Instituto Politécnico de Viana do Castelo

ipvc estg

Licenciatura em Engenharia Informática

Computação Móvel

TaskTracker

Alexandre Santos, n.º 24585

António Gomes, n.º 26780

Gilberto Parente, nº 15330

Prof. David Verde

Índice

1. Introdução e Objetivos

Com este trabalho prático da uc de Computação Móvel, temos como desafio desenvolver uma aplicação móvel com recurso às ferramentas Kotlin e Android Studio.

O nosso objetivo será, criar uma plataforma de gestão de projetos e tarefas, proporcionando aos utilizadores uma ferramenta eficiente para o acompanhamento e organização das suas atividades.

Este projeto não apenas visa fortalecer as nossas habilidades técnicas em programação móvel, mas também desafia a aplicar metodologias de desenvolvimento ágil, integrar uma base de dados e fornecer uma interface intuitiva e profissional para os utilizadores.

No final, esperamos não apenas cumprir os requisitos estabelecidos, mas também explorar funcionalidades adicionais que enriqueçam a experiência do utilizador e demonstrem a nossa capacidade de inovação e adaptação às necessidades do mercado.

2. Tecnologias utilizadas

2.1 Git/Github

Git é uma ferramenta de controlo de versão distribuído, amplamente utilizada no desenvolvimento de software para manter registo das alterações feitas em arquivos ao longo do tempo. Permite que várias pessoas trabalhem num mesmo projeto, coordenando as suas contribuições de forma eficiente. Com o Git, os colaboradores podem registar alterações, fazer merge de diferentes versões do código e reverter para versões anteriores, facilitando o trabalho colaborativo e a gestão do projeto de software.

O GitHub é uma plataforma baseada na web que utiliza o Git como sistema de controlo de versão. Além de fornecer serviços de armazenamento de projetos Git, o GitHub oferece recursos adicionais, como controlo de acesso, gestão de problemas (issues), colaboração em equipa e integração contínua. É amplamente utilizado pela comunidade de desenvolvimento de software para compartilhar e colaborar em projetos de código aberto, além de ser uma ferramenta comum para o desenvolvimento de projetos privados em equipas profissionais.

2.2 Trello

O Trello é uma ferramenta de gestão de projetos baseada em quadros, listas e cartões. Oferece uma abordagem visual e flexível para organizar tarefas e colaborar em projetos. No Trello, os utilizadores podem criar quadros para representar projetos, listas para categorizar tarefas e cartões para representar as próprias tarefas. Os cartões podem conter informações detalhadas, como descrições, listas de verificação, datas de vencimento, anexos e comentários. Os utilizadores podem mover os cartões entre as listas para indicar o progresso das tarefas e atribuir cartões a membros da equipa para distribuir responsabilidades. É amplamente utilizado por equipas de desenvolvimento de software, mas também é adaptável para uma variedade de outros tipos de projetos e workflows. Ele é conhecido pela sua simplicidade e facilidade de uso, tornando-se uma escolha popular para equipas de todos os tamanhos.

2.3 Android Studio/kotlin

Android Studio é o ambiente de desenvolvimento integrado (IDE) oficial para o desenvolvimento de aplicações Android, fornecido pela Google. Oferece uma ampla gama de ferramentas e recursos para programadores de aplicações Android, incluindo um editor de código, depurador, emulador de dispositivos, gestor de dependências e muito mais. Android Studio é construído sobre a plataforma IntelliJ IDEA da JetBrains e é altamente otimizado para o desenvolvimento de aplicações Android.

Kotlin é uma linguagem de programação concisa e segura que foi oficialmente suportada pela Google como uma linguagem de programação para o desenvolvimento de aplicações Android em 2017. É interoperável com o Java, o que significa que pode ser facilmente integrado em projetos Android existentes que utilizam Java. Oferece muitos recursos poderosos, como nulidade segura, extensões de funções, classes e tipos de dados imutáveis, entre outros. Kotlin é conhecida pela sua sintaxe limpa e expressiva, o que pode aumentar a produtividade dos programadores e tornar o código mais legível.

3. Planeamento Inicial

3.1 Utilização do Git

Para garantir um desenvolvimento colaborativo e uma gestão eficaz das versões do nosso código, optamos por utilizar o Git como sistema de controlo de versão.

Dessa forma, todas as alterações e novas funcionalidades desenvolvidas são registadas por meio de commits no branch principal (main), proporcionando transparência e organização ao nosso processo de desenvolvimento.

