Configuración, Implementación y Mantenimiento de Sistemas Informáticos
Tópico 5

Integrantes:

- Francisco Alexis Morales Vallado
- Mohammed Amine Bennani
- José Luis Corvo Rosado
- Juan Diego Villalobos Quirós
- Fernando Pardo Beltrán
- Juan López de la Calle
- Alejandro Cores

ÍNDICE

1. RESUMEN

Hemos realizado una app Android con el objetivo de informar sobre el estado de las estaciones de Sevici, en la ciudad de Sevilla, para informar a los usuarios de la disponibilidad de bicicletas, así como de espacios disponibles para estacionarlas en tiempo real.

Hemos implementado, además diversas herramientas y servicios externos: por un lado, Firebase Firestore, para poder almacenar datos sobre estaciones en tiempo real; en relación a la capa de autenticación, hemos implementado el SDK oficial de Google, Google Auth for Android, para proveer de mayor seguridad, así como de una experiencia de usuario personalizada. Por último, de cara a monitorizar, desplegar y desarrollar código de forma mucho más ágil, hemos implementado Sentry (monitorización de apps móviles en ejecución), CircleCI (pipeline CI/CD), Google Analytics (obtención de métricas y KPIs) y Mockito (implementación de tests unitarios).

Además, usaremos la API (conjunto de definiciones y protocolos que se utiliza para integrar el software de las aplicaciones) de Citybikes de dominio público para proveer datos a la app. También hemos usado OpenStreetMap como apoyo gráfico para ubicar las estaciones de bicicletas con mayor exactitud.

2. INTRODUCCIÓN

El uso de dispositivos inteligentes, como smartphones, han implicado un gran avance en nuestras vidas. Día tras día usamos estos dispositivos para comunicarnos con otras personas, consultar información sobre diversos temas, o simplemente visualizar noticias de temas que nos parezcan relevantes. En definitiva, han cambiado la forma en la que vivimos. También lo han hecho en relación a cómo nos transportamos.

Con la digitalización de la vida de las personas, se abre un nuevo sector: la movilidad compartida(Shared Mobility). Actualmente, diversos servicios como el *carsharing*, *motosharing* o *bikesharing* son comunes en nuestras ciudades, siendo algunos de estos servicios el principal modo de transporte de muchas personas. Y no es para menos. Según un estudio desarrollado por la consultora McKinsey & Company, el 25% de la población mundial reside en ciudades con más de un millón de habitantes. Para 2030, según el estudio, se estima que el número de megaciudades (definidas como ciudades con más de 10 millones de habitantes) aumente hasta 43. Es por esta razón por la cual ofrecer servicios ágiles y baratos de micromovilidad (medios de transporte utilizados para recorrer distancias cortas) en ciudades nos permite seguir avanzando hacia un mundo donde se consiga disminuir la contaminación por CO2.

Por suerte, actualmente existe una gran colaboración público-privada que apoya a startups y PYMEs que desarrollan soluciones de micromovilidad. Existen programas como Enisa, del Gobierno de España, que busca financiar proyectos de este tipo, así como financiación a fondo perdido como puede ser el fondo Next Generation de la Unión Europea, en la que podemos encontrar proyectos de empresas privadas en colaboración con universidades las cuales desarrollan estas soluciones software para diversos entes de la administración pública, como pueden ser los ayuntamientos.

Por ello, este proyecto tiene como objetivo principal agilizar la vida de los usuarios de estos servicios, concretamente, del servicio de *bikesharing* Sevici, implantado en la ciudad de Sevilla, ofreciendo información útil como puede ser la disponibilidad del servicio en la estación más cercana al usuario, entre otros...

3. ESTADO DEL ARTE

En la actualidad, existen diversas soluciones software que son capaces de proveer al usuario de información sobre los servicios bikesharing de diversas ciudades. Tenemos que hacer una distinción entre dos tipos de apps móviles: las que simplemente proveen rutas haciendo uso de APIs públicas, como puede ser CityBike (la cual usaremos en nuestra solución) u otras APIs de datos abiertos ofrecidas por administraciones públicas, o bien las que ofrecen un servicio privado de *bikesharing*, cuyo objetivo no es solo ofrecer la solución software, sino también la infraestructura en lo que concierne al medio de transporte (es decir, también ofrece bicicletas, por ejemplo). En relación a las apps que ofrecen información detallada, podemos destacar las siguientes:

- **Sevici**: se caracteriza por ser el servicio público por excelencia en la ciudad de Sevilla. Es, en la misma ciudad, el servicio que posee la red de estaciones más amplia.
- **Meep**: esta app, disponible para iOS y Android es un agregador de servicios de movilidad compartida. Ofrece información en tiempo real acerca de los servicios de bus, bicicletas compartidas y taxis de diversas ciudades de Malta, Portugal y España. No posee infraestructura propia, es decir, no ofrece los vehículos a los usuarios.
- **Moovit**: similar a Meep, con la única diferencia de que está disponible en más de 150 ciudades en todo el mundo. Además, posee una plataforma de Smart Mobility, la cual permite desarrollar soluciones a medida para clientes públicos y privados.

A modo de resúmen, a continuación encontrará una tabla con información valiosa sobre los distintos servicios ofrecidos en la actualidad (también incluímos los que poseen infraestructura):

Nombre	¿Ofrece infraestructura?	Público/Privado	Disponibilidad en ciudades
Sevici	Si	Público	Sevilla
Meep	No	Privado	17 ciudades repartidas entre España, Portugal y Malta
Moovit	No	Privado	150 ciudades en todo el mundo
Mobike	Si	Privado	8 ciudades repartidas en Asia, Norteamérica y Europa
Ofo	Si	Privado	10 ciudades en todos los continentes

4. DESCRIPCIÓN DE LA ARQUITECTURA

Haciendo uso de la API gratuita CityBike (http://api.citybik.es/v2/), podremos obtener información en tiempo real sobre el estado de las estaciones disponibles para ser usadas con Sevici. Entre la información disponible podemos destacar el número de slots disponibles para aparcar bicicletas, el número de bicicletas disponibles, el nombre y dirección de la estación, o la ubicación de la misma (haciendo uso de la longitud y latitud).

Para poder consumir esta API vamos a desarrollar una app móvil haciendo uso del SDK nativo de Android. Además de desarrollar la app haciendo uso de este SDK, así como del lenguaje de programación Kotlin, vamos a implementar distintos servicios, librerías o arquitecturas.

- CircleCI:

Es una plataforma CI/CD (Continuous Integration/Continuous Delivery) la cual permite a los desarrolladores testear, ejecutar builds y desplegar el código en tiempo real, automatizando todo este proceso, consiguiendo que los ingenieros se enfoquen sólo en desarrollar el código o arquitectura, disminuyendo el tiempo que es dedicado a tareas manuales las cuales son muy “mecánicas”. Por si fuera poco, CircleCI posee integración con otras herramientas, como Jira, la cual nos permite modificar el estado de una tarea dependiendo del resultado de ejecución de una build. Por ejemplo, si hemos ejecutado una build satisfactoriamente, CircleCI podrá modificar el estado de la tarea relacionada a “Hecho”, entre otros.

- Firebase Firestore

Firestore es la base de datos no relacional (NoSQL) ofrecido por Firebase; un servicio que nos permite desplegar soluciones haciendo uso de una base de datos escalable, flexible, que haciendo uso de JSON es capaz de almacenar una gran cantidad de datos en tiempo real. Además, posee diversas funcionalidades, como el SDK para Android, que facilita la implementación de la misma.

- Sentry

Es un servicio (ofrecido en forma de SDK) que rastrea errores en el código fuente. Posee compatibilidad con diversas plataformas, como pueden ser front-ends desarrollados usando frameworks de JavaScript como ReactJS, VueJS o AngularJS, así como apps móviles usando los SDKs nativos de iOS y Android. Sentry registra todos los posibles crashes y bugs que se ejecutan en la app, y los almacena en una base de datos, para su posterior tratamiento.

- Google Analytics

Es el servicio estrella de Google para medir y analizar los resultados, así como obtener métricas en relación al funcionamiento de nuestra app. Nos ofrece diversos KPIs (valor medible que demuestra cuán eficientemente una organización está logrando sus objetivos), los cuales nos pueden ayudar a obtener información valiosa acerca del tiempo que dedica cada usuario en una sección, por ejemplo, o información sobre cómo mejorar la indexación en tiendas de apps (ASO - *App Store Optimization*).

- Picasso

Es una potente librería de caching y descarga de imágenes en tiempo real para Android. Nos permite mostrar imágenes en nuestra app dada una URL. Se puede instalar haciendo uso de Gradle o Maven.

- OpenStreetMap for Android

OpenStreetMap es la alternativa por excelencia a Google Maps, de código abierto. Nos permite mostrar un mapa en nuestra app, el cual se actualiza día a día, debido a que posee una gran comunidad detrás apoyando y siendo parte del proyecto, mejorándolo y solucionando bugs.

- AdMob

Google AdMob es una plataforma publicitaria para dispositivos móviles que puedes utilizar para generar ingresos con tu aplicación mediante el uso de anuncios de alta calidad. Mostrar anuncios a los usuarios de las aplicaciones le permite crear una fuente sustentable de ingresos que lo ayudan a expandir su empresa mientras se enfoca en crear y desarrollar aplicaciones de alta calidad.

