FIAP ON

ANDROID CLOUD INTEGRATION
& DEPLOYMENT

PERSISTÊNCIA DE DADOS COM FIREBASE E ANDROID

HEIDER PINHOLI LOPES

3

LISTA DE FIGURAS

Figura 3.1 - Criação do banco de dados Cloud Firestore 6
Figura 3.2 Regras do banco de dados 6
Figura 3.3 Local da instância do banco de dados 7
Figura 3.4 Criação da classe NewUser 12
Figura 3.5 Registro inserido no Authentication 17
Figura 3.6 Registro inserido no Cloud Firestore 18
Figura 3.7 Criação do pacote enums 19
Figura 3.8 Nomeação do pacote enums 19
Figura 3.9 Criando um arquivo enum 20
Figura 3.10 Criando o enum FuelType 20
Figura 3.11 Criando uma nova classe 21
Figura 3.12 Criando uma nova classe chamada Car 21
Figura 3.13 Criação do pacote watchers 23
Figura 3.14 Criação de uma nova classe 23
Figura 3.15 Criação da classe DecimalTextWatcher 23
Figura 3.16 Nomeação da classe DecimalTextWatcher 24

LISTA DE CÓDIGOS-FONTE

Código-fonte 3.1 – Objeto Firestore 5
Código-fonte 3.2 – Dependência do Firestore.....7
Código-fonte 3.3 – Tipos de dados do Cloud Firestore 9
Código-fonte 3.4 – Objetos personalizados no Cloud FIrestore.....9
Código-fonte 3.5 – Utilização de objetos personalizados.....9
Código-fonte 3.6 – Definindo um documento 10
Código-fonte 3.7 – Definindo um documento com merge 10
Código-fonte 3.8 – Adicionando um documento no Cloud Firestore 1 11
Código-fonte 3.9 – Adicionando um document no Cloud Firestore 2 11
Código-fonte 3.10 – Atualizando um documento no Cloud Firestore 12
Código-fonte 3.11 – Criação da classe NewUser.....13
Código-fonte 3.12 – SignUpViewModel.....15
Código-fonte 3.13 – Implementação do SignUpFragment 17
Código-fonte 3.14 – Ação para tela de criação de conta.....17
Código-fonte 3.15 – Enum FuelType.....20
Código-fonte 3.16 – Código da classe Car.....22
Código-fonte 3.17 – Código da classe BetterFuelViewModel 22
Código-fonte 3.18 – Código da classe DecimalTextWatcher 25
Código-fonte 3.19 – Método para exibir um AlertDialog.....28
Código-fonte 3.20 – Código do setup das views do fragment BetterFuelFragment ..31
Código-fonte 3.21 – Consulta Simples – Retorna estados de SP 32
Código-fonte 3.22 – Consulta Simples – Retorna as capitais.....32
Código-fonte 3.23 – Executar uma consulta 33
Código-fonte 3.24 – Operadores de consulta.....33
Código-fonte 3.25 – Consulta composta 34
Código-fonte 3.26 – Consulta composta válida.....34
Código-fonte 3.27 – Consulta composta inválida 34
Código-fonte 3.28 – Código para recuperar os dados do carro.....36
Código-fonte 3.29 – Busca dos dados do carro 38
Código-fonte 3.30 – Método para limpar os campos do formulário.....38

SUMÁRIO

3 PERSISTÊNCIA DE DADOS COM FIREBASE 5
3.1 Criando o banco no Cloud Firestore.....6
3.2 Gravando dados no Firestore.....7
3.3 Tipos de dados.....8
3.4 Definir um documento 9
3.5 Adicionar um documento.....10
3.6 Atualizar um documento.....12
3.7 Implementando o Firestore para gravar usuário.....12
3.8 Implementando o Firestore no Projeto 18
3.8.1 Salvando os dados no Firestore.....19
3.8.2 Acessando os dados do Firestore 31
3.8.3 Consulta simples 32
3.8.4 Operadores de consulta 33
3.8.5 Consulta composta.....33
3.8.6 Carregando os dados do veículo.....34
CONCLUSÃO.....40
REFERÊNCIAS.....41

3 PERSISTÊNCIA DE DADOS COM FIREBASE

Caso seu aplicativo necessite persistir outros dados, além do e-mail e telefone, na autenticação do usuário (pois estas informações ficam na base do próprio Firebase Authentication), você pode utilizar o Firebase Firestore como base de dados.

O Firestore armazena dados NoSQL com sintaxe JSON, a qual chamamos de documentos. Ele possui uma infraestrutura na qual seus dados são divididos em collections (coleções) e documents (documentos).

Um documento é como se fosse um objeto. Em cada objeto, temos alguns valores com suas respectivas chaves. Esses valores podem ser do tipo boolean, byte, int, float, String, arrays, datas e até null. Por exemplo:

```
{
  "id" : 1,
  "nome": "Heider Lopes",
  "email" : "heiderlopes@exemplo.com
}
```

Código-fonte 3.1 – Objeto Firestore
Fonte: Elaborado pelo autor (2020)

As coleções são conjuntos de documentos. Elas não podem conter outras coleções e os documentos não podem conter outros documentos. Sendo assim, uma coleção só pode conter documentos e os documentos só podem conter subcoleções.

Com esse banco de dados, é possível realizar queries e filtragens de dados, o que torna seu uso bem abrangente, cobrindo boa parte das necessidades de persistências de dados em aplicações.

3.1 Criando o banco no Cloud Firestore

No console do Firebase, crie o banco de dados.

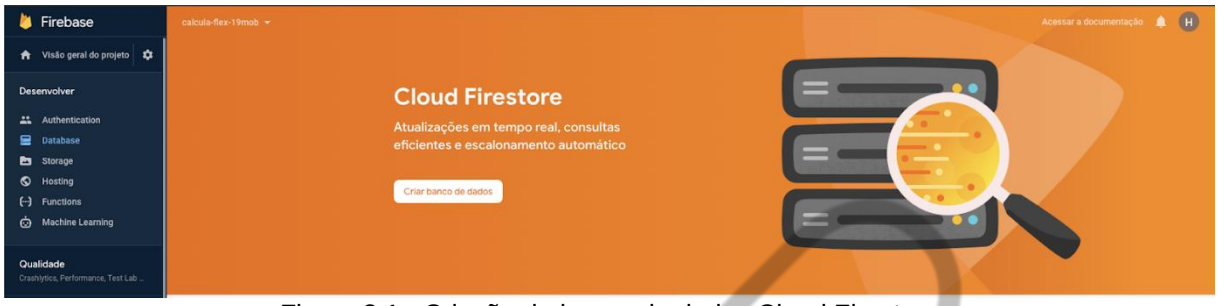

Figura 3.1 - Criação do banco de dados Cloud Firestore
Fonte: Elaborado pelo autor (2020)

Nesse momento, o projeto ficará aberto para gravações. É importante, quando o projeto for para produção, realizar as configurações mais restritivas da base de dados.

