
Trabajo de Fin de Curso

Desarrollo de una aplicación basada en Scrum con nuevas tecnologías

Jordi Ortegón Rivera

jatz

2019/05/27

Índice

Resumen Documental	3
Resumen	3
Abstract	3
Introducción	4
¿Qué son la metodologías ágiles?	4
¿Qué es Scrum?	4
¿Por qué Scrum?	5
¿Por qué Android?	5
Objetivo	5
Objetivo general	5
Objetivos parciales	5
Desarrollo	6
Solución tecnológica	6
Lenguaje utilizado	6
Herramientas utilizadas	7
Funcionamiento de la aplicación	7
Diseño de Base de datos	38
Esquema de Diálogo	38
Inicio de Sesión	39
Creación Cuenta	40
Tableros	41
Notas	43
Conclusiones	45
Mejoras	45
Bibliografía	46

Resumen Documental

Resumen

En este documento se presenta el trabajo de investigación y desarrollo para la creación de una aplicación en **Android**, en el lenguaje de programación **Kotlin**, con una base de datos proporcionada por **Parse Server** y basada en la metodología ágil **Scrum**.

Se realizará una pequeña introducción del proyecto en la cuál se describirán las elecciones tomadas para su desarrollo, explicando la metodología, el ámbito en el que se ha realizado, entre otras cosas. A continuación se dará un repaso sobre el objetivo general y sus objetivos parciales, seguido por la solución tecnológica ofrecida para la realización del proyecto. Nos encontraremos también con un esquema de funcionamiento de la misma y por último se reflejarán las conclusiones y posibles mejoras para un versión avanzada de la aplicación/programa.

Abstract

This document presents the research and development work for the creation of an application in **Android**, developed in **Kotlin**, with a **Parse Server** backend and based on **Scrum**, an agile methodology.

There will be a brief introduction of the project in which the choices made for its development will be described, explaining the methodology, the scope in which it has been carried out, among other things. Afterwards is a review of the general objective and its partial objectives, followed by the technological solution offered for the realization of the project. We will also find a scheme of operation of the application and finally reflect the conclusions and possible improvements for a future version of the application.

Introducción

El proyecto **Jatz** nació de mi afán por procurar la organización en ámbitos generales para de esta forma, obtener un mejor conocimiento de las tareas y su estado. Esta idea llegó a mí tras el aprendizaje, por parte de mis profesores y por distintas áreas de trabajo, de las metodologías ágiles, más concretamente, **Scrum**.

A través de esto surgió la idea de crear una aplicación para ayudar a la organización diaria y personal. Tomando en cuenta el ya conocido auge y mantenimiento de los dispositivos móviles se decidió construir el proyecto en una tecnología que permitiese la facilidad de acceso y la mayor globalidad posible.

¿Qué son la metodologías ágiles?

Podemos definir las metodologías ágiles como un conjunto de pautas de trabajo flexibles y adaptables, diferenciándose así de los antiguos métodos estrictos, sin eliminar su eficacia y permitiendo el ahorro de recursos para cada tarea.

¿Qué es Scrum?

Como ya se ha comentado forma parte de las presentes metodologías ágiles pero entrando en profundidad, podemos decir que **Scrum** se basa en una serie de ciclos continuos de corto tiempo conocidos como *Sprint*, para conseguir de esta forma la finalización de la tarea o producto. Se busca que gracias a la división por etapas se pueda: encontrar los posibles fallos o mejoras, tener un conocimiento del tiempo estimado, división de las tareas y prevención de cuellos de botella. Además entre otras características por las cuales se puede diferenciar a **Scrum** de otras metodologías tenemos:

- **Ligereza:** es una metodología que se basa en el alcance global y esto lo permite gracias a que no necesita una infraestructura compleja como por ejemplo, *Kanban* que aunque reduce el tiempo también reduce el orden.
- **Entendible:** muy fácil de entender y aplicar. No se necesitan grandes conocimientos para comprender la estructura básica y los ciclos de vida.
- **Flexibilidad:** es aplicable tanto en grandes grupos como de forma individual. De ambas formas el resultado es totalmente independiente.

¿Por qué Scrum?

Jatz se basa en **Scrum** por su facilidad de comprensión, ya que no exige un cambio brusco en la organización normal de las tareas del día a día, o proyectos, de una persona de a pie. También se apoya en dicha metodología debido a su escalabilidad, es decir, permite ser aplicada no solo a proyectos grupales sino que también de forma individual, la cuál a sido la idea principal de **Jatz**. Además cabe destacar que es conocida por una gran grupo de personas en todo tipo de áreas.

¿Por qué Android?

En la actualidad el sistema operativo para tabletas que domina el mercado es sin duda alguna **Android**, debido a la mayor popularidad, facilidad de acceso y pre-conocimiento del desarrollo en esta plataforma, se ha considerado acertado realizar el proyecto para **Android**.

Objetivo

Objetivo general

El objetivo principal de este proyecto es la realización de una versión funcional de la aplicación **Jatz** para dispositivos móviles con sistema operativo **Android**, para así ofrecer al mayor número de personas posible el acceso a la metodología **Scrum** de forma simple sin perder la esencia de la misma.

Objetivos parciales

En base al objetivo principal del desarrollo de la aplicación podemos encontrar que es divisible en pequeños objetivos parciales, los cuales se listan a continuación.

- Análisis de requisitos para la creación de la aplicación.
- Elección de herramientas de desarrollo.
- Estudio de la metodología Scrum.
- Adquisición de conocimientos sobre el sistema en el que se realiza el desarrollo y lenguajes a utilizar.
- Control de versiones y creación de prototipos.

Desarrollo

En este apartado se elaborará una descripción detallada del desarrollo realizado con el fin de cumplir el objetivo principal de este proyecto. Se empezará describiendo tanto el lenguaje como las herramientas usadas en el desarrollo, seguido de una explicación breve del funcionamiento de la aplicación y finalmente, el diseño de la base de datos.

Solución tecnológica

Lenguaje utilizado

Actualmente y desde hace ya unos años el desarrollo en **Android** ya no esta completamente ligado a un lenguaje particular, ni a una herramienta concreta. Mi elección para este trabajo a sido **Kotlin**.

¿Qué es Kotlin?

Kotlin es un lenguaje de programación desarrollado en su mayoría por la empresa de software **Jet-Brains**. Es un lenguaje de tipado estático que trabaja sobre la máquina virtual de Java y que también puede ser compilado a código fuente de JavaScript. Ya que corre sobre la máquina virtual de Java es inevitable compararlo con el mismo, por esto serán señaladas la principales diferencias entre ambos a continuación:

Kotlin	Java
Expresiones Lambda	Excepciones comprobadas
Funciones de extensión	Tipos primitivos
Seguridad de nulos	Miembros estáticos
Llamada inteligente	Campos no privados
Singletons	Tipos comodín
Expresiones de rango	Operador ternario $a ? b : c$

Destacar que **Kotlin** ofrece muchas más funcionalidades que **Java** no. Solo se han anotado las más destacables.

Herramientas utilizadas

Aquí se presentarán las diferentes herramientas utilizadas para la elaboración del proyecto, desde las herramientas de software a diferentes fuentes de datos (webs, libros) para la consulta y aprendizaje.

Software

- Entorno de desarrollo *Android Studio* para llevar a cabo el desarrollo de la aplicación
- Backend multiplataforma proporcionado por Parse Server
- Herramienta de control de versiones Git y Github.
- Emulador de Android Genymotion para el lanzamiento y depuración de la aplicación

Fuentes de conocimiento

- Diferentes repositorios de la antes mencionada Github
- Kotlin for Android Developers escrito por Antonio Leiva
- Documentación para android proporcionada por **Parse Server**
- Documentación proporcionada por **Alphabet** para la implementación de diferentes componentes y diseño
- Resolución de excepciones por diferentes hilos de Stackoverflow

Funcionamiento de la aplicación

Definiendo de forma general la aplicación podemos decir que es sencilla y ágil de utilizar, pese a esto se expresará el funcionamiento en grandes rasgos.

El objetivo de la aplicación es crear una cuenta o iniciar sesión si ya ha sido creada para luego llevar al usuario a la página de tableros donde podrá crear uno o varios. Desde allí podrá acceder al tablero, ahí se le mostrarán las notas en los diferentes estados, además le será posible crear notas, editarlas y cambiarlas de estados.

Desarrollo

Pasando a la explicación programática del proyecto hay que definir previamente el uso del patrón *MVP*.

¿Qué es el modelo MVP?