- Google Authentication Service

Hoy en día, existen diversas alternativas a la hora de implementar un sistema de autenticación en nuestros desarrollos de apps móviles, como pueden ser Twitter Login o Twilio. Sin embargo, en este caso hemos elegido Google Auth Service, un servicio de rápida implementación que nos ofrece la capa de autenticación que necesitamos.

- MVVM (Model-View-ViewModel)

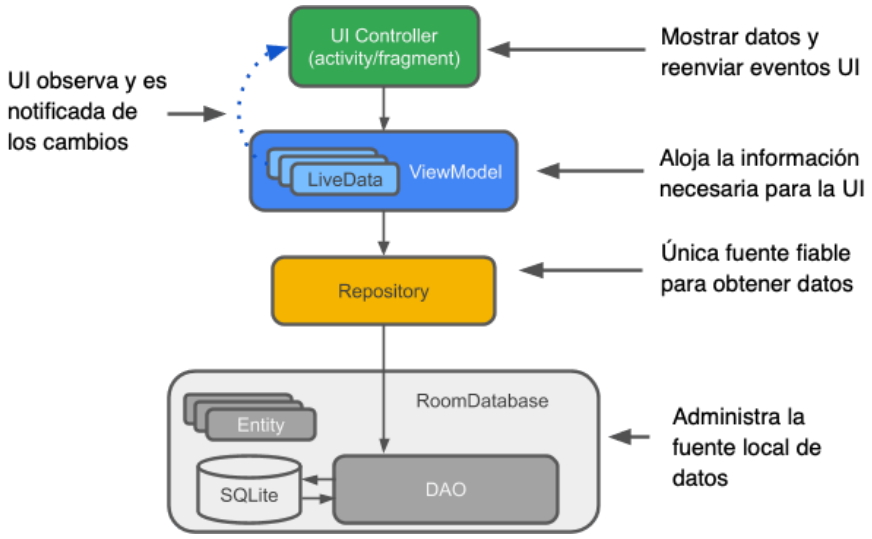
Es el patrón de arquitectura de software más avanzado en la actualidad. Se usa en el desarrollo de software para dispositivos Android e iOS, así como proyectos que hacen uso de microservicios. Su principal característica es aislar al máximo posible la interfaz de usuario de la lógica de negocio de la aplicación.

Posee 3 elementos distintos: el modelo, la vista y el modelo de vista. El modelo representa la capa de datos, o lógica de negocio de nuestro proyecto. Hace uso de dominios (*domain*); clases que contienen la información, los datos a usar.

La vista, por su parte, posee la misión de representar la información a través de la interfaz gráfica de usuario de nuestra app. Las vistas de MVVM están continuamente observando posibles cambios en los valores del modelo, de forma que la UI pueda ser actualizada en consecuencia.

Por último, nos encontramos con el modelo de vista. Es una clase ubicada entre el modelo y la vista. Contiene toda la lógica de representación: esto es, "*settea*" la información en la interfaz gráfica de usuario.

A continuación, mostramos un pequeño resúmen gráfico de la arquitectura MVVM:



- Dagger

Dagger es una librería, actualmente de Google, para inyección de dependencia (DI). La DI es una solución que tienen los desarrolladores para modularizar la creación de objetos y encapsular sus instancias.

A pesar de que la DI tiene una integración compleja, su implementación es prácticamente obligatoria en cualquier proyecto actual, ya que su potencial y posibilidad de personalización para la creación de instancias con Dagger son muy potentes.

Los módulos y componentes son las clases principales de Dagger. En los módulos es en donde se declaran las instancias de las clases que nosotros vamos a usar a lo largo de nuestros proyectos. Para ello se usan dos anotaciones principales con las cuales acompañar las instancias: “@Provider” y “@Binds”.

- @Provider nos permite devolver la instancia de un objeto en un bloque de código y controlar cómo se instancia, mientras que @Binds sólo devuelve la instancia de un objeto de manera simple y sin que podamos controlar cómo se instancia.

- @Binds autogenera menos código para instanciar los objetos y lo hace en menos tiempo. Por tanto, siempre que sea posible, debemos usar @Binds (normalmente cuando queremos devolver clases abstractas o interfaces).

- Mockito

Es un framework para testing de código abierto, principalmente usado en Java/Android, lanzado bajo la licencia MIT. Este framework permite ejecutar código haciendo uso de objetos mock, los cuales no son más que objetos falsos cuyo principal objetivo es testear los inputs y outputs del código desarrollado.

Es usado principalmente en desarrollo de software TDD (Test-driven development). TDD, por otra parte, es el proceso de desarrollar software generando casos de uso antes de que el software esté completamente desarrollado, haciendo especial énfasis en lo que queremos que ocurra, repitiendo una y otra vez la ejecución de los test cases, y monitorizando que todos son ejecutados continuamente de manera satisfactoria.

- AndroidX

AndroidX es una biblioteca rediseñada para que los nombres de los paquetes sean más claros. La jerarquía de Android es solo para las clases predeterminadas de Android, que vienen con el propio sistema operativo. Y el resto de las bibliotecas ó dependencias formarán parte de AndroidX.

AndroidX es una mejora significativa respecto de la biblioteca de compatibilidad de Android original. Los paquetes de AndroidX reemplazan por completo la biblioteca de compatibilidad, ya que proporcionan paridad de funciones y bibliotecas nuevas.

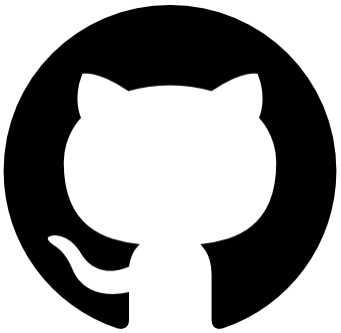
A diferencia de la biblioteca de compatibilidad de Android , cuya última actualización fue la versión 28.0.0, los paquetes de AndroidX se mantienen y actualizan por separado, mediante un control semántico de versiones.

- Retrofit

Retrofit es una librería para Android y Java compatible con Kotlin para hacer llamadas de red, obtener el resultado, y “parsearlo” de forma automática a su objeto. Esto facilita mucho realizar peticiones a un API y procesar las respuestas.

- Android Studio

Entorno de desarrollo integrado (IDE) oficial para el desarrollo de apps para Android, basado en Intellij IDEA. Será nuestro entorno de trabajo. Este software cuenta con herramientas y servicios que son necesarios para que un desarrollador sea capaz de crear nuevas aplicaciones. Esto incluye desde el código, al diseño de la interfaz de usuario de la aplicación. Android Studio es un entorno de desarrollo multiplataforma, de manera que es posible utilizarlo en Windows, MacOS, Linux y ChromeOS.

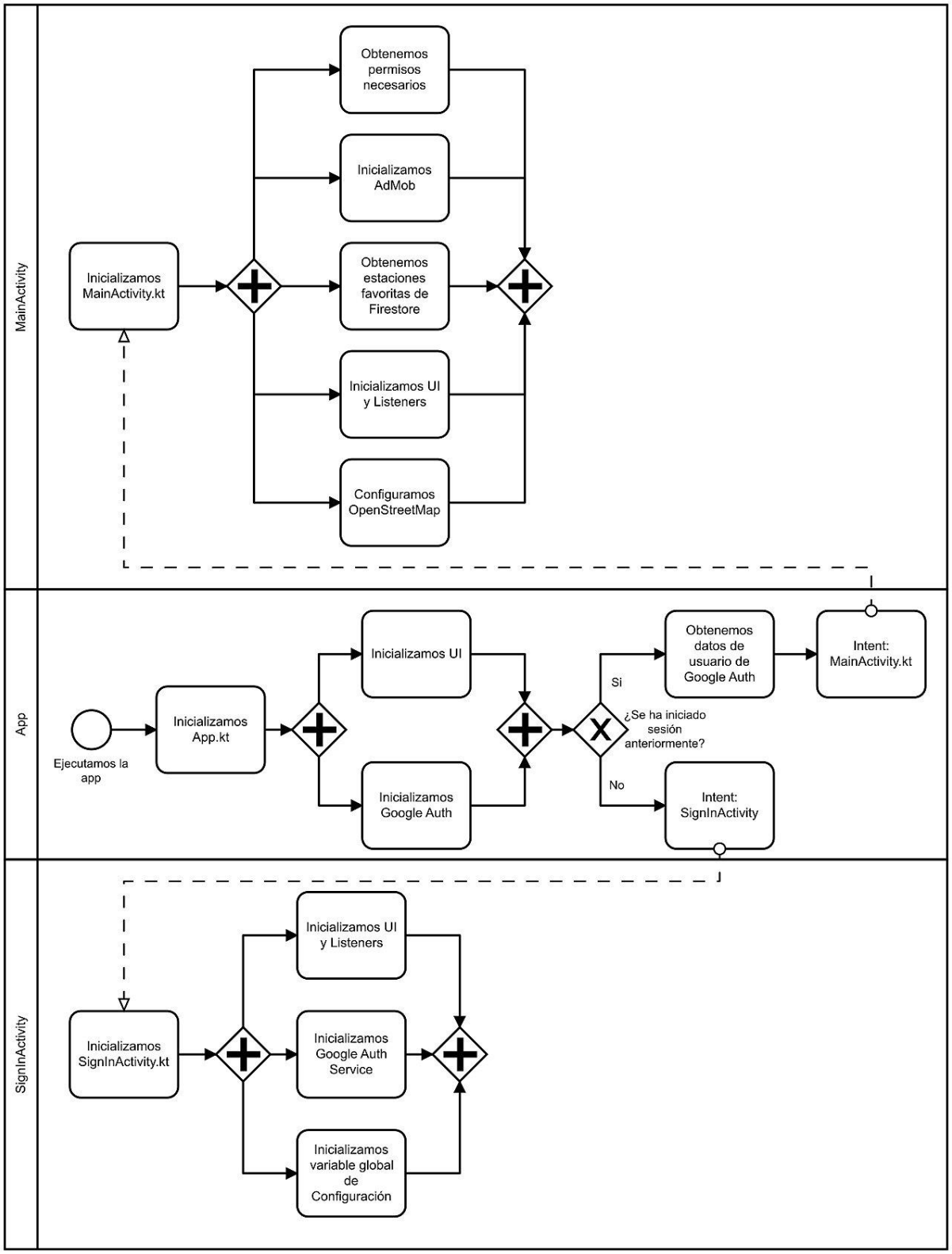


- GitHub

GitHub es un portal creado para alojar el código de las aplicaciones de cualquier desarrollador, y como usuario no sólo puedes descargarte la aplicación, sino también entrar a su perfil para leer sobre ella o colaborar en su desarrollo. Las principales características de la plataforma es que ofrece las mejores características de este tipo sin perder la simplicidad. Es multiplataforma y tiene multitud de interfaces de usuario.