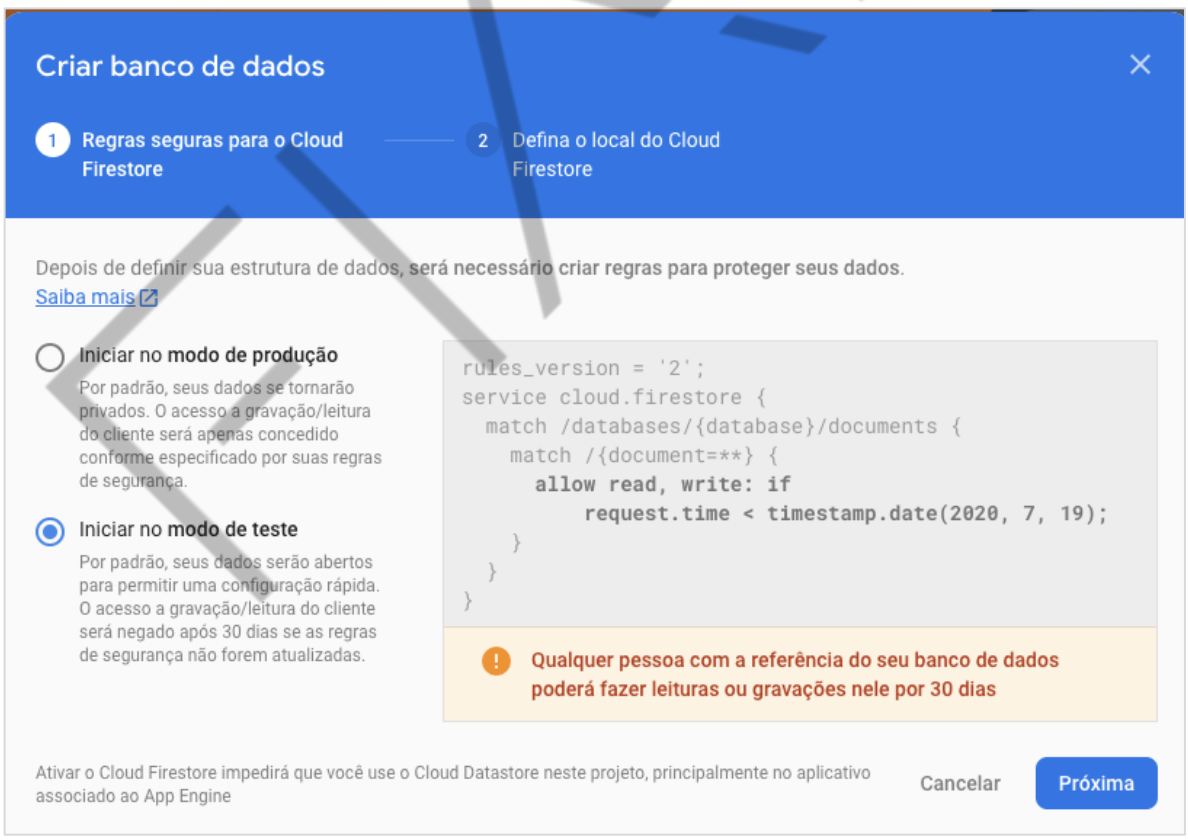

Figura 3.2 Regras do banco de dados
Fonte: Elaborado pelo autor (2020)

Defina o local em que irá criar seu banco. Quanto mais próximo da sua base de usuários, menor será a latência de acesso aos dados, ou seja, ficará mais rápido.

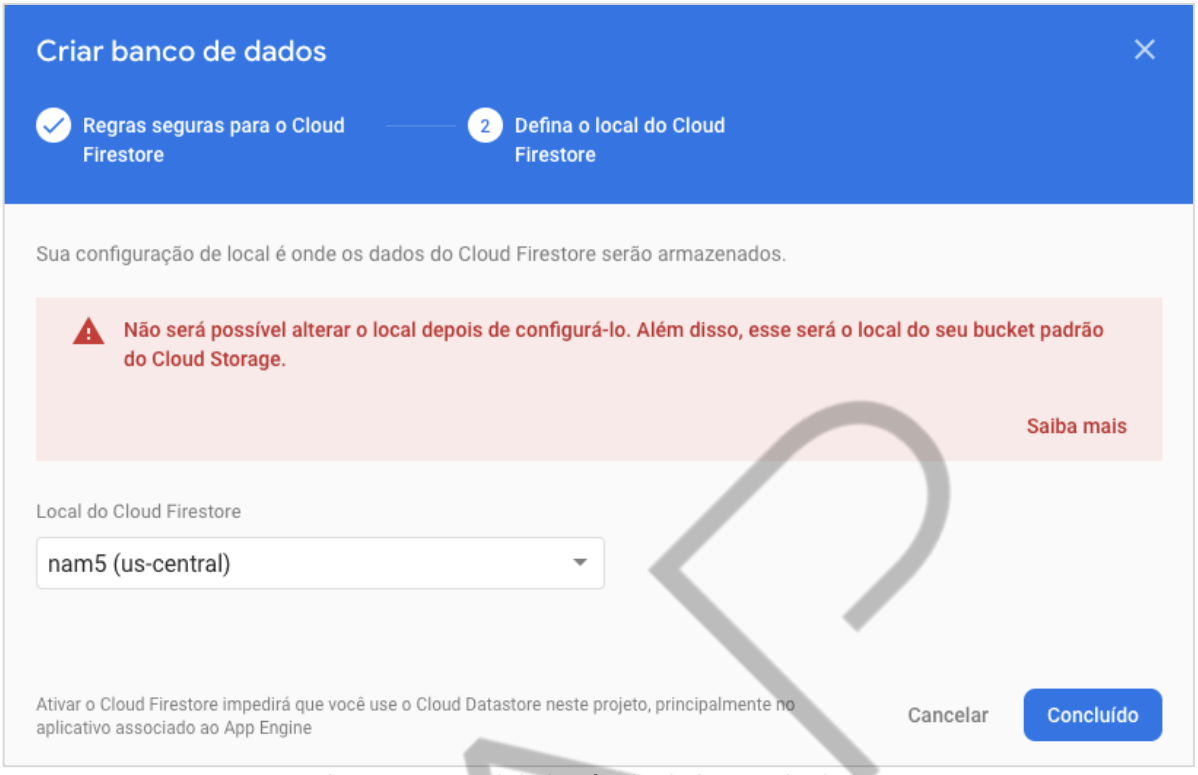

Figura 3.3 Local da instância do banco de dados
Fonte: Elaborado pelo autor (2020)

Para adicionar a dependência do Firestore ao seu projeto, abra o arquivo **build.gradle (app)** e adicione a seguinte dependência:

```
implementation 'com.google.firebase:firebase-firestore-ktx:21.5.0'
```

Código-fonte 3.2 – Dependência do Firestore
Fonte: Elaborado pelo autor (2020)

3.2 Gravando dados no Firestore

No Firestore existem duas formas de você realizar gravação de dados. Elas podem ser feitas de forma individual ou por meio de lotes. Abordaremos a gravação de forma individual. Neste contexto, podemos destacar as seguintes formas de realizar a gravação de dados:

- Definir os dados de um documento em uma coleção, especificando explicitamente um identificador de documento.

- Adicionar um novo documento a uma coleção. Neste caso, o Cloud Firestore gera automaticamente o identificador de documento (ID).
- Criar um documento vazio com um identificador gerado automaticamente e atribuir dados a ele posteriormente.

3.3 Tipos de dados

O Cloud Firestore permite gravar uma variedade de tipos de dados em um documento, incluindo strings, booleanos, números, datas, nulos, além de matrizes e objetos aninhados. O Cloud Firestore sempre armazena números como duplos, independentemente do tipo de número que você usa no código.

```
val docData = hashMapOf(
        "stringExemplo" to "Hello world!",
        "booleanExample" to true,
        "numberExample" to 3.14159265,
        "dateExample" to Timestamp(Date()),
        "listExample" to arrayListOf(1, 2, 3),
        "nullExample" to null
)

val nestedData = hashMapOf(
        "a" to 5,
        "b" to true
)

docData["objectExample"] = nestedData

db.collection("data").document("one")
        .set(docData)
        .addOnSuccessListener { Log.d(TAG, "DocumentSnapshot successfully
```

```
written!")                                                    }
        .addOnFailureListener { e -> Log.w(TAG, "Error writing document", e) }
```

Código-fonte 3.3 – Tipos de dados do Cloud Firestore
Fonte: firebase.google.com (s.d.)

Além de ser possível utilizar objetos **Map** ou **Dictionary** para representar seus documentos, o Cloud Firestore é compatível com a gravação de documentos com classes personalizadas. Ele converte os objetos em tipos de dados compatíveis.