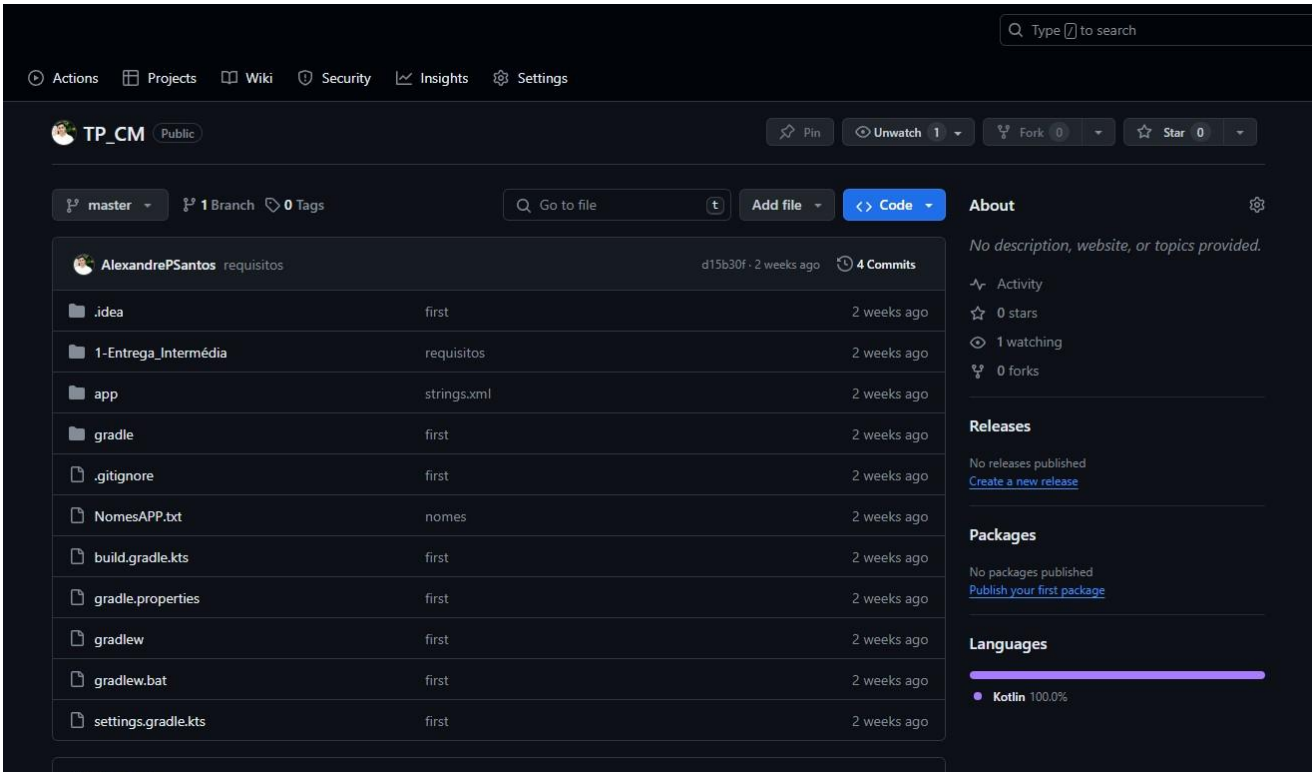

Figura 1 - Repositório criado no Github

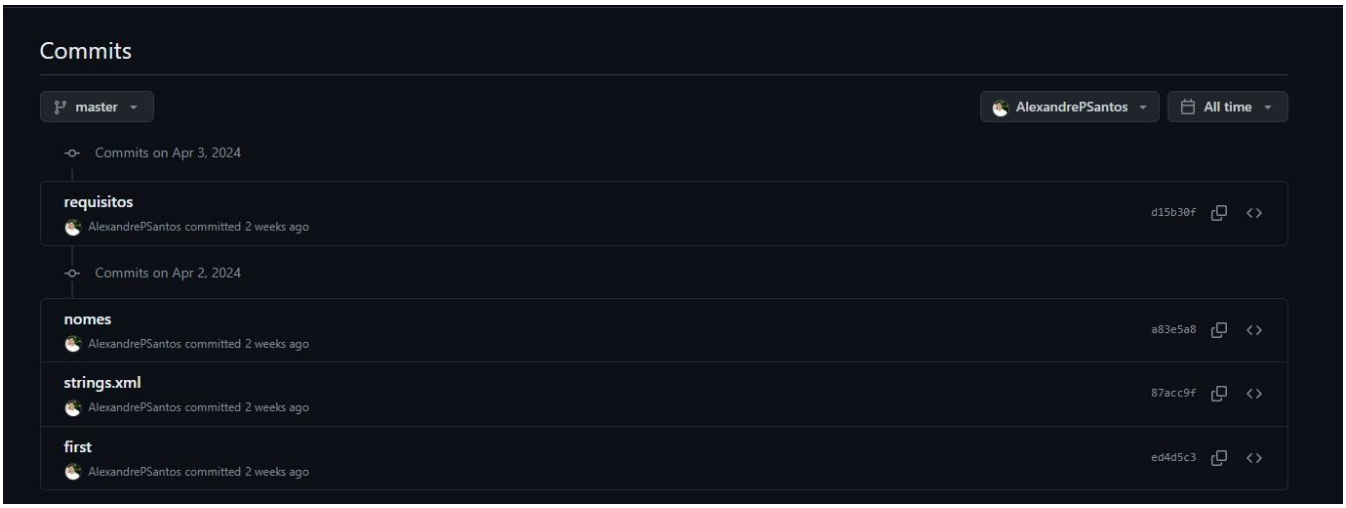

Figura 2 - Primeiros commits

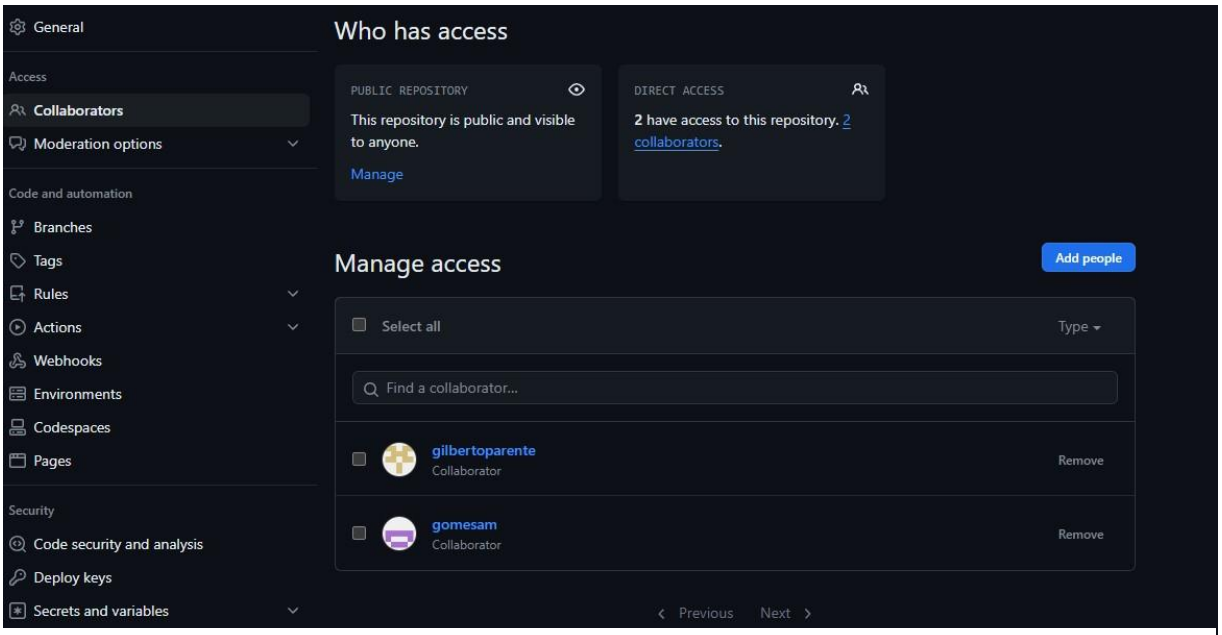

Figura 3 - Colaboradores do projeto

3.2 Trello para gestão do projeto

Para gerir eficazmente o projeto, dividir tarefas e cumprir prazos, optamos por utilizar o Trello.

Esta ferramenta permite manter a organização como equipa e garantir que o projeto siga um desenvolvimento fluido.

Com o Trello, podemos visualizar o progresso das tarefas, atribuir responsabilidades e manter uma comunicação eficiente entre os membros da equipa, garantindo assim uma colaboração harmoniosa e uma execução eficaz do projeto.

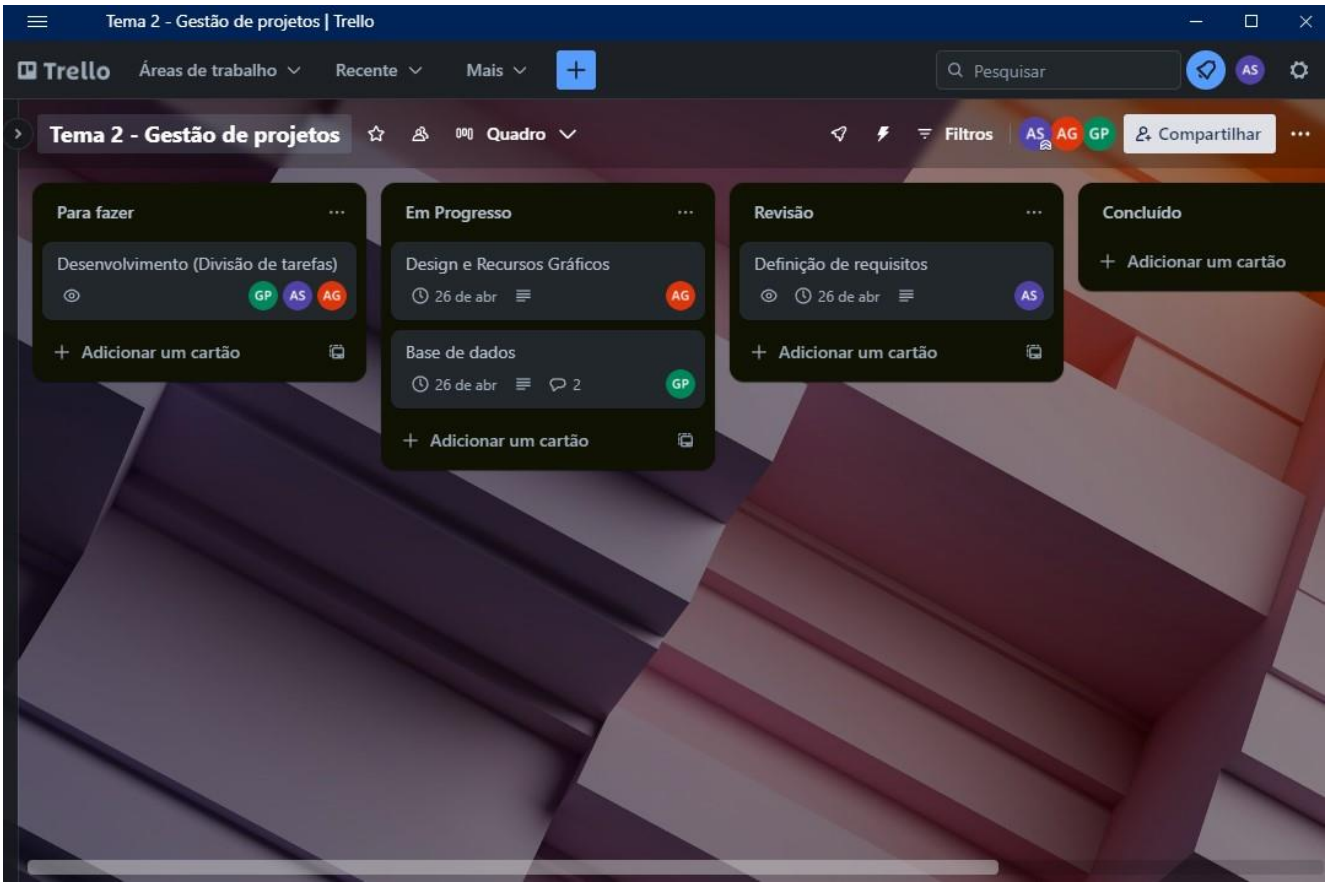

Figura 4 - Quadro com tarefas iniciais

3.3 Requisitos de software

Funcionais:

Gerais:

- A aplicação deve apresentar slides introdutórios para os novos utilizadores explicando as funcionalidades da aplicação;
- Os utilizadores devem ter a capacidade de criar uma conta na aplicação fornecendo informações como nome, username, email, fotografia e senha;
- Os utilizadores devem poder fazer login;
- Os utilizadores devem poder visualizar e editar o seu perfil, incluindo informações como nome, username, email, fotografia e senha;
- Deve haver pelo menos três tipos de perfis: Administrador, Gestor de Projeto e Utilizador;

Administrador:

- O administrador deve ter a capacidade de criar, editar e remover projetos;
- O administrador pode gerir os perfis dos utilizadores e dos gestores de projeto;
- O administrador pode associar um gestor de projeto a um projeto;
- O administrador pode exportar estatísticas por utilizador, projeto ou tarefa;

Gestor de projeto:

- O gestor de projeto deve ser capaz de associar tarefas a projetos;
- O gestor de projeto deve poder atribuir utilizadores a projetos e respetivas tarefas;
- O gestor de projeto pode visualizar as tarefas concluídas e por concluir num determinado projeto;
- O gestor de projeto pode marcar o projeto como concluído e avaliar o desempenho de cada utilizador;
- O gestor de projeto pode exportar estatísticas por utilizador, projeto ou tarefa;

Utilizador:

- O utilizador associado a uma tarefa de um projeto deve ser capaz de registar detalhes como data, local, percentagem de conclusão e tempo gasto na realização da tarefa;
- O utilizador pode associar observações e fotografias às tarefas conforme necessário;
- O utilizador pode marcar a tarefa como concluída;
- O utilizador pode visualizar uma lista de tarefas a serem realizadas e o histórico de tarefas concluídas.

Não funcionais:

- Os dados devem ser gravados localmente em casos de não haver conexão com a internet e sincronizados posteriormente com a API;
- A aplicação deve ser capaz de comunicar com uma API para sincronização de dados quando conectado à internet;
- A API deve ser segura para garantir a privacidade e integridade dos dados dos utilizadores;
- A aplicação deve ser responsiva e eficiente;
- A interface deve ser intuitiva e fácil de usar para todos os tipos de utilizador.

