INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: FIREBASE - FIRESTORE DATABASE

Prática 06

Atenção: Esta prática é continuação da Prática 05. Use controle de versões

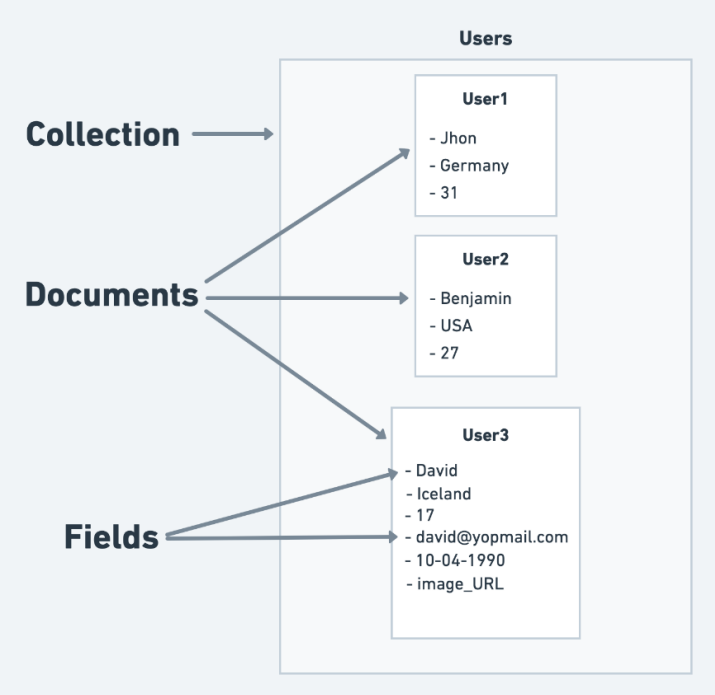
Parte 1: Preparação para usar o Firebase Firestore

Passo 1: No Android Studio, adicione suporte ao Firebase Firestore: *Tools* > *Firebase*.

Selecione a opção *Cloud Firestore* e depois *Get started ...*. Execute o passo 2 nessa nova aba (o 1º passo já deve estar feito pela Prática 05). Serão adicionadas as dependências necessárias ao *script gradle* da aplicação. Os demais passos são trechos de código para manipulação dos dados.

Passo 2: Habilitando e visualizando a base de dados no Firebase Console:

No Firebase Console, crie um banco de dados do tipo Cloud Firestore no projeto da prática (Criação > *Firestore Database*). Se perguntado(a) sobre permissões, deixe aberto para gravação e leitura ("**modo de teste"**); as regras de acesso devem ser revistas numa aplicação em produção. O *Firestore Database* organiza os dados na forma de coleções de documentos, que podem apontas para outras coleções, e assim sucessivamente.



https://docs.flutterflow.io/data-and-backend/firebase/firestore-database-cloud-firestore/creating-collections

Passo 3: Faça o build, rode e teste a aplicação. Dê commit se não houve problemas.

Parte 2: Refatoração e Arrumação

Passo 1: Tire `City` do arquivo de `MainViewModel` para um arquivo próprio no pacote `pdm.weatherapp.model` com o código abaixo:

```
data class City(
    val name: String,
    var weather: String,
    var location: LatLng? = null
)
```

Passo 2: Crie a classe `User` no pacote `pdm.weatherapp.model` com o código:

```
data class User(var name: String, var email: String)
```

Essa classe modela o usuário atualmente logado na aplicação.

Passo 3: Declare um objeto `User` dentro de `MainViewModel`:

```
private val _user = mutableStateOf (User("", ""))
val user : User
    get() = _user.value
```

Passo 4: Em `MainActivity`, mude o título da `topBar` para incluir o nome do usuário:

```
        title = { Text("Bem-vindo/a ${viewModel.user.name}") },
```

Passo 5: Rodar e testar.

Nesse ponto não deve haver mudanças significativas no comportamento do App. Se estiver tudo correto, faça um novo commit.

Parte 3: Criando do componente de banco de dados

Passo 1: Crie a classe `FBCity` em `pdm.weatherapp.db.fb`:

```
class FBCity {
    var name : String? = null
    var lat : Double? = null
    var lng : Double? = null

    fun toCity(): City {
        val latlng =  LatLng(lat ?: 0.0, lng ?: 0.0)
        return City(name!!, weather = "", location = latlng)
    }
}

fun City.toFBCity() : FBCity {
    val fbCity = FBCity()

    fbCity.name = this.name
    fbCity.lat = this.location?.latitude ?: 0.0
    fbCity.lng = this.location?.longitude ?: 0.0

    return fbCity
}
```

Essa classe é usada para serializar as cidades no Firebase Firestore. Ela precisa ter construtor *default* vazio e atributos "setáveis" que podem ser nulos. Também adicionamos métodos para transforma de/para `model.City`.

Passo 2: Crie a classe `FBUser` em `pdm.weatherapp.db.fb`:

```
class FBUser {
    var name : String ? = null
    var email : String? = null
    fun toUser() = User(name!!, email!!)
}

fun User.toFBUser() : FBUser {
    val fbUser = FBUser()
    fbUser.name = this.name
    fbUser.email = this.email
    return fbUser
}
```

Idem para o usuário e classe `model.User`.