Por exemplo:

```
data class City(
    val name: String? = null,
    val state: String? = null,
    val country: String? = null,
    val isCapital: Boolean? = null,
    val population: Long? = null,
    val regions: List<String>? = null
)
```

Código-fonte 3.4 – Objetos personalizados no Cloud FIrestore
Fonte: firebase.google.com (s.d.)

```
val city = City("Los Angeles", "CA", "USA",
        false, 5000000L, listOf("west_coast", "socal"))
db.collection("cities").document("LA").set(city)
```

Código-fonte 3.5 – Utilização de objetos personalizados
Fonte: firebase.google.com (s.d.)

3.4 Definir um documento

Para criar ou substituir um único documento, use o método **set()**. Se o documento não existir, ele será criado.

```
val cidade = hashMapOf(
    "nome" to "São Paulo",
    "estado" to "SP",
    "pais" to "Brasil"
)

db.collection("cidades").document("SP")
    .set(cidade)
    .addOnSuccessListener { Log.d(TAG, "Gravado com sucesso") }
    .addOnFailureListener { e -> Log.w(TAG, "Erro na gravação", e) }
```

Código-fonte 3.6 – Definindo um documento
Fonte: Adaptado de firebase.google.com (s.d.)

Se o documento existir, o conteúdo dele será substituído pelos dados recém-fornecidos, como os do código fonte “Definindo um document com merge”, a menos que você especifique que os dados devem ser incorporados ao documento existente:

```
// Atualize um campo, criando o document se ele ainda não existir.
val data = hashMapOf("capital" to true)

db.collection("cidades").document("SP")
    .set(data, SetOptions.merge())
```

Código-fonte 3.7 – Definindo um documento com merge
Fonte: firebase.google.com (s.d.)

3.5 Adicionar um documento

Ao usar set() para criar um documento, você precisa especificar um ID para ele, conforme exemplo apresentado anteriormente. Porém, é possível que o Cloud Firestore gere um automaticamente. Para fazer isso, basta chamar **add()**:

```
val data = hashMapOf(
    "name" to "Tokyo",
    "country" to "Japan"
)

db.collection("cities")
    .add(data)
    .addOnSuccessListener { documentReference ->
        Log.d(TAG, "DocumentSnapshot written with ID: ${documentReference.id}")
    }
    .addOnFailureListener { e ->
        Log.w(TAG, "Error adding document", e)
    }
```

Código-fonte 3.8 – Adicionando um documento no Cloud Firestore 1
Fonte: firebase.google.com (s.d.)

Em alguns casos, pode ser útil criar uma referência de documento com um ID gerado automaticamente para usá-la mais tarde. Para esse caso de uso, chame **doc()**:

```
val data = HashMap<String, Any>()

val newCityRef = db.collection("cities").document()

// Later...
newCityRef.set(data)
```

Código-fonte 3.9 – Adicionando um document no Cloud Firestore 2
Fonte: firebase.google.com (s.d.)

3.6 Atualizar um documento

Para atualizar alguns campos de um documento sem substituir o documento inteiro, use o método **update()**:

```
val washingtonRef = db.collection("cities").document("DC")

// Set the "isCapital" field of the city 'DC'
washingtonRef
    .update("capital", true)
    .addOnSuccessListener { Log.d(TAG, "DocumentSnapshot successfully updated!") }
    .addOnFailureListener { e -> Log.w(TAG, "Error updating document", e) }
```

Código-fonte 3.10 – Atualizando um documento no Cloud Firestore
Fonte: firebase.google.com (s.d.)

3.7 Implementando o Firestore para gravar usuário

Dentro do pacote **models,** adicione uma nova classe chamada **NewUser.** Essa classe irá representar o novo usuário em nosso sistema.

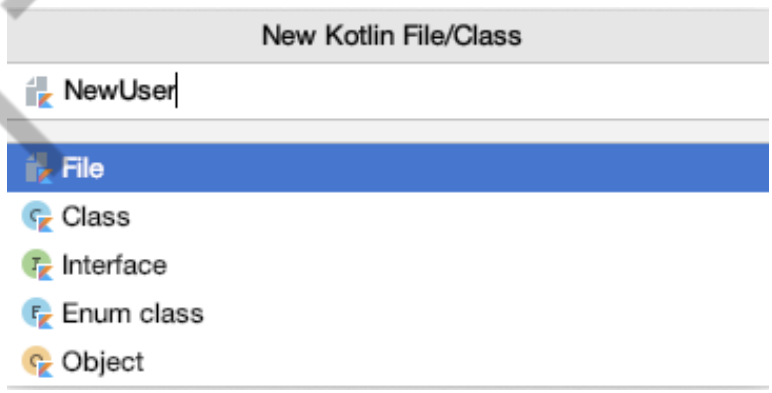

Figura 3.4 Criação da classe NewUser
Fonte: Elaborado pelo autor (2020)

Adicione o código do código-fonte “Criação da classe NewUser” à classe criada. O @Exclude foi utilizado para remover o campo senha e não o adicionar no banco de

dados, com isso, não trazemos a senha do usuário no banco ou qualquer dado mais sensível, como cartões de crédito etc.

```
data class NewUser(
    val username: String? = null,
    val email: String? = null,
    val phone: String? = null,
    @get:Exclude val password: String? = null
)
```

Código-fonte 3.11 – Criação da classe NewUser
Fonte: Elaborado pelo autor (2020)

Dentro do pacote signup, crie uma classe chamada **SignUpViewModel** e adicione o código do Código-fonte “SignUpViewModel”. Observe que, no método signUp, primeiro é criado o usuário dentro do Authentication, caso haja sucesso na criação, pegamos os demais dados do usuário e persistimos no firestore por meio do método saveInFirestore.

```
class SignUpViewModel : ViewModel() {

    private var mAuth: FirebaseAuth = FirebaseAuth.getInstance()
    private val db = FirebaseFirestore.getInstance()

    val signUpState = MutableLiveData<RequestState<FirebaseUser>>()

    fun signUp(newUser: NewUser) {

        signUpState.value = RequestState.Loading

        if (validateFields(newUser)) {
            mAuth.createUserWithEmailAndPassword(
                newUser.email ?: "",
                newUser.password ?: ""
            )
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        saveInFirestore(newUser)
```

```
                    }                    else                    {
                        signUpState.value    =    RequestState.Error(
                            Throwable(
                                task.exception?.message   ?:   "Não   foi
possível              realizar              a              requisição"
                            )
                        )
                    }
                }
            }
        }
        private      fun      saveInFirestore(newUser:      NewUser)      {
            db.collection("users")
                //.document(FirebaseAuth.getInstance().currentUser?.uid!!)
                .add(newUser)
                .addOnSuccessListener      {      documentReference      ->
                    sendEmailVerification()
                }
                .addOnFailureListener          {          e          ->
                    signUpState.value         =         RequestState.Error(
                        Throwable(e.message)
                    )
                }
        }
        private          fun          sendEmailVerification()          {
            mAuth.currentUser?.sendEmailVerification()
                ?.addOnCompleteListener          {          task          ->
                    signUpState.value                                    =
RequestState.Success(mAuth.currentUser!!)
                }
        }
        private    fun    validateFields(newUser:    NewUser):    Boolean    {

            if       (newUser.username?.isEmpty()       ==       true)       {
                signUpState.value = RequestState.Error(Throwable("Informe o
nome                          do                          usuário"))
                return                                                 false
            }

            if       (newUser.email?.isNotEmpty()       ==       false)       {
                signUpState.value   =   RequestState.Error(Throwable("E-mail
```

```
inválido"))
            return                                                    false
        }

        if       (newUser.phone?.isEmpty()       ==       true)       {
            signUpState.value = RequestState.Error(Throwable("Informe o
telefone                          do                              usuário"))
            return                                                    false
        }

        if       (newUser.password?.isEmpty()       ==       true)       {
            signUpState.value   =   RequestState.Error(Throwable("Informe
uma                                                              senha"))
            return                                                    false
        }

        if     (newUser.password?.length     ?:     0     <     6)     {
            signUpState.value                                         =
                RequestState.Error(Throwable("Senha   com   no   mínimo   6
caracteres"))
            return                                                    false
        }

        return                                                        true
    }
}
```

