Kingdom Town For Gods

Desarrollo de Aplicaciones Multiplataforma

Angel Cendrero y Juan Ajenjo

Vicente Rama Alguilar

9 de Junio de 2024

IES EL CAÑAVERAL

Departamento de Informática

Informática y Comunicaciones

Memoria del proyecto

Documento técnico

Índice

1.- Introducción

1.1.- Descripción y contexto del proyecto

El juego es un simulador de gestión de una aldea en estilo idle, donde los jugadores administran una comunidad desde sus inicios con 10 aldeanos.

La población aumenta periódicamente si hay suficiente comida disponible.

Los recursos principales son troncos, tablones, piedra, hierro, comida y oro, cada uno obtenido de edificios específicos: la casete del leñador para troncos, la carpintería para tablones, la mina para piedra, hierro y oro, la cabaña de caza y la granja para comida.

La obtención de estos recursos permite mejorar la aldea y sus distintos edificios para aumentar el máximo de población y de recursos que se pueden tener así como desbloquear nuevos edificios y nuevas funciones como los ataques a otros jugadores y un mercado para obtener recursos de manera más rápida.

1.2.- Motivación del proyecto

La motivación detrás del desarrollo surge de la popularidad de los juegos de gestión y simulación, especialmente aquellos que pueden ser jugados de forma casual en dispositivos móviles. Este tipo de juegos ofrece una experiencia atractiva y relajante, permitiendo a los jugadores ver cómo sus decisiones afectan el progreso de su aldea a lo largo del tiempo.

1.3.- Beneficios esperados

- **Desarrollo de Competencias Técnicas**: El proyecto permitirá aplicar y desarrollar habilidades técnicas en programación, diseño de interfaces de usuario y gestión de bases de datos.
- **Experiencia en Gestión de Proyectos**: Los estudiantes adquirirán experiencia en la planificación, ejecución y evaluación de un proyecto de software complejo.
- **Aplicación de Conocimientos Teóricos**: Permite poner en práctica conocimientos adquiridos en diversas asignaturas, como algoritmos, estructuras de datos, desarrollo de software y diseño de sistemas.
- **Trabajo en Equipo y Comunicación**: El proyecto fomenta habilidades de colaboración y comunicación, esenciales para el trabajo en equipo y la resolución de problemas.

2.- Objetivo/s generales del proyecto

Desarrollar un simulador de gestión de aldea en estilo idle implica la creación de un juego que permite a los jugadores administrar una aldea virtual. Este simulador combinará la gestión de recursos, la construcción y mejora de edificios, así como mecánicas de combate y defensa multijugador. Los jugadores tendrán la tarea de recolectar y administrar recursos como madera, piedra, alimentos, entre otros, para expandir su aldea y mejorar sus infraestructuras.

La interfaz de usuario de este juego deberá ser intuitiva y fácil de usar, permitiendo a los jugadores navegar por las diferentes opciones y actividades de manera sencilla. Además, se incorporará un sistema de ranking competitivo que motivará a los jugadores a competir entre ellos y demostrar sus habilidades de gestión y estrategia.

Para garantizar la escalabilidad y mantenibilidad del software para futuras expansiones, se implementarán estructuras de código eficientes y flexibles que permitan añadir nuevas funcionalidades, edificios, personajes u opciones de juego de manera sencilla. De esta forma, el juego podrá crecer y evolucionar con el tiempo, ofreciendo a los jugadores nuevas experiencias y desafíos.

En resumen, el desarrollo de este simulador de gestión de aldea en estilo idle requerirá un enfoque integral que combine aspectos fundamentales como la gestión de recursos, la construcción de edificios, las mecánicas de combate, la competencia multijugador, una interfaz amigable y un sistema de ranking, todo ello respaldado por una arquitectura de software robusta y adaptable.

3.- Objetivos específicos

- Identificar los requisitos funcionales y no funcionales del juego y representarlos adecuadamente para guiar el desarrollo.
- Diseñar y desarrollar la interfaz de usuario, asegurando que sea intuitiva y atractiva para facilitar la interacción del jugador.
- Implementar los sistemas de recolección y gestión de recursos, desarrollando edificios específicos para la producción de troncos, tablones, piedra, hierro, comida y oro.
- Desarrollar las mecánicas de construcción y mejora de edificios, permitiendo a los jugadores incrementar la eficiencia de producción y la capacidad de la aldea.
- Crear y optimizar las mecánicas de combate y defensa, incluyendo el desarrollo del castillo y el sistema de incursiones contra otros jugadores.
- Implementar un sistema de ranking competitivo que registre y muestre las puntuaciones de los jugadores, incentivando la mejora continua.
- Planificar y ejecutar pruebas exhaustivas para asegurar la funcionalidad, estabilidad y usabilidad del juego.
- Desarrollar un plan de mantenimiento y actualización, asegurando la escalabilidad y la capacidad de añadir futuras expansiones y mejoras al juego.

- Documentar todo el proceso de desarrollo, incluyendo los requisitos, diseño, implementación, pruebas y mantenimiento, para facilitar la comprensión y futura gestión del proyecto.

4.- Contexto actual

Estado del arte.-

En el ámbito de los videojuegos de gestión y simulación, destacan varios títulos que han capturado la atención de los jugadores:

- Clash of Clans: Estrategia y gestión de recursos con énfasis en combate y defensa.
- SimCity BuildIt: Construcción y gestión de ciudades, equilibrando recursos y expansión.
- Forge of Empires: Gestión de ciudades y recursos con elementos de combate y desarrollo histórico.
- Rise of Kingdoms: Estrategia en tiempo real con gestión de recursos, exploración y combate.
- Hay Day: Gestión de una granja y sus distintos cultivos y con mecánicas de producción y comercio

Conceptos clave.-

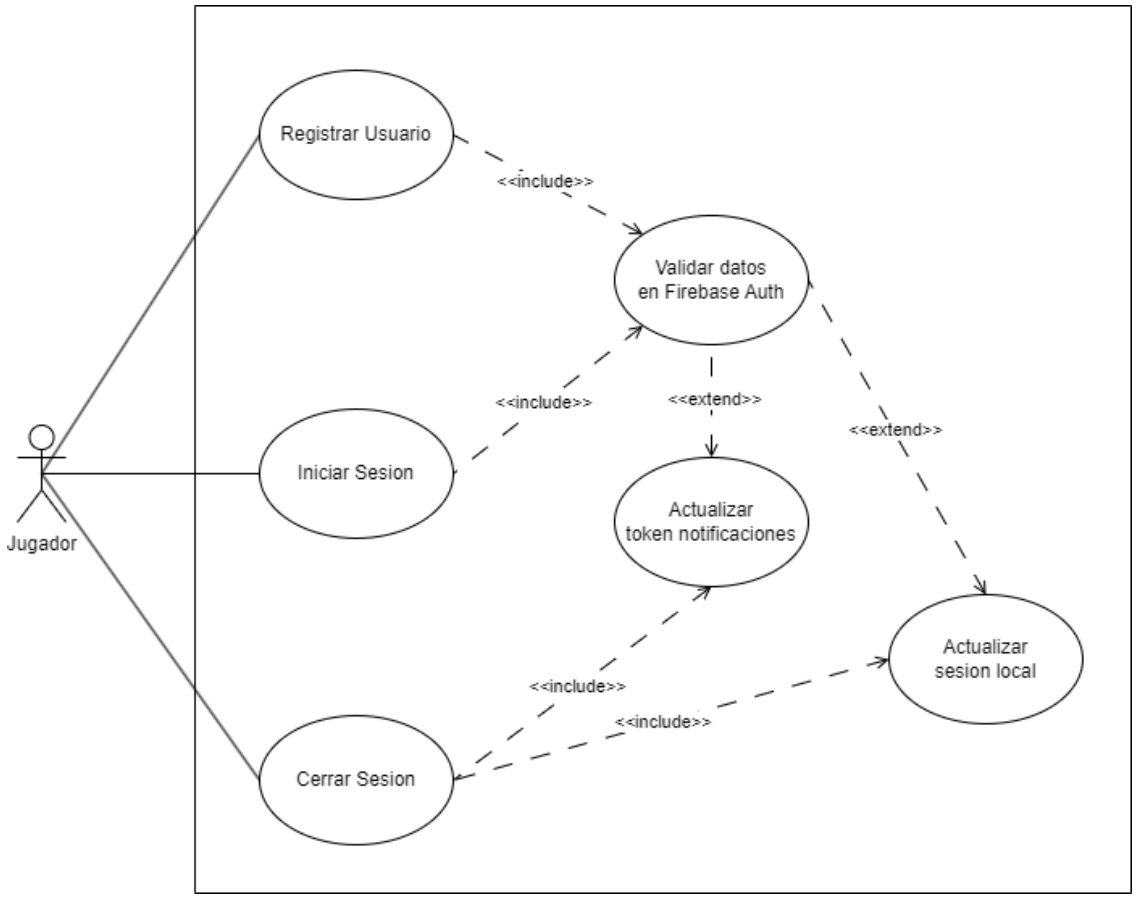
- **Simulador de Gestión**: Videojuego enfocado en la administración de recursos y estructuras.
- **Idle Game**: Género donde el juego progresa incluso cuando no está activo.
- **Recursos**: Elementos esenciales como troncos, tablones, piedra, hierro, comida y oro.
- **Edificios**: Estructuras para producir recursos, mejorables hasta nivel 10.
- **Mejora de Edificios**: Incremento de capacidad y eficiencia mediante inversión de recursos.
- **Senado**: Edificio central que aumenta la capacidad de la aldea y desbloquea nuevas funcionalidades.
- **Incursiones y Defensa**: Mecánicas de combate entre jugadores.
- **Ranking**: Clasificación de jugadores basada en logros y victorias en incursiones.

5.- Análisis de requisitos

5.1.- Diagrama de casos de uso.

- Casos de uso de la sesión del usuario:
  - **Registrar usuario**: Permitir al usuario registrarse en caso de ser nuevo, validando que las credenciales introducidas son válidas y asegurándose de que pueda recibir notificaciones.

- **Iniciar sesión**: Permitir al usuario iniciar sesión con cualquiera de sus cuentas, validando que las credenciales introducidas son válidas, guardar la sesión de manera local para que no sea necesario que introduzca sus datos cada vez que entra en el juego y asegurar que pueda recibir notificaciones.
- **Cerrar sesión**: Permitir al usuario cerrar su sesión actual y asegurar las notificaciones de su cuenta se desvinculan del dispositivo.