3.4 Casos de uso

Para criar o diagrama de casos de uso, utilizámos como base os requisitos funcionais, sendo estes as principais ações dos utilizadores.

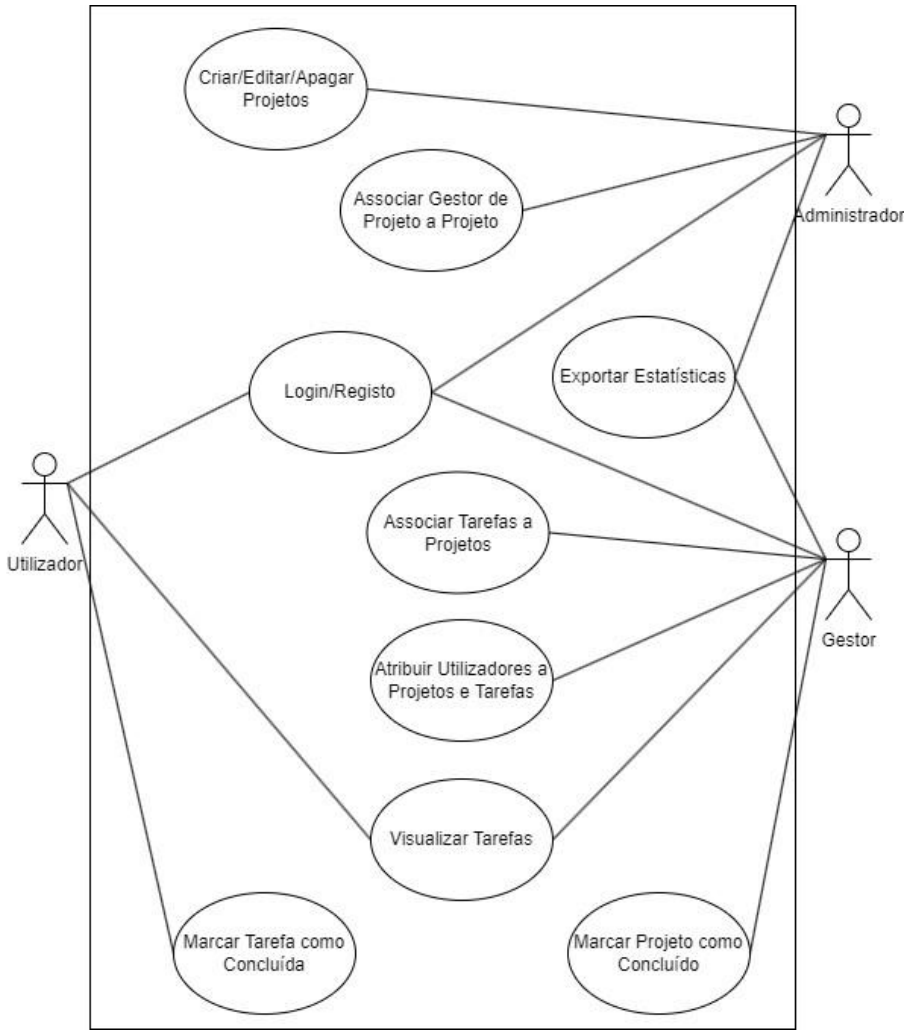

Figura 5 - Diagrama de casos de uso

3.5 Design e recursos gráficos

Para desenvolver os recursos gráficos, utilizámos maioritariamente o Figma para desenvolvimento das interfaces e o canva para ícones e logotipos.

Figura 6 - Logotipo desenvolvido para a aplicação TaskTracker

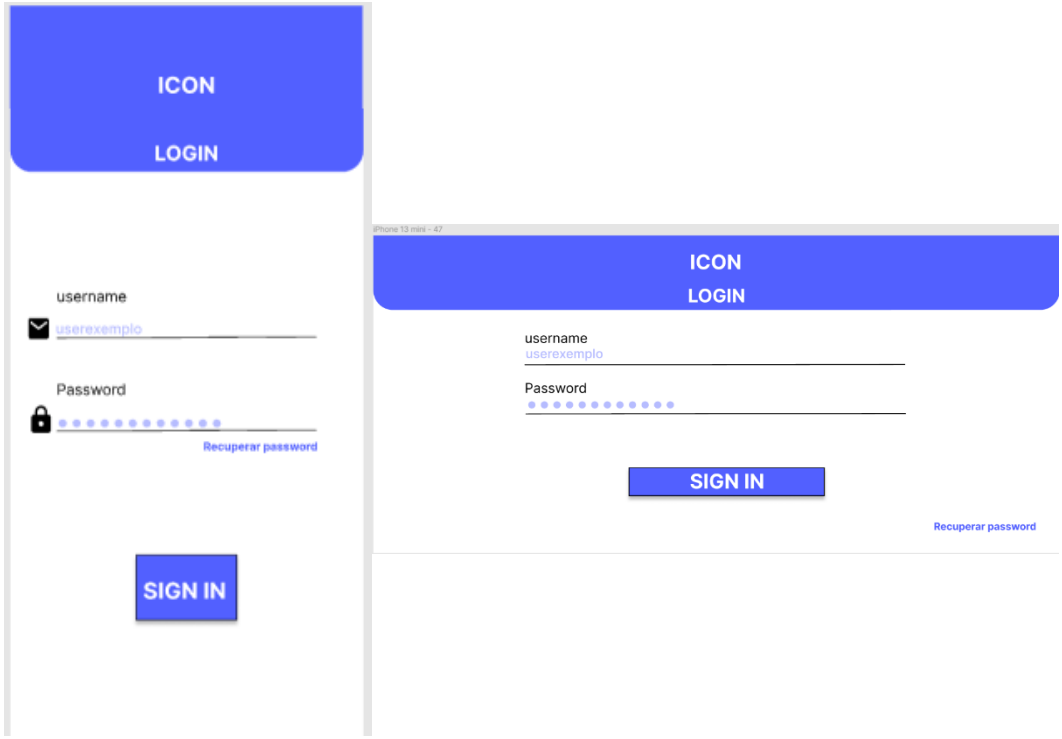

Figura 7 - Ecrã de login (horizontal e vertical)