Es un derivado del conocido **MVC** (Controlador de vista del modelo), y uno de los patrones más populares para organizar la capa de presentación en aplicaciones de Android. De la misma forma se divide en tres capas, las cuales son:

- Modelo: se considera la puerta de entrada a la capa de dominio o lógica de negocio.
- Vista: suele estar implementada por una actividad. Contiene cualquier parte de interacción con el usuario.
- Presentador: es el responsable de realizar las acciones entre la vista y el modelo. Un ejemplo claro es un adaptador

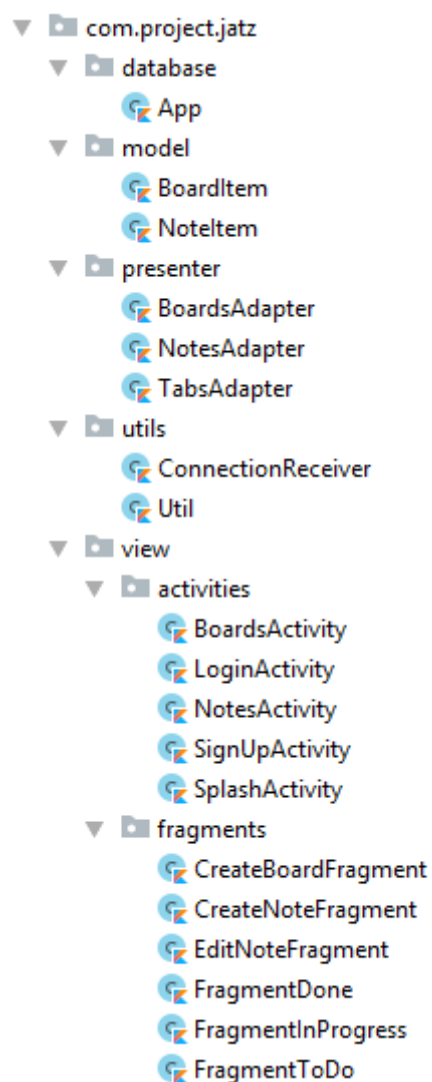


Figura 1: Presentación del modelo completo del proyecto (Sin desarrollo de la parte gráfica)

Database

```
/**
 * Class that contains Parse initialization
 */
class App: Application() {

    companion object {
        @get:Synchronized
        lateinit var instance: App
    }

    override fun onCreate() {
        super.onCreate()
        ParseObject.registerSubclass(BoardItem::class.java)
        ParseObject.registerSubclass(NoteItem::class.java)
        Parse.initialize(
            Parse.Configuration.Builder(context: this)
                .applicationId("WpK0tvBCi90tvdNFO2t5QF0gepOOj7jNLGHmNFyY")
                .clientId("sOZdZqL8cnZaJ4bb3bhE9RmnOSU4VQ3tkOrlRhMC")
                .server("https://parseapi.back4app.com/")
                .build()
        )

        instance = this

        ParseInstallation.getCurrentInstallation().saveInBackground()
    }

    fun setConnectionListener(listener: ConnectionReceiver.ConnectionReceiverListener) {
        ConnectionReceiver.connectionReceiverListener = listener
    }
}
```

Figura 2: En el sub-paquete *database* nos encontramos con la clase *App*. Esta clase contiene la inicialización de Parse, necesaria para la posterior conexión por parte de otras clases. también se puede ver la función *setConnectionListener* la cual se encarga de establecer un escuchador para el resto de clases o actividades, llamado por la clase *ConnectionReceiver* del sub-paquete *utils*.

Utils

En el sub-paquete *utils* nos encontraremos con las clases *Util* y *ConnectionReceiver*

```
class ConnectionReceiver: BroadcastReceiver() {

    companion object{
        var connectionReceiverListener: ConnectionReceiverListener? = null

        val isConnected: Boolean
        get(){
            val cm = App.instance.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
            val activeNetwork = cm.activeNetworkInfo

            return (activeNetwork != null && activeNetwork.isConnected)
        }
    }

    /**
     * Interface sheltering function that gets the connection change
     */
    interface ConnectionReceiverListener{
        fun onNetworkConnectionChanged(isConnected: Boolean)
    }

    override fun onReceive(context: Context, intent: Intent?) {
        val isConnected = checkConnection(context)

        if(connectionReceiverListener != null){
            connectionReceiverListener!!.onNetworkConnectionChanged(isConnected)
        }
    }

    private fun checkConnection(context: Context): Boolean {
        val cm = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
        val activeNetwork = cm.activeNetworkInfo

        return (activeNetwork != null && activeNetwork.isConnected)
    }
}
```

Figura 3: La clase *ConnectionReceiver* se encarga de la escucha de la conexión a la red. Esto permite crear diferentes métodos en las actividades para tener constancia de la conexión y actuar en consecuencia.

```
class Util{  
    companion object{  
        fun showToast(context: Context?, message: String?) {  
            Toast.makeText(context, message, Toast.LENGTH_SHORT).show()  
        }  
    }  
}
```

Figura 4: La clase `Util` contiene una función estática sencilla que permite la creación y muestra de un `toast`

Model

En este sub-paquete nos encontraremos las clases `BoardItem` y `NoteItem`, las cuales referencian directamente a tablas de la base de datos. Contienen anotaciones proporcionadas por Parse para ser relacionadas y getters y setters para los diferentes campos o columnas de la base de datos

```
@ParseClassName( value: "BoardItem")  
class BoardItem: ParseObject() {  
  
    /**  
     * This method sets the board's title put() is an inbuilt method in the ParseObject class  
     */  
    fun setTitle(title: String) = put("title", title)  
  
    /**  
     * This method sets the board's user put() is an inbuilt method in the ParseObject class  
     */  
    fun setUser(user: ParseUser) = put("createdBy",user)  
  
    /**  
     * This method returns the note's title getString() is an inbuilt method in the ParseObject class  
     */  
    fun getTitle() = getString( key: "title")  
  
    /**  
     * This method returns the note's details getString() is an inbuilt method in the ParseObject class  
     */  
    fun getUser() = getParseUser( key: "createdBy")  
}
```

```
@ParseClassName( value: "NoteItem")
class NoteItem: ParseObject() {

    fun setTitle(title: String) = put("title", title)

    fun setUser(user: ParseUser) = put("createdBy",user)

    fun setBoard(board: ParseObject) = put("parentBoard",board)

    fun setState(state: String) = put("state", state)

    fun setDescription(description: String) = put("description", description)

    fun setComment(comment: String) = put("comment", comment)

    fun getTitle() = getString( key: "title")

    fun getUser() = getParseUser( key: "createdBy")

    fun getBoard() = getParseObject( key: "parentBoard")

    fun getState() = getString( key: "state")

    fun getDescription() = getString( key: "description")

    fun getComment() = getString( key: "comment")
}
```

Presenter

En este sub-paquete nos encontraremos las clases referentes a los Adapters, los cuales son los encargados de enlazar los datos de una lista a los items de un `RecyclerView`, en este caso a los de los tableros y notas. También tienen contenido un `clickListener` el cual proporciona un método al clicar el item.

```
class BoardsAdapter(private val boardList: ArrayList<BoardItem>,
    val clickListener: (BoardItem) -> Unit = { boardItem : BoardItem ->BoardsActivity.clickedBoard(boardItem)}) :
    RecyclerView.Adapter<BoardsAdapter.BoardViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BoardViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.board_item, parent, attachToRoot: false)
        return BoardViewHolder(view)
    }

    override fun onBindViewHolder(holder: BoardViewHolder, position: Int) {
        val currentItem: BoardItem = boardList.get(position)

        holder.title.text = currentItem.getTitle()
        holder.bind(boardList[position], clickListener)
    }

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

    class BoardViewHolder(var rowView: View) : RecyclerView.ViewHolder(rowView){
        val title: TextView = rowView.findViewById(R.id.boarditem_textview)

        fun bind(boardItem: BoardItem, listener: (BoardItem) -> Unit) = with(rowView) { this: View
            setOnClickListener { listener(boardItem) }
        }
    }
}
```