La web utiliza el sistema de control de versiones Git, con el que los desarrolladores pueden administrar su proyecto, ordenando el código de cada una de las nuevas versiones que sacan de sus aplicaciones para evitar confusiones. Git es un proyecto de código abierto maduro y con mantenimiento activo, ya que un asombroso número de proyectos dependen de él.

- **Flujo de datos en la ejecución**



- **Código fuente**

  - **Dominios**

· **NetworkResponse:** representa la respuesta del endpoint que usamos para obtener información sobre una estación. Su contenido es un objeto de tipo Network.

· **Network:** posee información, tal como el nombre del servicio, así como la ubicación del mismo (de la ciudad) y la lista de estaciones.

· **NetworkLocation:** posee información relacionada con la ciudad, país, latitud y longitud.

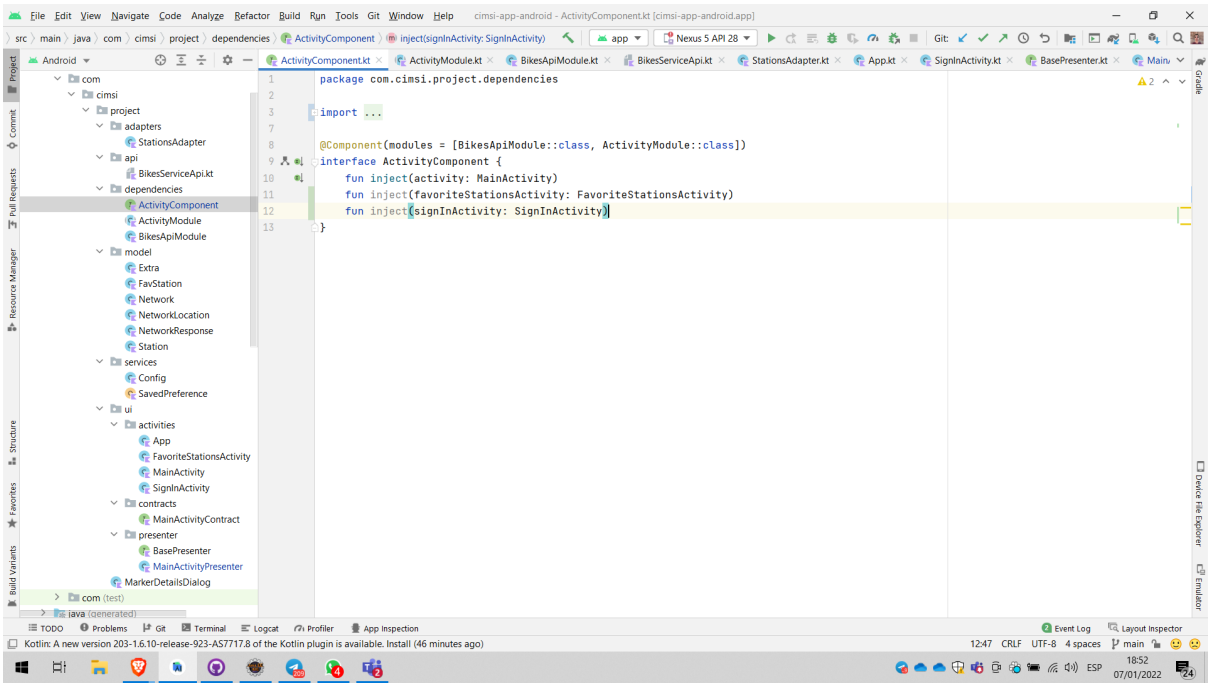
· **Station:** representa una estación (es decir, una Network posee una lista de Station). La información que podemos encontrar en este objeto es el nombre, latitud, longitud, slots libres, bicicletas libres e información extra.

· **Extra:** posee información relacionada con la dirección de una estación, así como el estado (“OPEN” o “CLOSED”).

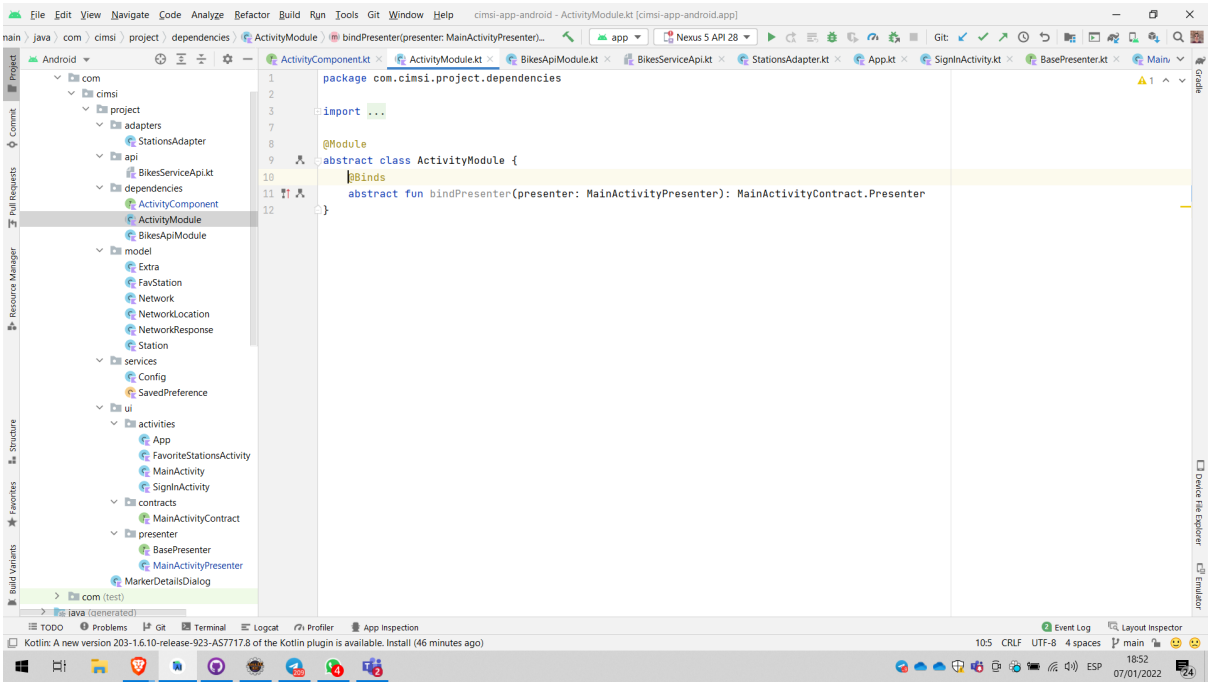
· **FavStation:** posee información relacionada con las estaciones señaladas como favoritas: nombre, dirección, latitud y longitud.