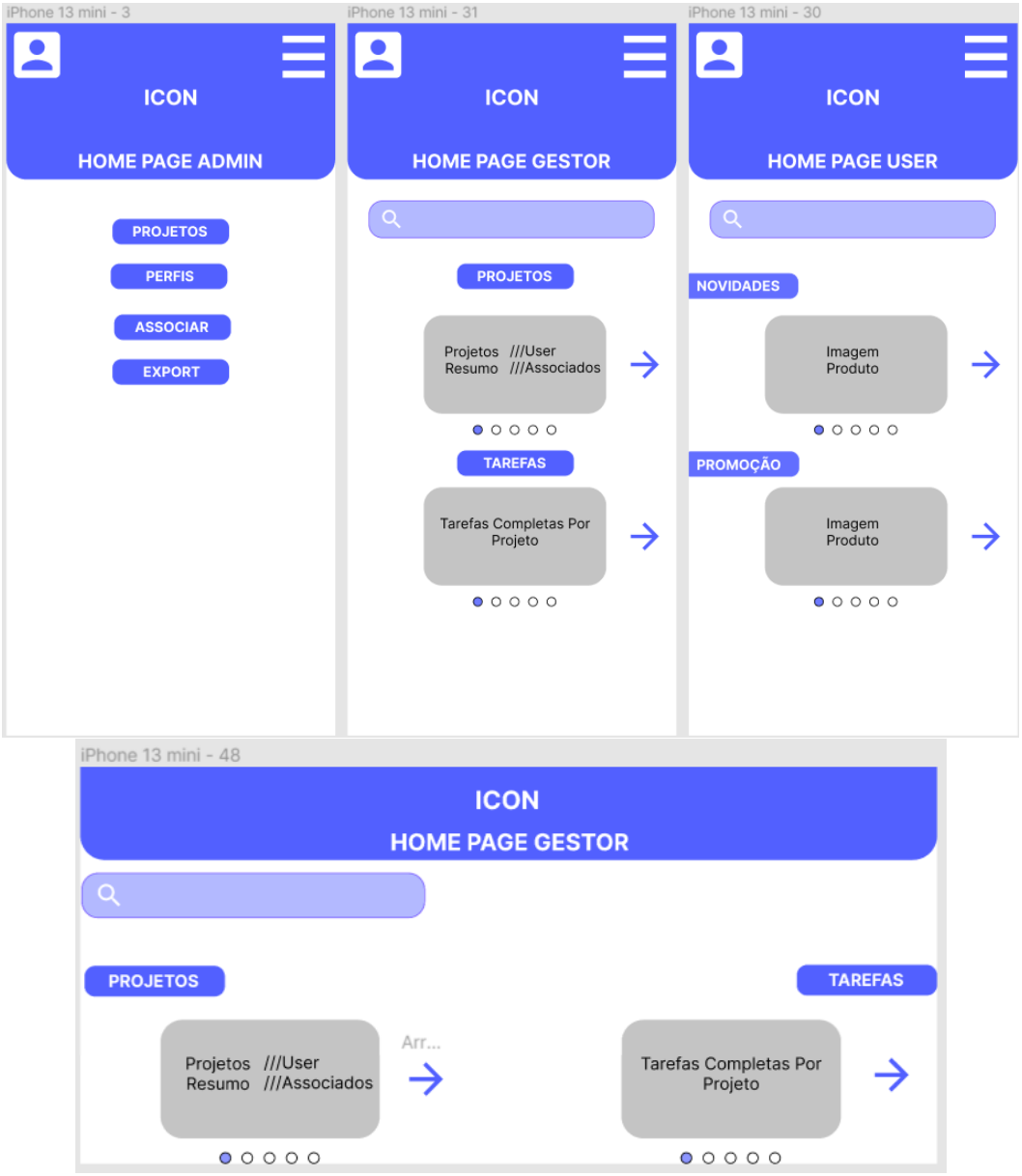

Figura 8 - Dashboard dos utilizadores

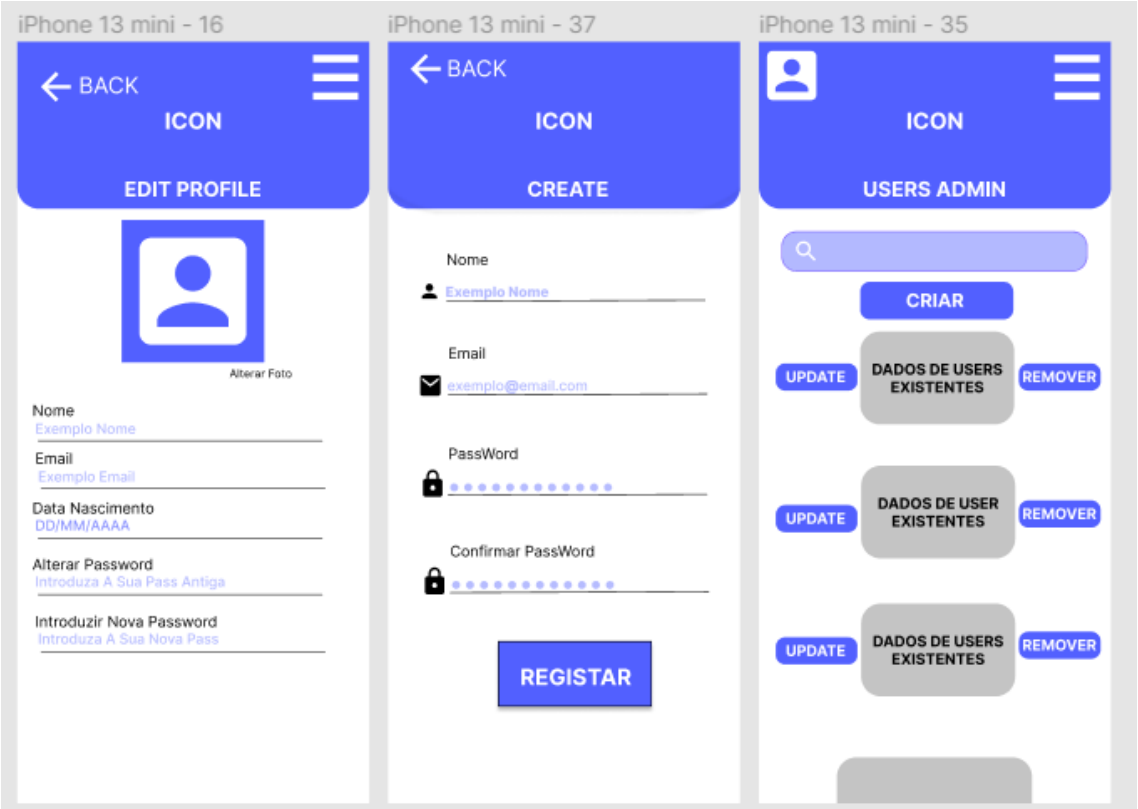

Figura 9 - Páginas CRUD de utilizadores

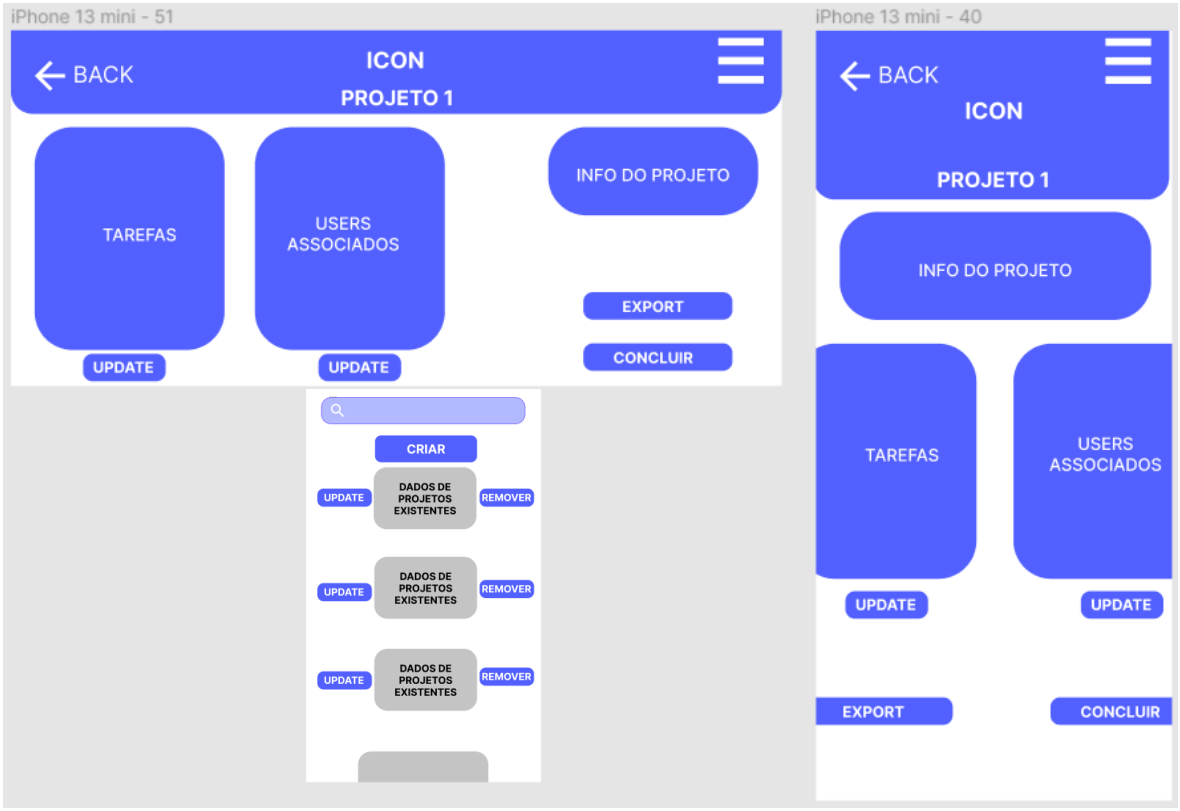

Figura 10 - Página para gestão de projetos

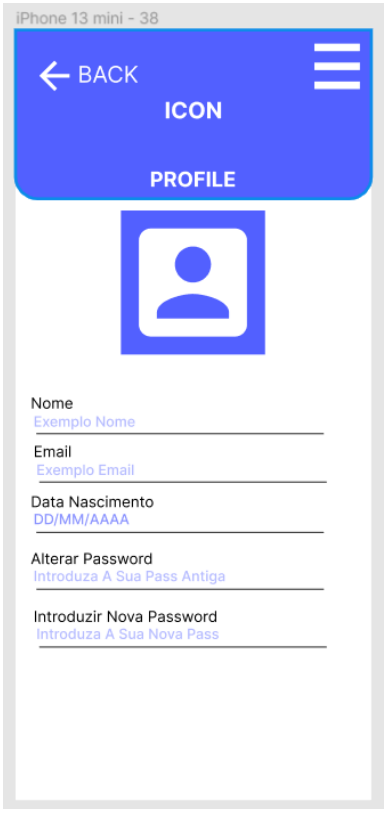

Figura 11 - Gestão do perfil de utilizador

3.6 Base de dados

O esquema da base de dados representa o modelo desenvolvido para o trabalho prático tendo em conta os requisitos descritos acima.

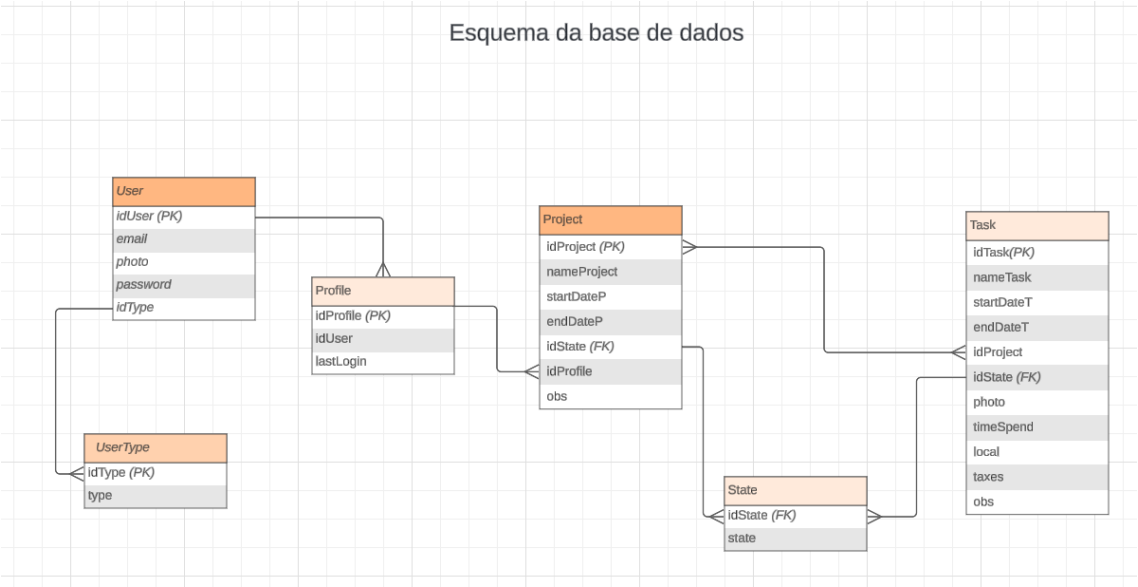

Figura 12 - Diagrama ER

4. Base de Dados

4.1 API

Para a API, utilizamos como exemplo o repositório disponibilizado pelos docentes no link https://github.com/marcelowolfsmartindustries/AW-P-EXEMPLO-1778. As tecnologias utilizadas para o seu desenvolvimento foram Node.JS, Prisma, Javascript e Vercel.