Figura 5: Adapter para tableros

```
class NotesAdapter(val itemList: ArrayList<NoteItem>,
    val clickListener: (NoteItem) -> Unit = { noteItem : NoteItem-> NotesActivity.clickedNote(noteItem) }):
    RecyclerView.Adapter<NotesAdapter.NoteViewHolder>(){

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): NoteViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.note_item, parent, attachToRoot: false)

        return NoteViewHolder(view)
    }

    override fun onBindViewHolder(holder: NoteViewHolder, position: Int) {
        val currentItem: NoteItem = itemList.get(position)

        holder.title.text = currentItem.getTitle()
        holder.description.text = currentItem.getDescription()
        holder.bind(itemList[position], clickListener)
    }

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

    fun getNoteAt(position: Int): NoteItem{
        return itemList.get(position)
    }

    class NoteViewHolder (var rowView: View): RecyclerView.ViewHolder(rowView){
        var title: TextView = rowView.findViewById(R.id.noteitem_title_textview)
        var description: TextView = rowView.findViewById(R.id.noteitem_description_textview)

        fun bind (noteItem: NoteItem, listener: (NoteItem) -> Unit) = with(rowView) { this: View
            | setOnClickListener { listener(noteItem) }
        }
    }
}
```

Figura 6: Adapter para notas

```
class TabsAdapter(manager: FragmentManager): FragmentPagerAdapter(manager) {

    val fragmentList: ArrayList<Fragment> = ArrayList()
    val titleList: ArrayList<String> = ArrayList()

    override fun getItem(position: Int): Fragment {
        return fragmentList[position]
    }

    override fun getCount(): Int {
        return titleList.size
    }

    override fun getPageTitle(position: Int): CharSequence? {
        return titleList[position]
    }

    fun addFragment(fragment: Fragment, title: String){
        fragmentList.add(fragment)
        titleList.add(title)
    }
}
```

Figura 7: Al utilizar un `viewPager`, mostrando tres pestañas necesitaremos un adaptador con un método `addFragment` para añadir las instancias de los fragmentos que contendrá la lista de fragmentos. Este será utilizado para la actividad `NotesActivity`

View

En este sub-paquete nos encontraremos las clases derivadas de actividades y fragmentos. A continuación se describirá cada una de ellas y los métodos personales que contiene.

SplashActivity: esta es la primera actividad que se ejecuta al lanzar la aplicación y la que decide hacia cual seguir. Esto lo hace a través de una consulta a la base de datos.

- `launchBoards`: crea un intent de la actividad `BoardsActivity` y lo lanza.

```
private fun launchBoards() {
    val boardsActivityLaunch = Intent(packageContext, BoardsActivity::class.java)
    startActivity(boardsActivityLaunch)
    finish()
}
```

- launchLogin: crea un intent de la actividad LoginActivity y lo lanza.

```
private fun launchLogin(){
    val loginActivityLaunch = Intent( packageContext: this, LoginActivity::class.java)
    startActivity(loginActivityLaunch)
    finish()
}
```

- isNetworkAvailable: comprueba la conexión del dispositivo. Retorna un **boolean** en función de si hay conexión o no.

```
private fun isNetworkAvailable(): Boolean {
    val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE)
    return if (connectivityManager is ConnectivityManager) {
        val networkInfo: NetworkInfo? = connectivityManager.activeNetworkInfo
        networkInfo?.isConnected ?: false
    } else false
}
```

LoginActivity: esta actividad es la encargada del inicio de sesión de la aplicación.

- onNetworkConnectionChanged: cuando la conexión cambia también lo hace el botón, habilitándose o no, además de cambiar su texto.

```
override fun onNetworkConnectionChanged(isConnected: Boolean) {
    if(!isConnected){
        login_button.isEnabled = false
        login_button.text = "NO CONNECTION"
    }else{
        login_button.text = "LOGIN"
        login_button.isEnabled = true
    }
}
```

- onPause: aquí borraremos y registraremos el receptor de conexión tras haberse pausado la actividad. Esto tras haberlo registrado en el metodo onCreate.

```
override fun onPause() {
    super.onPause()
    unregisterReceiver(connectionReceiver)

    baseContext.registerReceiver(connectionReceiver, IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION))
    App.instance.setConnectionListener(this)
}
```


- `validate`: valida que los datos introducidos en los campos sean correctos; de ser así retorna **true**. Es utilizado en conjunto con el método que sigue.

```
private fun validate(): Boolean {
    var valid = true

    val email = login_email_edittext.text.toString()
    val password = login_password_edittext.text.toString()

    if (email.isEmpty() || !android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
        login_email_edittext.setError("Enter a valid email address")
        valid = false
    } else {
        login_email_edittext.setError(null)
    }

    if (password.isEmpty() || password.length < 4 || password.length > 10) {
        login_password_edittext.setError("Between 4 and 10 alphanumeric characters")
        valid = false
    } else {
        login_password_edittext.setError(null)
    }

    return valid
}
```

- `loginUser`: inicia sesión con los datos introducidos y tras haber realizado la validación.

```
private fun loginUser() {
    if (!validate()) {
        return
    }

    var userEmailQuery: ParseQuery<ParseUser> = ParseUser.getQuery().whereEqualTo("email", login_email_edittext.text.toString())

    userEmailQuery.findInBackground(FindCallback { users, e ->
        if (e == null && users.size > 0) {
            ParseUser.logInInBackground(users.get(0).username, login_password_edittext.text.toString(), LogInCallback { user, a ->
                if (user != null) {
                    Toast.makeText(context: this, text: "Welcome ${user.get("name")}!", Toast.LENGTH_SHORT).show()

                    val intent = Intent(packageContext: this, BoardsActivity::class.java)
                    startActivity(intent)
                    finish()
                } else {
                    ParseUser.logOut();
                    Toast.makeText(context: this, a.message, Toast.LENGTH_LONG).show();
                }
            })
        } else {
            login_email_edittext.setError("Email address does not mach any user")
        }
    })
}
```

- `isNetworkAvailable`: comprueba la conexión del dispositivo. Retorna un **boolean** en función de si hay conexión o no.

```
private fun isNetworkAvailable(): Boolean {  
    val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE)  
    return if (connectivityManager is ConnectivityManager) {  
        val networkInfo: NetworkInfo? = connectivityManager.activeNetworkInfo  
        networkInfo?.isConnected ?: false  
    } else false  
}
```

SignUpActivity: esta actividad es la encargada de la creación de cuentas de la aplicación.

- `onNetworkConnectionChanged`: cuando la conexión cambia también lo hace el botón, habilitándose o no, además de cambiar su texto.

```
override fun onNetworkConnectionChanged(isConnected: Boolean) {  
    if (!isConnected) {  
        signup_button.isEnabled = false  
        signup_button.text = "NO CONNECTION"  
    } else {  
        signup_button.text = "CREATE ACCOUNT"  
        signup_button.isEnabled = true  
    }  
}
```

- `validate`: valida que los datos introducidos en los campos sean correctos; de ser así retorna **true**. Es utilizado en conjunto con el método que sigue.

```
private fun validate(): Boolean{
    var valid = true

    val name = signup_name_edittext.text.toString()
    val surname = signup_surname_edittext.text.toString()
    val username = signup_nick_edittext.text.toString()
    val email = signup_email_edittext.text.toString()
    val password = signup_password_edittext.text.toString()
    val repassword = signup_repeat_password_edittext.text.toString()

    if (name.isEmpty() || name.length < 3) {
        signup_name_edittext.setError("At least 5 characters")
        valid = false
    } else {
        signup_name_edittext.setError(null)
    }

    if (surname.isEmpty() || name.length < 3) {
        signup_surname_edittext.setError("At least 5 characters")
        valid = false
    } else {
        signup_name_edittext.setError(null)
    }

    if (username.isEmpty() || name.length < 3) {
        signup_nick_edittext.setError("At least 3 characters")
        valid = false
    } else {
        signup_nick_edittext.setError(null)
    }

    if (email.isEmpty() || !android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
        signup_email_edittext!!.setError("Enter a valid email address")
        valid = false
    } else {
        signup_email_edittext!!.setError(null)
    }

    if (password.isEmpty() || password.length < 4 || password.length > 10) {
```

- signingUp: crea la cuenta con los datos introducidos y habiendo realizado la validación.

```
private fun signingUp(user: ParseUser){

    if (!validate()) {
        return
    }

    signup_progress_bar.visibility = View.VISIBLE

    user.signUpInBackground(SignUpCallback { it: ParseException!
        if (it == null) {
            signup_progress_bar.visibility = View.INVISIBLE
            Toast.makeText( context: this, text: "User created", Toast.LENGTH_SHORT).show()

            val intent = Intent( packageContext: this, LoginActivity::class.java)
            startActivity(intent)
            finish()
            overridePendingTransition(R.anim.push_left_in, R.anim.push_left_out)

        } else {
            signup_progress_bar.visibility = View.INVISIBLE
            ParseUser.logout()
            Toast.makeText( context: this, it.message, Toast.LENGTH_LONG).show();
        }
    })
}
```

- isNetworkAvailable: comprueba la conexión del dispositivo. Retorna un **boolean** en función de si hay conexión o no.

```
private fun isNetworkAvailable(): Boolean {
    val connectivityManager = getSystemService(Context.CONNECTIVITY_SERVICE)
    return if (connectivityManager is ConnectivityManager) {
        val networkInfo: NetworkInfo? = connectivityManager.activeNetworkInfo
        networkInfo?.isConnected ?: false
    } else false
}
```

BoardsActivity: en esta actividad se encuentra la lista de tableros y la interacción con ellos.