  - **Dependencias**

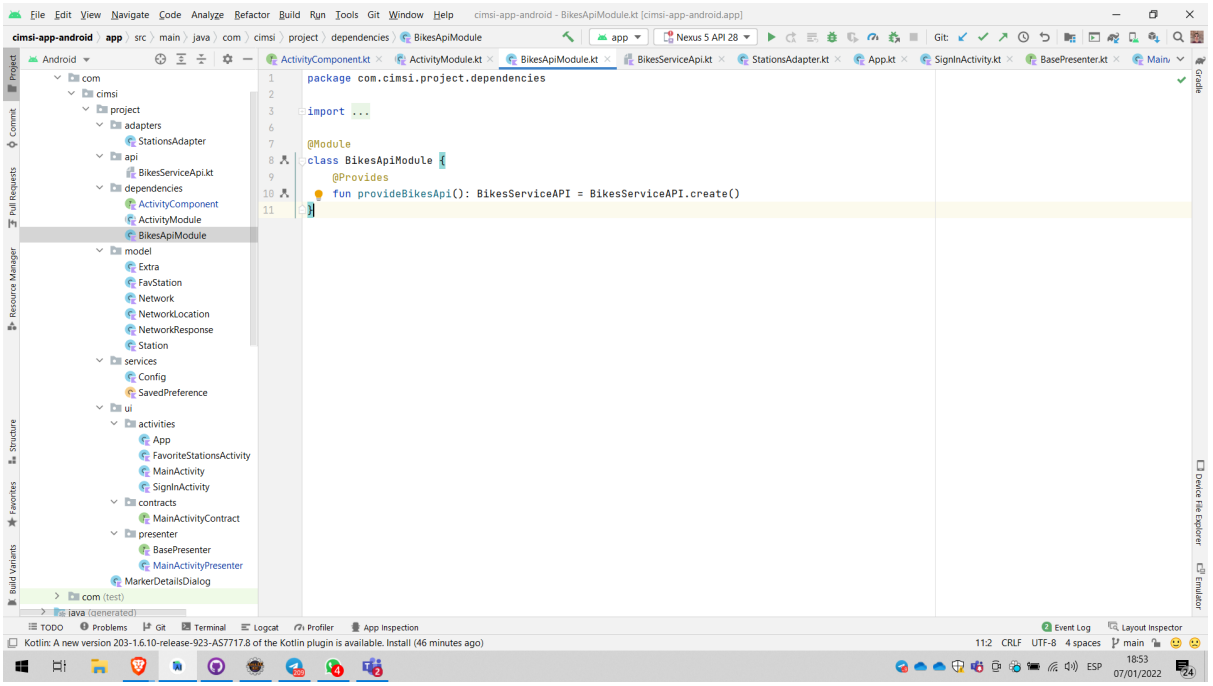
· **ActivityComponent:** nos permite crear las distintas inyecciones de dependencias (en nuestro caso crearemos tres: MainActivity, SignInActivity y FavoriteStationsActivity).



· **ActivityModule:** nos permite crear los binder de los presentadores (es decir, crear la función que nos permite generar la función que recibe la independencia). En este caso, vamos a crear un binder para las tres activities (bindPresenter).

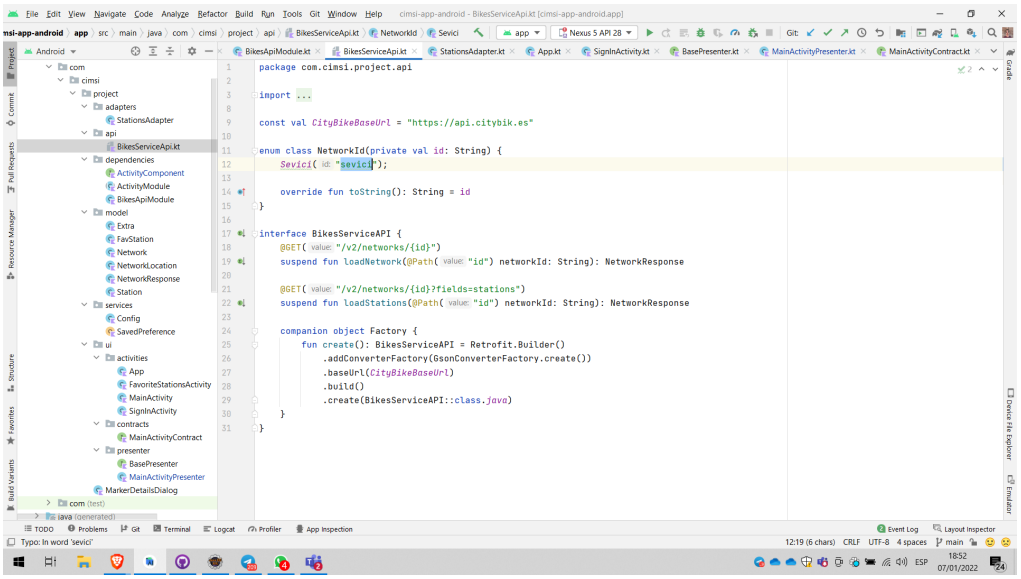


· **BikesApiModule:** posee los métodos que devuelven las distintas APIs. En nuestro caso, tan solo tenemos implementada una API RESTful, concretamente la de CityBikes. Por esa razón, solo poseemos un método (providesBikesApi).



- **API**

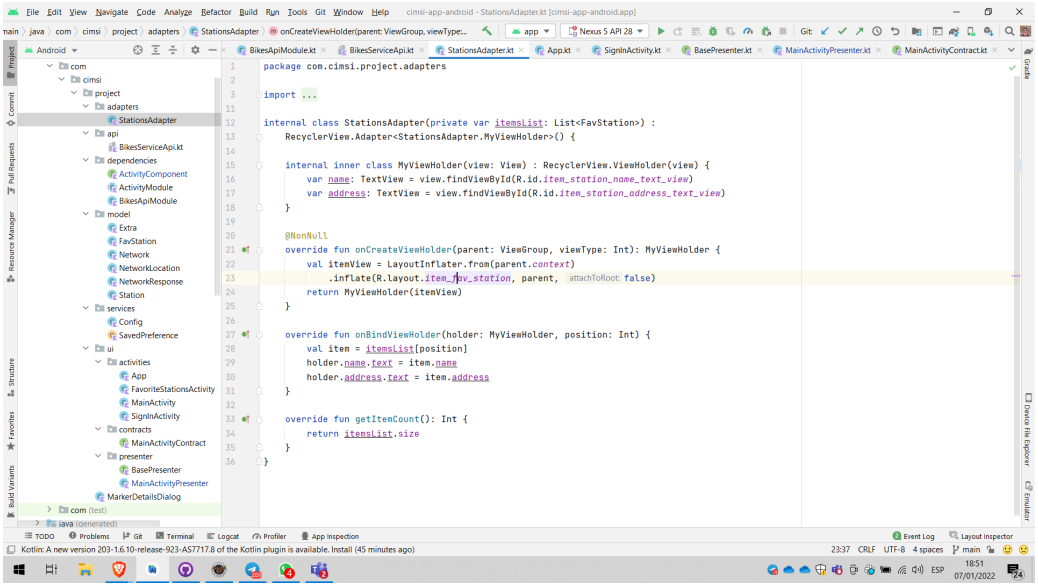
· **BikesServiceApi:** posee métodos, enumerados, variables e interfaces necesarias para la correcta ejecución de un endpoint. Podemos observar, por ejemplo, que poseemos la URL base de la API, los distintos NetworkIds (actualmente solo poseemos Sevici), así como una interfaz (BikesServiceAPI) que posee dos funciones, loadNetwork y loadStations, además de un companion object que hace las veces de factoría (devuelve el objeto Retrofit que nos permite ejecutar la llamada al endpoint).



```
package com.cimsi.project.api

import ...

const val CityBikeBaseUrl = "https://api.citybik.es"

enum class NetworkId(private val id: String) {
    Sevici( id: "sevici");

    override fun toString(): String = id
}

interface BikesServiceAPI {
    @GET( value: "/v2/networks/{id}")
    suspend fun loadNetwork(@Path( value: "id") networkId: String): NetworkResponse

    @GET( value: "/v2/networks/{id}?fields=stations")
    suspend fun loadStations(@Path( value: "id") networkId: String): NetworkResponse

    companion object Factory {
        fun create(): BikesServiceAPI = Retrofit.Builder()
            .addConverterFactory(GsonConverterFactory.create())
            .baseUrl(CityBikeBaseUrl)
            .build()
            .create(BikesServiceAPI::class.java)
    }
}
```

- **Adaptadores**

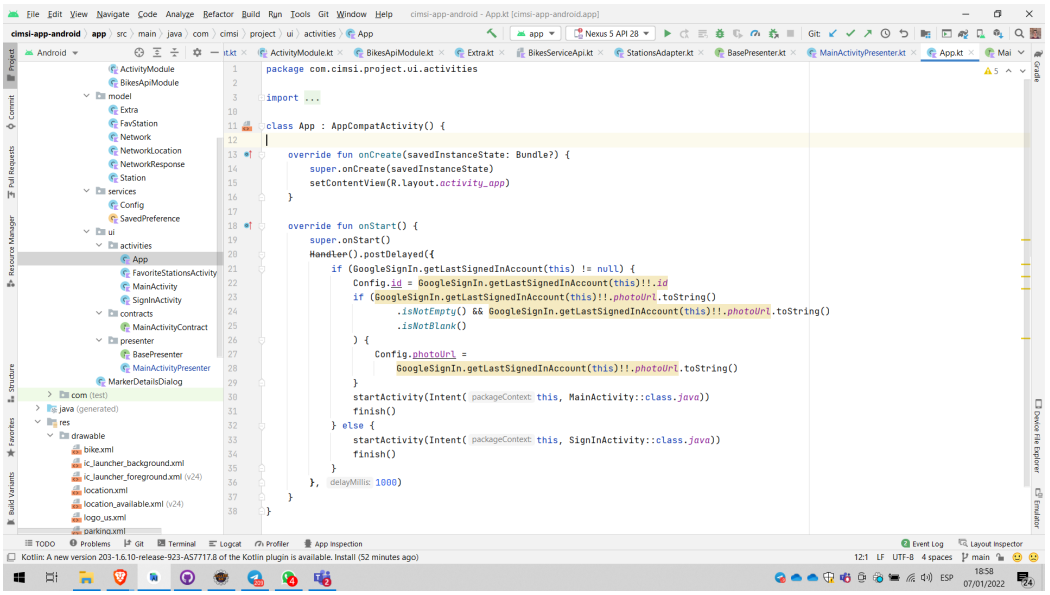
· **StationsAdapter:** inicializa el adaptador que populará de estaciones la lista de estaciones marcadas como favoritas del usuario. Es decir, este adaptador inicializa los elementos de la interfaz gráfica de cada ítem de la lista, y los inicializa con los datos de la lista estaciones (por cada estación, un ítem).



```
package com.cimsi.project.adapters

import ...

internal class StationsAdapter(private var itemsList: List<FavStation>) :
    RecyclerView.Adapter<StationsAdapter.MyViewHolder>() {

    internal inner class MyViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        var name: TextView = view.findViewById(R.id.item_station_name_text_view)
        var address: TextView = view.findViewById(R.id.item_station_address_text_view)
    }

    @NonNull
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
        val itemView = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_fav_station, parent,  attachToRoot: false)
        return MyViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        val item = itemsList[position]
        holder.name.text = item.name
        holder.address.text = item.address
    }

    override fun getItemCount(): Int {
        return itemsList.size
    }
}
```