Começamos por desenvolver os controllers, onde toda a lógica dos endpoints é implementada, desde **creates**, **updates** e **gets**.

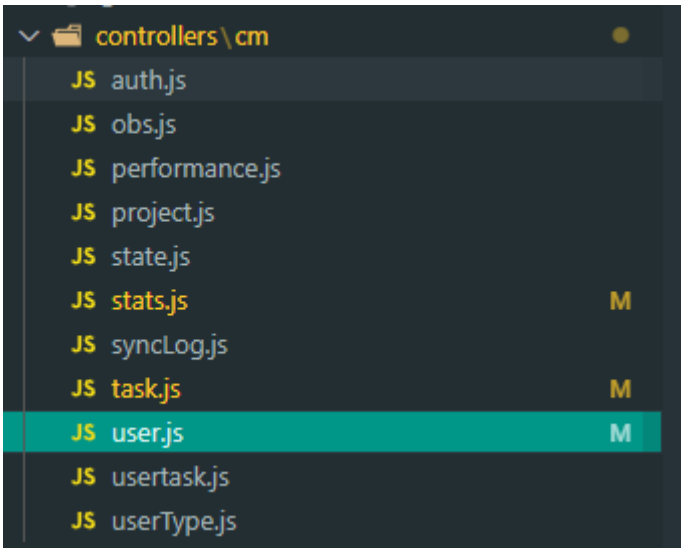

Figura 13- Controllers

```
// Create user
exports.create = async (req, res) => {
    const { email, photo, password, idtype, username, name, last_login } = req.body;
    var hashedPassword = bcrypt.hashSync(password, 8);
    try {
        const user = await prisma.user.create({
            data: {
                email: email,
                photo: photo,
                password: hashedPassword,
                idtype: idtype,
                username: username,
                name: name,
                last_login: last_login
            },
        })
        res.status(201).json(user)
    } catch (error) {
        res.status(400).json({ msg: error.message })
    }
}
```

Figura 14 - Exemplo de Create

Para a base de dados ser criada no **vercel**, descrevemos através do **prisma**, a estrutura completa da base de dados.

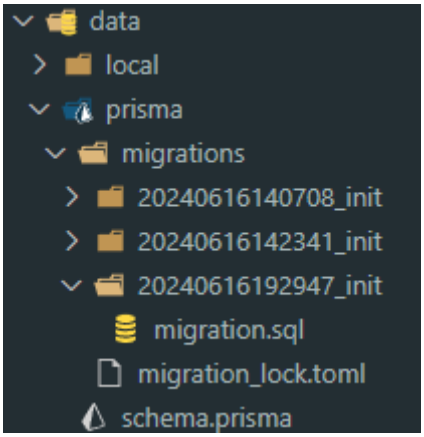

Figura 15 - Prisma e migrações

```
generator client {
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model usertype {
  idtype      Int          @id @default(autoincrement())
  type        String       @db.VarChar(50)
}

model user {
  iduser      Int          @id @default(autoincrement())
  email       String       @unique @db.VarChar(100)
  photo       String?      @db.Text
  password    String       @db.VarChar(255)
  idtype      Int
  username    String?      @unique @db.VarChar(50)
  name        String?      @db.VarChar(100)
  last_login  DateTime?
}
```

Figura 16 - Exemplos de tabelas no prisma

Para gerir o acesso às rotas, definimos um midlleware de autenticação, ou seja, apenas conseguem aceder ás rotas os utilizadores com autenticação feita e um token válido.

```
const authenticateUtil = require('../utils/authenticate.js');

module.exports = async (req, res, next) => {
    const accessToken = req.headers['authorization']; // req.headers['x-access-token'];

    if (!accessToken) {
        return res.status(401).send("unauthorized");
    }

    try {
        const bearer = accessToken.split(' ');
        const bearerToken = bearer[1];

        const result = await authenticateUtil.certifyAccessToken(bearerToken);
        req.body.loggedUserName = result.Name;

        return next();
    } catch (err) {
        return res.status(401).send("unauthorized");
    }
}
```

Figura 17 - Middleware

Por fim, definimos as rotas, é nestes ficheiros que são especificados os endpoints para termos acesso á informação através da aplicação.

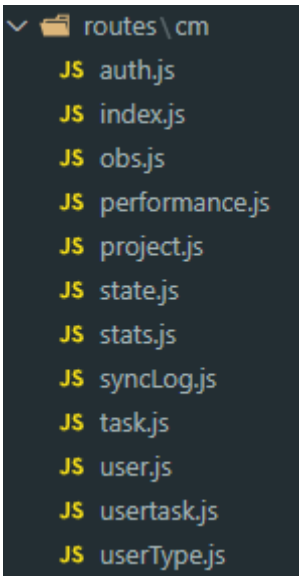

Figura 18 - Ficheiros das rotas

```
const userRouter = require('express').Router();
const controller = require('../../controllers/cm/user');
const authMiddleware = require('../../middlewares/auth');

//use auth middleware
userRouter.use(authMiddleware);

//user CRUD
userRouter.get('/', controller.getAll); //read all
userRouter.get('/:iduser', controller.getById); //read one by his id
userRouter.post('/create', controller.create); //create new user
userRouter.put('/update/:iduser', controller.update); //update user
userRouter.delete('/delete/:iduser', controller.delete); //delete user

module.exports = userRouter;
```

Figura 19 - Rotas para a tabela User

```
const router = require('express').Router();

router.use('/auth', require('./auth'));
router.use('/project', require('./project'));
router.use('/syncLog', require('./syncLog'));
router.use('/task', require('./task'));
router.use('/user', require('./user'));
router.use('/userType', require('./userType'));
router.use('/stats', require('./stats'));
router.use('/obs', require('./obs'));
router.use('/usertask', require('./usertask'));
router.use('/state', require('./state'));
router.use('/performance', require('./performance'));

module.exports = router;
```

Figura 20 - Index principal das rotas

Para finalizar e ter toda a API funcional e acessível, carregamos para o vercel de forma a estar exposta constantemente. Nesta ferramenta fica também armazenada a base de dados que é gerada em função das migrations criadas.