- clickedBoard: función estática que es llamada al clicar un `BoardItem` y lanza la actividad `NotesActivity` junto con el título del `BoardItem`.

```
fun clickedBoard(boardItem: BoardItem) {  
    val intent = Intent(contexto, NotesActivity::class.java)  
    intent.putExtra( name: "currentBoard", value: "${boardItem.getTitle()}")  
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)  
    contexto!!.startActivity(intent)  
}
```

- onPause: aquí borraremos y registraremos el receptor de conexión tras haberse pausado la actividad. Esto tras haberlo registrado en el metodo `onCreate`.

```
override fun onPause() {  
    super.onPause()  
    unregisterReceiver(connectionReceiver)  
  
    baseContext.registerReceiver(connectionReceiver, IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION))  
    App.instance.setConnectionListener(this)  
}
```

- onRestart: aquí actualizaremos el adapter, realizando una consulta, tras reiniciarse la actividad.

```
override fun onRestart() {  
    super.onRestart()  
  
    val boardsQuery = ParseQuery.getQuery(BoardItem::class.java)  
    boardsQuery .whereEqualTo("createdBy", ParseUser.getCurrentUser())  
  
    boardsQuery.findInBackground{ boardList, e ->  
        | boards_recycler_view.adapter = BoardsAdapter(ArrayList(boardList))  
        |  
    }  
}
```

- `onOptionsItemSelected`: aquí se establecen los escuchadores para cuando un ítem del menú es seleccionado.

```

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    when (item!!.itemId) {
        R.id.top_app_add -> {
            createBoard()
        }
        R.id.top_app_logout -> {
            ParseUser.logout()
            val loginIntent = Intent(packageContext: this, LoginActivity::class.java)
            startActivity(loginIntent)
            finish()
        }
    }

    return true
}

```

- `setAdapter`: en esta función se realizará una consulta a la base de datos y se establecerá el adapter con la lista obtenida.

```

private fun setAdapter(){

    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)

    boardQuery.findInBackground{ boardList, e ->
        if(e == null){
            if(boardList.count() > 0){
                boards_recycler_view.layoutManager = LinearLayoutManager(context: this)
                boards_recycler_view.adapter = BoardsAdapter(ArrayList(boardList))
            }else{
                boards_recycler_view.visibility = View.GONE
                boards_empty_layout.visibility = View.VISIBLE
            }
        }else{
            Util.showToast(context: this, message: "${e.message}")
        }
    }
}

```

- `createBoard`: esta función creará una instancia del fragmento `CreateBoardFragment`

```

private fun createBoard(){
    val dialogCreateBoard = CreateBoardFragment()
    dialogCreateBoard.show(supportFragmentManager, dialogCreateBoard.tag)
}

```

NotesActivity: en esta actividad se encuentra la lista de notas y la interacción con ellas.

- clickedNote: función estática que es llamada al clicar un `NoteItem` y lanza el fragmento `EditNoteFragment` junto con el título del `BoardItem`.

```
fun clickedNote(noteItem: NoteItem) {  
    val editNoteDialog = EditNoteFragment()  
  
    bundleEditNote.putString("currentBoard", currentBoard)  
    bundleEditNote.putString("currentNote", "${noteItem.getTitle()}")  
    bundleEditNote.putInt("currentPager", mainPager!!)  
  
    editNoteDialog.arguments = bundleEditNote  
    editNoteDialog.show(supportFragmentManager, editNoteDialog.tag)  
}
```

- onPause: aquí borraremos y registraremos el receptor de conexión tras haberse pausado la actividad. Esto tras haberlo registrado en el metodo `onCreate`.

```
override fun onPause() {  
    super.onPause()  
    unregisterReceiver(connectionReceiver)  
  
    baseContext.registerReceiver(connectionReceiver, IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION))  
    App.instance.setConnectionListener(this)  
}
```

- onOptionsItemSelected: aquí se establecen los escuchadores para cuando un item del menu es seleccionado.

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {  
  
    when (item?.itemId) {  
  
        R.id.bottom_app_home -> {  
            main_view_pager.setCurrentItem(1)  
        }  
        R.id.bottom_app_logout -> {  
            ParseUser.logout()  
            val loginIntent = Intent(packageContext, LoginActivity::class.java)  
            startActivity(loginIntent)  
            finish()  
        }  
        R.id.bottom_app_delete -> {  
            main_progress_bar.visibility = View.VISIBLE  
            deleteBoard()  
        }  
    }  
  
    return true  
}
```

- `handleFab`: esta función se encarga de la acción del botón flotante. Crea una instancia del fragmento `CreateNoteFragment`

```
private fun handleFab(floatButton: FloatingActionButton) {  
    floatButton.setOnClickListener { it: View!  
        val createNoteDialog = CreateNoteFragment()  
        bundleCreateNote.putString("currentBoard", currentBoard)  
        createNoteDialog.arguments = bundleCreateNote  
        createNoteDialog.show(supportFragmentManager, createNoteDialog.tag)  
    }  
}
```


- createTabs: está función añade al adaptador las instancias estáticas de los fragmentos de cada *página* y también contiene el método `onPageChangeListener` el cual pasará argumentos a los fragmentos que se encargan de crear notas (`CreateNoteFragment`) y editarlas (`EditNoteFragment`)

```
private fun createTabs(adapter: TabsAdapter) {
    adapter.addFragment(todoFragment, title: "To Do")
    adapter.addFragment(progressFragment, title: "In Progress")
    adapter.addFragment(doneFragment, title: "Done")

    sendBoard()

    main_view_pager.adapter = adapter
    main_tab_layout.setupWithViewPager(main_view_pager)

    main_view_pager.addOnPageChangeListener(object : ViewPager.OnPageChangeListener {

        override fun onPageScrollStateChanged(state: Int) {}
        override fun onPageScrolled(position: Int, positionOffset: Float, positionOffsetPixels: Int) {}

        override fun onPageSelected(position: Int) {
            when(position){
                0 ->{
                    bundleEditNote.putInt("currentPage", 0)
                    bundleCreateNote.putInt("currentPage", 0)
                }

                1 ->{
                    bundleEditNote.putInt("currentPage", 1)
                    bundleCreateNote.putInt("currentPage", 1)
                }

                2 -> {
                    bundleEditNote.putInt("currentPage", 2)
                    bundleCreateNote.putInt("currentPage", 2)
                }
            }
        }
    })
}
```

- getCurrentBoardIntent: obtiene el intent pasado por `BoardsActivity` con la cadena del título del tablero. Esto permitirá realizar las consultas a la base de datos cuando se quiera crear editar o borrar notas.

```
private fun getCurrentBoardIntent() {
    val intent = intent
    currentBoard = intent.extras!!.getString(key: "currentBoard")
}
```

- sendBoard: envía el título del tablero recogido en la función anterior a los fragmentos pertenecientes al `ViewPager`

```
private fun sendBoard(){
    val bundle = Bundle()
    bundle.putString("currentBoard", currentBoard)

    todoFragment.arguments = bundle
    progressFragment.arguments = bundle
    doneFragment.arguments = bundle
}
```

- deleteBoard: elimina el tablero actual y las notas que contenga. También lanzará un intent hacia la actividad `BoardsActivity`

```
private fun deleteBoard(){

    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.whereEqualTo("title", currentBoard)

    boardQuery.getFirstInBackground{ boardItem, e ->
        if(e == null ){

            val notesQuery = ParseQuery.getQuery(NoteItem::class.java)
            notesQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
            notesQuery.whereEqualTo("parentBoard", boardItem)

            notesQuery.findInBackground{ noteList, f ->
                if (f == null){
                    for (x in noteList){
                        x.delete()
                    }

                    boardItem.delete()

                    val boardsQuery = ParseQuery.getQuery(BoardItem::class.java)
                    boardsQuery .whereEqualTo("createdBy", ParseUser.getCurrentUser())

                    boardsQuery.findInBackground{ boardList, g ->

                        val intent = Intent( packageContext: this, BoardsActivity::class.java)
                        startActivity(intent)
                        finish()
                    }
                }
            }
        }
    }else{
        e.printStackTrace()
    }
}
```

CreateBoardFragment: este es un fragmento que extiende de DialogFragment, el cual se encarga de la creación de tableros.

- createBoard: tras ser llamado por un clickListener realiza las consultas para la validacion de unicidad del tablero y tras esto llama a la función `uploadBoard`. Actualiza también el adapter a través del método `updateAdapter`.