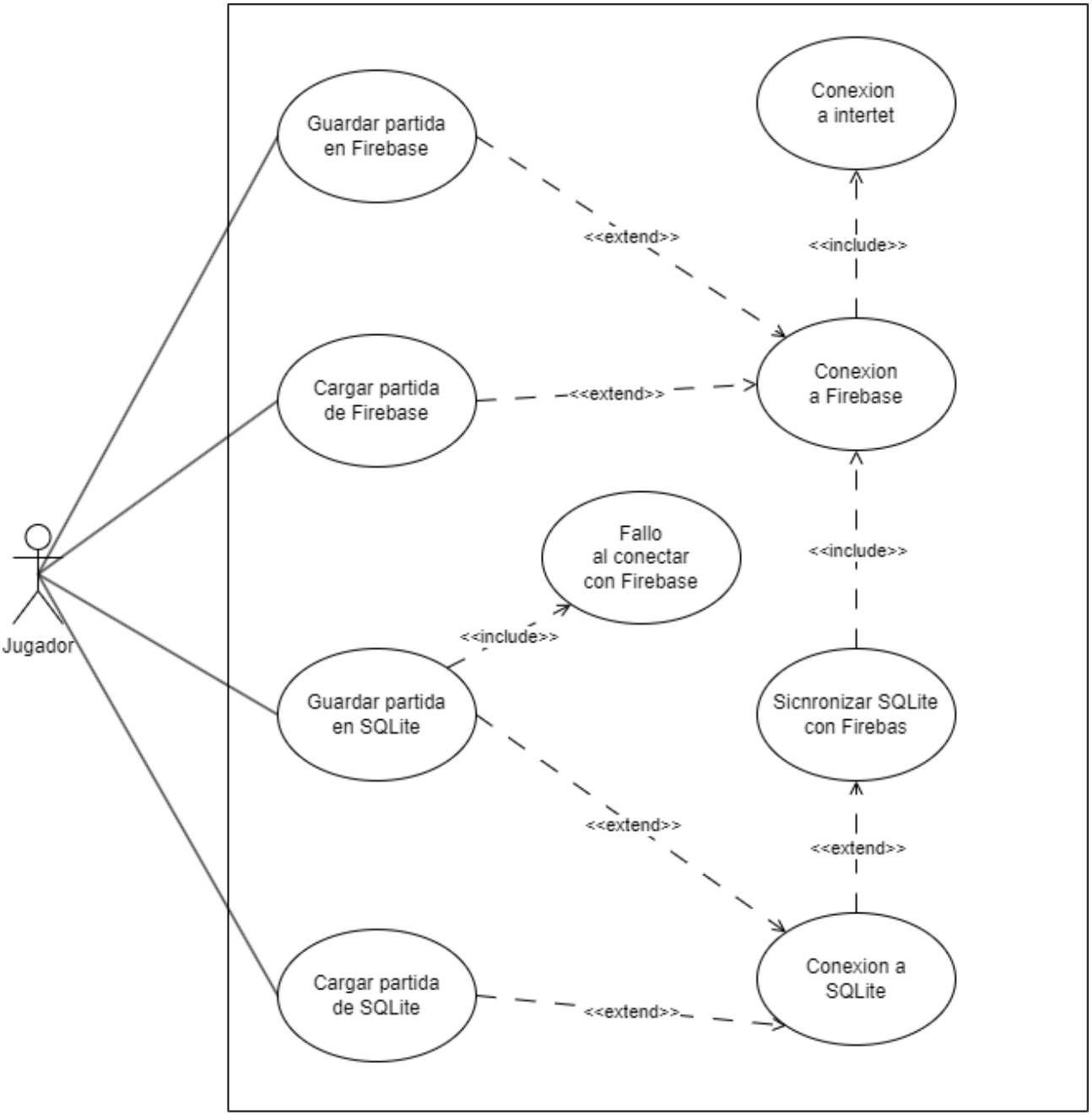


- Casos de uso de la gestión de los datos del usuario:

  Estos casos de uso son realizados por el usuario de manera indirecta por acciones como entrar y salir del juego. Y todos los casos de uso que requieran conexiones a la base de datos requerirán de conexión a internet, ya que si no hay no se iniciará ninguno de estos casos de uso, y por lo tanto no se iniciará el juego.

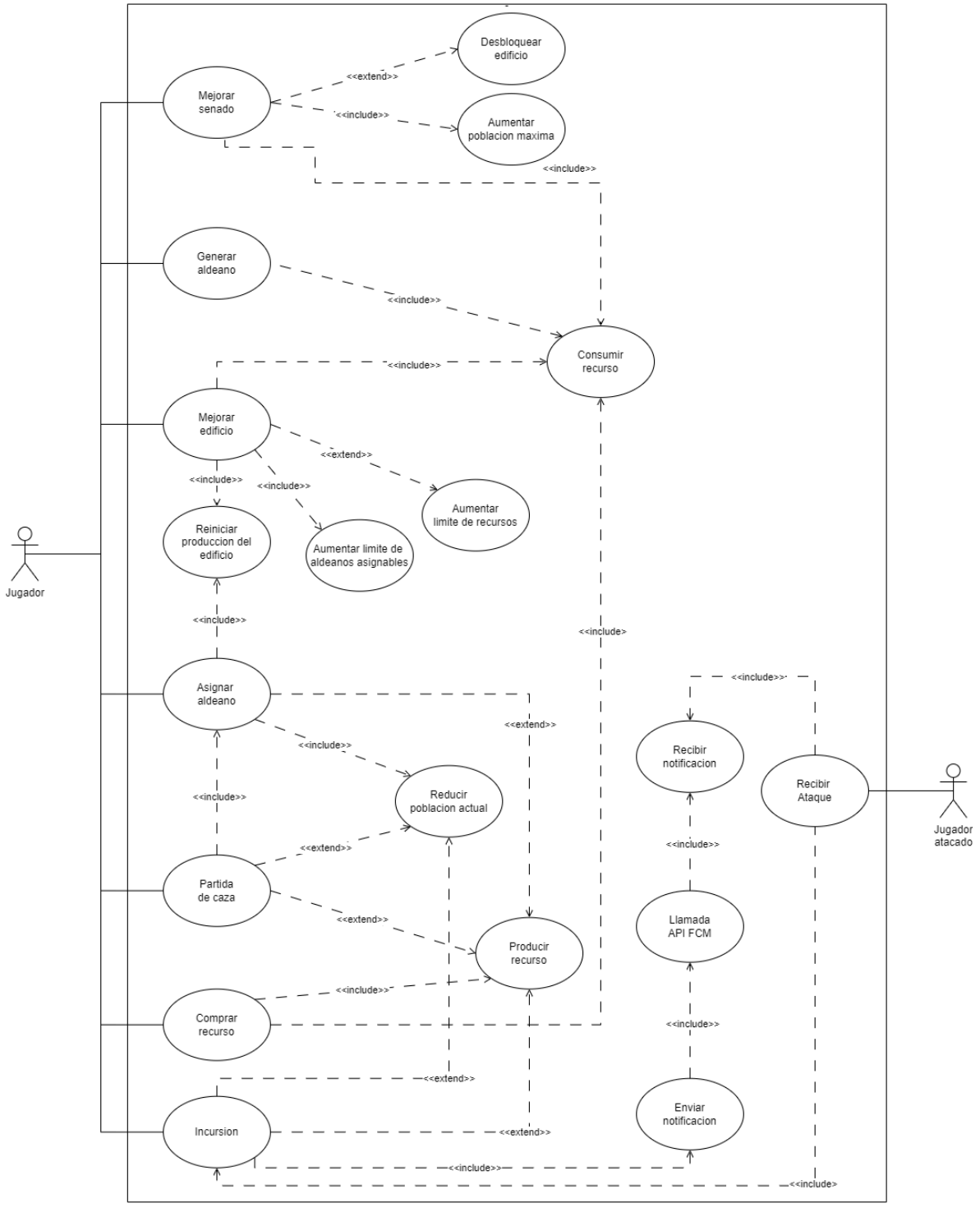
  - **Guardar partida en Firebase**: Cuando el usuario sale del juego automáticamente se guardarán todos sus progresos y los datos relacionados con el sistema online del juego referentes al usuario en Firebase, pero en caso de que haya un error se guardarán de manera local en SQLite (Otro caso de uso) .
  - **Cargar partida de Firebase**: Cuando el usuario entra al juego automáticamente se comprobará si la partida local está sincronizada con la del servidor (en caso de existir partida local) y en caso de que esté sincronizada cargará los datos de Firebase, si no lo está se cargará la partida loca (Otro caso de uso)l.
  - **Guardar partida en SQLite**: Como se ha mencionado en uno de los casos de uso anteriores este caso de uso solo ocurrirá si se produce cualquier error al guardar los datos en el servidor de Firebase.

- **Cargar partida de SQLite**: Como se ha mencionado en uno de los casos de uso anteriores este caso de uso solo ocurrirá si la partida local y la del servidor no estan sincronizadas



- Casos de uso de las principales funcionalidades del juego:
  - **Mejorar Senado**: Mejorar el senado consumirá recursos y siempre aumentará la población máxima de la aldea y en ciertos niveles concretos desbloquear nuevos edificios.
  - **Mejorar Edificio**: Mejorar cualquier edificio consumirá recursos, aumentará el máximo de aldeanos que se le pueden asignar, también se reinicia la producción de recursos del edificio para ajustarla al nuevo nivel y en la mayoría de casos aumentará la cantidad máxima del recurso generado por el edificio.

- **Asignar Aldeano**: Asignar un aldeano a un edificio reducirá la población actual de la aldea ya que ese aldeano se quedará en el edificio hasta que el usuario decida quitar al aldeano del edificio (asignar y desasignar son la misma acción ya que si el usuario asigna a un edificio menos aldeanos de los que hay se devolverán los restantes a la aldea), al asignar un aldeano se reiniciará la producción del edificio para recalcular los recursos generados según los aldeanos asignados.
- **Partida de Caza**: Para realizar una partida de caza el usuario debe asignar varios aldeanos a esta tarea en su propio menú y durante 10 segundos varios de esos aldeanos pueden morir y los que no mueran traerán toda la comida (producción de un recurso), si todos han muerto no traen comida.
- **Generación de Aldeanos**: Mientras la aldea tenga comida esta se irá consumiendo para generar nuevos aldeanos hasta cierto máximo que dependerá del nivel del Senado.
- **Comprar Recurso**: Cuando el usuario desbloquee el mercado podrá comprar ciertos recursos a cambio de oro para acelerar la obtención de recursos.
- **Incursión**: Una incursión es un ataque a otro jugador que el usuario podrá realizar cada 10 minutos una vez desbloqueado el Castillo, en caso de que el usuario mande más soldados a atacar de los que el usuario atacado tiene asignados a su defensa este ganara, y viceversa. Si el usuario atacante pierde perderá todos los soldados enviados y si gana; el usuario atacado perderá todas sus defensas y varios recursos que serán robados por el atacante y se enviará una notificación al usuario atacado de que ha recibido un ataque.