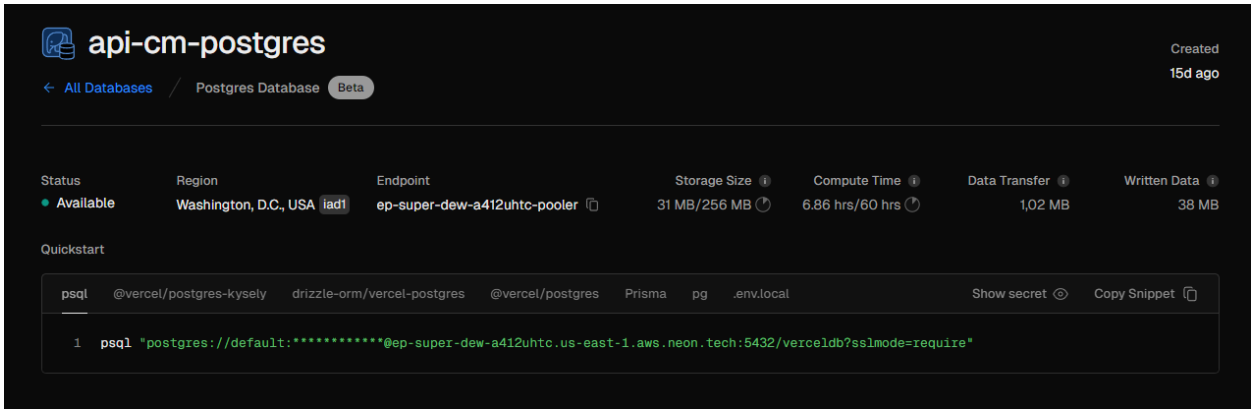

Figura 21 - Base de dados hospedada no vercel

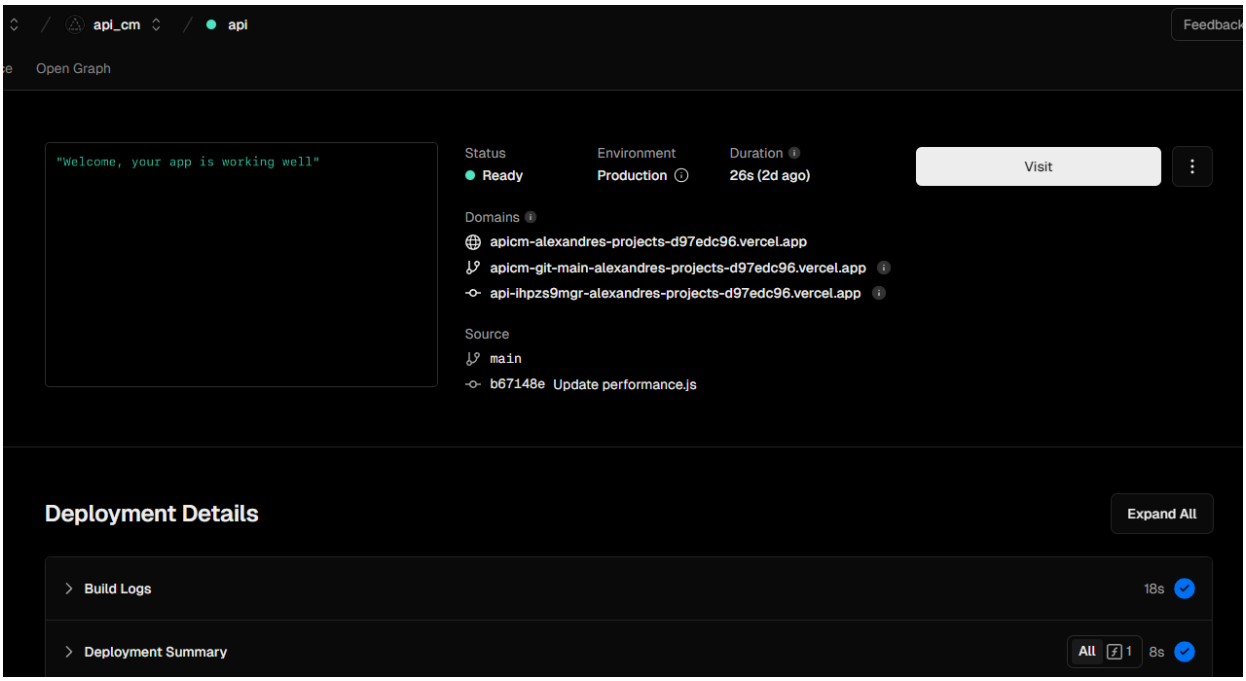

Figura 22 - Deploy da API no vercel

4.2 Base de dados local – ROOM

Outro método alternativo á API para armazenar os dados, é o ROOM. Esta ferramenta utilizamos na aplicação em kotlin, com o objetivo de exemplificar uma implementação, neste caso, quando um utilizador pretende criar uma observação numa tarefa e não tem acesso á internet, se for o caso, a observação é guardada localmente na base de dados ROOM, para posteriormente ser sincronizada com a API.

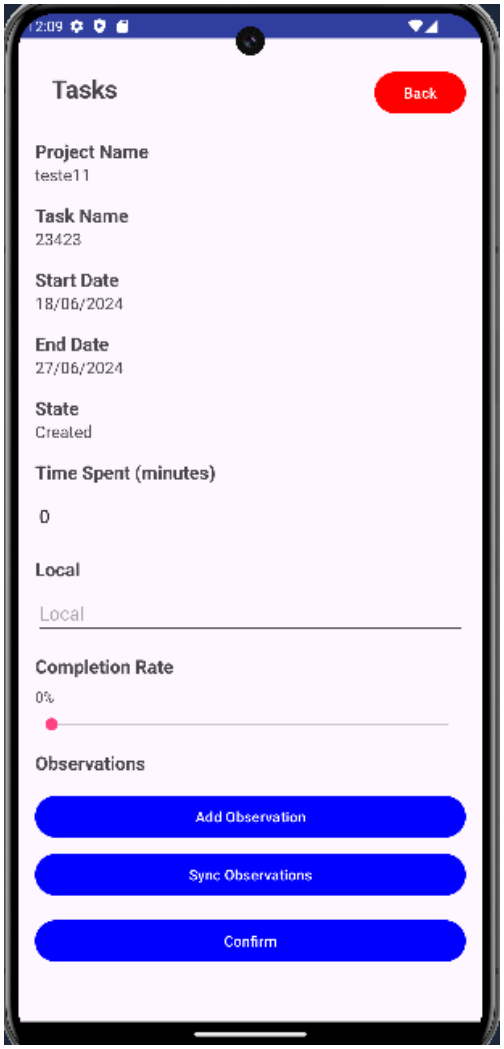

Figura 23 - Ecrã para edição de tarefas

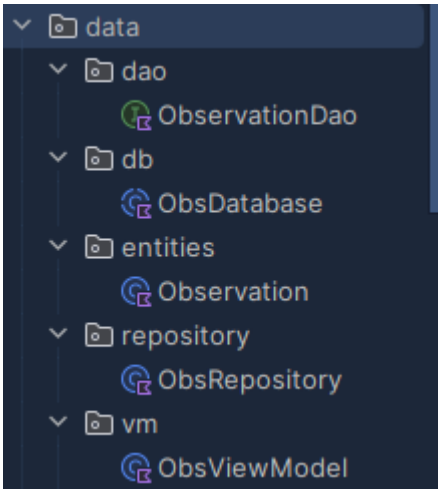

Figura 24 - Estrutura de ficheiros

Para criar a base de dados em ROOM, criámos os seguintes ficheiros:

1. Entidade: Uma entidade representa uma tabela na base de dados. Define os campos da tabela e os seus tipos de dados.

2. DAO (Data Access Object): Um DAO é uma interface que fornece métodos para aceder e manipular dados na base de dados. Encapsula as operações de CRUD para a entidade correspondente.

3. Base de Dados: O arquivo de base de dados armazena os dados reais da aplicação. Contém as tabelas e seus registos.

4. Repositório: Um repositório é uma camada de abstração que encapsula o DAO e fornece uma interface mais flexível para aceder aos dados. Pode ser usado para implementar lógica de negócio e regras de validação.

5. ViewModel: Um ViewModel é uma classe que observa os dados do repositório e expõe os dados para a interface do utilizador. É responsável por preparar os dados para serem exibidos no ecrã e lidar com eventos da interface.

```
@Dao  AlexandrePSantos
interface ObservationDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)  AlexandrePSantos
    suspend fun insert(observation: Observation)

    @Update  AlexandrePSantos
    suspend fun update(observation: Observation)

    @Delete  AlexandrePSantos
    suspend fun delete(observation: Observation)

    @Query("SELECT * FROM obs")  AlexandrePSantos
    fun getAllObservations(): LiveData<List<Observation>>

    @Query("SELECT * FROM obs WHERE id = :id")  AlexandrePSantos
    suspend fun getObservationById(id: Int): Observation?

    @Query("SELECT * FROM obs WHERE issynced = 0")  AlexandrePSantos
    suspend fun getUnsyncedObservations(): List<Observation>
}
```

Figura 25 - Ficheiro DAO

```kotlin
@Database(entities = [Observation :: class], version = 1, exportSchema = false)
abstract class ObsDatabase : RoomDatabase(){
    abstract  fun  obsDao(): ObservationDao  AlexandrePSantos

    companion object {  AlexandrePSantos
        @Volatile
        private var INSTANCE: ObsDatabase? = null

        fun getDatabase(context: Context): ObsDatabase{  AlexandrePSantos
            val tempInstance = INSTANCE
            if(tempInstance != null){
                return tempInstance
            }
            synchronized( lock: this){
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    ObsDatabase::class.java,
                     name: "obs_database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Figura 26 - Ficheiro Database

```kotlin
@Entity(tableName = "obs")  AlexandrePSantos
class Observation(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val idtask: Int,
    val iduser: Int,
    @ColumnInfo(name = "content") val content: String,
    @ColumnInfo(name = "issynced") val issynced: Boolean = false
)
```

Figura 27 - Ficherio exemplo Entity

```kotlin
class ObsRepository(private val obsDao: ObservationDao) {  AlexandrePSantos
    val readAllObservations: LiveData<List<Observation>> = obsDao.getAllObservations()

    suspend fun addObservation(observation: Observation) {  AlexandrePSantos
        obsDao.insert(observation)
    }

    suspend fun updateObservation(observation: Observation) {  AlexandrePSantos
        obsDao.update(observation)
    }

    suspend fun deleteObservation(observation: Observation) {  AlexandrePSantos
        obsDao.delete(observation)
    }

    suspend fun getUnsyncedObservations(): List<Observation> {  AlexandrePSantos
        return obsDao.getUnsyncedObservations()
    }
}
```

Figura 28 - Ficheiro Repository

```
class ObsViewModel(application: Application) : AndroidViewModel(application) {  AlexandrePSantos *
    val readAllObservations: LiveData<List<Observation>>
    private val repository: ObsRepository

    init {  AlexandrePSantos
        val obsDao = ObsDatabase.getDatabase(application).obsDao()
        repository = ObsRepository(obsDao)
        readAllObservations = repository.readAllObservations
    }

    fun addObservation(observation: Observation) {  AlexandrePSantos
        viewModelScope.launch(Dispatchers.IO) {
            repository.addObservation(observation)
        }
    }

    fun syncObservations(apiService: ApiService, callback: (Boolean) -> Unit) {  AlexandrePSantos
        viewModelScope.launch(Dispatchers.IO) {
            val unsyncedObservations = repository.getUnsyncedObservations()
            if (unsyncedObservations.isNotEmpty()) {
                var success = true
                for (observation in unsyncedObservations) {
                    val obsRequest = ObsRequest(observation.idtask, observation.iduser, observation.content)
                    val response = apiService.createObs(obsRequest).execute()
                    if (response.isSuccessful) {
                        repository.deleteObservation(observation)
                    } else {
                        success = false
                    }
                }
                withContext(Dispatchers.Main) {
                    callback(success)
                }
            } else {
                withContext(Dispatchers.Main) {
                    callback(false)
                }
            }
        }
    }
}
```

Figura 29 - Ficheiro ViewModel

5. Desenvolvimento

5.1. Dependências

As dependências que definimos para além das já existentes, resumem-se a dependências para utilizar o **ROOM** para base de dados local, **Retrofit** para chamadas á API, **ViewModel** e **LiveData** para utilizar com a base de dados local e **Coroutine** para gerir a concorrência de maneira assíncrona e simplificada em aplicações Android.