```
private fun createBoard(boardEditText: EditText) {

    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.whereEqualTo("title", boardEditText.text.toString())

    try {
        if (boardQuery.find().size > 0) {
            boardEditText.setError("There is a board with this name")
        } else {
            if (boardEditText.text.toString().isEmpty()) {
                boardEditText.setError("Missing board's name")
            } else {
                uploadBoard(boardEditText)

                val boardsQuery = ParseQuery.getQuery(BoardItem::class.java)
                boardsQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)
                boardsQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())

                try {
                    val boardList = ArrayList(boardsQuery.find())

                    if (activity!!.boards_empty_layout.visibility == View.VISIBLE) {
                        activity!!.boards_empty_layout.visibility = View.GONE
                        activity!!.boards_recycler_view.visibility = View.VISIBLE
                    }

                    updateAdapter(boardList)
                    dismiss()
                } catch (a: ParseException) {
                    a.printStackTrace()
                }
            }
        }
    } catch (e: ParseException) {
        e.printStackTrace()
    }
}
```

- uploadBoard: crea una instancia de `BoardItem` y lo crea en la base de datos con los datos pasados.

```
private fun uploadBoard(boardEditText: EditText) {  
  
    val board = BoardItem()  
    board.setTitle(boardEditText.text.toString())  
    board.setUser(ParseUser.getCurrentUser())  
  
    board.save()  
}
```

- updateAdapter: crea una instancia de `BoardsAdapter` y actualiza la del recyclerView de la actividad.

```
private fun updateAdapter(boardList: ArrayList<BoardItem>) {  
    val layoutManager = LinearLayoutManager(activity)  
    val adapter = BoardsAdapter(boardList)  
  
    activity!!.boards_recycler_view.adapter = adapter  
    activity!!.boards_recycler_view.layoutManager = layoutManager  
  
    activity!!.boards_recycler_view.adapter!!.notifyDataSetChanged()  
}
```

CreateNoteFragment: este es un fragmento que extiende de DialogFragment, el cual se encarga de la creación de notas.

- createNote: tras ser llamado por un clickListener realiza las consultas para la obtención del tablero, tras esto valida la unicidad de la nota y una vez hecho esto, con el índice pasado antes para obtener la página en la que se encuentra del viewPager llama al metodo uploadNote y tras esto al updateAdapter correspondiente con la página.

```
private fun createNote(title: EditText, description: EditText, comment: EditText, currentPage: Int?) {  
  
    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)  
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)  
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
    boardQuery.whereEqualTo("title", currentBoard)  
  
    boardQuery.getFirstInBackground { boardItem, e ->  
        if (e == null) {  
            val noteQuery = ParseQuery.getQuery(NoteItem::class.java)  
            noteQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
            noteQuery.whereEqualTo("parentBoard", boardItem)  
            noteQuery.whereEqualTo("title", title.text.toString())  
  
            noteQuery.getFirstInBackground { noteItem, a ->  
                if (noteItem != null) {  
                    title.setError("There is a note with this name")  
                } else {  
                    if (title.text.toString().isEmpty()) {  
                        title.setError("Fill in the field")  
                    } else {  
                        title.setError(null)  
                    }  
  
                    if (description.text.toString().isEmpty()) {  
                        description.setError("Fill in the field")  
                    } else {  
                        description.setError(null)  
                    }  
  
                    when (currentPage) {  
                        0 -> {  
                            uploadNote(title, description, comment, state: "todo")  
                            updateToDoAdapter()  
                            dismiss()  
                        }  
  
                        1 -> {  
                            uploadNote(title, description, comment, state: "inprogress")  
                            updateInProgressAdapter()  
                            dismiss()  
                        }  
  
                        2 -> {  
                            uploadNote(title, description, comment, state: "done")  
                            updateDoneAdapter()  
                            dismiss()  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

- uploadNote: realiza la consulta del tablero y consiguientemente crea la instancia de NoteItem y la salva en la base de datos.

```
private fun uploadNote(title: EditText, description: EditText, comment: EditText, state: String){  
  
    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)  
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
    boardQuery.whereEqualTo("title", currentBoard)  
  
    val boardItem = boardQuery.first  
  
    val note = NoteItem()  
    note.setTitle(title.text.toString())  
    note.setDescription(description.text.toString())  
    note.setComment(comment.text.toString())  
    note.setState(state)  
    note.setUser(ParseUser.getCurrentUser())  
    note.setBoard(boardItem)  
  
    note.save()  
}
```

- updateToDoAdapter: crea una instancia de `BoardsAdapter` y actualiza la del `recyclerView` del fragmento de forma estática.

```
private fun updateToDoAdapter() {
    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.whereEqualTo("title", currentBoard)

    boardQuery.getFirstInBackground(boardItem, e ->
        if (boardItem != null) {
            val notesQuery = ParseQuery.getQuery(NoteItem::class.java)
            notesQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
            notesQuery.whereEqualTo("parentBoard", boardItem)
            notesQuery.whereEqualTo("state", "todo")

            notesQuery.findInBackground(notesList, a ->
                if (a == null) {
                    val layoutManager = LinearLayoutManager(activity)
                    val adapter = NotesAdapter(ArrayList(notesList))

                    FragmentToDo.adapter = adapter
                    FragmentToDo.recyclerView!!.adapter = adapter
                    FragmentToDo.recyclerView!!.layoutManager = layoutManager

                    FragmentToDo.recyclerView!!.adapter!!.notifyDataSetChanged()
                } else {
                    Util.showToast(activity, a.message.toString())
                    a.printStackTrace()
                }
            )
        }
    )
}
```

EditNoteFragment: este es un fragmento que extiende de DialogFragment, el cual se encarga de la edición de notas.

- loadNoteData: realizando una consulta a la base de datos carga los datos de la nota en el fragmento para su posterior tratamiento.

```
private fun loadNoteData(title: EditText, description: EditText, comment: EditText){
    val noteQuery = ParseQuery.getQuery(NoteItem::class.java)
    noteQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    noteQuery.whereEqualTo("title", currentNote)

    noteQuery.getFirstInBackground{ noteItem, e ->
        if(noteItem != null){
            title.setText("${noteItem.getTitle()}")
            description.setText("${noteItem.getDescription()}")
            comment.setText("${noteItem.getComment()}")
        }
    }
}
```

- saveNote: esta función es igual a `createNote` del fragmento `CreateNoteFragment`


```
private fun saveNote(title: EditText, description: EditText, comment: EditText){

    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.whereEqualTo("title", currentBoard)

    boardQuery.getFirstInBackground{boardItem, e ->
        val noteQuery = ParseQuery.getQuery(NoteItem::class.java)
        noteQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
        noteQuery.whereEqualTo("parentBoard", boardItem)
        noteQuery.whereEqualTo("title", title.text.toString())

        noteQuery.getFirstInBackground{ noteItem, a ->
            if (noteItem != null){
                title.setError("There is a note with this name")
            }else{
                if(title.text.toString().isEmpty()){
                    title.setError("Fill in the field")
                }else{
                    title.setError(null)

                    if(description.text.toString().isEmpty()){
                        description.setError("Fill in the field")
                    }else{
                        description.setError(null)

                        if(comment.text.toString().isEmpty()){
                            comment.setError("Fill in the field")
                        }else{
                            comment.setError(null)

                            when(currentPage) {
                                0 -> {
                                    uploadNote(title,description, comment)
                                    updateToDoAdapter()
                                    dismiss()
                                }

                                1 -> {
                                    uploadNote(title, description, comment)
                                    updateInProgressAdapter()
                                    dismiss()
                                }

                                2 -> {
                                    uploadNote(title, description, comment)
                                    updateDoneAdapter()
                                    dismiss()
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

- `deleteNote`: tras ser llamada por un `clickListener` se encarga de realizar la consulta del tablero que contiene la nota, a continuación obtendrá la misma de la base de datos y la eliminará. Por último y de la misma forma que la función `saveNote` llamará al método `updateAdapter` correspondiente a la página actual del `viewPager`.

