Timebox7 – UnlockFrag

Oversigt

OpgaveNavn	Implementering af UnlockFrag		
Implementering af krav	Hel implementering af: APP-4, APP-5, APP-6, APP-7, APP-8		
Udført af	Jan	**Dato**	08-11-2021
Timebox	7	**Område**	App

Contents

Introduktion

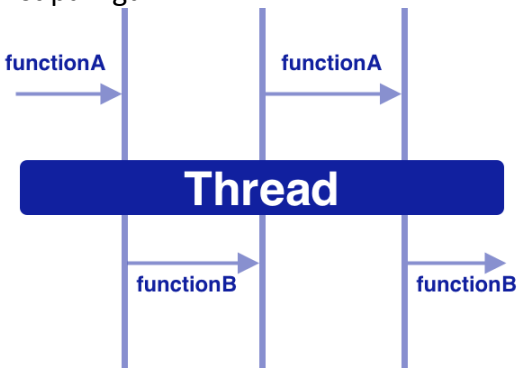
Det skal være muligt igennem denne del af appen at:

- Hente en nøgle fra restAPI'en, til en specifik facilitet, på hjemmesiden.
- Sende en log "entry" til databasen på hjemmesiden, gennem hjemmesidens restAPI.
- Sende nøglen til embed låsen vedhjælp af Bluetooth.

Analyse + implementering

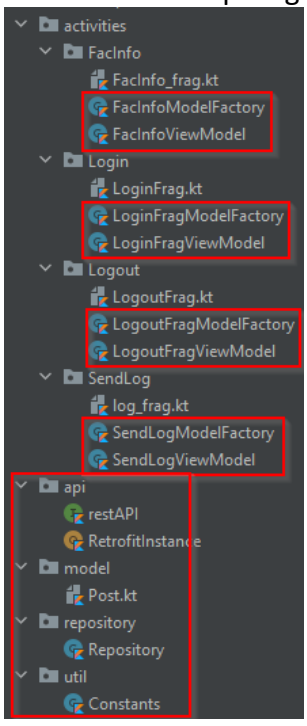
Corutines:
Da der trækkes meget på telefonens resurser til forholdsvis Bluetooth, restAPI kald og display af ens layout, er det nødvendigt at enten bruge threads eller coroutines i appen, i tilfælde af disse ikke bruges, vil der være mulighed for ”skipped frames”, hvilket vil kunne have seriøse konsekvenser for om appen virker.
Til dette fragment i appen, er der valgt at blive brugt coroutines.

Coroutines er en simpel version af threading, dog i stedet for at lave en ny thread til opgaverne, kører de alle sammen på samme thread, men man ”styrer” hvilke opgaver der skal køre hvornår, som er vist på Figur 1.



Figur 1 coroutines eksempel [1]

RestAPI:
For at kunne kontakte restAPI’en bliver der brugt retroFit biblioteket, hvor de forskellige nødvendige filer for at restAPI kaldet virker kan ses på Figur 2.



Figur 2 retrofit filer

[1] https://blog.mindorks.com/mastering-kotlin-coroutines-in-android-step-by-step-guide

Filerne gør følgende:

- Util -> Constants:
  - De forskellige faste variabler, som fx hjemmeside URL'en.

```
const val BASE_URL = "http://www.control-center.xyz/"
```

Figur 3 constants eksempel

- Repository -> Repository:
  - Hvilke funktioner der skal være til restAPI'en, som håndterer kaldene til restAPI'en, som yderlig er sat til at køre separat fra fragmenterne de er kaldt i, med kotlins suspend funktion.

```
suspend fun pushPostSendLog(Authorization: String, postSendLog: PostSendLog): Response<PostSendLog> {
    return RetrofitInstance.api.pushPostSendLog(Authorization, postSendLog)
}
```

Figur 4 repository eksempel

- Api -> Retrofit instance:
  - Opbygningen af retrofit kaldende:
    - Indsæt url -> konverter til Json -> indsæt body.

```
object RetrofitInstance {
    private val retrofit by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
```

Figur 5 retrofit instance eksempel

- Api -> restAPI:
  - opbygningen af de forskellige kald.

```
@POST( value: "rest_api/key_api/")
suspend fun pushPostGetFacInfo(
    @Header( value: "Authorization") Authorization: String,
    @Body postGetFacInfo: PostGetFacInfo
): Response<PostGetFacInfo>
```

Figur 6 restAPI eksempel

- model -> Post
  - classes med de forskellige variable der skal sendes/modtages.

```
data class PostLogin (
    val email: String?,
    val password: String?,
    val auth_token: String?
)
```

Figur 7 post eksempel

- *ModelFactory:
    - Override af ViewModels, baseret på den custom skrevet viewModel udvidelse (*ViewModel).

```
class SendLogModelFactory(private val repository: Repository) : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return SendLogViewModel(repository) as T
    }
}
```

