



Arquiteturas Móveis

## Trabalho Prático 1

Docente: Álvaro Nuno Santos

Gonçalo Leite	-	2018014643
Gonçalo Pereira	-	2019139650
Inês Estevão	-	2011016171

# Índice

1.	Introdução .....	3
2.	Código Fonte.....	4
3.	Estruturas do Trabalho .....	5
4	Implementação .....	10

# 1. Introdução

Este trabalho surge no âmbito da Unidade Curricular de Arquiteturas Móveis, no 1º Semestre do ano letivo de 2023/2024. A sua implementação foi feita recorrendo à linguagem Kotlin.

Este trabalho tem como foco desenvolver uma aplicação Android que facilite as visitas turísticas a diversas localizações, disponibilizando informação sobre os locais de interesse existentes nesses locais, podendo também consultar locais de interesse e categorias. É composto por três componentes essenciais: a Localização, os Locais de Interesse e as Categorias.

No âmbito da gestão e persistência de dados, utilizamos uma base de dados FireBase, consolidando uma abordagem eficiente e confiável

Ao longo deste trabalho, exploraremos detalhadamente cada componente do sistema, analisando sua importância e função específica.

## 2. Código Fonte

O nosso código encontra-se dividido em vários ficheiros separados em várias packages (data.models, data.repos, ui.screens, ui.theme, ui.viewmodels):

A nossa package “models” representa um conjunto de data classes responsáveis por definir e estruturar os dados fundamentais da aplicação.

A nossa package “repos” é utilizada para armazenar as classes relacionadas aos repositórios da aplicação, onde é feita a comunicação com a FireBase.

A nossa package “theme” é utilizada para definir os temas visuais e estilos da aplicação.

A nossa package “viewmodels” é utilizada para gerir a lógica de cada um dos screens, comunicando com os repositórios e fornecendo estados para a interface com o utilizador.

A nossa package “screens” representa as diferentes páginas da aplicação. É onde centralizamos todas as implementações relacionadas à interface do usuário, incluindo a lógica de apresentação para diferentes páginas.

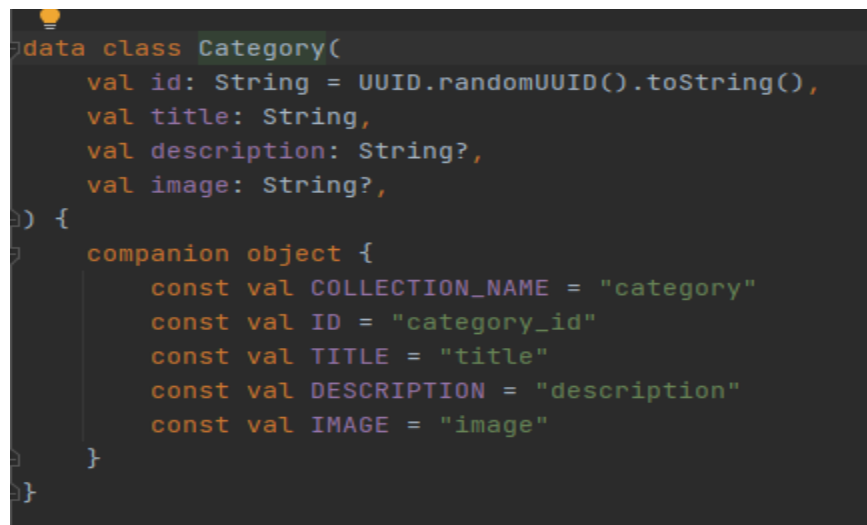
### 3. Estrutura do Trabalho

Neste trabalho, dividimos o nosso código todo por cinco diferentes packages como anunciado no ponto anterior, sendo elas :

- models
- repos
- screens
- Theme
- viewmodels

#### Package models

Nesta package vamos ter as data classes das categorias, dos locais e dos pontos de interesse.



```
data class Category(  
    val id: String = UUID.randomUUID().toString(),  
    val title: String,  
    val description: String?,  
    val image: String?,  
) {  
    companion object {  
        const val COLLECTION_NAME = "category"  
        const val ID = "category_id"  
        const val TITLE = "title"  
        const val DESCRIPTION = "description"  
        const val IMAGE = "image"  
    }  
}
```

Nesta imagem temos a data classe de uma categoria, em que vai ter que ser guardado um id, um titulo, uma descrição e uma imagem.

```
data class Location(  
    val id: String = UUID.randomUUID().toString(),  
    val title: String,  
    val description: String,  
    val latitude: Double,  
    val longitude: Double,  
) {  
    companion object {  
        const val ID = "location_id"  
        const val COLLECTION_NAME = "location"  
        const val TITLE = "title"  
        const val DESCRIPTION = "description"  
        const val LATITUDE = "latitude"  
        const val LONGITUDE = "longitude"  
        const val IMAGE = "image"  
        const val COORDINATES = "coordinates"  
        const val SCORE = "score"  
        const val STATE = "state"  
    }  
}
```

Nesta imagem temos a data classe de uma localização, em que vai ser guardado um id, um titulo, uma descrição, uma latitude e uma longitude.

```

data class PointOfInterest(
    val id: String = UUID.randomUUID().toString(),
    val title: String,
    val description: String,
    val latitude: Double,
    val longitude: Double,
    val categoryId: String,
    val categoryTitle: String,
    val locationId: String,
    val locationTitle: String,
) {
    companion object {
        const val ID = "point_of_interest_id"
        const val CATEGORY_ID = "category_id"
        const val CATEGORY_TITLE = "category_title"
        const val LOCATION_ID = "location_id"
        const val LOCATION_TITLE = "location_title"
        const val COLLECTION_NAME = "point_of_interest"
        const val TITLE = "title"
        const val DESCRIPTION = "description"
        const val LATITUDE = "latitude"
        const val LONGITUDE = "longitude"
        const val IMAGE = "image"
        const val COORDINATES = "coordinates"
        const val SCORE = "score"
        const val STATE = "state"
    }
}

```

Nesta imagem temos a data classe de um ponto de interesse onde guardamos um id, um titulo, uma descrição, a latitude, a longitude, um id de uma categoria, uma descrição de uma categoria, um id de uma localização e uma descrição de uma localização.