```
private fun deleteNote(title: EditText, description: EditText, comment: EditText) {  
  
    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)  
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)  
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
    boardQuery.whereEqualTo("title", currentBoard)  
  
    boardQuery.getFirstInBackground { boardItem, e ->  
        val noteQuery = ParseQuery.getQuery(NoteItem::class.java)  
        noteQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
        noteQuery.whereEqualTo("parentBoard", boardItem)  
        noteQuery.whereEqualTo("title", title.text.toString())  
  
        noteQuery.getFirstInBackground { noteItem, a ->  
            noteItem.deleteInBackground { it: ParseException!  
                when(currentPage) {  
                    0 -> {  
                        updateToDoAdapter()  
                        dismiss()  
                    }  
  
                    1 -> {  
                        updateInProgressAdapter()  
                        dismiss()  
                    }  
  
                    2 -> {  
                        updateDoneAdapter()  
                        dismiss()  
                    }  
                }  
            }  
        }  
    }  
}
```

A continuación y para la facilidad de la lectura presentaré uno de los tres fragmentos que forman parte del `viewPager`. Estos son similares por lo que es innecesario repetir la definición de las funciones. Ante cualquier duda revisar la documentación adjunta en el repositorio Github proporcionado.

FragmentToDo: este es un fragmento que extiende de Fragment en el cual se mostrarán las notas que correspondan al estado *todo*.

- loadData: carga las notas que correspondan con su estado tras realizar la consulta del tablero en el que se encuentra y la de las notas validas.

```
private fun loadData() {  
  
    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)  
    val boardIsInCache = boardQuery.hasCachedResult()  
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.CACHE_ELSE_NETWORK)  
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
    boardQuery.whereEqualTo("title", currentBoard)  
  
    if (!boardIsInCache) {  
        val boardItem = boardQuery.first  
  
        val notesQuery = ParseQuery.getQuery(NoteItem::class.java)  
        notesQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)  
  
        notesQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())  
        notesQuery.whereEqualTo("parentBoard", boardItem)  
        notesQuery.whereEqualTo("state", "todo")  
  
        try {  
            val notesList = ArrayList(notesQuery.find())  
  
            val layoutManager = LinearLayoutManager(activity)  
            adapter = NotesAdapter(notesList)  
  
            recyclerView!!.adapter = adapter  
            recyclerView!!.layoutManager = layoutManager  
        } catch (e: com.parse.ParseException) {  
            Util.showToast(activity, e.message.toString())  
            e.printStackTrace()  
        }  
    }  
}
```

- onSwipeNote: contiene una instancia de `ItemTouchHelper` que se encargará de escuchar el movimiento de las notas y llamará a la función `changeNoteState`

```
private fun onSwipeNote() {  
    val itemTouchHelperCallback = object: ItemTouchHelper.SimpleCallback( dragDirs: 0, ItemTouchHelper.RIGHT) {  
        override fun onSwiped(viewHolder: RecyclerView.ViewHolder, position: Int) {  
            changeNoteState(viewHolder)  
        }  
  
        override fun onMove(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder, target: RecyclerView.ViewHolder): Boolean {  
            return false  
        }  
    }  
  
    val itemTouchHelper = ItemTouchHelper(itemTouchHelperCallback)  
    itemTouchHelper.attachToRecyclerView(recyclerView)  
}
```

- changeNoteState: el progreso es el siguiente; obtiene el título de la nota actual, consulta el tablero padre, consulta la nota y cambia el estado, consulta dos listas para cada uno de los fragmentos y actualiza sus adapters para mostrar los últimos cambios.

```

private fun changeNoteState(viewHolder: RecyclerView.ViewHolder) {

    val title = adapter!!.getNoteAt(viewHolder.adapterPosition).getTitle().toString()

    val boardQuery = ParseQuery.getQuery(BoardItem::class.java)
    boardQuery.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)
    boardQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
    boardQuery.whereEqualTo("title", currentBoard)

    boardQuery.getFirstInBackground { boardItem, e ->
        progressBar!!.visibility = View.VISIBLE

        val noteQuery = ParseQuery.getQuery(NoteItem::class.java)
        noteQuery.whereEqualTo("createdBy", ParseUser.getCurrentUser())
        noteQuery.whereEqualTo("parentBoard", boardItem)
        noteQuery.whereEqualTo("title", title)

        noteQuery.getFirstInBackground { noteItem, a ->
            noteItem.setState("inprogress")
            noteItem.save()

            val notesQueryTODO = ParseQuery.getQuery(NoteItem::class.java)
            notesQueryTODO.whereEqualTo("createdBy", ParseUser.getCurrentUser())
            notesQueryTODO.whereEqualTo("parentBoard", boardItem)
            notesQueryTODO.whereEqualTo("state", "todo")
            notesQueryTODO.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)

            val notesListTODO = ArrayList(notesQueryTODO.find())
            val layoutManagerTODO = LinearLayoutManager(activity)

            //Update static adapter
            adapter = NotesAdapter(notesListTODO)
            recyclerView!!.adapter = adapter
            recyclerView!!.layoutManager = layoutManagerTODO

            recyclerView!!.adapter!!.notifyDataSetChanged()

            progressBar!!.visibility = View.INVISIBLE

            val notesQueryINPROGRESS= ParseQuery.getQuery(NoteItem::class.java)
            notesQueryINPROGRESS.whereEqualTo("createdBy", ParseUser.getCurrentUser())
            notesQueryINPROGRESS.whereEqualTo("parentBoard", boardItem)
            notesQueryINPROGRESS.whereEqualTo("state", "inprogress")
            notesQueryINPROGRESS.setCachePolicy(ParseQuery.CachePolicy.NETWORK_ELSE_CACHE)

            notesQueryINPROGRESS.findInBackground { notesListINPROGRESS, e ->
                if (e == null) {
                    val adapterINPROGRESS = NotesAdapter(ArrayList(notesListINPROGRESS))
                    val layoutManagerINPROGRESS = LinearLayoutManager(activity)

                    FragmentInProgress.adapter = adapterINPROGRESS
                    FragmentInProgress.recyclerView!!.adapter = adapterINPROGRESS
                    FragmentInProgress.recyclerView!!.layoutManager = layoutManagerINPROGRESS