Código-fonte 3.12 – SignUpViewModel
Fonte: Elaborado pelo autor (2020)

Agora, abra a classe **SignUpFragment** e implemente o seguinte código:

```
      class        SignUpFragment        :        BaseFragment()        {

    override      val      layout      =      R.layout.fragment_sign_up

    private   val   signUpViewModel:   SignUpViewModel   by   viewModels()

    private      lateinit      var      etUserNameSignUp:      EditText
    private       lateinit       var       etEmailSignUp:       EditText
```

```
    private lateinit var etPhoneSignUp: EditText
    private lateinit var etPasswordSignUp: EditText
    private lateinit var btCreateAccount: Button

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        setUpView(view)

        registerObserver()
    }
    private fun setUpView(view: View) {

        etUserNameSignUp = view.findViewById(R.id.etUserNameSignUp)
        etEmailSignUp = view.findViewById(R.id.etEmailSignUp)
        etPhoneSignUp = view.findViewById(R.id.etPhoneSignUp)
        etPasswordSignUp = view.findViewById(R.id.etPasswordSignUp)
        btCreateAccount = view.findViewById(R.id.btCreateAccount)
        setUpListener()
    }
    private fun setUpListener() {
        btCreateAccount.setOnClickListener {
            val newUser = NewUser(
                etUserNameSignUp.text.toString(),
                etEmailSignUp.text.toString(),
                etPhoneSignUp.text.toString(),
                etPasswordSignUp.text.toString()
            )
            signUpViewModel.signUp(
                newUser
            )
        }
    }

    private fun registerObserver() {
        this.signUpViewModel.signUpState.observe(viewLifecycleOwner,
Observer {
            when (it) {
                is RequestState.Success -> {
                    hideLoading()
```

```
                    NavHostFragment.findNavController(this)
                        .navigate(R.id.main_nav_graph)
                }
                is          RequestState.Error          ->          {
                    hideLoading()
                    showMessage(it.throwable.message)
                }
                is  RequestState.Loading  ->  showLoading("Criando  sua
conta")
            }
        })
    }
}
```

Código-fonte 3.13 – Implementação do SignUpFragment
Fonte: Elaborado pelo autor (2020)

Para chamar a tela de criação de conta, abra o arquivo **LoginFragment** e adicione o seguinte método dentro de **setUpView**:

```
    tvNewAccount.setOnClickListener                                    {
    findNavController().navigate(R.id.signUpFragment)
}
```

Código-fonte 3.14 – Ação para tela de criação de conta
Fonte: Elaborado pelo autor (2020)

Execute o aplicativo e crie um usuário para validar o fluxo desenvolvido até o momento.

Figura 3.5 Registro inserido no Authentication
Fonte: Elaborado pelo autor (2020)

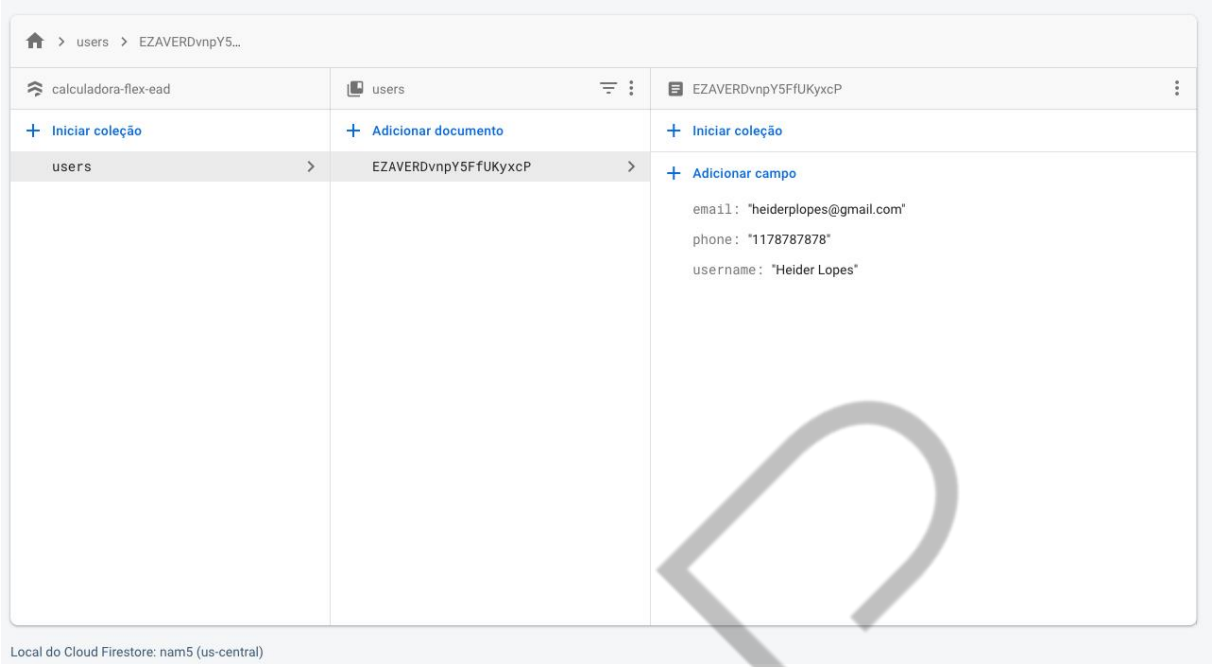

Figura 3.6 Registro inserido no Cloud Firestore
Fonte: Elaborado pelo autor (2020)

3.8 Implementando o Firestore no Projeto

Para implementação da tela principal do aplicativo, é necessário saber que o cálculo do rendimento do carro é importante, pois o motorista poderá verificar também qual combustível é mais econômico em função do preço na bomba.

Existe a convenção de que o etanol é mais econômico se custar até 70% do preço da gasolina ou 30% mais barato (baseado no teste do Inmetro).

Mas testes atuais realizados pelo Instituto Mauá mostram que o motorista poderá verificar que seu carro rende muito mais e, então, economizará usando etanol mesmo se o percentual estiver acima de 70%, podendo chegar até 75%.

Segue a lógica para resolver nossa tela:

1. Divida o desempenho do etanol pelo desempenho da gasolina (se seu carro faz 7,3 km/litro com etanol e 10 km/l com gasolina, você deve dividir 7,3 por 10 = a 0,73 ou 73%. Pronto, você achou o rendimento do carro com etanol!

2. Faça agora o cálculo da relação de preço etanol/gasolina na bomba: divida o preço do etanol pelo preço da gasolina (ex.: atualmente, o litro do etanol está em R$ 2,74 e da gasolina, R$ 4,64 = relação, então, de 0,59 ou 59%).
3. A relação atual de preço na bomba (59%) dá uma enorme economia ao consumidor se abastecer com etanol e quando este cálculo estiver em 73%, pelo rendimento do carro neste exemplo, também estará economizando se usar etanol.