Figur 8 ModelFactory eksempel

- *ViewModel:
    - Udvidelse af viewModel, så funktionerne fra repository kan bruges.

```
class SendLogViewModel(private val repository: Repository): ViewModel() {

    val myResponse: MutableLiveData<Response<PostSendLog>> = MutableLiveData()

    fun pushPost(Authorization: String, postSendLog: PostSendLog) {
        viewModelScope.launch { this: CoroutineScope
            val response: Response<PostSendLog> = repository.pushPostSendLog(Authorization, postSendLog)
            myResponse.value = response
        }
    }

}
```

Figur 9 viewModel udvidelses eksempel

Når alle disse filer er oprettet, kan der sendes til restAPI'en fra et fragment I appen, ved at lave en variable instans af den viewModel der skal bruges, hvilket gør at man kan tilgå funktionen der bruges til at sende til restAPI'en.

Bluetooth:
For at Bluetooth enheden til at virke, oprettes der et Bluetooth adapter object, hvor fra et Bluetooth client socket object kan oprettes, der gør det muligt at sende til en Bluetooth server, såfremt man har Mac adressen til Bluetooth server enheden, samt har givet sin klient et UUID, hvilket er gjort gennem hardcoding I appen.

Efter Bluetooth client socket objektet er færdig med sit arbejde, skal der lukkes for socketen.

De forskellige funktioner til Bluetooth socket, er forholdsvis selvsigende, fx. for at oprette forbindelse er kaldet ".connect".

Fragment:
I stedet for at køre appen I forskellige aktiviteter, er der blevet valgt at bruge fragmenter som nævnt I "TB6-Generel struktur.docx".

Fragmentet der skal navigeres videre til, er et success fragment, ved afslutning af besked sendelse eller hvis der sker en fejl, hvor der skal sendes følgende data med over til:

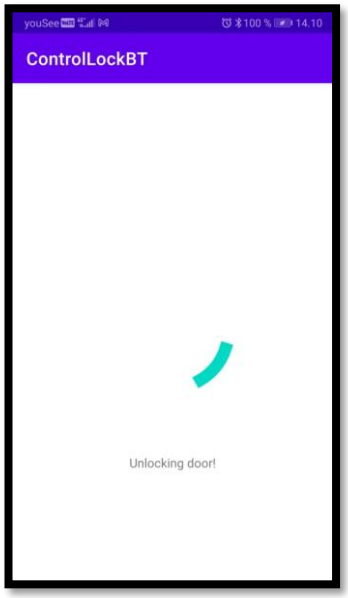
- Login token.

- Bruger email.
- Facilitets navn.
- Status kode for om noget gik galt.

BackButton håndtering:
Da coroutinerne kan ødelægge appen, hvis de bliver afbrudt, er der valgt at overskrive back knappen på telefon, så denne knap ikke kan gøre noget.