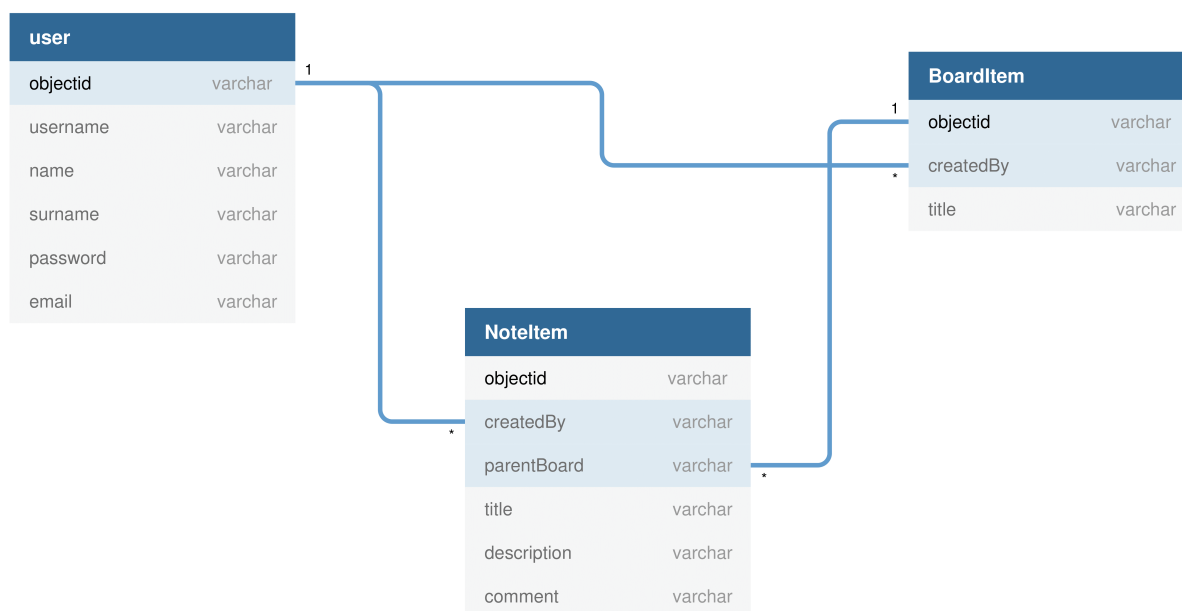
                    FragmentInProgress.recyclerView!!.adapter!!.notifyDataSetChanged()
                } else {
                    Util.showToast(activity, e.message.toString())
                    e.printStackTrace()
                }
            }
        }
    }
}
}
}
}

```

Diseño de Base de datos

Entrando en el desarrollo de la base de datos nos encontramos con **Parse Server**, el cuál puede correr tanto con una base de datos de **MongoDB** como **PostgreSQL**. Tras una larga investigación me tope con un servicio llamado **Back4App** el cuál basado en **Parse Server**, proporciona de forma gratuita una base de datos y servicio de *Dashoboard*, es decir, una manera más sencilla y amable de trabajar con la base de datos.

Dicho esto presentaré aquí el modelo de base de datos con el que trabaja **Jatz**



holistics

Esquema de Diálogo

En este capítulo veremos descrita la parte visual del proyecto, informando de esta forma el modo de uso de la aplicación e un ámbito más visual. Debido a que es un aplicación Android, este esquema se concretará en función de las actividades que conforman **Jatz**.

Inicio de Sesión

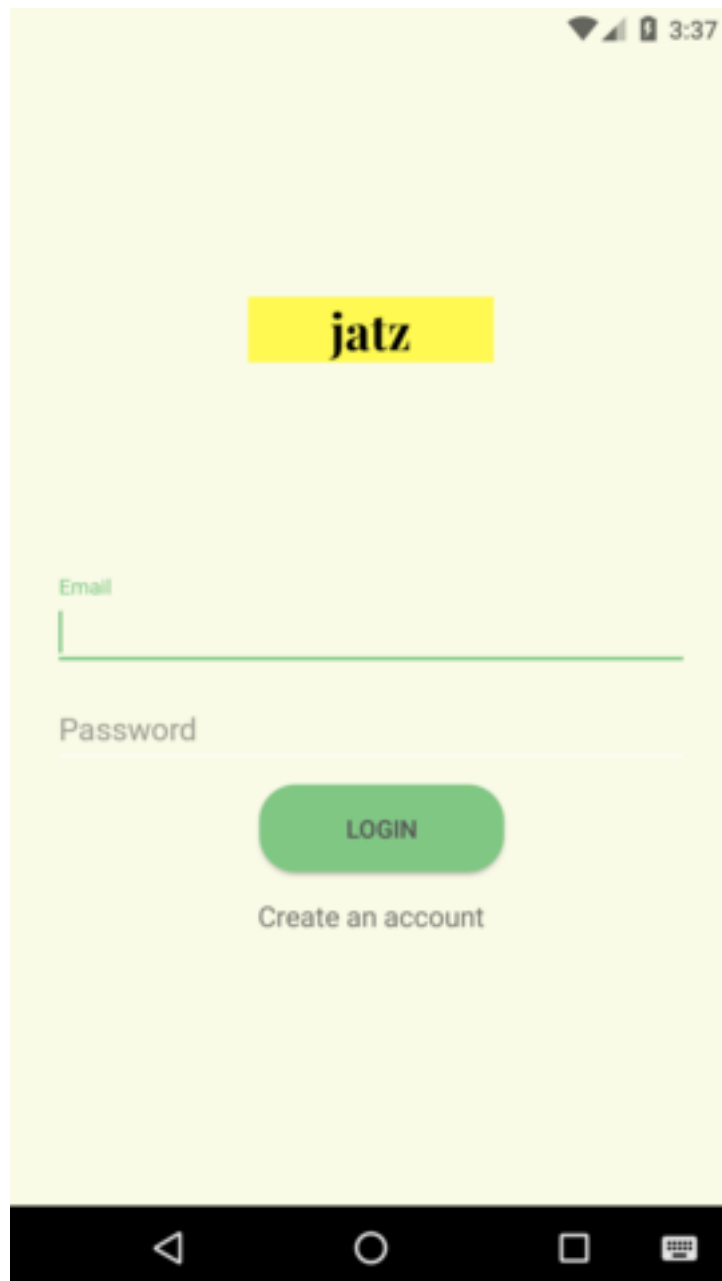


Figura 8: Tras la actividad `SplashActivity` nos encontramos con la actividad conocida como `LoginActivity`, en ella podemos ver dos campos `EditText` referentes al email y a la contraseña, un botón `Login` que ejecutará el intento de inicio de sesión y bajo él un `TextView` que no llevará a la actividad `SignUpActivity`.

Creación Cuenta

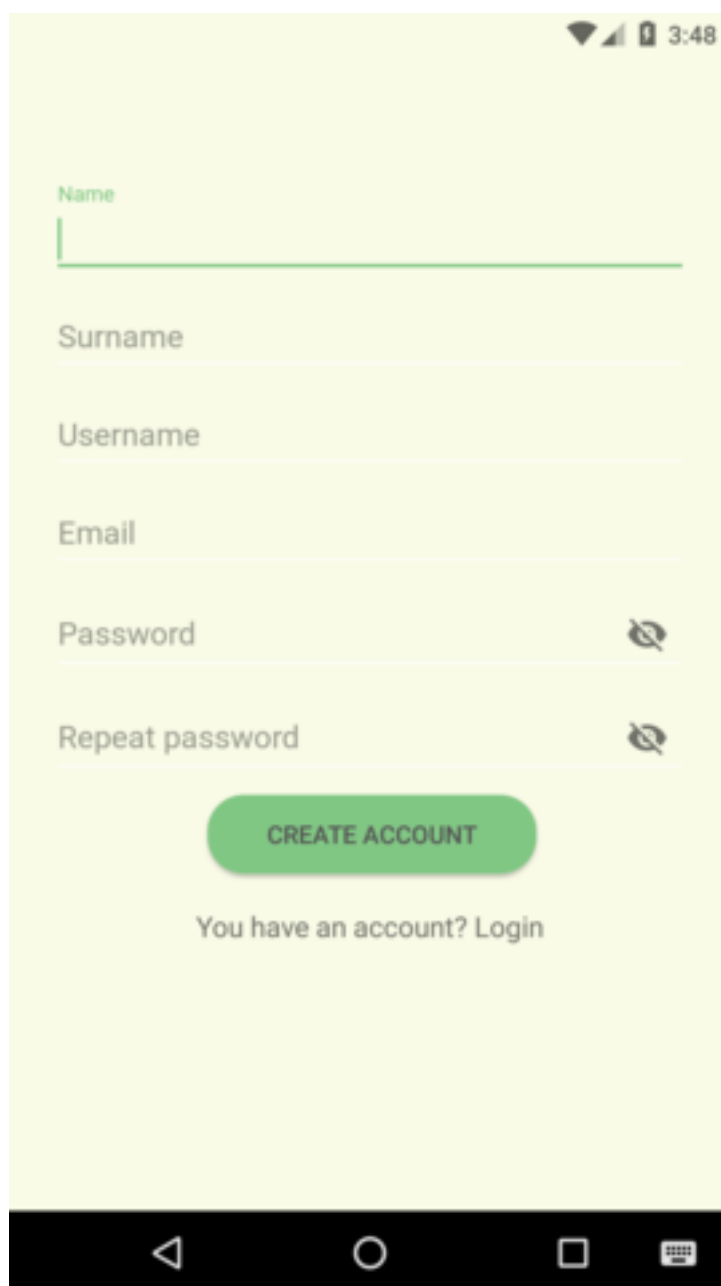


Figura 9: Una vez clicado el ítem `TextView Create an account` de la actividad `LoginActivity` pasaremos a esta pantalla, la actividad `SignUpActivity`. En esta rellenaremos los campos y seguido a esto pulsaremos el botón `Create account` para realizar el registro. El último ítem de esta actividad es un `TextView` que nos permitirá ir a la actividad anterior `LoginActivity`

Tableros

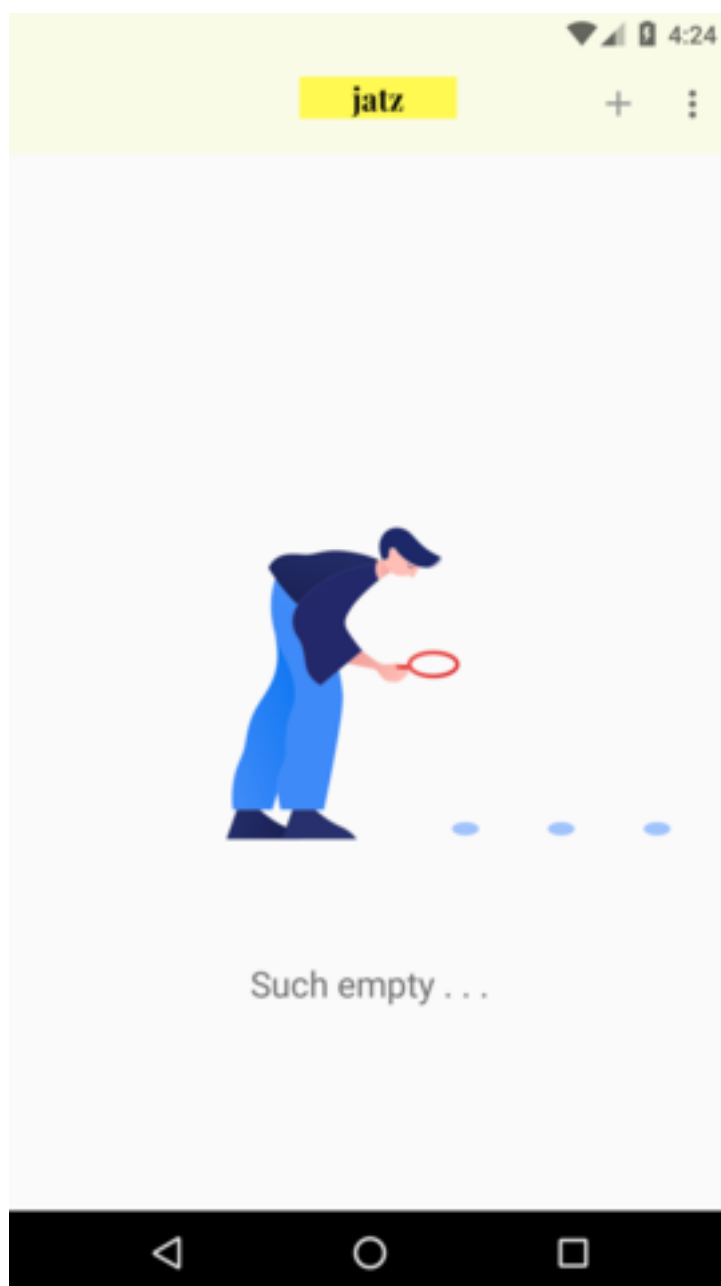


Figura 10: Tras iniciar sesión por primera vez saltará la actividad `BoardsActivity`, en un principio vacía, con dos botones en la parte superior; uno de adición que dará paso al fragmento `CreateBoardFragment`, en el que se podrá crear un tablero, y otro a modo de menú, en el cuál se podrá cerrar la sesión.

Notas