3.8.1 Salvando os dados no Firestore

Crie um pacote chamado **enums** dentro do pacote **models**:

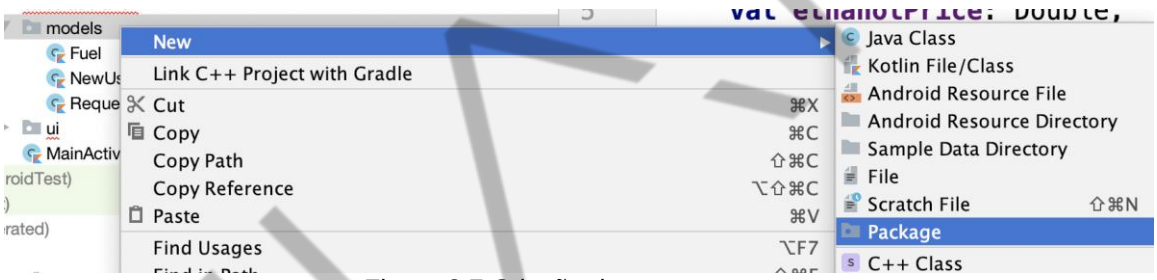

Figura 3.7 Criação do pacote enums
Fonte: Elaborado pelo autor (2020)

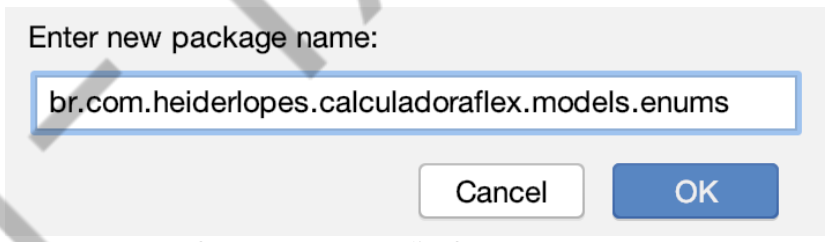

Figura 3.8 Nomeação do pacote enums
Fonte: Elaborado pelo autor (2020)

Dentro desse pacote, será criado o enum **FuelType**.

***Enums** são tipos de campos que consistem em um conjunto fixo de constantes, sendo como uma lista de valores pré-*definidos.

Neste exemplo, criaremos os tipos de combustíveis GASOLINE e ETHANOL.

Para isso, clique com o botão direito do mouse sobre o pacote **enums,** em seguida, **New → Kotlin File/Class.**

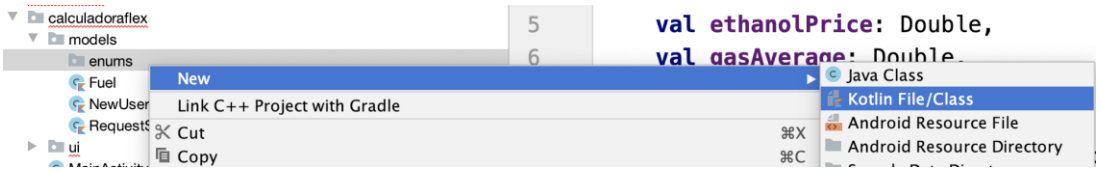

Figura 3.9 Criando um arquivo enum
Fonte: Elaborado pelo autor (2020)

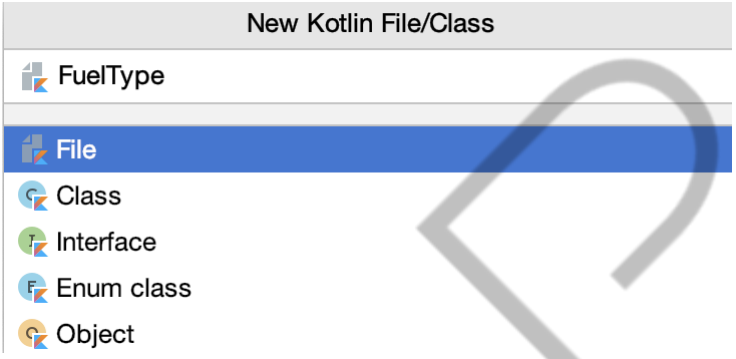

Figura 3.10 Criando o enum FuelType
Fonte: Elaborado pelo autor (2020)

```
enum class FuelType {
    GASOLINE,
    ETHANOL
}
```

Código-fonte 3.15 – Enum FuelType

Fonte: Elaborado pelo autor (2020)

Crie uma classe dentro do pacote **models** chamado **Car**.

Nesta classe, serão colocados os atributos referentes ao carro, como nome do veículo, preço da gasolina, preço do álcool, média de quilômetros percorridos com gasolina e a média de quilômetros percorridos com álcool. Esses dados serão persistidos no **Firestore**.

Clique com o botão direito do mouse sobre o pacote **models** → **New** → **Package** → **Kotlin/File Class**.

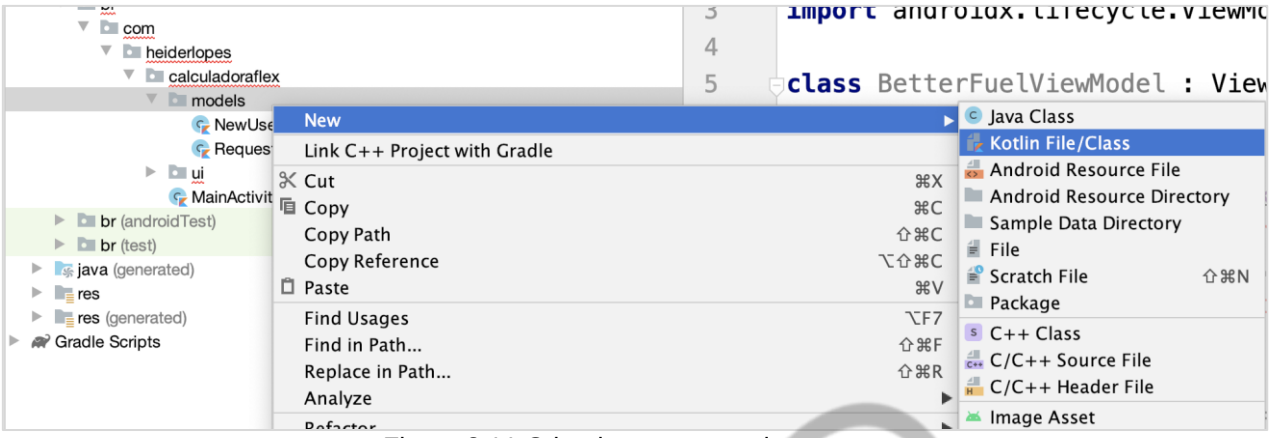

Figura 3.11 Criando uma nova classe
Fonte: Elaborado pelo autor (2020)

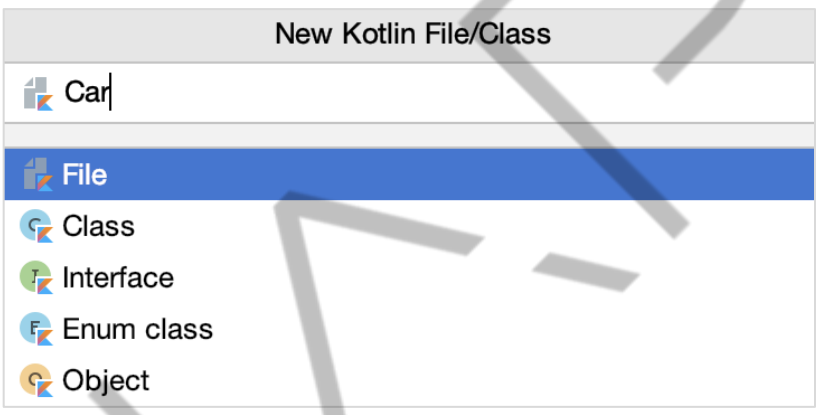

Figura 3.12 Criando uma nova classe chamada Car
Fonte: Elaborado pelo autor (2020)

Agora adicione o seguinte código:

```
data class Car(
    val model: String = "",
    val gasPrice: Double = 0.0,
    val ethanolPrice: Double = 0.0,
    val gasAverage: Double = 0.0,
    val ethanolAverage: Double = 0.0
) {
    fun getPerformanceOfMyCar(): Double {
        return ethanolAverage / gasAverage
    }

    fun getPriceOfFuelIndice(): Double {
        return ethanolPrice / gasPrice
    }

    fun getBetterFuel(): FuelType {
```

```
        return if (getPriceOfFuelIndice() <= getPerformanceOfMyCar())
            FuelType.ETHANOL
        else
            FuelType.GASOLINE
    }
}
```

Código-fonte 3.16 – Código da classe Car
Fonte: Elaborado pelo autor (2020)

Abra a classe BetterFuelViewModel e adicione o método para calcular o melhor combustível. Ao acionar esse método, serão gravados os dados no Firestore para futuras consultas. No código-fonte “Código da classe BetterFuelViewModel” está o código a ser implementado:

```
class BetterFuelViewModel : ViewModel() {

    private val db = FirebaseFirestore.getInstance()
    val calculateState = MutableLiveData<RequestState<FuelType>>()

    fun calculate(car: Car) {
        calculateState.value = RequestState.Loading
        db.collection("cars")
            .document(FirebaseAuth.getInstance().currentUser?.uid!!)
            .set(car)
            .addOnSuccessListener { _ ->
                calculateState.value =
RequestState.Success(car.getBetterFuel())
            }
            .addOnFailureListener { e ->
                calculateState.value = RequestState.Error(
                    Throwable(e.message)
                )
            }
    }
}
```

Código-fonte 3.17 – Código da classe BetterFuelViewModel
Fonte: Elaborado pelo autor (2020)

No Código-fonte “Código da classe BetterFuelViewModel”temos:

.collection("cars") → nome da coleção gravada no Firestore. No caso, cars.