Desbloquear edificio
Mejorar senado
<<extend>>
<<include>>
Aumentar poblacion maxima
<<include>>
Generar aldeano
<<include>>
Consumir recurso
<<include>>
Mejorar edificio
<<extend>>
<<include>>
<<include>>
Aumentar limite de recursos
Reiniciar produccion del edificio
Aumentar limite de aldeanos asignables
Jugador
<<include>>
<<include>
Asignar aldeano
<<include>>
<<extend>>
<<include>>
Reducir poblacion actual
<<extend>>
Partida de caza
<<extend>>
Producir recurso
<<include>>
Comprar recurso
<<extend>>
Incursion
<<extend>>
<<include>>
<<include>>
Recibir notificacion
<<include>>
Llamada API FCM
<<include>>
Enviar notificacion
<<include>>
Recibir Ataque
Jugador atacado
<<include>>

5.2.- Requisitos funcionales principales: funcionalidades que debe tener la aplicación.

- **Registrar nuevo jugador**: Registrarse utilizando un correo electrónico y una contraseña para iniciar sesión.
- **Iniciar sesión**: Usar el correo y contraseña guardados en el registro para acceder a nuestra aldea.
- **Cerrar sesión**: Desvincular la cuenta de usuario y todo lo relacionado con ella (notificaciones, partida guardada, etc.) del dispositivo.
- **Cargar partida**: Cargar todos los datos del progreso del jugador cuando entre al juego.
- **Guardar partida**: Guardar los datos del progreso del jugador en la base de datos, en caso de que se pierda la conexión a internet se usará una base de datos local que se sincronizará posteriormente al cargar la partida.
- **Construir Edificios**: Permitir al jugador construir nuevos edificios mejorando el Senado para la producción de recursos.
- **Mejorar Edificios**: Permitir al jugador mejorar los edificios existentes para aumentar la eficiencia y capacidad de producción.
- **Asignar Aldeanos**: Permitir al jugador asignar aldeanos a diferentes tareas y edificios para aumentar la producción.
- **Mercado**: Permitir al jugador comprar recursos en el mercado.
- **Partidas de Caza**: Permitir al jugador enviar aldeanos a cazar y recolectar comida.
- **Incursiones**: Permitir al jugador enviar guerreros a atacar a otros jugadores y recolectar botines.
- **Mejorar Senado**: Permitir al jugador mejorar el senado para aumentar la capacidad de aldeanos, desbloquear nuevos edificios y funcionalidades.
- **Asignar Tareas desde el Senado**: Gestionar y asignar aldeanos a diferentes edificios y tareas desde el senado.
- **Generación de recursos**: Los edificios con aldeanos asignados generarán ciertos recursos para que avance la partida.
- **Establecer Defensas**: Permitir al jugador asignar guerreros al castillo para defender la aldea de ataques.
- **Ranking**: Permitir al jugador ver su posición en el ranking global.
- **Navegar por las Pantallas**: Permitir al jugador moverse entre las diferentes pantallas del juego (Aldea, Mercado, Senado, Expediciones, Ranking).
- **Configuración**: Permitir al jugador modificar configuraciones del juego (sonido, etc.).
- **Tutorial**: Mostrar al usuario mensajes de ayuda según va avanzando en el juego y desbloqueando nuevas funcionalidades.

5.3.- Requisitos no funcionales: rendimiento, seguridad, usabilidad, etc.

- **Rendimiento**:
  - El sistema debe ofrecer un tiempo de respuesta medio inferior a 1 segundo para las operaciones básicas (como la construcción, mejora de edificios y la recolección de recursos).
  - La aplicación debe manejar simultáneamente un alto número de usuarios concurrentes sin degradar significativamente el rendimiento.
- **Seguridad**:
  - El sistema debe garantizar la autenticidad, integridad y confidencialidad de los datos de los usuarios.
  - Se debe implementar autenticación segura para el acceso de los usuarios, evitando accesos no autenticados.
  - El sistema debe registrar todas las acciones importantes realizadas por los usuarios para asegurar la trazabilidad.
  - Los datos sensibles deben ser encriptados tanto en tránsito como en reposo.
- **Usabilidad**:
  - La interfaz de usuario debe ser intuitiva y fácil de usar, permitiendo a los jugadores realizar acciones de manera eficiente.
  - El diseño debe ser responsive, asegurando una experiencia de usuario consistente en dispositivos móviles..
  - Se debe proporcionar una guía o tutorial inicial para ayudar a los nuevos jugadores a familiarizarse con las mecánicas del juego.
- **Disponibilidad**:
  - El sistema debe estar disponible al menos el 99.9% del tiempo, asegurando que los jugadores puedan acceder al juego en cualquier momento.
  - Se deben implementar mecanismos de recuperación ante fallos para minimizar el tiempo de inactividad en caso de problemas técnicos.
  - Funciones de recuperación de los datos y progreso en caso de fallo o cierres inesperados.
- **Escalabilidad**:
  - La arquitectura del sistema debe ser escalable para manejar un crecimiento en el número de usuarios y en la cantidad de datos generados sin pérdida de rendimiento.
  - El sistema debe permitir la adición de nuevas funcionalidades y expansiones de manera modular y eficiente.
- **Mantenimiento:**
  - El código del sistema debe estar bien documentado para facilitar su mantenimiento y la incorporación de futuras mejoras.
  - Se deben establecer procedimientos de monitoreo y actualización regulares para asegurar que el sistema permanezca seguro y eficiente.

- **Compatibilidad**:
  - La aplicación es compatible con el 95.4% de los dispositivos haciendo uso de la API Android 8.
  - Se debe garantizar el correcto funcionamiento en dispositivos con diferentes especificaciones de hardware y software, cumpliendo las especificaciones mínimas del SDK 26 en adelante.

5.4.- Descripción de los usuarios y sus necesidades.

Administradores: Gestionan los distintos sistemas a los que se conecta el juego desde la web de Firebase (base de datos, envío de notificaciones, etc.), pero no en la propia aplicación.

- **Necesidades:**
  - Mantenimiento y monitoreo del sistema: Asegurar que las funcionalidades del juego, como la autenticación, base de datos y mensajería, funcionan correctamente.
  - Gestión de usuarios y datos: Verificar registros y sesiones de los jugadores, gestionar datos en Firestore y Realtime Database.
  - Supervisión de notificaciones: Gestionar y enviar notificaciones a los jugadores.
  - Optimización del sistema: Mejorar la eficiencia y seguridad de las bases de datos y otros servicios.
- **Gestión en Firebase:**
  - Autenticación (Authentication):
    - Supervisar y gestionar los registros y autenticaciones de los jugadores.
    - Asegurar la seguridad de las credenciales y gestionar accesos indebidos o cuentas comprometidas.
  - Firestore Database:
    - Gestionar y monitorear la base de datos para asegurar la integridad y disponibilidad de los datos de los jugadores.
  - Realtime Database:
    - Supervisar y administrar las sincronizaciones en tiempo real para asegurar que los datos se actualizan correctamente y de manera eficiente.
  - Cloud Messaging:
    - Configurar y enviar mensajes de notificación a los jugadores para mantenerlos informados sobre eventos, actualizaciones y alertas de seguridad.
    - Gestionar campañas de notificaciones para involucrar a los jugadores y mantenerlos activos en el juego.

Usuarios:

Estos solo tienen acceso a las funcionalidades de la app.

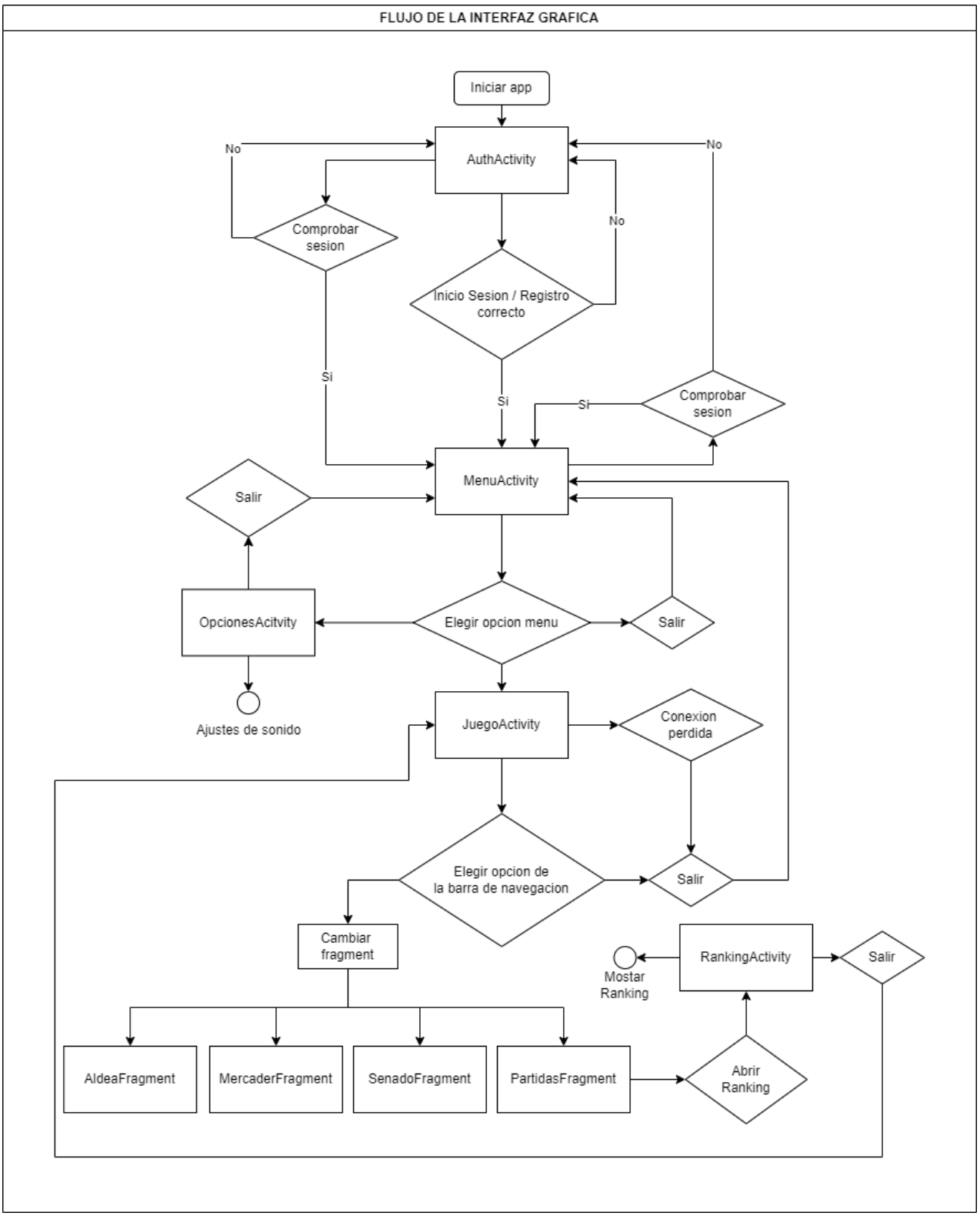
- **Necesidades:**
  - Registro y autenticación: Registrar e iniciar sesión en el juego con sus credenciales.
  - Gestión de progreso: Guardar y cargar su partida, ya sea en Firebase a través del servidor del juego o localmente en SQLite.
  - Notificaciones: Recibir alertas sobre eventos importantes, como ataques de otros jugadores.
  - Mejoras y gestión: Mejorar el Senado y otros edificios, asignar aldeanos a tareas específicas.
  - Interacción con el juego: Navegar por diferentes pantallas del juego, comprar recursos en el mercado, realizar incursiones y participar en partidas de caza.
  - Configuración: Modificar opciones del juego y recibir tutoriales de nuevas funcionalidades.