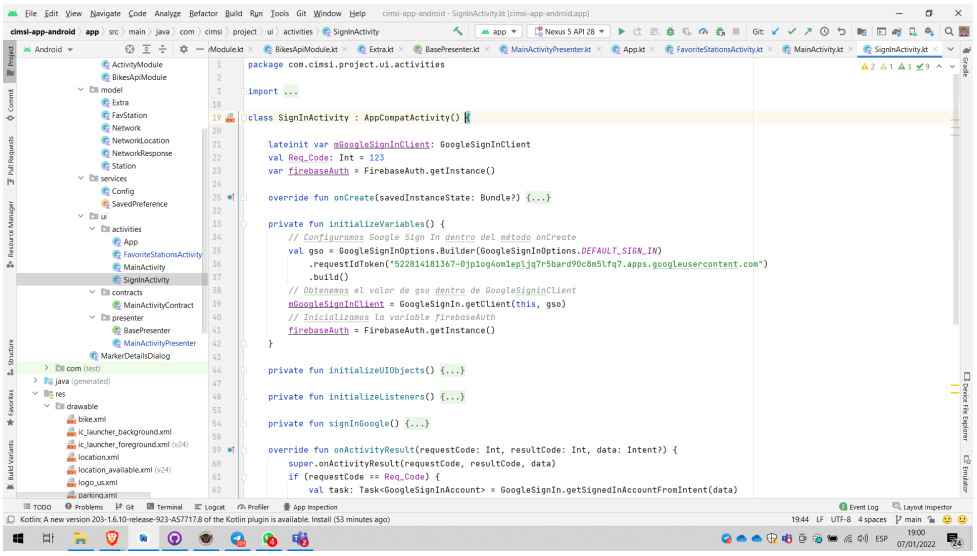
- **Activities**

· **App:** esta actividad posee una funcionalidad principal: inicializar la autenticación con Google. Para ello, mientras la aplicación comprueba si el usuario ha iniciado sesión o no en la app con su cuenta, se muestra una pantalla en blanco, con el logo de la Universidad de Sevilla en el centro. En caso de que el usuario haya iniciado sesión anteriormente, nos redirigirá a la pantalla principal de la app. En caso contrario, nos redirigirá a la pantalla de inicio de sesión

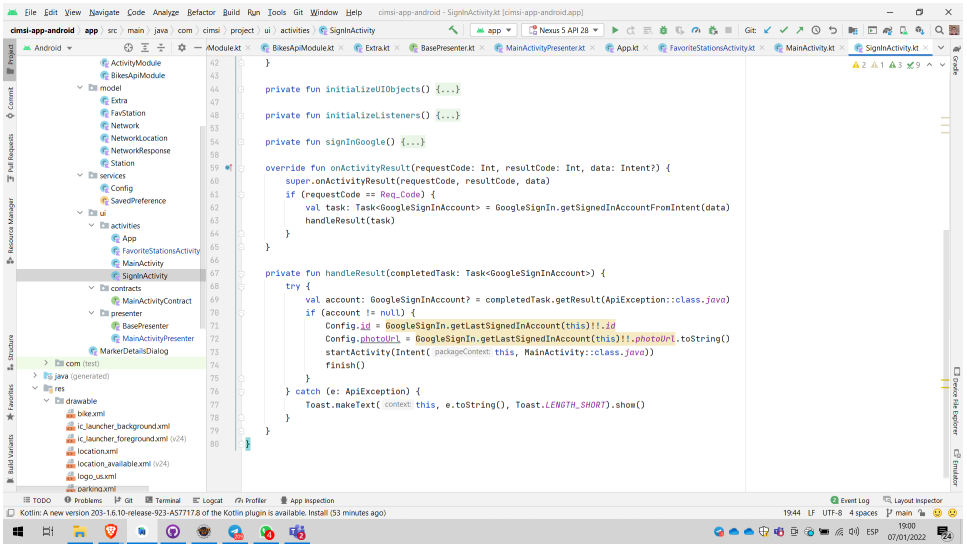


```
package com.cimsi.project.ui.activities

import ...

class App : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_app)
    }

    override fun onStart() {
        super.onStart()
        Handler().postDelayed({
            if (GoogleSignIn.getLastSignedInAccount(this) != null) {
                Config.id = GoogleSignIn.getLastSignedInAccount(this)!!.id
                if (GoogleSignIn.getLastSignedInAccount(this)!!.photoUrl.toString()
                        .isNotEmpty() && GoogleSignIn.getLastSignedInAccount(this)!!.photoUrl.toString()
                        .isNotBlank()
                ) {
                    Config.photoUrl =
                        GoogleSignIn.getLastSignedInAccount(this)!!.photoUrl.toString()
                }
                startActivity(Intent( packageContext: this, MainActivity::class.java))
                finish()
            } else {
                startActivity(Intent( packageContext: this, SignInActivity::class.java))
                finish()
            }
        }, delayMillis: 1000)
    }
}
```

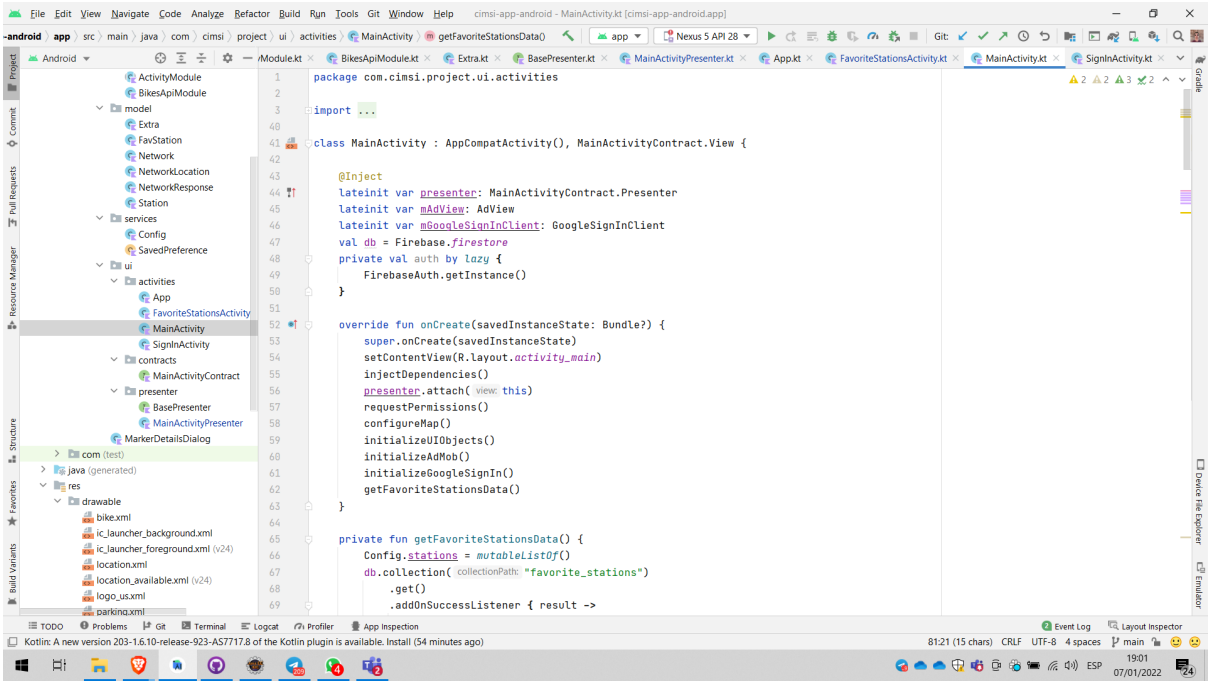
· **SignInActivity:** esta actividad nos permitirá, una vez inicializado el servicio de autenticación, hacer click en un botón, en el cual al hacer click sobre el mismo podremos ver un diálogo con las distintas cuentas de Google configuradas en el dispositivo, con las cuales podremos iniciar sesión. Una vez hemos realizado click sobre una de ellas, la app nos redirigirá a la pantalla principal.