```
dependencies {
        // Core libraries
        implementation("org.jetbrains.kotlin:kotlin-stdlib:1.5.31")
        implementation("androidx.core:core-ktx:1.7.0")
        implementation("androidx.appcompat:appcompat:1.4.0")
        implementation("com.google.android.material:material:1.4.0")
        implementation("androidx.constraintlayout:constraintlayout:2.1.2")
        implementation("androidx.viewpager2:viewpager2:1.1.0")
        implementation("com.fasterxml.jackson.core:jackson-databind:2.13.0")

        // ViewModel and LiveData
        implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0")
        implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.2.0")
        implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.4.0")

        // Retrofit for API calls
        implementation("com.squareup.retrofit2:retrofit:2.9.0")
        implementation("com.squareup.retrofit2:converter-gson:2.9.0")
        implementation("com.squareup.okhttp3:logging-interceptor:4.9.0")

        // Room for local database
        implementation("androidx.room:room-runtime:2.6.1")
        implementation("androidx.room:room-ktx:2.6.1")
        implementation("androidx.room:room-common:2.6.1")
        kapt("androidx.room:room-compiler:2.6.1")
        testImplementation("androidx.room:room-testing:2.6.1")

        // Coroutine
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2")
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.2")
```

Figura 29 - Dependências utilizadas

5.2. Funcionalidades desenvolvidas

Devido a questões de tempo não nos foi possível desenvolver todas as funcionalidades propostas pelos docentes, pelo que ficou por implementar a exportação de estatísticas, no entanto conseguimos implementar as restantes.

5.2.1. Administrador

Começando pelo Admin, este tem a possibilidade de adicionar, editar e apagar projetos e utilizadores.

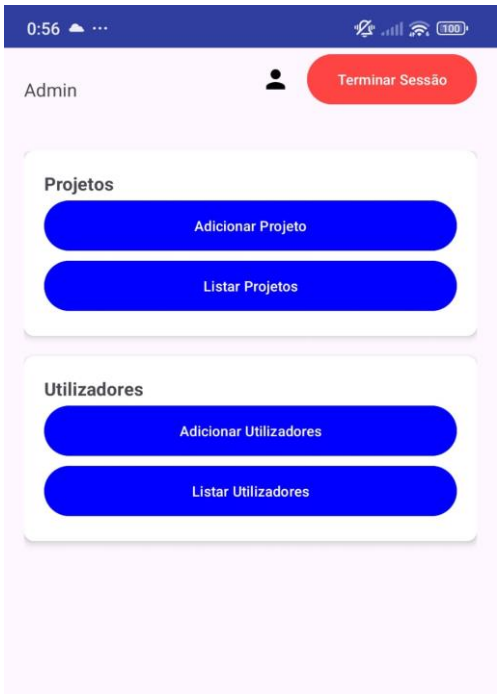

Figura 20 - Dashboard Admin

Figura 31 - Edição de projetos

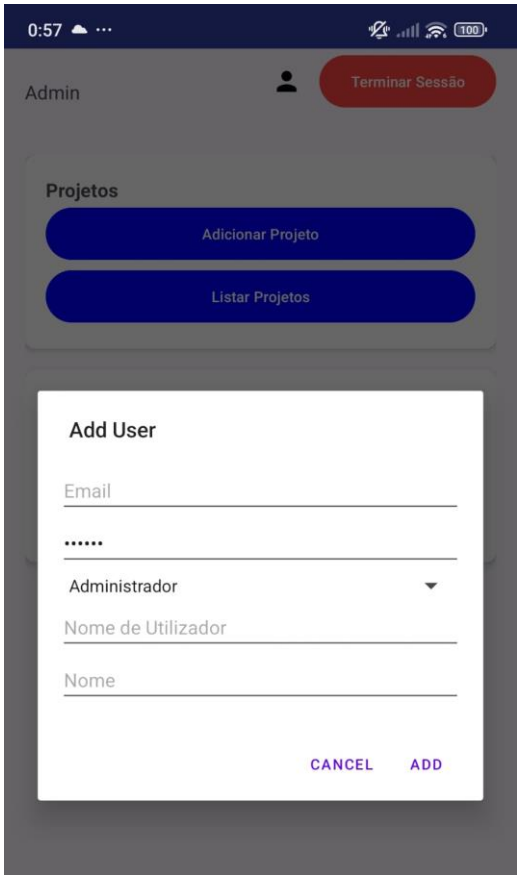

Figura 32 - Criação de utilizadores

Figura 33 - Listagem de utilizadores

Figura 34 - Edição de utilizadores

5.2.2. Gestor

O Gestor, é provavelmente o tipo de utilizador mais trabalhoso de implementar, pois, está interligado a todas as atividades principais da aplicação. Pode gerir projetos, desde a criação de tarefas, associação de utilizadores às mesmas e avaliação da sua performance em determinada tarefa.

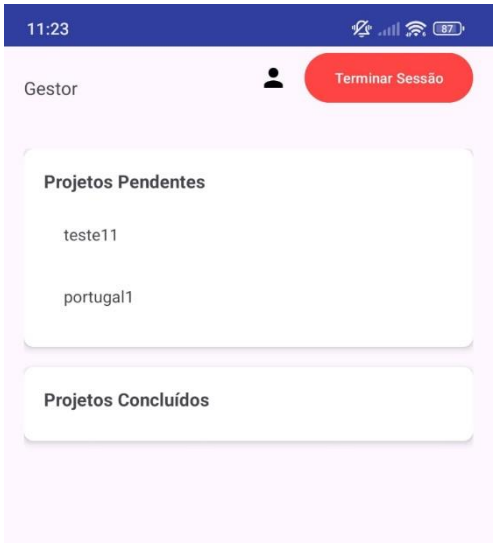

Figura 35 - Dashboard gestor

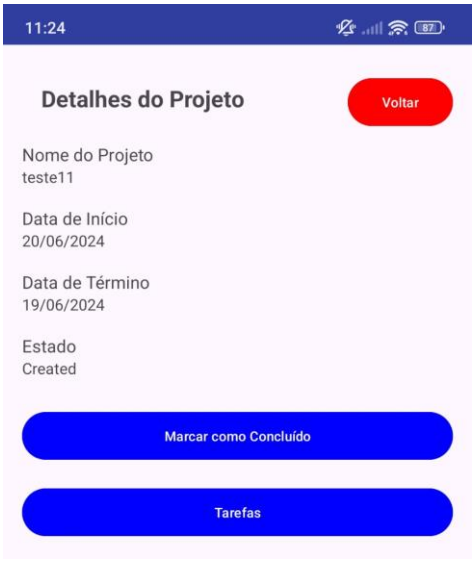

Figura 36 - Detalhes projeto

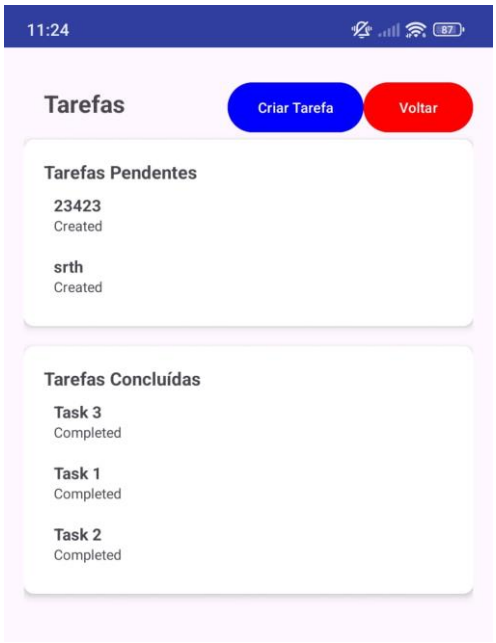

Figura 37 - Tarefas de um determinado projeto

Figura 38 - Detalhes tarefa

Figura 39 - Listagem de observações

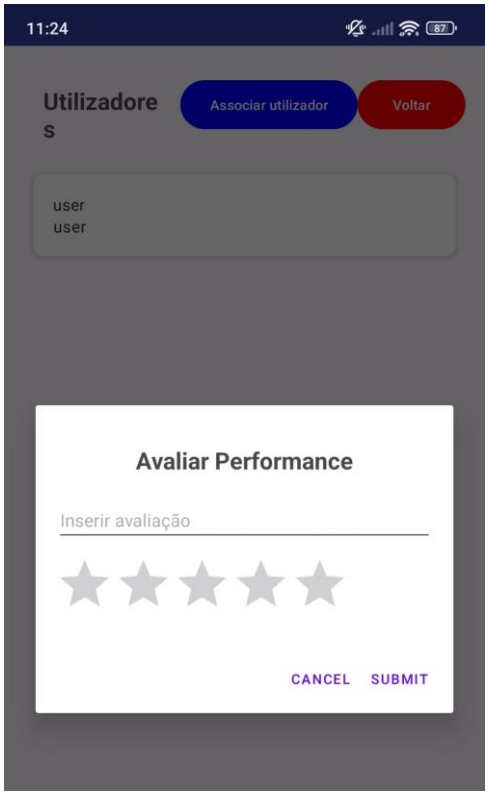

Figura 40 - Listagem e avaliação de utilizadores

5.2.3. Utilizador

Por fim implementamos o Utilizador, este é o tipo de utilizador com menos requisitos, apenas pode alterar o estado da tarefa á qual está associado, adicionar observações e é onde implementamos a base de dados local, quando não há internet as observações são guardadas localmente.

Figura 41 - Dashboard utilizador

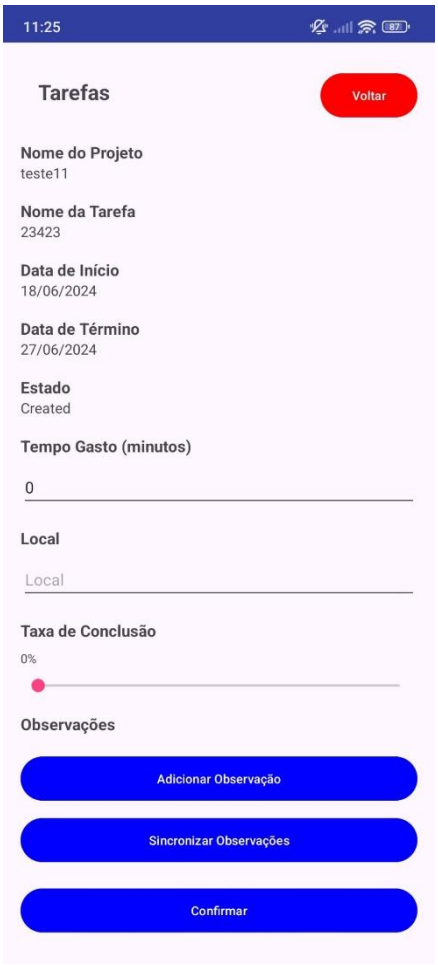

Figura 42 - Edição da tarefa atribuida a um utilizador

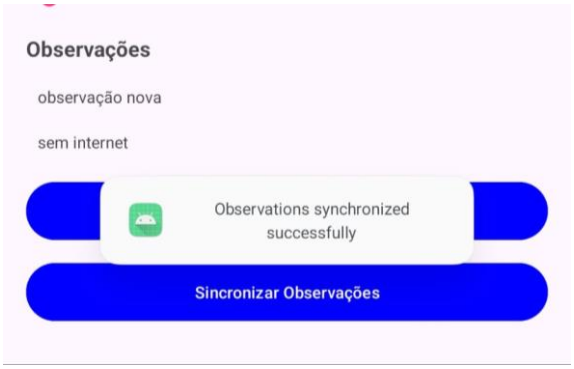

Figura 43 - Sincronização de observações

5.3. Testes realizados

Para realizar testes durante o desenvolvimento, utilizámos maioritariamente **Logs**, através do código onde pretendíamos obter mais informação utilizávamos a função **Log.d(…)**, com esta ferramenta tínhamos acesso a informação dos pedidos da APP á API, para resolver erros ou mesmo para saber se estavam a ser passados os parâmetros corretos em determinadas ocasiões.

Figura 44 - Logs para testar funcionalidades