  Acceso indirecto a Firebase a través del juego. Los jugadores no interactúan directamente con Firebase; en cambio, sus acciones en el juego son gestionadas por la app, que interactúa con Firebase en su nombre.

  - Autenticación (Authentication):
    - Registrar e iniciar sesión en el juego utilizando credenciales seguras.
    - Mantener la sesión activa para facilitar el acceso continuo al juego sin necesidad de autenticarse repetidamente.
  - Gestión de progreso:
    - Guardar y cargar el progreso del juego en el servidor del juego, que a su vez utiliza Firebase para almacenar y sincronizar los datos.
    - Sincronizar datos locales en SQLite con el servidor cuando se restablece la conexión a Internet.
  - Mensajería:
    - Recibir notificaciones sobre eventos importantes, como ataques.

6.- Diseño de la aplicación

6.1.- Mockups o wireframes o prototipos de la interfaz gráfica de usuario.



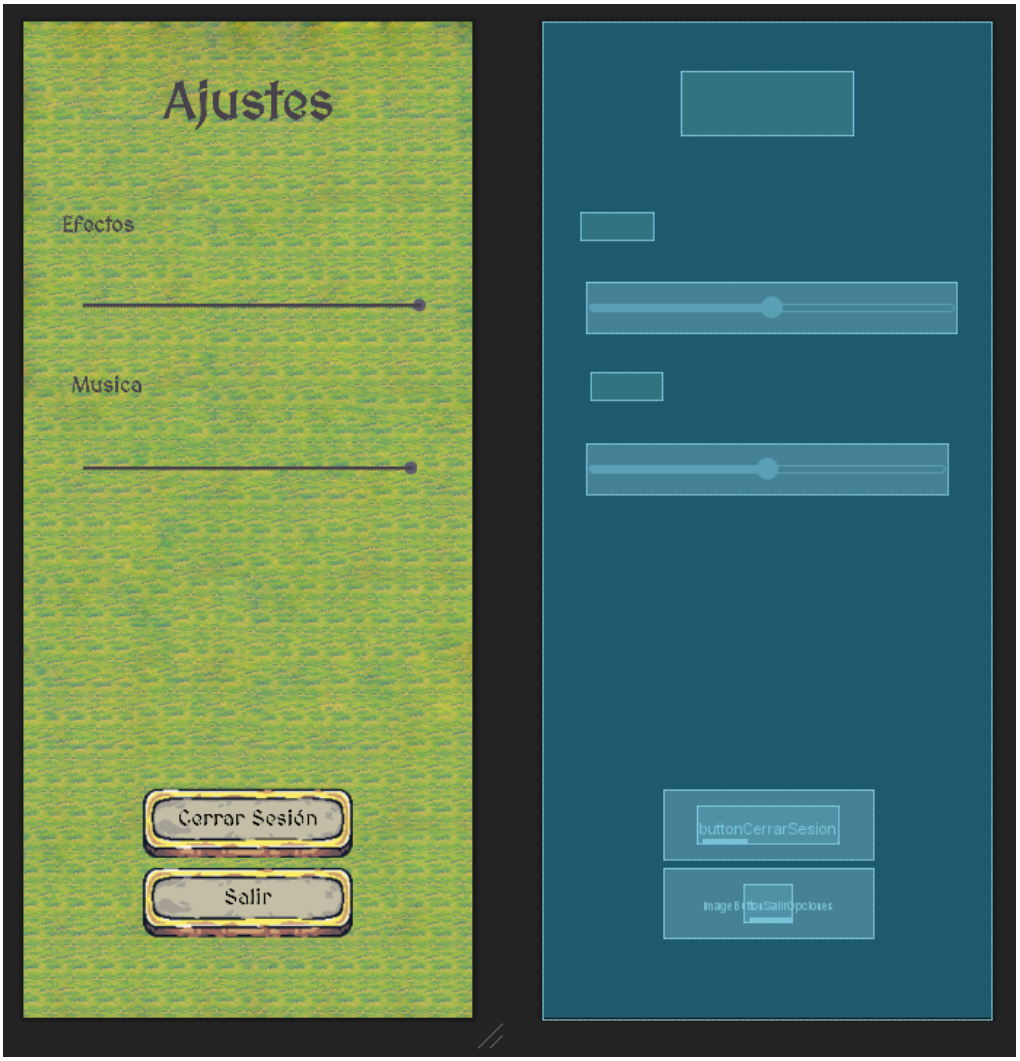
- **AuthActivity**:

  - Es el punto de entrada de la aplicación y en caso de que haya una sesión guardada redirigirá automáticamente al usuario a OpcionesActivity.
  - El botón de registro y el botón de iniciar sesión llevarán al usuario a MenuActivity si las credenciales introducidas son correctas.

- **MenuActivity**:

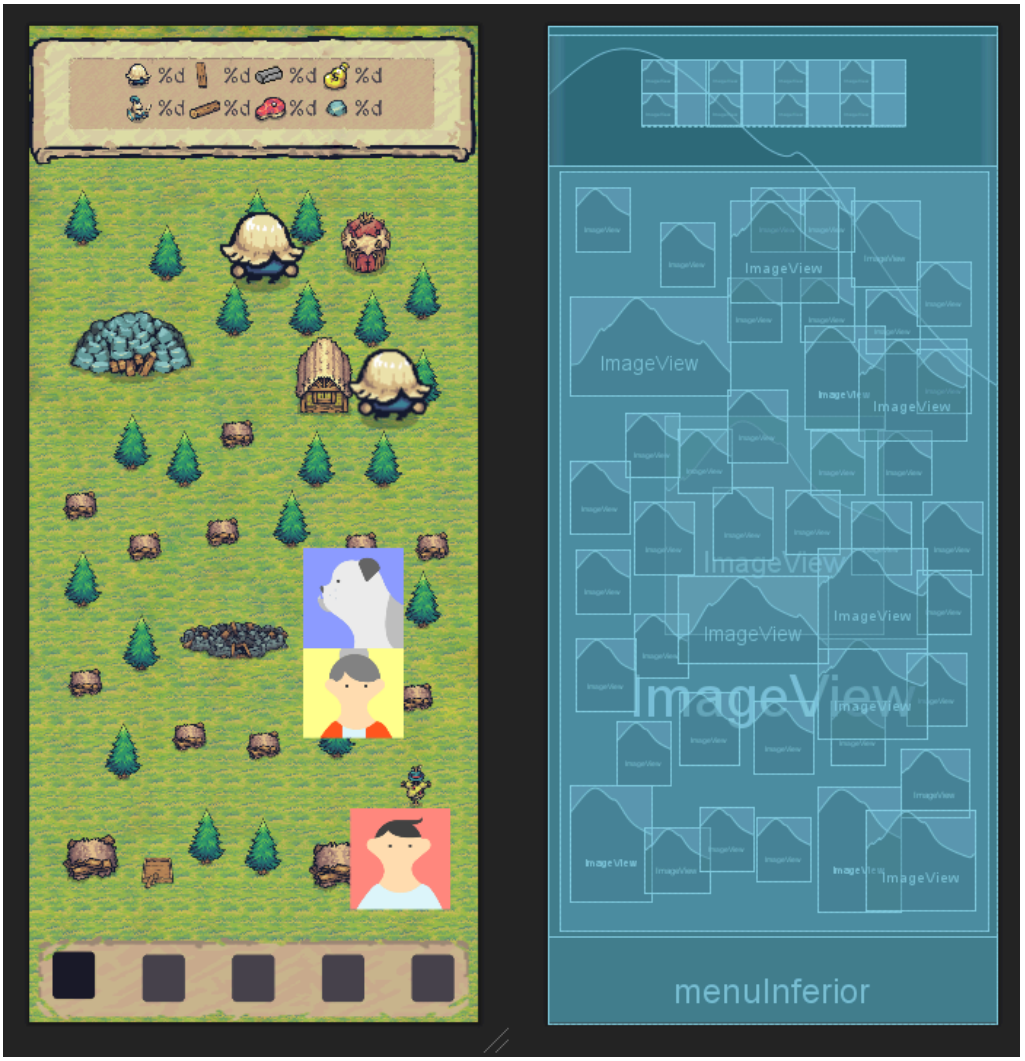
  - El botón de jugar llevará al usuario a JuegoActivity.
  - El botón de opciones llevará al usuario a OpcionesActivity.
  - El botón salir cierra la aplicación.

- **OpcionesActivity**:



  - Las seekbars de efectos y música permitirán al usuario regular sus respectivos volúmenes.
  - El botón de cerrar sesión cerrará la sesión actual y finaliza la activity volviendo a MenuActivity, la cual se finalizará al comprobar que no hay ninguna sesión activa redirigiendo al usuario a AuthActivity.