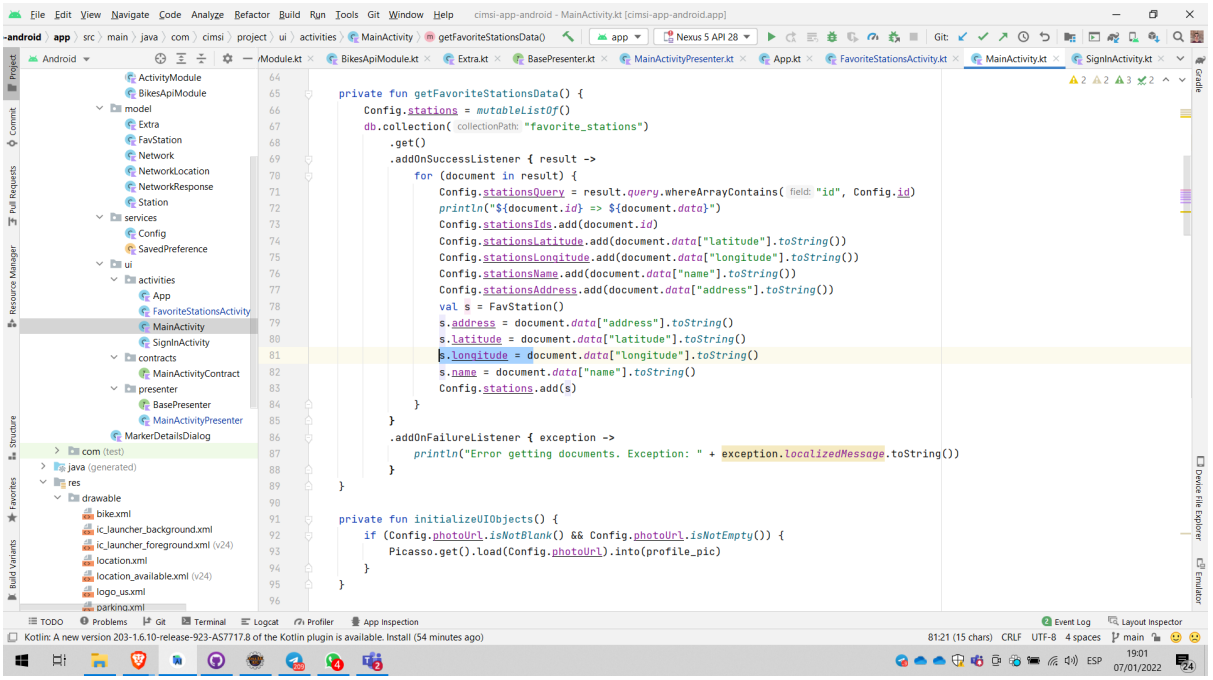


```
package com.cimsi.project.ui.activities

import ...

class SignInActivity : AppCompatActivity() {

    lateinit var mGoogleSignInClient: GoogleSignInClient
    val Req_Code: Int = 123
    var firebaseAuth = FirebaseAuth.getInstance()

    override fun onCreate(savedInstanceState: Bundle?) {...}

    private fun initializeVariables() {
        // Configuramos Google Sign In dentro del método onCreate
        val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
            .requestIdToken("522814181367-0jp1og4om1epljq7r5bard90c8m5lfq7.apps.googleusercontent.com")
            .build()
        // Obtenemos el valor de gso dentro de GoogleSignInClient
        mGoogleSignInClient = GoogleSignIn.getClient(this, gso)
        // Inicializamos la variable firebaseAuth
        firebaseAuth = FirebaseAuth.getInstance()
    }

    private fun initializeUIObjects() {...}

    private fun initializeListeners() {...}

    private fun signInGoogle() {...}

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        super.onActivityResult(requestCode, resultCode, data)
        if (requestCode == Req_Code) {
            val task: Task<GoogleSignInAccount> = GoogleSignIn.getSignedInAccountFromIntent(data)
```

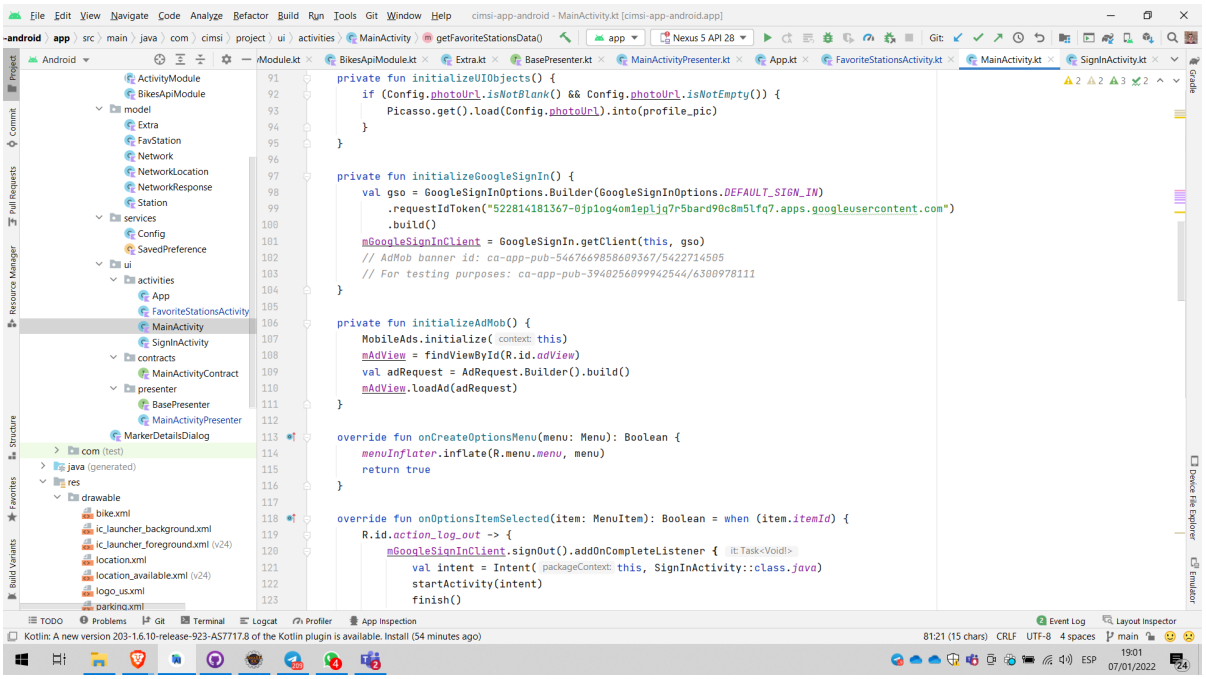


· **MainActivity:** es la pantalla principal de la app. Inicializa OpenStreetMap, lo configura, y popula el mismo con markers obtenidos del endpoint de CityBike. Además, muestra una foto de perfil en la parte superior izquierda de la app. Además, en la barra de navegación superior, si hacemos click en la derecha, podremos acceder a tres menús: GitHub, el cual nos redirige al código fuente de la app; Estaciones favoritas, el cual nos redirige a la lista con las estaciones marcadas como favoritas por nosotros; y cerrar sesión, el cual cierra la sesión de Google y nos redirige a la pantalla de inicio de sesión de la app.