## Package repos

Nesta package temos os repositórios de cada uma das data classes juntamente com a autenticação de um utilizador. É também nesta package que comunicamos com a base de dados, onde podemos guardar os dados e listar os mesmos, como podemos ver na próxima imagem sobre uma categoria :

```
class CategoryRepository {  
  
    private val db = Firebase.firestore  
  
    fun saveCategory(category: Category, onSuccess: (Boolean, Throwable?) -> Unit) {  
        val c = db.collection(Category.COLLECTION_NAME).document(category.id)  
        db.runTransaction { transaction ->  
            val doc = transaction.get(c)  
            if(doc.exists()){  
                //if exists, updates document  
                transaction.update(c, Category.TITLE, category.title)  
                transaction.update(c, Category.DESCRPTION, category.description)  
                transaction.update(c, Category.IMAGE, category.image)  
                null ^runTransaction  
            } else {  
                //if not, creates one  
                c.set(hashMapOf(  
                    Category.ID to category.id,  
                    Category.TITLE to category.title,  
                    Category.DESCRPTION to category.description,  
                    Category.IMAGE to category.image  
                )) ^runTransaction  
            }  
        }.addOnCompleteListener { result ->  
            onSuccess(result.isSuccessful, result.exception)  
        }  
    }  
}
```



```

suspend fun getAllCategories(): List<Category> {
    return try {
        db.collection(Category.COLLECTION_NAME) CollectionReference
            .get() Task<QuerySnapshot!>
            .await() QuerySnapshot!
            .documents (Mutable)List<DocumentSnapshot!>
            .mapNotNull { it: DocumentSnapshot!
                it.toCategory()
            }
    } catch (e: Exception) {
        // Trate exceções ou retorne uma lista vazia em caso de erro
        emptyList()
    }
}

```

```

DocumentSnapshot.toCategory() : Category {
    val id = this.getString(Category.ID) ?: ""
    val title = this.getString(Category.TITLE) ?: ""
    val description = this.getString(Category.DESCRPTION)
    val image = this.getString(Category.IMAGE)
    return Category(

```

```

class CategoryViewModel(
    private val categoryRepository: CategoryRepository,
    private val pointOfInterestRepository: PointOfInterestRepository,
) : ViewModel() {

    val title = mutableStateOf<String>{ value: "" }
    val description = mutableStateOf<String>{ value: "" }
    val imagePath: MutableState<String?> = mutableStateOf<String?>{ value: null }

    private val _categories = MutableStateFlow<List<Category>>(emptyList())
    val categories: StateFlow<List<Category>> = _categories

    private val _categoryAddedStatus = mutableStateOf<Boolean>{ value: false }
    val categoryAddedStatus: State<Boolean>
    {
        get() = _categoryAddedStatus
    }

    private val _error = mutableStateOf<String?>{ value: null }
    val error : MutableState<String?>
    {
        get() = _error
    }

    var categoryDetails : Category? = null

    private val _pointsOfInterestOfCategory = MutableStateFlow<List<PointOfInterest>>(emptyList())
    val pointsOfInterestOfCategory: StateFlow<List<PointOfInterest>> = _pointsOfInterestOfCategory

    init {
        getCategories()
    }
}

```

```

fun getCategories() {
    viewModelScope.launch { this: CoroutineScope
        _categories.value = categoryRepository.getAllCategories()
    }
}

fun getPointsOfInterestOfCategory() {
    categoryDetails?.let { category ->
        viewModelScope.launch { this: CoroutineScope
            _pointsOfInterestOfCategory.value =
                pointOfInterestRepository.getPointsOfInterestOfCategory(category.id)
        }
    }
}

fun addCategory() {
    categoryRepository.saveCategory(
        Category(
            title = title.value,
            description = description.value,
            image = imagePath.value
        )
    ) { isSuccessfull, exception ->
        if (isSuccessfull) {
            getCategories()
            _error.value = null
        } else {
            _error.value = exception?.message
        }
    }
}

```

Nesta package é onde gerimos a lógica de cada screen, neste caso da categoria. Aqui a viewmodel interage com o repositório, onde podemos adicionar e listar todas as categorias.

## 4. Implementação

A implementação foi codificada da seguinte forma:

- Ao executar o programa é pedido ao utilizador se quer efetuar um login ou se quer registar um novo user. Será feita a autenticação do utilizador caso este exista ou se quer criar um novo, com email e password, através do repositório de autenticação (AuthenticationRepository). Isto é feito através do uso da Firebase. Caso seja autenticado, é nos apresentado um menu com várias opções onde pode escolher entre locais, locais de interesse, categorias, mapa e os créditos do trabalho. Se escolher locais, é apresentado uma lista

de locais presentes na FireBase, e selecionando um deles é nos apresentado a descrição desse local e as suas coordenadas. A viewmodel deste screen (Local) faz ligação com o repositório, listando todos os locais e podendo adicionar um local novo na FireBase. O resto da aplicação funciona assim com este método. Se escolher locais de interesse, é apresentado uma lista com todos os locais de interesse, mostrando o local e a categoria do mesmo. Temos dois tipos de mapa, um para os locais e outro para os pontos de interesse, quando selecionado é nos mostrado os detalhes de cada um, tanto para o local tanto para o ponto de interesse.