- **JuegoActivity**:



  - Al iniciar la activity se mostrará una pantalla de carga hasta que el juego haya terminado de cargar todo lo necesario.
  - Los imageviews se irán actualizando según el usuario vaya desbloqueando nuevos edificios y progresando en el juego para representar el estado actual de la aldea.
  - Esta activity contiene un layout que puede mostrar diferentes fragments según el botón pulsado del menú inferior:
    - El primer botón muestra el AldeaFragment.
    - El segundo botón muestra el MercaderFragment.
    - El tercer botón muestra el SenadoFragment.
    - El primer botón muestra el PartidasFragment.
  - El quinto botón finalizara la activity volviendo a MenuActivity.
  - Si se pierde la conexión a internet se finalizara la activity volviendo a MenuActivity.
  - Esta activity es la que se encarga de cargar y guardar los datos del progreso del usuario manejando correctamente todos los posibles estados de la activity (finalización y suspensión).

- **AldeaFragment**:
  - No hay imagen por que simplemente es un fragment vacío para mostar la aldea que se encuentra en JuegoActivity.
- **MercaderFragment**:

  - Hasta que el senado no esté a nivel 8 solo aparecerá un texto para indicar al usuario que debe subir a nivel 8 para usar el mercado.
  - Permite al usuario comprar distintos recursos gastando oro.

- **SenadoFragment**:

  - Permite al usuario mejorar el senado y los edificios y asignar aldeanos en los edificios que lo permitan.
  - Sólo se mostrarán los edificios desbloqueados.
  - Contiene una ScrollView para asegurar que si no hay espacio suficiente para todos los elementos se pueda navegar por la interfaz correctamente.

- **PartidasFragment**:

  - Hasta que el senado no esté a nivel 4 el menú de incursiones no estará disponible, y se mostrará un texto explicando qué se debe desbloquear el castillo para acceder a esta funcionalidad.
  - Permite iniciar una partida de caza con varios aldeanos que durará 10 segundos, y si no mueren todos traen comida.
  - Permite iniciar un ataque con varios aldeanos a otro jugador cada 10 minutos, lo que permitirá al usuario obtener puntos si gana el ataque y perderlos en caso de derrota.
  - El botón de ranking llevará al usuario a RankingActivity.

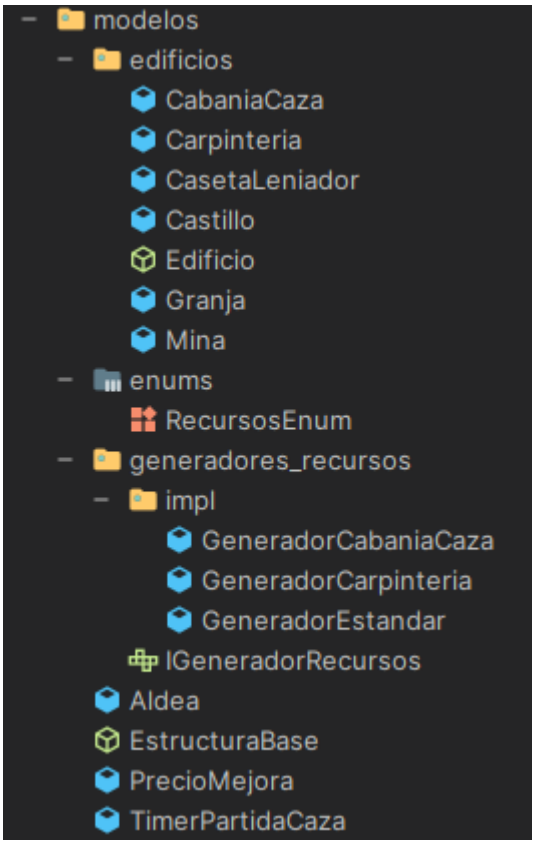
- **RankingActivity**:

  - Muestra la puntuación del usuario y el top 10 mundial.
  - El botón de salir finaliza la activity volviendo a JuegoActivity.

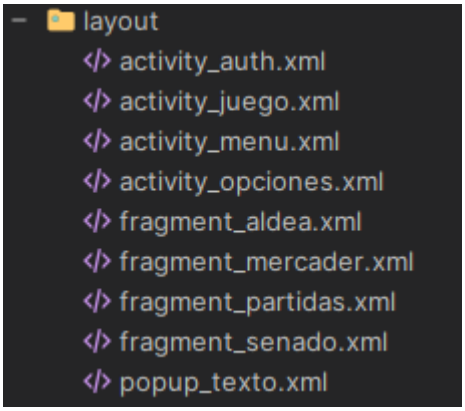
6.2.- Arquitectura del sistema.

- **Arquitectura cliente-servidor:** La implementación de esta arquitectura en el proyecto se realiza durante las peticiones desde el dispositivo android (cliente) a los distintos sistemas de firebase (servidor).

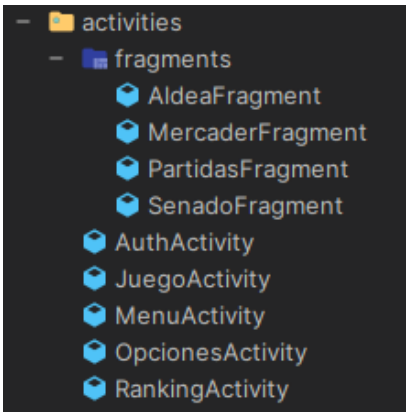
- **Arquitectura MVC (Modelo Vista Controlador):** Esta arquitectura se basa en separar la aplicación en tres capas distintas: modelo de datos, interfaz de usuario, y la lógica de las interacciones entre los modelos y la interfaz.
    - Modelos: En este caso los modelos son todas las entidades que representan las bases del juego y las funcionalidades principales. La mayor parte de la lógica de ejecución del juego se encuentra en esta parte.

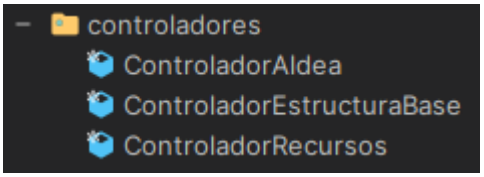


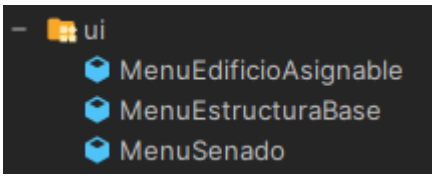
    - Vistas: En cualquier proyecto de Android Studio las vistas son representadas mediante layouts xml que representan la interfaz de usuario.



- Controladores: Los controladores son las clases de tipo Activity y Fragment que van vinculadas a los distintos layouts xml, que permiten gestionar todas las entradas del usuario y mostrar correctamente los datos según sea necesario y además hemos creado controladores personalizados para realizar un correcto tratamiento y validaciones de los datos modificados por el usuario para asegurar un correcto funcionamiento de la aplicación.





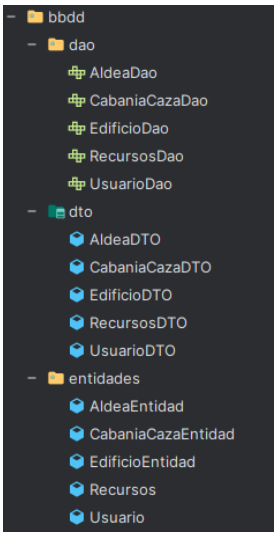


- **Patron Singleton:** Este patrón de diseño se basa en que las clases que lo implementen solo podrán tener una única instancia en el código. Lo hemos utilizado para las clases Aldea y SoundManager. Ejemplo del patrón singleton:
  - Singleton Aldea:

```
public class Aldea extends EstructuraBase implements
Runnable {
   // Campo estático para almacenar la instancia única
   private static Aldea instance;
   private Aldea() {
       // Constructor privado para evitar multiples
instancias
   }
   // Método estático para obtener la instancia única de
Aldea
   public static Aldea getInstance() {
       if (instance == null)instance = new Aldea();
       return instance;
   }
  // El resto del codigo...
```

```
}
```

- **Patrón DAO (Data Acces Object):** Este patrón se basa en la utilización de un ORM (Object Relational Mapping) que consiste en realizar todas las operaciones de la base de datos utilizando directamente los objetos del codigo para tener una capa de abstracción en las operaciones realizadas en la base de datos y ofrecer una mayor seguridad contra inyecciones SQL. En nuestro caso lo hemos utilizado para todas las operaciones realizadas sobre la base de datos local SQLite y como ORM hemos utilizado Room Persistence Library que es la librería oficial de google para SQLite en Android. Y para la base de datos de Firebase al ser de tipo documental simplemente se realizan las operaciones mediante DTOs (Data Transfer Objects) para gestionar la información de manera más sencilla. Ejemplo de DAO usando Room Persistence Library:
    - Estructura de paquetes:



    - Mapeo de tablas mediante anotaciones:

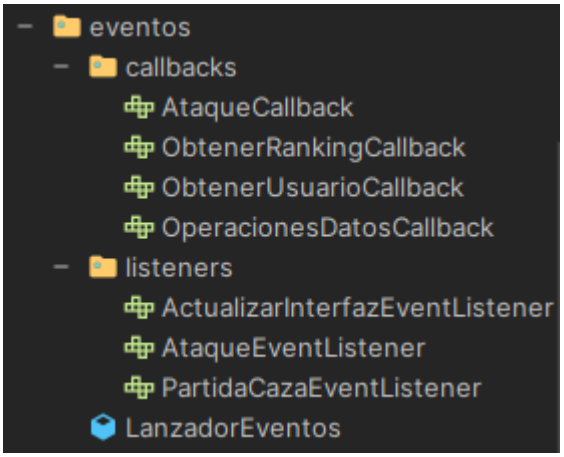
```
@Data
@Entity(tableName = "datos_aldea", foreignKeys =
@ForeignKey(
                entity = Usuario.class,parentColumns =
"email",
                childColumns = "usuario",
                onDelete = ForeignKey.CASCADE)
)
public class AldeaEntidad {
   @PrimaryKey @NonNull
   private String usuario;
   @ColumnInfo(name = "nivel") @NonNull
   private Integer nivel;
   @ColumnInfo(name = "poblacion") @NonNull
   private Integer poblacion;
}
```

- Interfaz DAO (Room Persistence library la implementará automáticamente en tiempo de compilación basándose en las querys):

```
@Dao
public interface AldeaDao {
  @Insert(onConflict = OnConflictStrategy.REPLACE)
  void insertar(AldeaEntidad aldeaEntidad);

  @Query("SELECT * FROM datos_aldea WHERE usuario =
:usuario")
  AldeaEntidad getAldeaByEmail(String usuario);
}
```