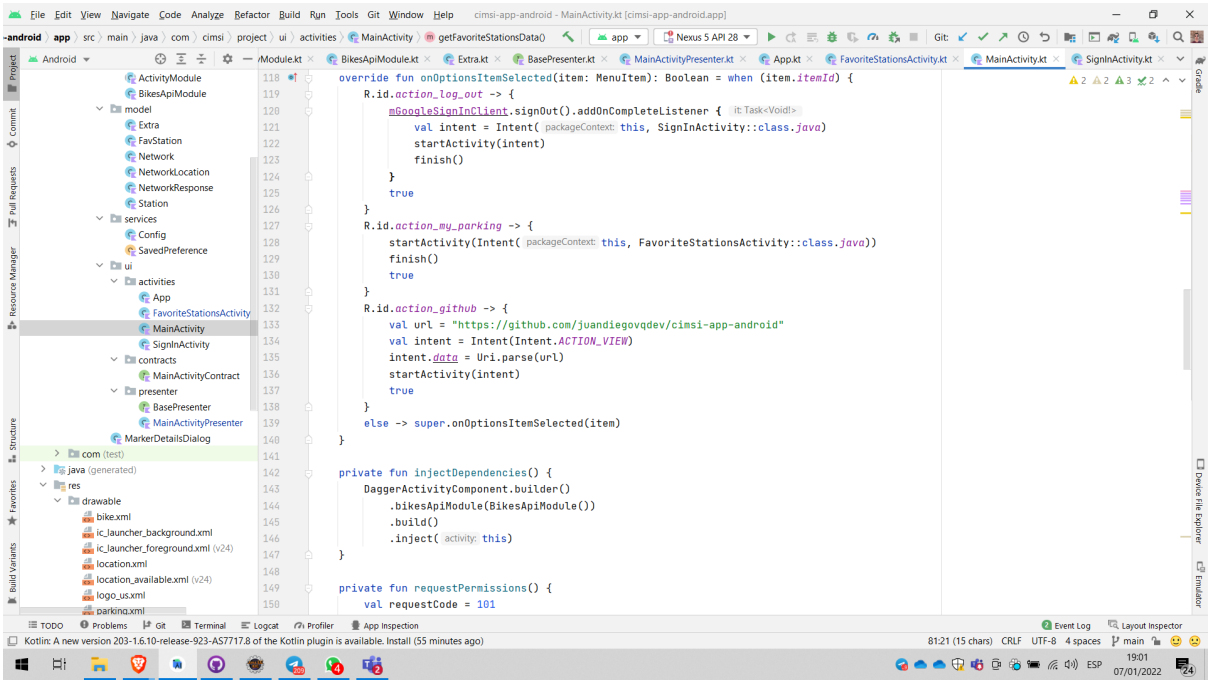




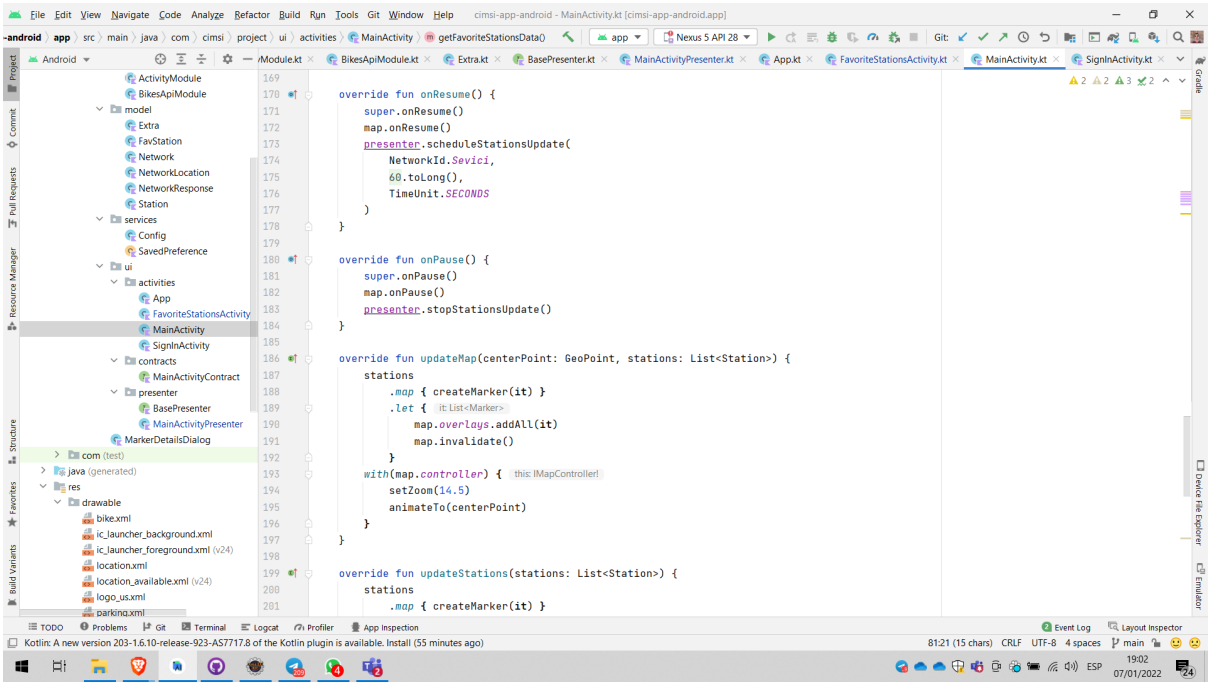
```
private fun getFavoriteStationsData() {
    Config.stations = mutableListOf()
    db.collection( collectionPath: "favorite_stations")
        .get()
        .addOnSuccessListener { result ->
            for (document in result) {
                Config.stationsQuery = result.query.whereArrayContains( field: "id", Config.id)
                println("${document.id} => ${document.data}")
                Config.stationsIds.add(document.id)
                Config.stationsLatitude.add(document.data["latitude"].toString())
                Config.stationsLongitude.add(document.data["longitude"].toString())
                Config.stationsName.add(document.data["name"].toString())
                Config.stationsAddress.add(document.data["address"].toString())
                val s = FavStation()
                s.address = document.data["address"].toString()
                s.latitude = document.data["latitude"].toString()
                s.longitude = document.data["longitude"].toString()
                s.name = document.data["name"].toString()
                Config.stations.add(s)
            }
        }
        .addOnFailureListener { exception ->
            println("Error getting documents. Exception: " + exception.localizedMessage.toString())
        }
}

private fun initializeUIObjects() {
    if (Config.photoUrl.isNotBlank() && Config.photoUrl.isNotEmpty()) {
        Picasso.get().load(Config.photoUrl).into(profile_pic)
    }
}
```



```
private fun initializeUIObjects() {
    if (Config.photoUrl.isNotBlank() && Config.photoUrl.isNotEmpty()) {
        Picasso.get().load(Config.photoUrl).into(profile_pic)
    }
}

private fun initializeGoogleSignIn() {
    val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken("522814181367-0jp1og4om1eplqj7r5bard90c8m5lfq7.apps.googleusercontent.com")
        .build()
    mGoogleSignInClient = GoogleSignIn.getClient(this, gso)
    // AdMob banner id: ca-app-pub-5467669858609367/5422714505
    // For testing purposes: ca-app-pub-3940256099942544/6300978111
}

private fun initializeAdMob() {
    MobileAds.initialize( context: this)
    mAdView = findViewById(R.id.adView)
    val adRequest = AdRequest.Builder().build()
    mAdView.loadAd(adRequest)
}

override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.menu, menu)
    return true
}

override fun onOptionsItemSelected(item: MenuItem): Boolean = when (item.itemId) {
    R.id.action_log_out -> {
        mGoogleSignInClient.signOut().addOnCompleteListener { it: Task<Void!>
            val intent = Intent( packageContext: this, SignInActivity::class.java)
            startActivity(intent)
            finish()
```

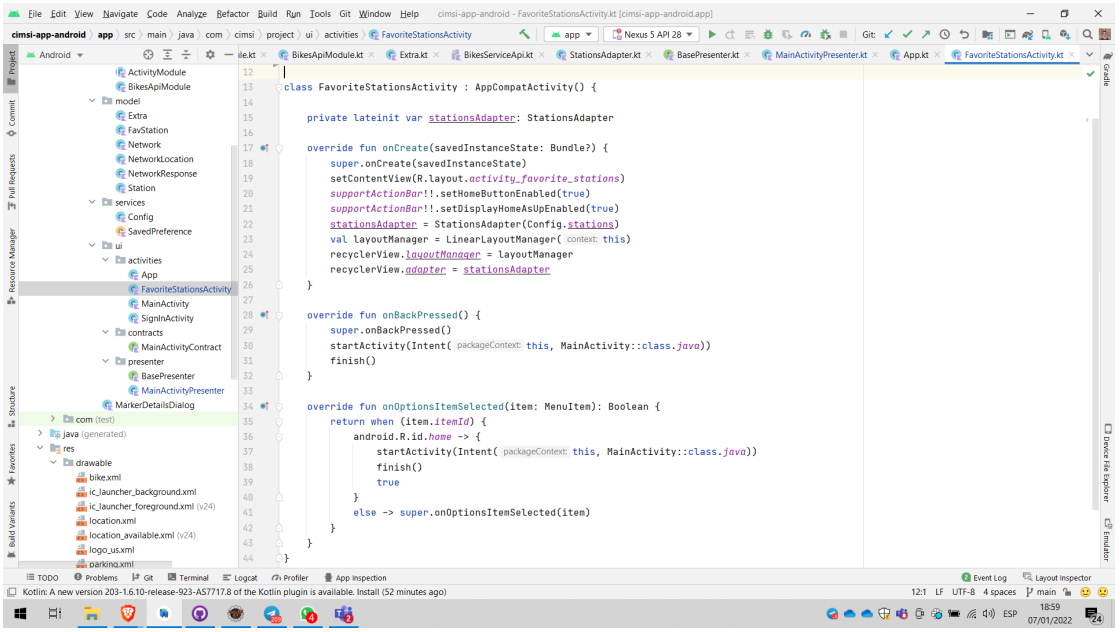


```
override fun onOptionsItemSelected(item: MenuItem): Boolean = when (item.itemId) {
    R.id.action_log_out -> {
        mGoogleSignInClient.signOut().addOnCompleteListener {
            val intent = Intent( packageContext: this, SignInActivity::class.java)
            startActivity(intent)
            finish()
        }
        true
    }
    R.id.action_my_parking -> {
        startActivity(Intent( packageContext: this, FavoriteStationsActivity::class.java))
        finish()
        true
    }
    R.id.action_github -> {
        val url = "https://github.com/juandiegovqdev/cimsi-app-android"
        val intent = Intent(Intent.ACTION_VIEW)
        intent.data = Uri.parse(url)
        startActivity(intent)
        true
    }
    else -> super.onOptionsItemSelected(item)
}

private fun injectDependencies() {
    DaggerActivityComponent.builder()
        .bikesApiModule(BikesApiModule())
        .build()
        .inject( activity: this)
}

private fun requestPermissions() {
    val requestCode = 101
```



```
override fun onResume() {
    super.onResume()
    map.onResume()
    presenter.scheduleStationsUpdate(
        NetworkId.Sevici,
        60.toLong(),
        TimeUnit.SECONDS
    )
}

override fun onPause() {
    super.onPause()
    map.onPause()
    presenter.stopStationsUpdate()
}

override fun updateMap(centerPoint: GeoPoint, stations: List<Station>) {
    stations
        .map { createMarker(it) }
        .let {
            map.overlays.addAll(it)
            map.invalidate()
        }
    with(map.controller) {
        setZoom(14.5)
        animateTo(centerPoint)
    }
}

override fun updateStations(stations: List<Station>) {
    stations
        .map { createMarker(it) }
```

· **FavoriteStationsActivity:** posee una lista con todas las estaciones marcadas como favoritas por el usuario. Hacemos uso de la variable global Config.stations, así como del adaptador StationsAdapter, para rellenar la lista.

