold fra API´et. ViewModelScopeLaunch gjør at alt det som skjer i bakgrunnen med oppkobling til API´et ikke gjør appen treger mens den henter dataene. Denne filen sender listen med karakterer til Screen.

Screen har viewmodelen som parameter, som innhenter data. Her opprettes en variabel som inneholder alle karakterene og bruker items for å gå igjennom listen og printe det ut i henhold til «reglene»/layouten som er definert i CharacterItem-komponenten.

```
class CharacterListViewModel : ViewModel(){

    private val _characterRepository : CharacterRepository = CharacterRepository

    private val _showCharacters = MutableStateFlow<List<Character>>(emptyList())
    val showCharacters = _showCharacters.asStateFlow()

    fun loadCharacters(){
        //etter endring blir det "emitet" til composable screen
        viewModelScope.launch {
            _showCharacters.value = _characterRepository.getAllCharacters()
    }
}
```

```
@Composable
fun CharacterListScreen(characterListViewModel: CharacterListViewModel) { // Koble

    // State
    val characters = characterListViewModel.showCharacters.collectAsState()

    // Koden som henter info - asynkront/coroutine
    LaunchedEffect(Unit) {
            characterListViewModel.loadCharacters()
    }
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .padding(2.dp)
            .background(Color( red: 154, green: 205, blue: 50)),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = "Rick & Morty Characters",
            fontSize = 30.sp,
            color = Color( red: 12, green: 96, blue: 13),
            fontWeight = FontWeight.Bold
        )
            LazyColumn {
                items(characters.value) { character ->
                    CharacterItem(character)
                }
            }
    }
}
```

Navigasjon

Definerer skjermene med @Serializable

```
//Fire skjermer
@Serializable
object Home

@Serializable
object UsersCharacters

@Serializable
object CreateCharacter

@Serializable
object SearchCharacter
```

Oppretter navController og bruker state for å vite hvilken skjerm brukeren er på med remember. Oppretter deretter NavigationBar og NavigationBarItem, hvorav sistnevnte er en enkelt knapp for en skjerm. Knappene har filled, altså uthevet ikon, ut ifra hvilken currentScreen som er aktivert, hvilket sjekkes med if else. De andre knappene er da ikke filled. NavHost blir brukt til å håndtere hvilken composable screen som skal tilhøre hvilken Serializable-objekt.

```
val navController = rememberNavController()

var currentScreen by remember {
    mutableStateOf( value: 0)
}
```

```
// 1. knapp - Hjem
NavigationBarItem(
    selected = currentScreen == 0, // == sammenligner
    onClick = {
        currentScreen = 0 // = er lik
        navController.navigate(Home)
    },
    icon = {
        if (currentScreen == 0) {
            Icon(
                imageVector = Icons.Filled.Home,
                contentDescription = null
            )
        } else {
            Icon(
                imageVector = Icons.Outlined.Home,
                contentDescription = null
            )
        }
    },
    label = {
        Text(text = "Hjem")
    },
    colors = colorScheme
)
```

```
) {
    NavHost(
        navController = navController,
        startDestination = Home
    ) {
        composable<Home> {
            CharacterListScreen(characterListViewModel)
        }
        composable<UsersCharacters> {
            ShowUsersCharactersScreen(showUsersCharactersViewModel)
        }
        composable<CreateCharacter> {
            CreateCharacterScreen(createCharacterViewModel)
        }
        composable<SearchCharacter> {
            SearchCharacterScreen(searchCharacterViewModel)
        }
    }
```

4. Kvalitet og struktur

Jeg har delt opp prosjektet i fire mapper som håndterer komponentene, dataene, navigasjonen og skjermene. Dataene er videre fordelt i undermapper for å håndtere database og api. Hver skjerm har sin egen mappe, inkl. sin egen viewModel.

Jeg foretrekker å dele opp koden over flere linjer fremfor å ha lange linjer med kode, som for eksempel i modifikasjonene i Column og i Text. Angående navngivning har jeg forsøkt å holde meg til Kotlins kodekonvensjon. Klasser har UpperCamelCase og funksjoner har lowerCamelCase. Alt er navngitt på engelsk, men kommentarene er på norsk.

Jeg har lagt inn linker der jeg har brukt andre kilder enn pensum, mye fordi jeg selv ofte har lyst til å vite hvor noe er hentet fra ved senere anledninger.