.document(FirebaseAuth.getInstance().currentUser?.uid!!) → aqui é definido o documento, neste exemplo, o id do usuário é utilizado como chave do documento.

.set(car) → objeto que será gravado no banco de dados.

Crie um pacote dentro de ui chamado de **watchers.** Botão direito → **New** → **Package**.

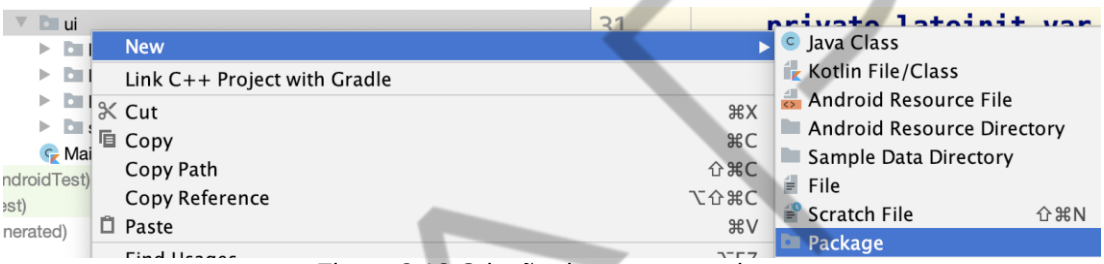

Figura 3.13 Criação do pacote watchers
Fonte: Elaborado pelo autor (2020)

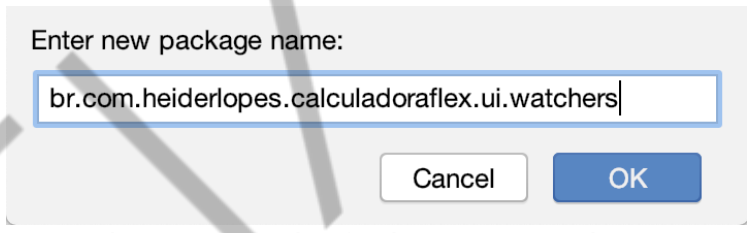

Figura 3.14 Criação de uma nova classe
Fonte: Elaborado pelo autor (2020)

Crie uma classe **DecimalTextWatchers** clicando com o botão direito do mouse sobre o pacote **watchers → New → Kotlin File/Class.** Essa classe será responsável por adicionar, nas caixas de textos, um listener que irá formatar os valores digitados pelo usuário com casas decimais.

Figura 3.15 Criação da classe DecimalTextWatcher
Fonte: Elaborado pelo autor (2020)

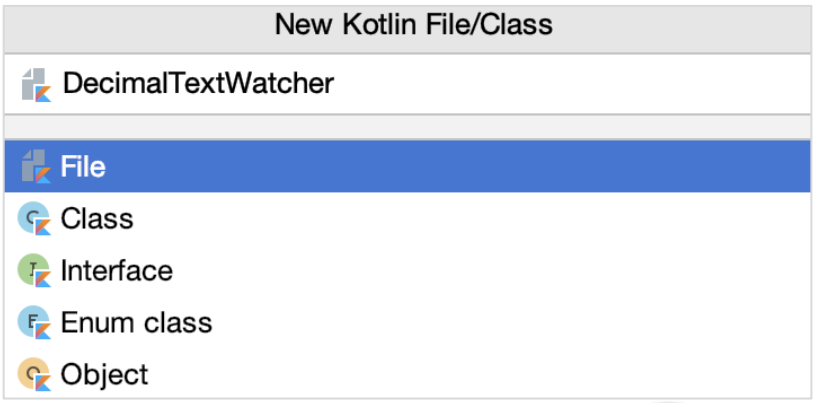

Figura 3.16 Nomeação da classe DecimalTextWatcher
Fonte: Elaborado pelo autor (2020)

Adicione o seguinte código referente à classe **DecimalTextWatcher:**

```
class DecimalTextWatcher(editText: EditText,
                         private val totalDecimalNumber: Int = 2) :
TextWatcher {

    private val editTextWeakReference: WeakReference<EditText> =
WeakReference(editText)

    init {
        formatDigits(editTextWeakReference.get()?.text)
    }

    override fun beforeTextChanged(s: CharSequence, start: Int, count:
Int, after: Int) {}

    override fun onTextChanged(s: CharSequence, start: Int, before: Int,
count: Int) {}

    override fun afterTextChanged(editable: Editable) {
        formatDigits(editable)
    }

    private fun getTotalDecimalNumber(): String {
        val decimalNumber = StringBuilder()
        for (i in 1..totalDecimalNumber) {
            decimalNumber.append("0")
        }
        return decimalNumber.toString()
    }
```

```
    private fun formatDigits(editable: Editable?) {
        val editText = editTextWeakReference.get() ?: return
        val cleanString = editable.toString().trim().replace(" ", "")
        editText.removeTextChangedListener(this)

        val number = Math.pow(10.toDouble(),
totalDecimalNumber.toDouble())

        val parsed = when (cleanString) {
            "" -> BigDecimal(0)
            "null" -> BigDecimal(0)
            else -> BigDecimal(cleanString.replace("\\D+".toRegex(),
""))
                .setScale(totalDecimalNumber, BigDecimal.ROUND_FLOOR)
                .divide(BigDecimal(number.toInt()),
                    BigDecimal.ROUND_FLOOR)
        }

        val dfnd = DecimalFormat("#,##0.${getTotalDecimalNumber()}")
        val formatted = dfnd.format(parsed)
        editText.setText(formatted.replace(',', '.'))
        editText.setSelection(formatted.length)
        editText.addTextChangedListener(this)
    }
}
```

Código-fonte 3.18 – Código da classe DecimalTextWatcher
Fonte: Elaborado pelo autor (2020)