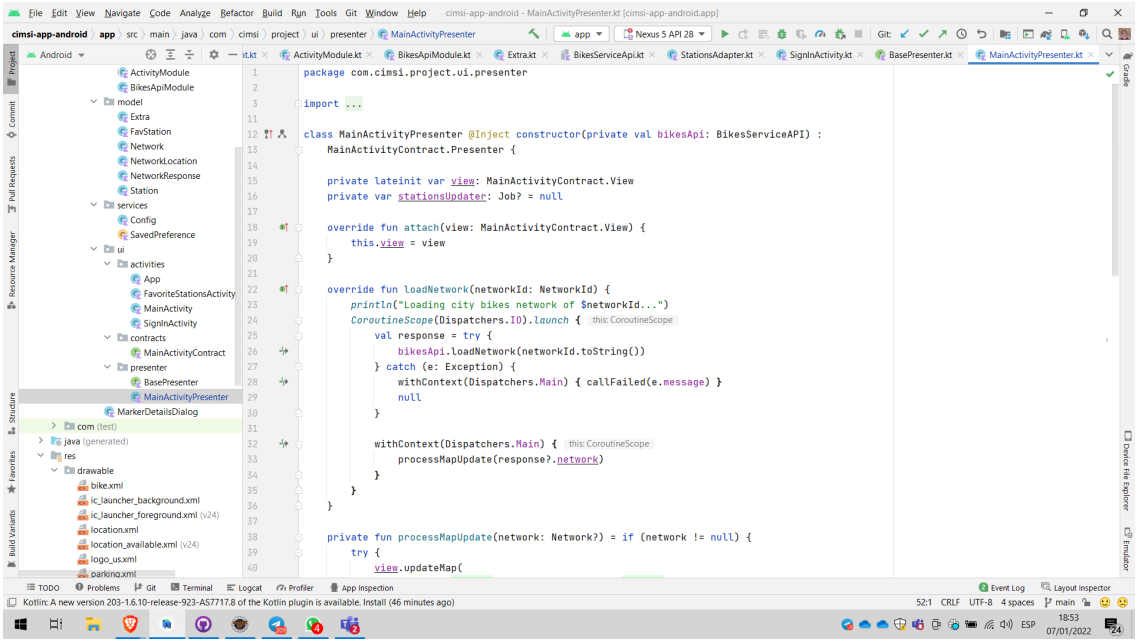


```
class FavoriteStationsActivity : AppCompatActivity() {

    private lateinit var stationsAdapter: StationsAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_favorite_stations)
        supportActionBar!!.setHomeButtonEnabled(true)
        supportActionBar!!.setDisplayHomeAsUpEnabled(true)
        stationsAdapter = StationsAdapter(Config.stations)
        val layoutManager = LinearLayoutManager( context: this)
        recyclerView.layoutManager = layoutManager
        recyclerView.adapter = stationsAdapter
    }

    override fun onBackPressed() {
        super.onBackPressed()
        startActivity(Intent( packageContext: this, MainActivity::class.java))
        finish()
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            android.R.id.home -> {
                startActivity(Intent( packageContext: this, MainActivity::class.java))
                finish()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
}
```

- **Presentadores**

· **BasePresenter:** inicializa la interfaz del presentador base.

· **MainActivityPresenter:** posee métodos los cuales usarán nuestras activities. Entre estos métodos, podemos encontrar el método loadNetwork (que ejecuta un servicio de bokesharing); processMapUpdate (actualiza los markers del mapa cada 60 segundos), etc…



```
package com.cimsi.project.ui.presenter

import ...

class MainActivityPresenter @Inject constructor(private val bikesApi: BikesServiceAPI) :
    MainActivityContract.Presenter {

    private lateinit var view: MainActivityContract.View
    private var stationsUpdater: Job? = null

    override fun attach(view: MainActivityContract.View) {
        this.view = view
    }

    override fun loadNetwork(networkId: NetworkId) {
        println("Loading city bikes network of $networkId...")
        CoroutineScope(Dispatchers.IO).launch {  this: CoroutineScope
            val response = try {
                bikesApi.loadNetwork(networkId.toString())
            } catch (e: Exception) {
                withContext(Dispatchers.Main) { callFailed(e.message) }
                null
            }

            withContext(Dispatchers.Main) {  this: CoroutineScope
                processMapUpdate(response?.network)
            }
        }
    }

    private fun processMapUpdate(network: Network?) = if (network != null) {
        try {
            view.updateMap(
```

- **Dialogs**

· **MarkerDetailsDialog:** inicializa el diálogo que aparece al hacer click en una estación. Inicializa el diseño, y asigna los valores a las variables (nombre, slots vacíos, bicis disponibles…).

- **Contracts**

· **MainActivtiyContract:** no es más que una interfaz la cual contiene los métodos a implementar por el presentador.

5. CARACTERÍSTICAS DE ALTA DISPONIBILIDAD

Una de las razones por las que hemos implementado Firestore es para aumentar la disponibilidad del servicio. Imaginemos por un momento que queremos consultar los últimos datos disponibles sobre una estación. En caso de que la API no se encuentre disponible en el mismo momento, podemos obtener desde Firebase Firestore todas las estaciones que hemos añadido como favoritas, y acceder a información como puede ser el nombre, dirección, latitud o longitud.

6. COMO ESCALAR EL SERVICIO

Actualmente, dada la arquitectura aplicada (MVVM - Model-View-ViewModel) el servicio es 100% escalable, debido a que la inyección de dependencias nos permite implementar más llamadas a la API, usando el código ya implementado. Es decir, en nuestro caso, *MainActivity* (que no es más que la clase que implementa la librería OpenStreetMap y lo muestra al usuario) simplemente tomaría de una variable global (una lista de POJOs, para ser más concretos) las distintas estaciones obtenidas desde la API, sin adentrarse en “cómo” se obtienen esas estaciones. Por tanto, para *MainActivity* no existe una forma de obtener estos datos, simplemente usa los que obtiene una clase Utilidades, y más tarde inicializa en esta variable global. Esta clase “utilidades” si está modularizada, y puede obtener información acerca de todas las estaciones disponibles en CityBike.

Esta clase utilidades, por otra parte, permite la inicialización de estaciones las cuales se proveen desde la API: por esta sencilla razón, cualquier cambio realizado en la API de CityBike, como puede ser la adición o eliminación de estaciones. De esta forma, no existe inconsistencia de datos entre los alojados en la API, y los que la app permite al usuario ejecutar (menos excepciones, y mayor desacople de las funcionalidades de la app, apoyándose en la API para ejecutarlas).

7. MANTENIMIENTO

A la hora de tratar el mantenimiento de una app móvil, tenemos que tener en cuenta varios aspectos. Las apps son sometidas a numerosas actualizaciones, ya sean propias del SDK de Android, de la versión del lenguaje de programación usado (Kotlin, Java, u otros si optamos por un framework multiplataforma), de Gradle o Maven (gestores de paquetes de Android), o de librerías de terceros que incluyen nuevas funcionalidades en su software.

Realmente, la gran carga de trabajo durante el mantenimiento de esta aplicación móvil va a ser mantener actualizado el SDK de Android a la última versión, así como las librerías de terceros, entre las que se encuentran las siguientes dependencias Gradle:

8. TRABAJO FUTURO

Como trabajo futuro, nos hemos propuesto mejorar el patrón de diseño (la arquitectura) haciendo uso de programación reactiva (concretamente RxKotlin - ReactiveKotlin), para poder ofrecer así un proyecto de ejemplo acerca de cómo usar programación reactiva en Android, mostrando además cómo se debe de usar diversas librerías como puede ser OpenStreetMap for Android. En definitiva, queremos alojar un proyecto en GitHub que sirva como ejemplo de buenas prácticas en el desarrollo de apps Android.

Otro de los objetivos de la aplicación es modularizar el uso de la API. Es decir, actualmente solo está disponible para mostrar datos sobre Sevici, y el objetivo marcado en el próximo trabajo es modularizar la carga de estaciones mediante la ubicación del dispositivo. De esta forma, dependiendo de la ubicación del dispositivo, se ejecutarán las llamadas a la API haciendo uso del servicio disponible más cercano.

9. CÓDIGO FUENTE DEL PROYECTO

El código fuente del proyecto puede ser encontrado en GitHub, concretamente en el siguiente link: https://github.com/juandiegovqdev/cimsi-app-android.

10. BIBLIOGRAFÍA / REFERENCIAS

- https://en.wikipedia.org/wiki/Android_Studio
- https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93modelo_de_vista#Historia
- https://en.wikipedia.org/wiki/Mockito
- https://dagger.dev/
- https://es.wikipedia.org/wiki/Firebase
- https://circleci.com/
- https://es.wikipedia.org/wiki/Google_Analytics
- https://www.atlassian.com/es/git/tutorials/what-is-git

Como información adicional, a continuación compartimos diversos cursos realizados a lo largo del desarrollo del proyecto, de cara a aprender nuevas técnicas para desarrollar el proyecto:

- Curso Android (2h 24m): https://www.youtube.com/watch?v=ebQphhLpJG0
- Firebase Js (53m): https://www.youtube.com/watch?v=Nb7NGjyx1eo
- MongoDB (1h 21m): https://www.youtube.com/watch?v=lWMemPN9t6Q
- AWS Amplify (1h): https://aws.amazon.com/es/getting-started/hands-on/build-android-app-amplify/module-one/