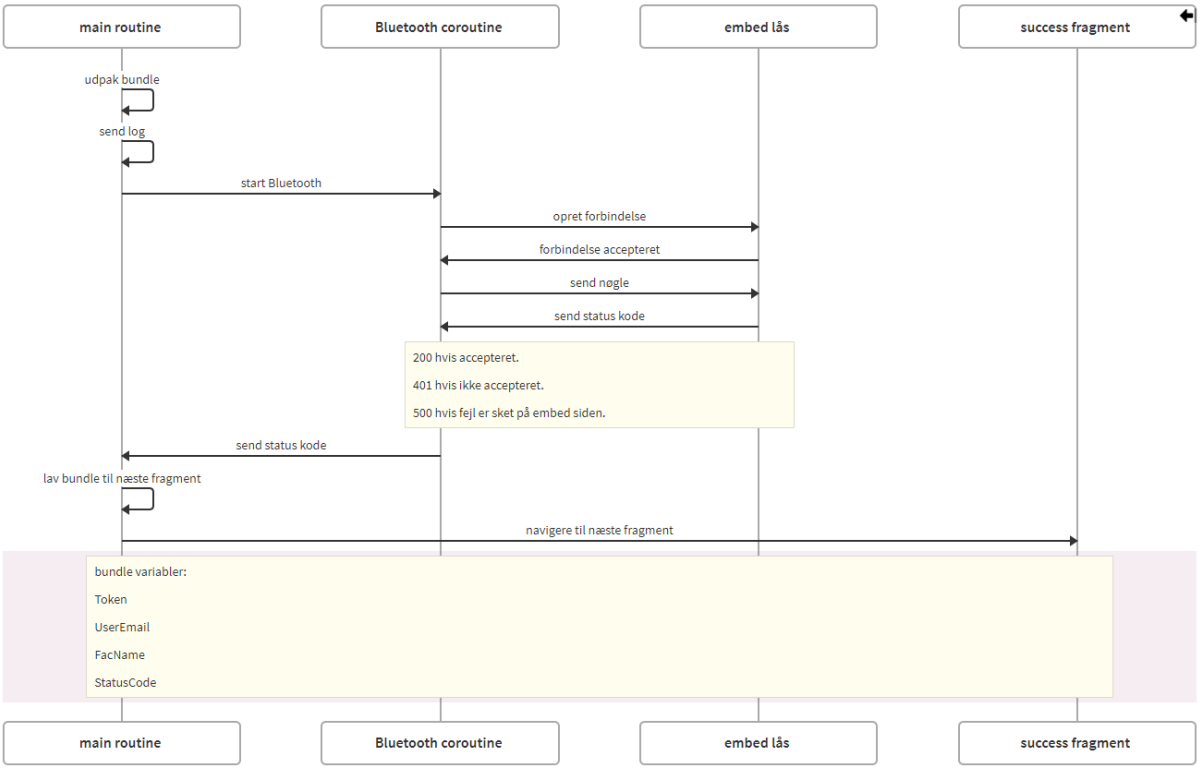
Design

Grafisk:
Eftersom der ikke er nogen bruger input på dette fragment, er der kun en besked og en loadingbar, så længe fragmentet er ved at udføre arbejde, som kan ses på Figur 10.



Figur 10 unlock grafisk design

Kode flow:
Først bliver bundlen fra forrige fragment packet ud, for derefter at lave en entry I loggen på hjemmesiden database, gennem restAPI'en, hvorefter der bliver startet en coroutine op, hvor der skal genereres en forbindelse til låsen gennem Bluetooth.
Når der er forbundet til embedden, bliver koden sendt, for derefter at vente på svar fra embedden.
Efter et svar fra embedden er modtaget, bliver Bluetooth forbindelsen afbrudt, og der bliver navigeret til success fragmentet sammen med en status kode, på om transaktion lykkedes eller fejlede.
Dette flow kan også ses på Figur 11.



Figur 11 kode flow

Verifikation

For at denne del af appen kan sættes som verificeret, vil den skulle kunne:

- Etablere forbindelse til en lås.
- Kunne sende log gennem restAPI'en.
- Kunne modtage log retur beskeden.
- Kunne sende nøglen til låsen.
- Navigere videre til næste fragment.

Der antages for at disse test virker, at der er sat et helt testmiljø op med en bruger der har adgang til test facilitet, samt at facilitet og bruger er oprettet på hjemmesiden, og de korrekte tilladelser er blevet givet til brugeren.

Tabel 1: Tests til verifikation af opgave

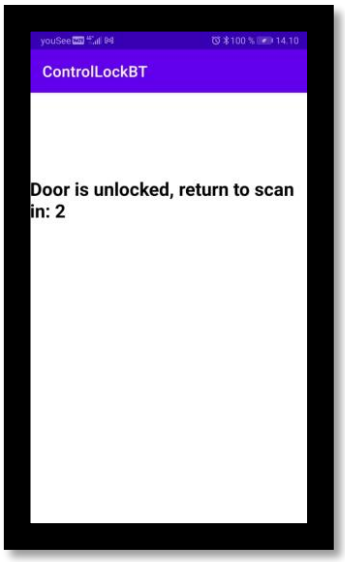
Test	Test Steps	Pass-betingelser	Resultat
Forbindelse	1. åben app 2. log ind på app 3. scan efter test facilitet 4. tryk på test facilitet 5. vent til der bliver navigeret til success fragmentet	låsen bliver låst op	Bestået
Send log	Som overstående	Der kan på hjemmesidens database ses en log entry	Bestået
Modtag log	Som overstående	Låsen bliver låst op	Bestået
nøgle	Som overstående	Låsen bliver låst op	Bestået
Næste frag.	Som overstående	Der bliver navigeret hen til success fragmentet	Bestået

Testresultat

Resultat for test ”forbindelse”, ”modtag log” og ”nøgle” kan ses på Figur 12.

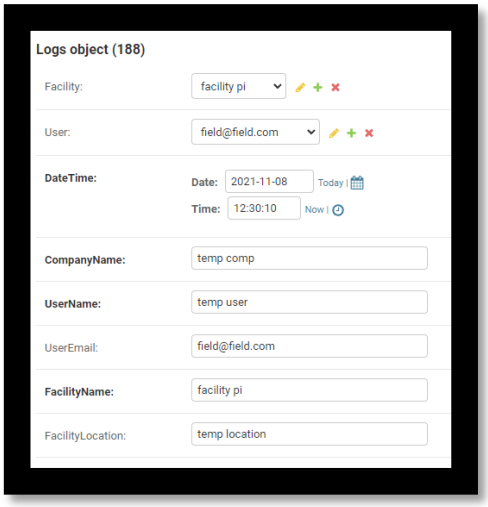
Figur 12 lås åben

Resultat for ”næste frag.” kan ses på Figur 13.



Figur 13 success frag

Resultat for ”send log” kan ses på Figur 14.



Figur 14 log entry

Konklusion

Efter implementeringen af denne opgave er det nu muligt og åbne låsen gennem appen, dog skal det noteres at dette fragment ikke virker, uden det foregående scan fragment!