```
private fun fetchUsers() {  AlexandrePSantos *
    apiService.getAllUsers().enqueue(object : Callback<List<UserDetailsResponse>> {
        override fun onResponse(
            call: Call<List<UserDetailsResponse>>,
            response: Response<List<UserDetailsResponse>>
        ) {
            if (response.isSuccessful) {
                response.body()?.let {
                    userAdapter.submitList(it)
                }
                Log.d( tag: "UsersActivity",  msg: "API response: ${response.body()}")

            } else {
                Toast.makeText( context: this@UsersActivity,  text: "Failed to fetch users", Toast.
                Log.e( tag: "UsersActivity",  msg: "Failed to fetch users: ${response.code()}")
            }
        }
```

Figura 45 - Código para os logs

5.4. Instruções de instalação

Para poder instalar a aplicação, temos primeiro de gerar o APK, com isto começamos por selecionar a opção **Build** -> **Build APP Bundle/APK.**

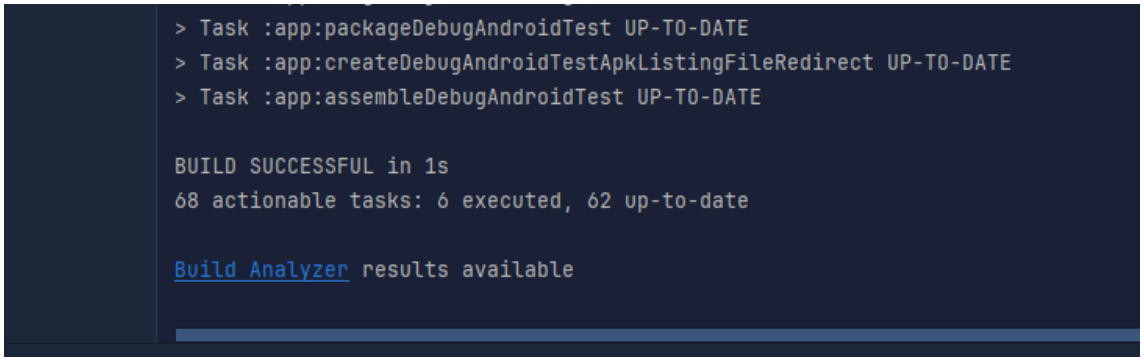

Figura 46 - Criação da build

Quando o APK estiver pronto, podemos passar do computador para o smartphone Android.

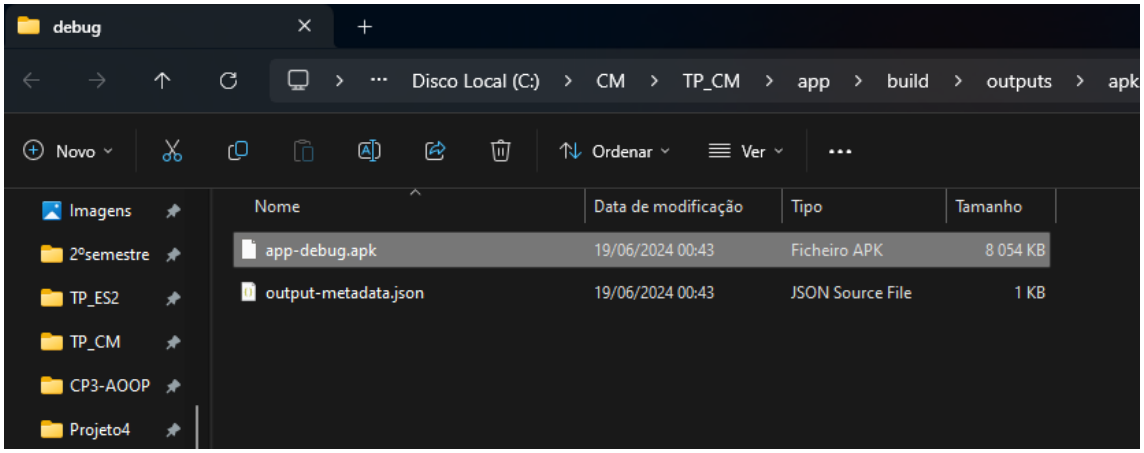

Figura 47 - Localização do APK

Tendo o APK no smartphone podemos passar á instalação e por fim, utilizar a APP.

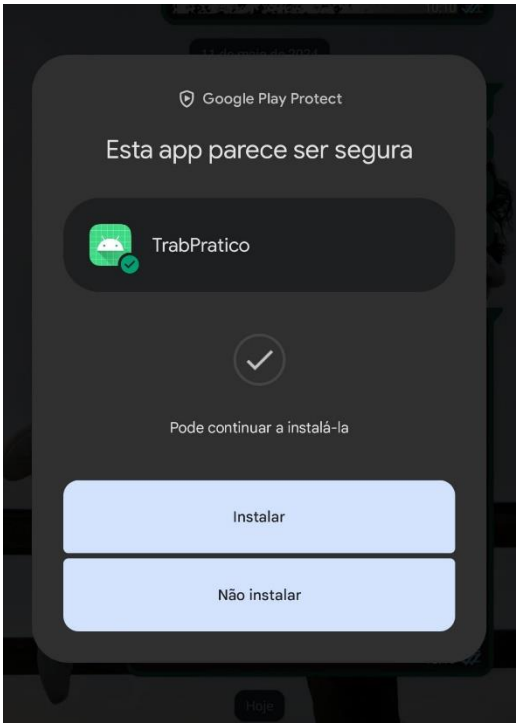

Figura 48 - Instalação da app

Figura 49 - App instalada

6. Conclusões

Com este trabalho, tivemos oportunidade de desenvolver a nossa capacidade de trabalho em equipa e organização. Desde a definição de requisitos, é criação da base de dados e design dos layouts, conseguimos aprimorar tecnologias e conhecimentos adquiridos em unidades curriculares anteriores.

Para além disso, tivemos oportunidade de entrar no mundo do desenvolvimento para equipamentos Android, apesar de haver dificuldades por ser algo novo, conseguimos implementar a aplicação e conseguimos reter diversos conhecimentos sobre o desenvolvimento mobile.

7. Referências

- Documentação Android Studio: https://developer.android.com/develop?hl=pt-br
- Draw.io: https://diagrams.net/
- Trello: https://trello.com/
- Figma: https://www.figma.com/