Passo 3: Crie o objeto `FBDatabase` em `pdm.weatherapp.db.fb`:

```
class FBDatabase(private val listener: Listener? = null) {
    private val auth = Firebase.auth
    private val db = Firebase.firestore
    private var citiesListReg: ListenerRegistration? = null

    interface Listener {
        fun onUserLoaded(user: User)
        fun onCityAdded(city: City)
        fun onCityRemoved(city: City)
    }

    init {  ...  }

    fun register(user: User) { ... }

    fun add(city: City) { ... }

    fun remove(city: City) { ... }
}
```

Essa classe nos fornecerá acesso ao nosso BD no Firebase. Ela define um *listener* que é chamado sempre que houver um evento de cidade adicionada ou removida (`onCityAdded`/`onCityRemoved`), ou quando as informações do usuário logado forem carregas (`onUserLoaded`). Vamos instânciar essa classe nas atividades, passando `viewModel` como `listener`.

Passo 4: Em `FBDatabase`, use o código baixo no bloco `init`:

```
init {
    auth.addAuthStateListener { auth ->

        if (auth.currentUser == null) {
            citiesListReg?.remove()
            return@addAuthStateListener
        }

        val refCurrUser = db.collection("users")
               .document(auth.currentUser!!.uid)
        refCurrUser.get().addOnSuccessListener {
            it.toObject(FBUser::class.java)?.let { user ->
                listener?.onUserLoaded(user.toUser())
            }
        }

        citiesListReg = refCurrUser.collection("cities")
            .addSnapshotListener { snapshots, ex ->
                if (ex != null) return@addSnapshotListener

                snapshots?.documentChanges?.forEach { change ->
                    val fbCity = change.document.toObject(FBCity::class.java)
                    if (change.type == DocumentChange.Type.ADDED) {
                        listener?.onCityAdded(fbCity.toCity())
                    } else if (change.type == DocumentChange.Type.REMOVED) {
                        listener?.onCityRemoved(fbCity.toCity())
                    }
                }
            }
    }
}
```

Esse código é chamado na criação do `FBDatabase`. Ele registra um *listener* que é disparado quando o usuário faz login ou logout. Em caso de login, é iniciada a carga dos dados do usuário atual, a qual ao ser completada dispara o evento `listener?.onUserLoaded(...)` (a `?` indica que o *listener* pode ser *null*, nesse caso nada é feito). Também é registrado o *listener* `citiesListReg` que notifica sempre que uma cidade for adicionada (`onCityAdded()`) ou removida (`onCityRemoved()`) no BD. Quando o usuário faz *sign out*, o `citiesListReg` registrado é removido.

Passo 5: Implemente o método `FBDatabase.register()`:

```
fun register(user: User) {
    if (auth.currentUser == null)
        throw RuntimeException("User not logged in!")
    val uid = auth.currentUser!!.uid
    db.collection("users").document(uid + "").set(user.toFBUser());
}
```

Esse método salva um objeto `FBUser` na coleção `users` no Firebase Firestore e deve ser chamado em `RegisterPage` (ver mais a frente).

Passo 6: Implemente o método `FBDatabase.add()`:

```
fun add(city: City) {
    if (auth.currentUser == null)
        throw RuntimeException("User not logged in!")

    val uid = auth.currentUser!!.uid

    db.collection("users").document(uid).collection("cities")
        .document(city.name).set(city.toFBCity())
}
```

Nesse método, caso haja um usuário logado, é criado um registro para a cidade na coleção `cities` do usuário atual, com o nome da cidade como *id*.

Passo 7: Implemente o método `FBDatabase.remove()`:

```
fun remove(city: City) {
    if (auth.currentUser == null)
        throw RuntimeException("User not logged in!")

    val uid = auth.currentUser!!.uid
    db.collection("users").document(uid).collection("cities")
        .document(city.name).delete()
}
```

Nesse método, a cidade é removida da lista de cidades associada ao usuário atualmente logado.

Passo 8: Compile para ver se não há problemas. Não deve fazer diferença ao rodar nesse ponto. Faça um novo *commit* caso esteja tudo certo.

Parte 4: Conectando a UI com o Firebase DB.

Passo 1: Modifique `MainViewModel` de forma a implementar `FBDababase.Listener`:

```
class MainViewModel : BaseViewModel(), FBDatabase.Listener {

    ...

    override fun onUserLoaded(user: User) {
        _user.value = user
    }

    override fun onCityAdded(city: City) {
        _cities.add(city)
    }

    override fun onCityRemoved(city: City) {
        _cities.remove(city)
    }
}
```

Essa mudança faz com que o `MainViewModel` trate os eventos de adição/remoção de cidades e carga do usuário do `FBDatabase`. Nesse caso, a cidade é adicionada ou removida à lista de cidades (`_cities`), e o usuário é configurado na propriedade `_user`.

Passo 2: Em remova `MainViewModel.add()` e `.remove()` e faça os ajustes abaixo:

A função `getFavoriteCities()` também desnecessária e pode ser removida.

Onde eram chamados `viewModel.add()` e `viewModel.remove()` ao longo do código, altere para as chamadas correspondentes à `FBDatabase`. Uma instância de `FBDatabase` deve ser criada em `MainActivity` e passada como parâmetro até chegar as páginas `HomePage`, `ListPage` e `MapPage`:

```
val fbDB = remember { FBDatabase (viewModel) }
```

Em `MapPage`, ao adicionar novas cidade (via *click* no mapa), dê nomes únicos, pois estes são usados para identificar a cidade no BD. Usar as coordenadas (latitude e longitude) ou gerar número aleatório para o nome da cidade.

Passo 3: Salvando os dados do usuário em `RegisterPage`:

Instancie `FBDatabase` sem parâmetro:

```
val fbDB = remember { FBDatabase() }
```

Chame o código abaixo se o registro for bem-sucedido:

```
fbDB.register(User(name, email))
```

Passo 4: Rode e teste a aplicação.

Registre um novo usuário e adicione e remova cidades usando o mapa e o diálogo. **Atenção:** testar com usuários criados na prática 5 deve causar problemas por falta de dados desse usuário no BD.

Faça um *commit* se estiver tudo correto.