O próximo passo é abrir o **fragment** referente ao formulário para calcular o melhor combustível e aplicar o watcher criado no “Código-fonte “Código da classe DecimalTextWatcher”:

```
class BetterFuelFragment : BaseAuthFragment() {

    private val betterFuelViewModel: BetterFuelViewModel by viewModels()

    override val layout: Int
        get() = R.layout.fragment_better_fuel
```

```
    private lateinit var etCar: EditText
    private lateinit var etKmGasoline: EditText
    private lateinit var etKmEthanol: EditText
    private lateinit var etPriceGasoline: EditText
    private lateinit var etPriceEthanol: EditText

    private lateinit var btSignOut: AppCompatImageView
    private lateinit var btCalculate: Button
    private lateinit var btClear: TextView

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        setUpView(view)

        registerObserver()
    }

    private fun setUpView(view: View) {
        etCar = view.findViewById(R.id.etCar)
        etKmGasoline = view.findViewById(R.id.etKmGasoline)
        etKmEthanol = view.findViewById(R.id.etKmEthanol)
        etPriceGasoline = view.findViewById(R.id.etPriceGasoline)
        etPriceEthanol = view.findViewById(R.id.etPriceEthanol)

        btSignOut = view.findViewById(R.id.btSignOut)
        btCalculate = view.findViewById(R.id.btCalculate)
        btClear = view.findViewById(R.id.btClear)


etKmGasoline.addTextChangedListener(DecimalTextWatcher(etKmGasoline,
1))

etKmEthanol.addTextChangedListener(DecimalTextWatcher(etKmEthanol, 1))

etPriceGasoline.addTextChangedListener(DecimalTextWatcher(etPriceGasoli
ne))
```

```
etPriceEthanol.addTextChangedListener(DecimalTextWatcher(etPriceEthanol
))



        btCalculate.setOnClickListener                                  {

        }

        btClear.setOnClickListener                                      {

        }

        btSignOut.setOnClickListener                                    {
            betterFuelViewModel.logout()
        }
    }

    private             fun                 registerObserver()          {

this.betterFuelViewModel.loggedState.observe(viewLifecycleOwner,
Observer                                                                {
            when                            (it)                        {
                is          RequestState.Success          ->          {
                    hideLoading()
                    NavHostFragment.findNavController(this)
                        .navigate(R.id.logInFragment)
                }
                is           RequestState.Error           ->          {
                    hideLoading()
                    showMessage(it.throwable.message)
                }
                is  RequestState.Loading  ->  showLoading("Calculando  o
melhor                                                  combustível")
            }
        })
    }
}
```

Código-fonte 3.17 – Código do setup das views do fragment BetterFuelFragment
Fonte: Elaborado pelo autor (2020)

Antes de aplicarmos a ação do clique, abra o arquivo **BaseFragment** e adicione o código do “Código-fonte “Método para exibir um AlertDialog” para exibir um alert dialog.

```
    fun showAlert(title: String, message: String, buttonText: String =
"OK") {
    val builder =
        AlertDialog.Builder(requireContext())
    builder.setTitle(title)
    builder.setMessage(message)
        .setCancelable(false)
        .setPositiveButton(buttonText) { dialog, id ->
            dialog.dismiss()
        }
    val alert = builder.create()
    alert.show()
}
```

Código-fonte 3.19 – Método para exibir um AlertDialog
Fonte: Elaborado pelo autor (2020)

Com isso, o próximo passo será adicionar a ação no botão que irá calcular o melhor combustível e gravar os dados no banco de dados (no exemplo, os dados serão persistidos no Firestore). Para isso, adicione o código em negrito do Código-fonte “Código do setup das views do fragment BetterFuelFragment”.

```
class BetterFuelFragment : BaseAuthFragment() {

    private val betterFuelViewModel: BetterFuelViewModel by viewModels()

    override val layout: Int
        get() = R.layout.fragment_better_fuel


    private lateinit var etCar: EditText
    private lateinit var etKmGasoline: EditText
    private lateinit var etKmEthanol: EditText
    private lateinit var etPriceGasoline: EditText
    private lateinit var etPriceEthanol: EditText
```

```
    private lateinit var btSignOut: AppCompatImageView
    private lateinit var btCalculate: Button
    private lateinit var btClear: TextView

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        setUpView(view)

        registerObserver()
    }

    private fun setUpView(view: View) {
        etCar = view.findViewById(R.id.etCar)
        etKmGasoline = view.findViewById(R.id.etKmGasoline)
        etKmEthanol = view.findViewById(R.id.etKmEthanol)
        etPriceGasoline = view.findViewById(R.id.etPriceGasoline)
        etPriceEthanol = view.findViewById(R.id.etPriceEthanol)

        btSignOut = view.findViewById(R.id.btSignOut)
        btCalculate = view.findViewById(R.id.btCalculate)
        btClear = view.findViewById(R.id.btClear)


etKmGasoline.addTextChangedListener(DecimalTextWatcher(etKmGasoline,
1))

etKmEthanol.addTextChangedListener(DecimalTextWatcher(etKmEthanol, 1))

etPriceGasoline.addTextChangedListener(DecimalTextWatcher(etPriceGasoli
ne))

etPriceEthanol.addTextChangedListener(DecimalTextWatcher(etPriceEthanol
))


        btCalculate.setOnClickListener {
            val car = Car(
                etCar.text.toString(),
```

```
                etPriceGasoline.text.toString().toDouble(),
                etPriceEthanol.text.toString().toDouble(),
                etKmGasoline.text.toString().toDouble(),
                etKmEthanol.text.toString().toDouble()

            )
            betterFuelViewModel.calculate(car)
        }

        btClear.setOnClickListener {

        }

        btSignOut.setOnClickListener {
            betterFuelViewModel.logout()
        }
    }

    private fun registerObserver() {

this.betterFuelViewModel.loggedState.observe(viewLifecycleOwner,
Observer {
            when (it) {
                is RequestState.Success -> {
                    hideLoading()
                    NavHostFragment.findNavController(this)
                        .navigate(R.id.logInFragment)
                }
                is RequestState.Error -> {
                    hideLoading()
                    showMessage(it.throwable.message)
                }
                is RequestState.Loading -> showLoading("Calculando o
melhor combustível")
            }
        })


this.betterFuelViewModel.calculateState.observe(viewLifecycleOwner,
Observer {
            when (it) {
```

```
                is          RequestState.Success          ->          {
                    hideLoading()
                    when                    (it.data)                   {
                        FuelType.GASOLINE             ->              {
                            showAlert("Calcula Flex", "Melhor abastecer
com                                                         Gasolina")
                        }
                        FuelType.ETHANOL              ->              {
                            showAlert("Calcula Flex", "Melhor abastecer
com                                                          Álcool")
                        }
                    }
                }
                is           RequestState.Error          ->          {
                    hideLoading()
                    showMessage(it.throwable.message)
                }
                is  RequestState.Loading  ->  showLoading("Calculando  o
melhor                                                  combustível")
            }
        })
    }
}
```

Código-fonte 3.20 – Código do setup das views do fragment BetterFuelFragment
Fonte: Elaborado pelo autor (2020)

3.8.2 Acessando os dados do Firestore

Há duas maneiras de recuperar dados armazenados no Cloud Firestore. Qualquer um desses métodos pode ser usado com documentos, coleções de documentos ou resultados de consultas:

- Chame um método para receber os dados.
- Defina um listener para receber eventos de mudança de dados.

Quando você configura um listener, o Cloud Firestore envia para ele um snapshot inicial dos dados e, em seguida, outro snapshot, cada vez que o documento é alterado.

3.8.3 Consulta simples

O Cloud Firestore tem uma funcionalidade de consulta eficiente que permite especificar os documentos que pretende recuperar de uma coleção ou grupo de coleções. Essas consultas também podem ser usadas com get() ou addSnapshotListener().

No exemplo do Código-fonte “Consulta Simples – Retorna estados de SP”, serão exibidas todas as cidades de SP:

```
// Create a reference to the cities collection
val citiesRef = db.collection("cities")

// Create a query against the collection.
val query = citiesRef.whereEqualTo("state", "SP")
```

Código-fonte 3.21 – Consulta Simples – Retorna estados de SP
Fonte: firebase.google.com (s.d.)

A consulta do Código-fonte “Consulta Simples – Retorna as capitais” retorna todas as capitais.

```
val capitalCities = db.collection("cities").whereEqualTo("capital", true)
```

Código-fonte 3.22 – Consulta Simples – Retorna as capitais
Fonte: firebase.google.com (s.d.)

Depois de criar um objeto de consulta, use a função **get()** para recuperar os resultados.

```
db.collection("cities")
                .whereEqualTo("capital", true)
                .get()
                .addOnSuccessListener { documents ->
                    for (document in documents) {
                        Log.d(TAG, "${document.id} => ${document.data}")
                    }
                }
                .addOnFailureListener { exception ->
                    Log.w(TAG, "Error getting documents: ", exception)
                }
```

Código-fonte 3.23 – Executar uma consulta
Fonte: firebase.google.com (s.d.)