Figura 11: Una vez creado y seleccionado el tablero se mostrará la actividad `NotesActivity` con tres pestañas en la parte superior referentes a cada uno de los estados posibles de la aplicación. En la parte inferior se visualizan tres botones interactivos: *Adición*, el cuál inicia el fragmento `CreateNoteFragment` en el que indicaremos los campos necesarios para crear una nota; *Home*, el que se encarga de llevarnos a la página *In Progress*; *Menu*, con dos opciones, una para eliminar el tablero antes seleccionado y otra para cerrar sesión.

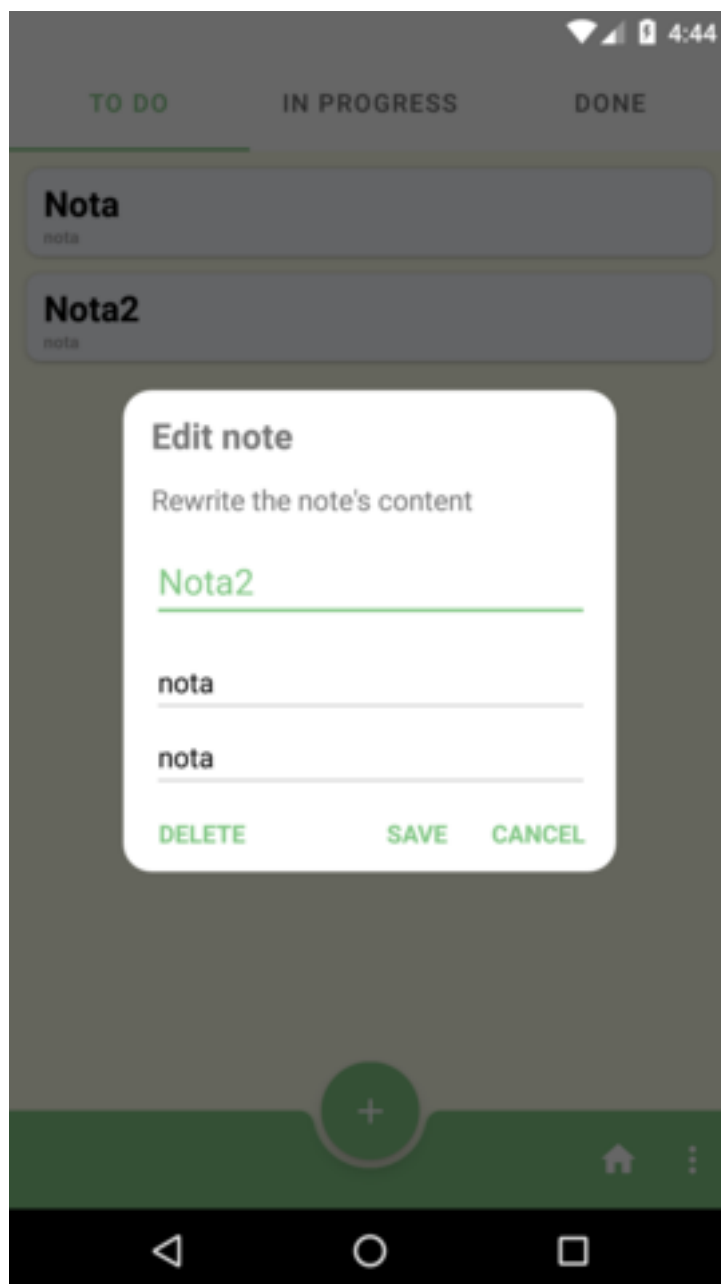


Figura 12: Habiendo creado una nota podremos acceder a la edición o borrado de la misma al seleccionar el ítem; todo esto proporcionado por la instancia del fragmento `EditNoteFragment`. También podremos deslizar a la izquierda o a la derecha para cambiar el estado de la nota.

Conclusiones

Como se ha podido ver a lo largo de todo este trabajo de fin de grado se han cumplido los objetivos planteados, ya que se ha conseguido el desarrollo de la aplicación descrita anteriormente para dispositivos móviles con sistema operativo **Android**, con un resultado bastante satisfactorio, tal y como demuestran las distintas evaluaciones que se han llevado a cabo en todo el proceso de construcción.

Al ser un proyecto individual y con tecnologías novedosas se han podido adquirir diversos conocimientos tanto con respecto al desarrollo móvil como a la programación general, ya que ha requerido aprender un lenguaje y más parte del desarrollo de forma autodidacta.

Mejoras

En este apartado se definirán las posibles mejoras que se pueden realizar para una próxima versión, además de las adiciones de *features* o características que se pueden hacer.

- Localizar en un servidor local la base de datos para así mejorar la velocidad de consultas y no depender de un servicio de terceros.
- Perfeccionar la eficiencia de los procesos para así conseguir una experiencia más fluida.
- Añadir la característica de grupos y la compartición de tableros.

Bibliografía

Referencias

- [1] Anónimo: Ejemplos java y C/Linux,
<http://www.chuidiang.org/ood/metodologia/scrum.php>
- [2] Statcounter: Mobile Operating System Market Share Worldwide,
<http://gs.statcounter.com/os-market-share/mobile/worldwide>
- [3] Anónimo: About,
<https://kotlinlang.org>
- [4] Janice J. Heiss: The Advent of Kotlin: A Conversation with JetBrains' Andrey Breslav,
<https://www.oracle.com/technetwork/articles/java/breslav-1932170.html>
- [5] Antonio Leiva: MVP for Android: how to organize the presentation layer,
<https://antonioleiva.com/mvp-android/>
- [6] Alvaro Arrarte: Scrum Master: Las metodologías ágiles que llevan tu proyecto al éxito,
<https://www.recomendacionesy tendencias.com/scrum-master-metodologias-agiles/>