3.8.4 Operadores de consulta

O método **where()** usa três parâmetros: um campo para filtrar, uma operação de comparação e um valor. A comparação pode ser **<, <=, ==, >, >= ou array-contains.** Para iOS, Android e Java, o operador de comparação é nomeado explicitamente no método. Veja alguns exemplos:

```
citiesRef.whereEqualTo("state", "CA")
citiesRef.whereLessThan("population", 100000)
citiesRef.whereGreaterThanOrEqualTo("name", "San Francisco")
```

Código-fonte 3.24 – Operadores de consulta
Fonte: firebase.google.com (s.d.)

3.8.5 Consulta composta

É possível encadear vários **métodos where()** para criar consultas mais específicas (lógica AND). Para combinar o operador de igualdade (==) com uma

cláusula range ou array-contains (<, <=, >, >= ou array-contains), é necessário criar um índice composto.

```
citiesRef.whereEqualTo("state", "CO").whereEqualTo("name", "Denver")
citiesRef.whereEqualTo("state", "CA").whereLessThan("population", 1000000)
```

Código-fonte 3.25 – Consulta composta
Fonte: firebase.google.com (s.d.)

É válido haver filtros de intervalo em apenas um campo.

```
citiesRef.whereGreaterThanOrEqualTo("state", "CA")
.whereLessThanOrEqualTo("state", "IN")
citiesRef.whereEqualTo("state", "CA")
.whereGreaterThan("population", 1000000)
```

Código-fonte 3.26 – Consulta composta válida
Fonte: firebase.google.com (s.d.)

É inválido haver filtros de intervalos em campos diferentes.

```
citiesRef.whereGreaterThanOrEqualTo("state", "CA")
.whereGreaterThan("population", 100000)
```

Código-fonte 3.27 – Consulta composta inválida
Fonte: firebase.google.com (s.d.)

3.8.6 Carregando os dados do veículo

Agora, ao entrar na tela, já serão carregados os dados gravados (caso existam) e serão exibidos para o usuário. Abra o arquivo **BetterFuelViewModel** e adicione o método buscar os dados dos usuários:

```
class BetterFuelViewModel : ViewModel() {

    private var mAuth: FirebaseAuth = FirebaseAuth.getInstance()
    private val db = FirebaseFirestore.getInstance()

    val calculateState = MutableLiveData<RequestState<FuelType>>()

    val loggedState = MutableLiveData<RequestState<Boolean>>()

    val carState = MutableLiveData<RequestState<Car>>()

    fun getCarData() {
        FirebaseAuth.getInstance().currentUser?.let {
            carState.value = RequestState.Loading
            db.collection("cars")
                .document(it.uid)
                .get()
                .addOnSuccessListener { documents ->
                    var car = Car()
                    if (documents.data != null) {
                        car = Car(
                            documents.data?.get("model") as String,
                            documents.data?.get("gasPrice") as Double,
                            documents.data?.get("ethanolPrice") as
Double,
                            documents.data?.get("gasAverage") as
Double,
                            documents.data?.get("ethanolAverage") as
Double
                        )
                    }
                    carState.value = RequestState.Success(car)
                }
                .addOnFailureListener { e ->
                    calculateState.value = RequestState.Error(
                        Throwable(e.message)
                    )
                }
        }
```

```
    }

    fun calculate(car: Car) {
        calculateState.value = RequestState.Loading
        db.collection("cars")
            .document(FirebaseAuth.getInstance().currentUser?.uid!!)
            .set(car)
            .addOnSuccessListener { _ ->
                calculateState.value =
RequestState.Success(car.getBetterFuel())
            }
            .addOnFailureListener { e ->
                calculateState.value = RequestState.Error(
                    Throwable(e.message)
                )
            }
    }

    fun logout() {
        loggedState.value = RequestState.Loading
        mAuth.signOut()
        loggedState.value = RequestState.Success(true)
    }
}
```

Código-fonte 3.28 – Código para recuperar os dados do carro
Fonte: Elaborado pelo autor (2020)

No Código-fonte “Código para recuperar os dados do carro”, temos, no objeto **db,** uma instância do banco de dados. Para realizar uma busca dos dados do usuário, será utilizado o id do usuário logado. Detalhando o código acima:

.collection(“cars”) → Nome da coleção

.document(FirebaseAuth.getInstance().currentUser?.uid!!) → id do documento que será pesquisado.

.get() → Utilizado para realizar a busca.

.addOnSuccessListener → Listener executado quando o resultado da busca for realizada com sucesso.

.addOnFailureListener → Listener executado quando o resultado da busca der algum erro.

O retorno do listener de sucesso retorna um objeto que podemos pegar por meio da chave para recuperar o valor obtido na busca.

Agora, para executar no fragmento **BetterFuelFragment**, adicione o seguinte código em negrito:

```
class BetterFuelFragment : BaseAuthFragment() {

    override val layout: Int
        get() = R.layout.fragment_better_fuel


    private val betterFuelViewModel: BetterFuelViewModel by viewModels()

    private lateinit var etCar: EditText
    private lateinit var etKmGasoline: EditText
    private lateinit var etKmEthanol: EditText
    private lateinit var etPriceGasoline: EditText
    private lateinit var etPriceEthanol: EditText

    private lateinit var btSignOut: AppCompatImageView
    private lateinit var btCalculate: Button
    private lateinit var btClear: TextView

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        setUpView(view)

        registerObserver()

        betterFuelViewModel.getCarData()
    }



    private fun registerObserver() {
```

```
        this.betterFuelViewModel.carState.observe(viewLifecycleOwner,
Observer                                                                {
            when                            (it)                        {
                is          RequestState.Success          ->            {
                    hideLoading()
                    fillFields(it.data)
                }
                is           RequestState.Error           ->            {
                    hideLoading()
                    showMessage(it.throwable.message)
                }
                is RequestState.Loading -> showLoading("Carregando seus
dados")
            }
        })
    }

    private          fun          fillFields(car:          Car)          {
        etCar.setText(car.model)
        etKmGasoline.setText(car.gasAverage.toString())
        etKmEthanol.setText(car.ethanolAverage.toString())
        etPriceGasoline.setText(car.gasPrice.toString())
        etPriceEthanol.setText(car.ethanolPrice.toString())
    }
}
```

Código-fonte 3.29 – Busca dos dados do carro
Fonte: Elaborado pelo autor (2020)

Pronto, para finalizar essa tela até o momento, adicione a ação para limpar o formulário:

```
      btClear.setOnClickListener                                        {
    fillFields(Car())
}
```

Código-fonte 3.30 – Método para limpar os campos do formulário
Fonte: Elaborado pelo autor (2020)

O projeto completo pode ser baixado neste repositório <https://github.com/FIAPON/CalculaFlexAndroid>.

CONCLUSÃO

Neste capítulo, foi possível conhecer o Cloud Firestore e como persistir e recuperar dados. O Firebase fornece a infraestrutura necessária para armazenar e manter os dados dos usuários/aplicação de forma segura, provendo mecanismos de segurança e dando suporte para o crescimento da sua aplicação.

REFERÊNCIAS

BRASIL. INMETRO. Informação ao consumidor. **Tabelas de Consumo de Combustível.** Brasil. 2020. Disponível em: <http://www.inmetro.gov.br/consumidor/tabelas_pbe_veicular.asp>. Acesso em: 10 set. 2020.

GOOGLE. **Documentação do Firebase.** EUA. 2020. Disponível em: <https://firebase.google.com/docs/auth/android/custom-auth?hl=pt-br>. Acesso em: 10 set. 2020.

ROMIO, R. **Rendimento dos carros abastecidos com etanol pode ser melhor do que se imaginava**. 2020. Disponível em: <https://maua.br/imprensa/infomaua/192/texto/812>. Acesso em: 10 set. 2020.