- **Patrón Observer y Programación Orientada a Eventos:** El patrón observer consiste en tener dos tipos de clases: los observadores y los observables, los observables tienen una lista de observers y en determinadas situaciones de la aplicación notificarán a todos los observers de la lista y estos implementaran la funcionalidad deseada según el tipo de evento. En nuestro caso hemos utilizado distintos tipos de event listeners para implementar este patrón y además también hemos creado eventos de tipo callback para asegurar la integridad de los datos en caso de llamadas asíncronas a la base de datos. Ejemplos:
  - Eventos y Callbacks de la aplicación:



- Observable (LanzadorEventos):

```
public class LanzadorEventos <T extends EventListener> {
    // Se usa un set y no una lista para evitar duplicados
    protected Set<T> listeners = new HashSet<>();

    public void addEventListener(T listener) {
        listeners.add(listener);
    }

    public void removeEventListener(T listener) {
        listeners.remove(listener);
    }

    // Consumer<T> permite pasar cualquier funcion que
    // del listener de tipo T, pero no de otros listeners
    public void lanzarEvento(Consumer<T> action) {
        listeners.forEach(action);
    }
}
```

- EventListener (Sería el Observer, deben ser implementados por la clase que se desee y esa clase se debe añadir a la lista de listeners de del lanzador de eventos correspondiente):

```
public interface AtaqueEventListener extends EventListener {

    void onAtaqueTerminado(boolean victoria);

    void onError(Exception e);

}
```

- CallBack (Los callbacks deben ser implementados por la clase que llama a la función y pasados como parámetro para asegurar que la implementación no se ejecuta hasta que se han realizado todas las operaciones deseadas) :

```
public interface OperacionesDatosCallback {

    void onDatosCargados();

    void onDatosGuardados();

}
```

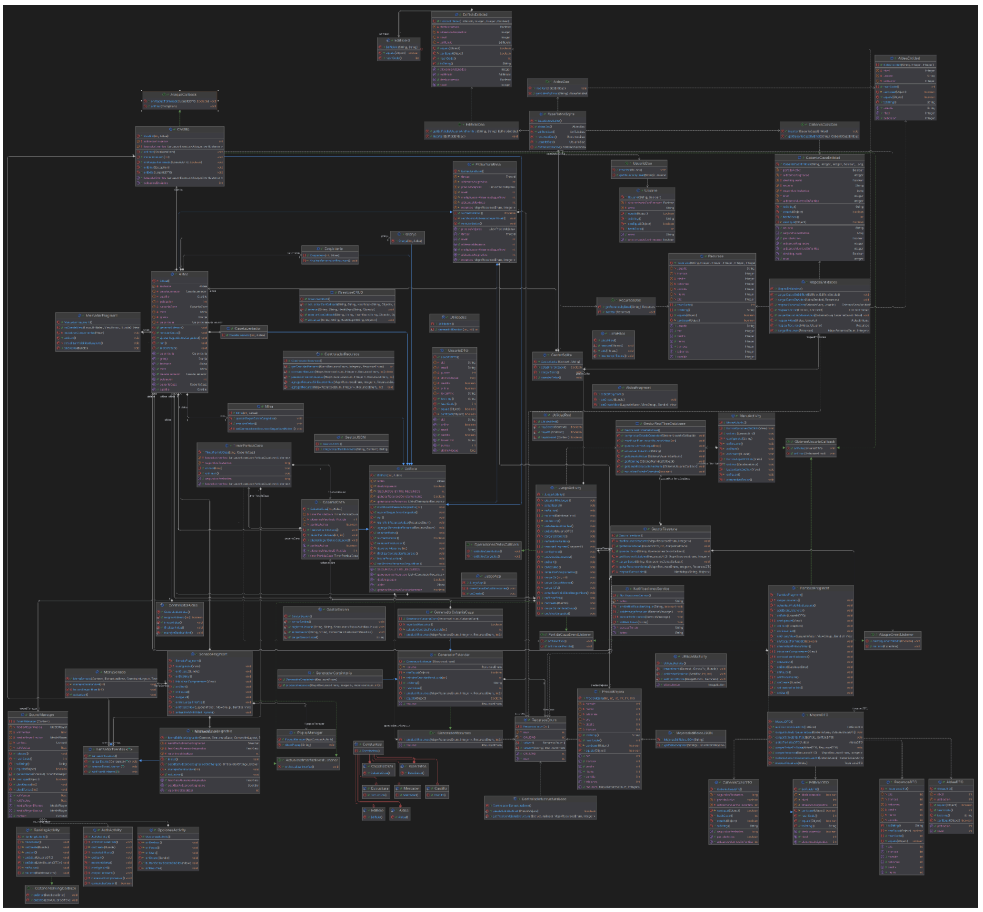
```
public static void actualizarConCallback(String coleccion,
String documento, HashMap<String, Object> datos,
OperacionesDatosCallback callback) {

    actualizar(coleccion, documento, datos);
```

```
    callback.onDatosGuardados();
}
```

6.3.- Diagramas de clases y de entidad-relación.

Diagrama General:



Versión en alta resolución para mejor visualización.

Diagrama de la Aldea y los Edificios:

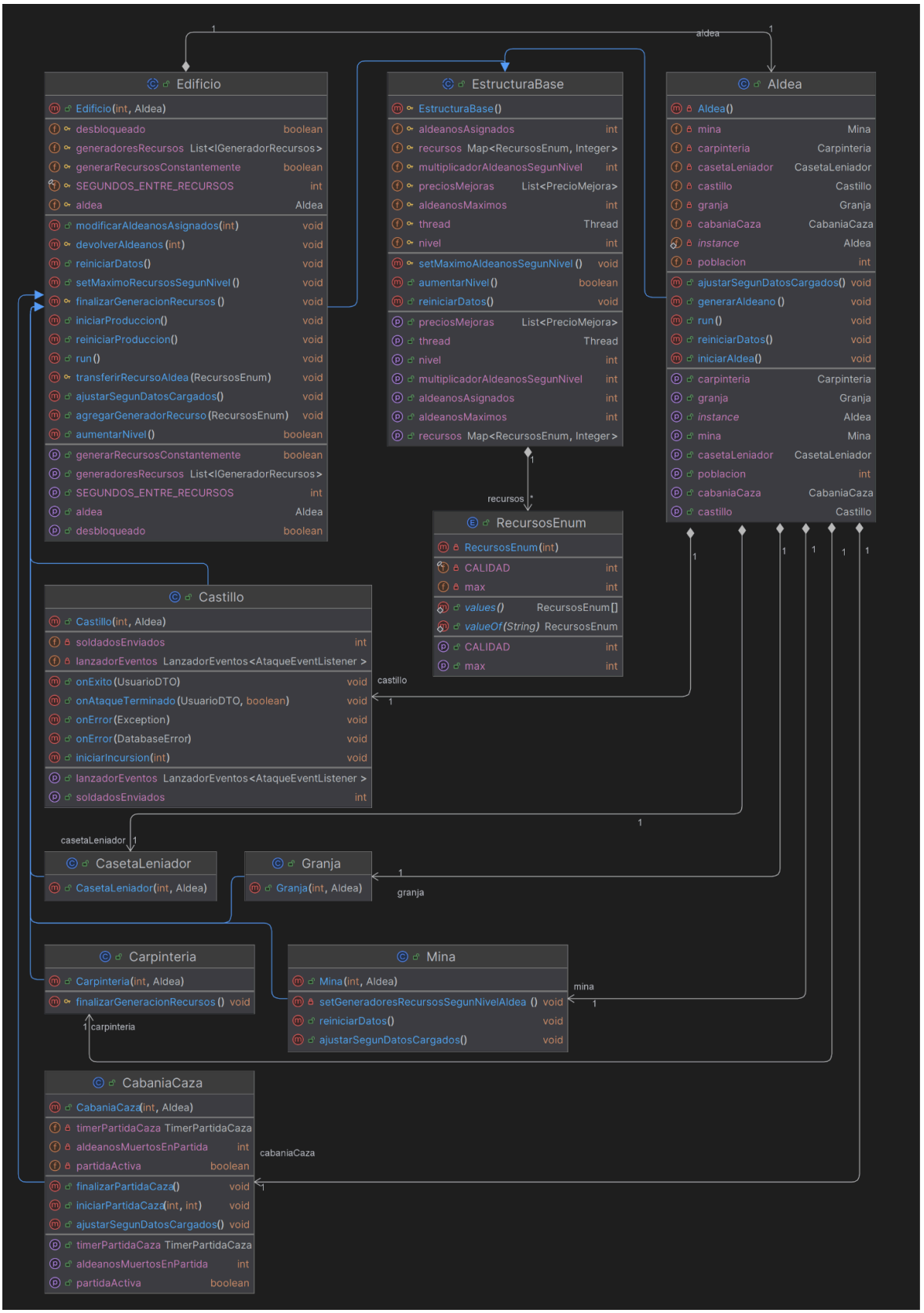


Diagrama de Eventos y Callbacks:

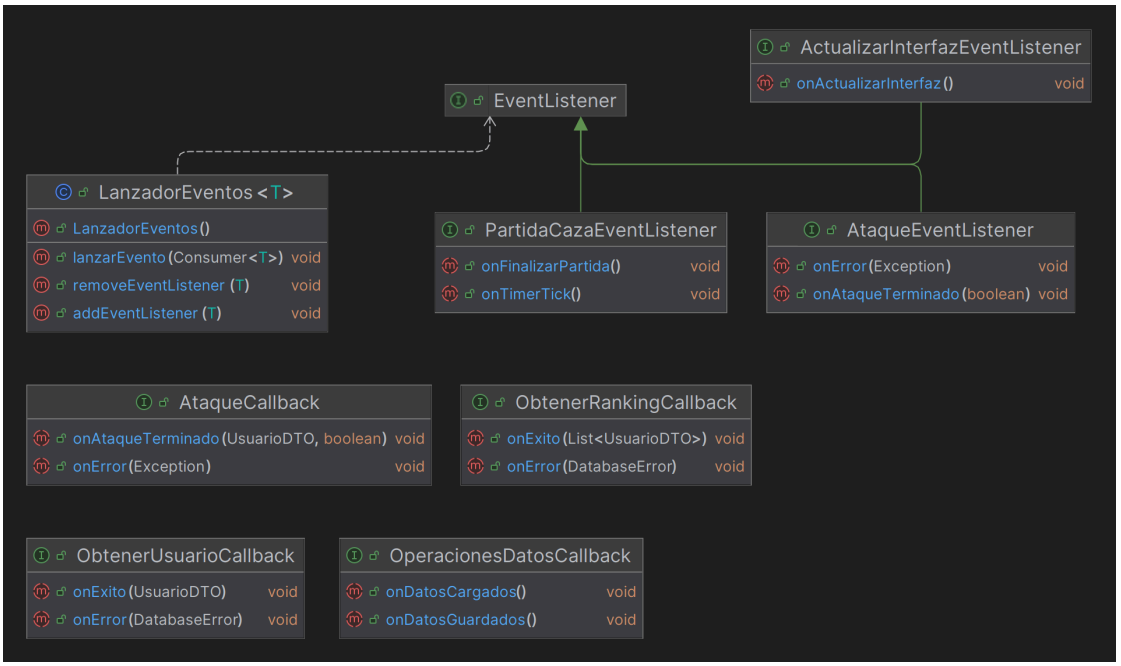
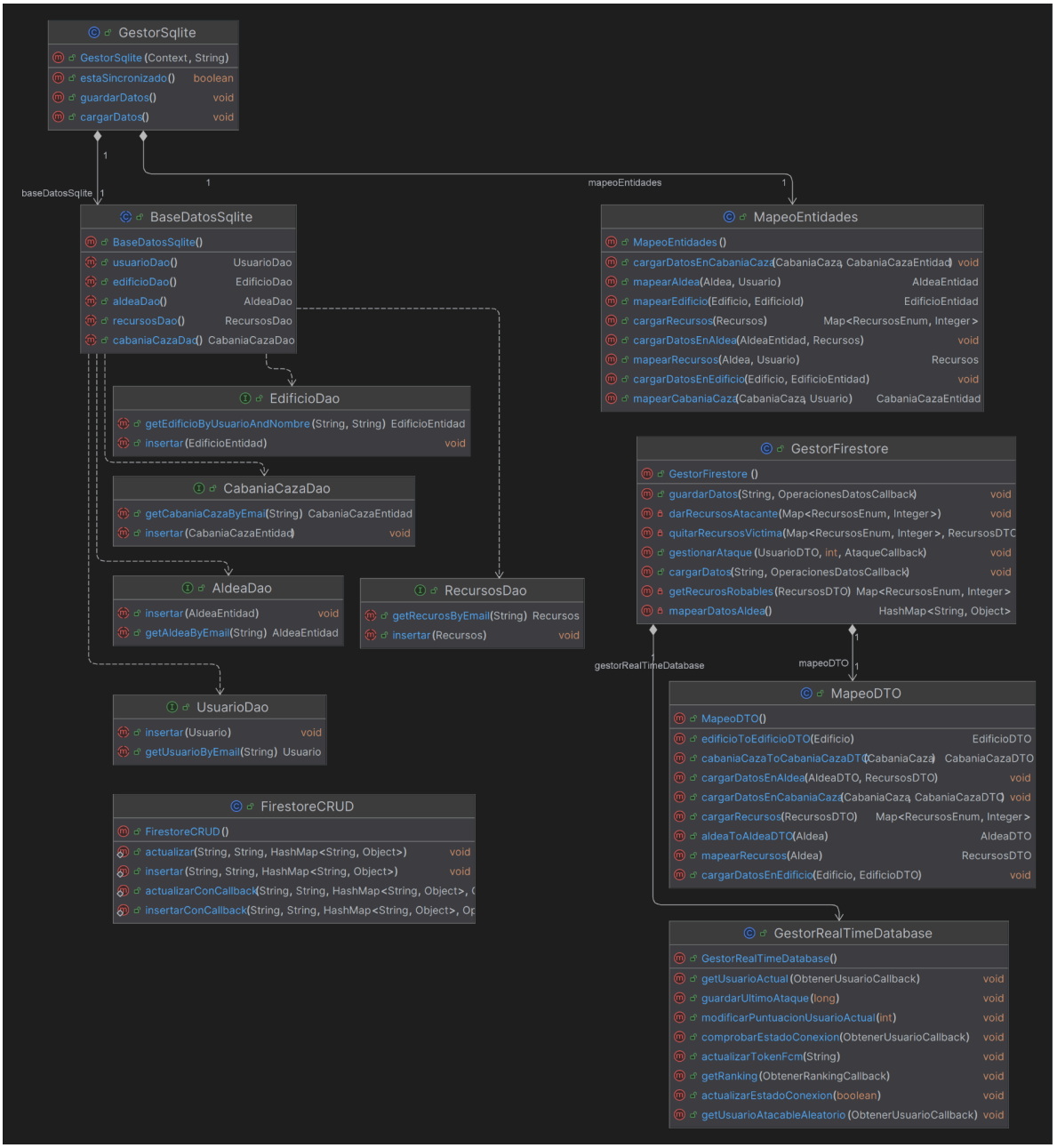
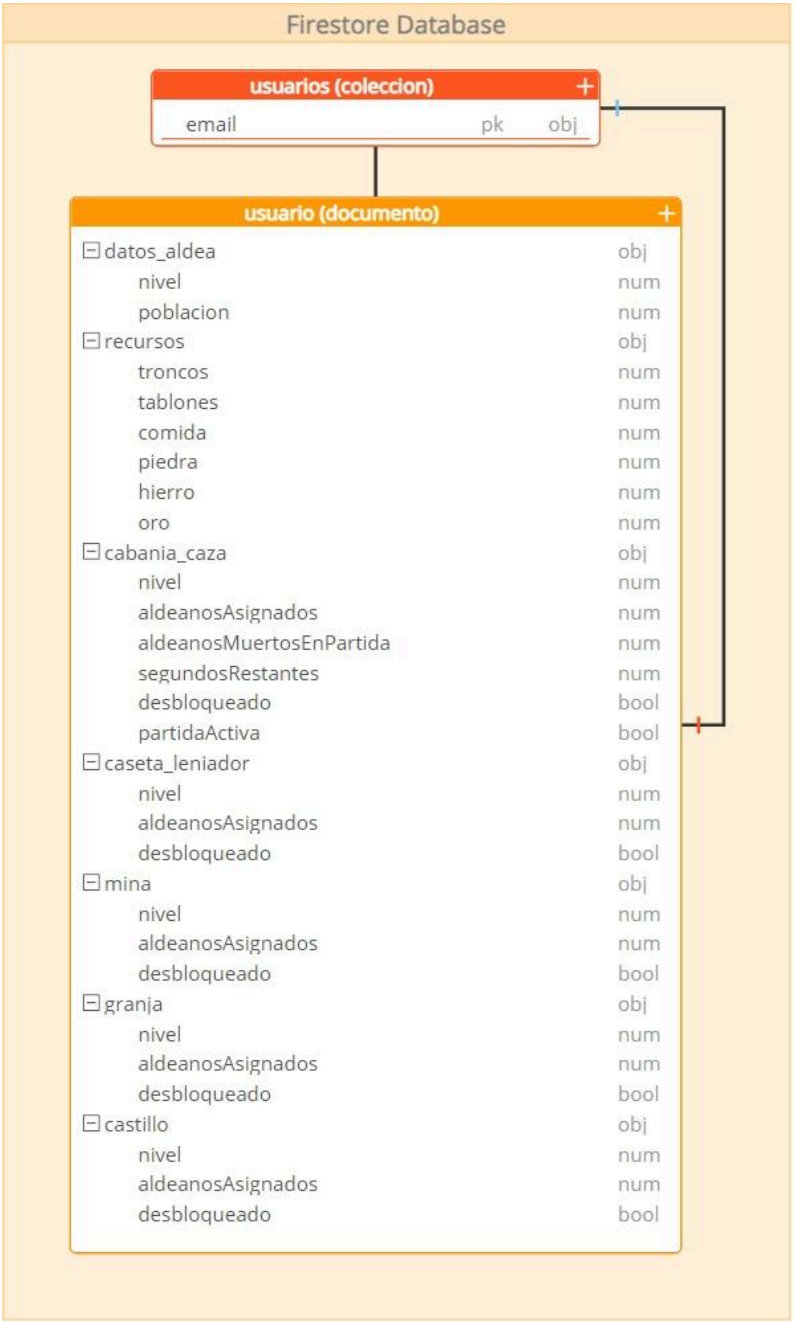


Diagrama de las Conexiones de Bases de Datos:



6.4.- Diseño de la base de datos: esquemas y tablas.

- **Firestore Database (Servidor de Firebase):** base de datos documental en la que se almacenan los datos del progreso de los jugadores en el juego, en la colección usuarios se guarda un documento asociado al email de cada usuario.

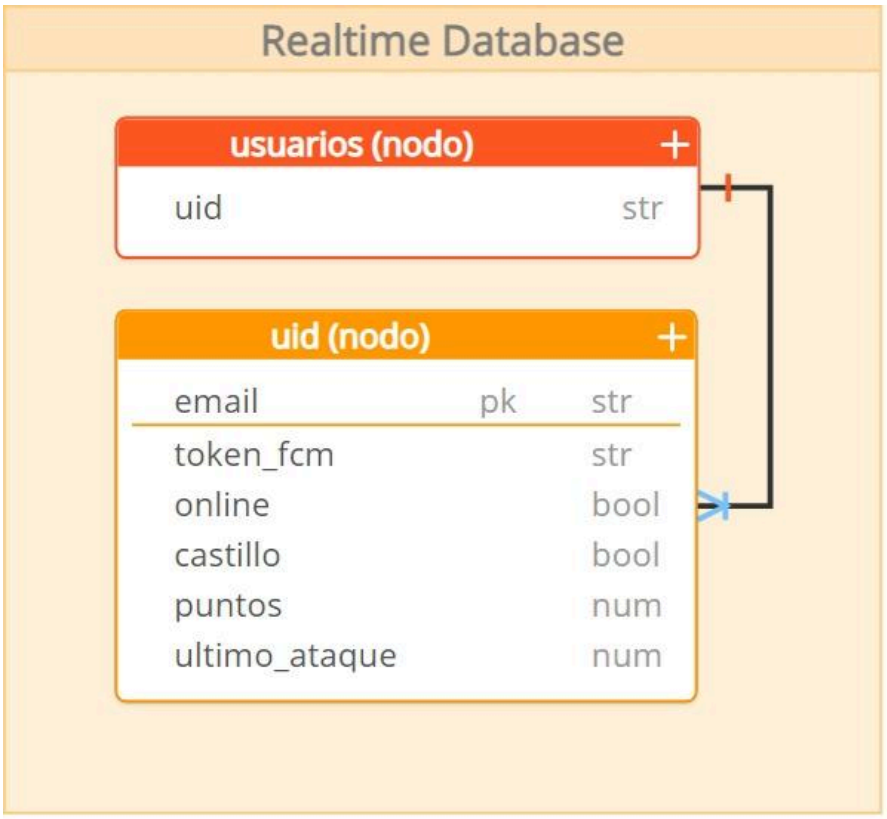


- Reglas de seguridad de Firestore:

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      // Permitir que cualquier usuario autenticado pueda leer y escribir
      allow read, write: if request.auth.uid != null;
    }
  }
}
```

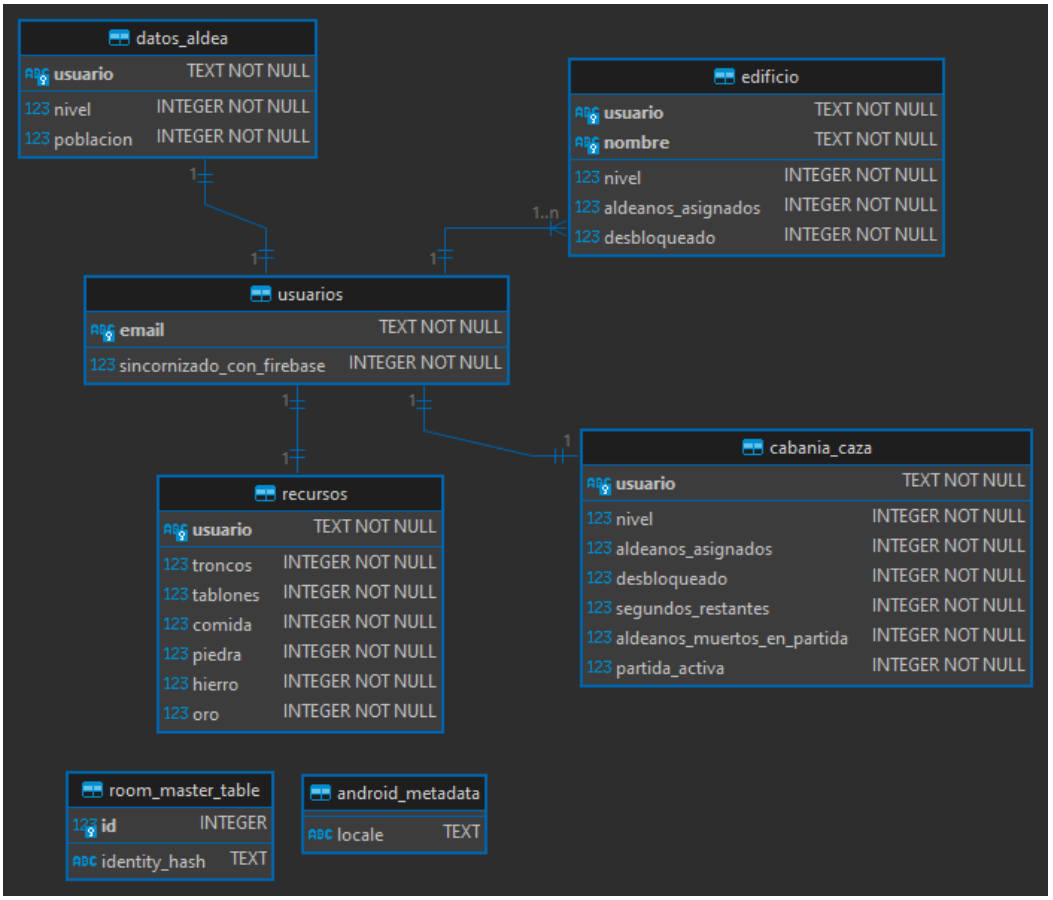
- **Realtime Database (Servidor de Firebase):** base de datos basada en nodos que almacena la información necesaria para los ataques entre jugadores (email, estado de la conexión y castillo desbloqueado) en un nodo asociado al uid del usuario y se utiliza para estos datos concretos por que permite manejar modificaciones de datos cuando la app termina de una forma no deseada (pérdida de conexión, crasheo, etc.).



- Reglas de seguridad de Realtime Database:

```
{
  // Los datos de un usuario solo podran ser modificados
  // por el propio usuario, pero cualquier usuario autenticado podra leer
  "rules": {
    "usuarios": {
      ".read": "auth != null",
      "$uid": {
        ".write": "auth != null && auth.uid === $uid"
      }
    }
  }
}
```

- **SQLite 3 (Almacenamiento Local):** base de datos relacional de tipo SQL que almacena la información del progreso del usuario de manera local en caso de que se pierda la conexión, para posteriormente sincronizar los datos con Firestore cuando se recupere la conexión.



Android genera automáticamente room_master_table y android_metadata.

7.- Desarrollo de la aplicación

7.1.- Tecnologías y herramientas utilizadas (lenguajes de programación, frameworks, bases de datos, etc.).

- **Java**: Lenguaje de programación orientado a objetos y con características de programación funcional, hemos decidido usar este lenguaje porque es el que más hemos trabajado durante el curso y por eso mismo tenemos más dominio de él que de otros lenguajes que hemos estudiado durante el curso.
- **Android Studio**: Hemos optado por usar este entorno para desarrollar el proyecto al ser una de las herramientas más sencillas para desarrollar en Android ya que es la herramienta oficial proporcionada por sus desarrolladores y además es una herramienta que hemos aprendido durante el curso en la asignatura de "Programación multimedia y dispositivos móviles", y otro motivo más es que permite programar en Java, que como ya hemos mencionado anteriormente es el lenguaje que mejor conocemos.
- **FireBase**: Plataforma de google que permite gestionar de manera sencilla la autenticación de usuarios, el acceso a sus datos mediante distintos tipos de bases de datos y el envío de notificaciones y además al ser de google su integración con android es muy fácil de implementar.
    - Firestore: Base de datos de tipo documental (las cuales aprendimos a manejar en la asignatura de "Acceso a datos"), la hemos utilizado para almacenar los datos del progreso de los jugadores.
    - Firebase Realtime Database: Base de datos basada en nodos que facilita guardar ciertos datos en caso de que la aplicación se cierre o desconecte de una forma no deseada, en este caso almacenamos datos relacionados con el estado de conexión del usuario, el token de notificaciones y si puede o no ser atacado por otros jugadores.
    - Firebase Authentication: Sistema de autenticación de fácil implementación que permite tanto autenticación mediante email y contraseña como con cuentas de terceros (Google, Facebook, Twitter, Microsoft, etc.) en nuestro caso hemos optado por usar la autenticación por email y contraseña, además ofrece una seguridad robusta al no permitir ver las contraseñas de los usuarios desde la consola de firebase, y también permite gestionar la sesión a nivel local para que el usuario no tenga que introducir la contraseña cada vez que entra a la aplicación.
    - Firebase Cloud Messaging: Sistema que permite programar notificaciones para distintos segmentos de usuarios de la consola de Firebase y que además permite el envío de notificaciones en reacción a ciertas acciones del usuario (en este caso un ataque a otro jugador) mediante el uso de una API. Y a nivel local permite mantener las notificaciones siempre activas mediante un servicio que se ejecuta en segundo plano y un token de notificaciones asociado al dispositivo.
- **SQLite 3**: Base datos de tipo relacional y SQL que tiene ciertas limitaciones pero que permite almacenar los datos de forma muy ligera, lo cual lo hace ideal para guardar datos de manera local en un dispositivo móvil, en el caso de esta aplicación lo hemos utilizado para guardar los datos de progreso del jugador de manera local en caso de que se pierda la conexión a internet para sincronizarlo posteriormente con Firebase.
- **Gradle**: Gestor de dependencias por defecto de Android Studio, que se encarga de compilar la aplicación y permite añadir librerías.

- **DBeaver**: Cliente de gráfico para sistemas de bases de datos, utilizado para ver la información de las tablas de la base de datos local SQLite y generar el diagrama de la misma.
- **Lombok**: Librería para java que permite automatizar tareas como la creación de constructores, getters, y setters mediante anotaciones en tiempo de compilación para mantener un código más limpio y legible. (requiere de instalar un plugin en Android Studio además de añadirla en Gradle)
- **Librerías de FireBase**: Todas las librerías necesarias para acceder a FireBase desde una app android en java:
    - firebase-bom: Dependencia base de Firebase.
    - firebase-analytics: Dependencia para enviar distintos tipos de analíticas al servidor de Firebase.
    - firebase-auth: Dependencia para implementar el sistema de autenticación.
    - firebase-firestore: Dependencia para manejar la conexión a Firestore
    - firebase-database: Dependencia para manejar la conexión a Realtime Database
    - firebase-messaging: Dependencia para manejar las notificaciones.
    - oauth2-http: Dependencia para manejar el token de seguridad de conexión a la api de notificaciones.
- **Glide**: Librería para poder cargar gifs y que se muestren correctamente.
- **Jackson**: Librería para la lectura y escritura de ficheros json y xml que hemos usado para guardar y cargar ciertos datos del juego.
- **Room Persistence Library**: Librería oficial de google para manejar SQLite en Android mediante el uso de un ORM y el patrón DAO, ofrece una capa de abstracción sobre todas las operaciones de la base de datos y permite su fácil manejo mediante anotaciones y creación automática de las tablas.
- **Android Core Splash Screen**: Librería para mostrar una Splash Screen personalizada al iniciar la aplicación.

7.2.- Descripción de las principales funcionalidades implementadas ilustrándolo con fragmentos de código relevantes.

- **Cargar Gifs:**

  Esta función pide 2 parámetros, el primero es la id de un ImageView y el segundo la id del gif.

```
private void cargarGif(int id, int d) {
    ImageView imageView = findViewById(id);
    Glide.with(this).asGif().fitCenter()
            .override(imageView.getWidth()*10,
imageView.getHeight()*10)
            .load(d).into(imageView);
}
```

- **Barra de navegación:**

Barra con los botones de las ventanas del juego.

Se hace de esta manera y no con un switch, por que los valores de las id se calculan en tiempo de ejecución y el switch no es capaz de manejar esto por lo que daría una excepción.

```
private final NavigationBarView.OnItemSelectedListener
 itemSelectedListener = menuItem -> {
   if (menuItem.getItemId() == R.id.aldea) {
       cambiarFragment(new AldeaFragment());
   } else if (menuItem.getItemId() == R.id.mercader) {
       cambiarFragment(new MercaderFragment());
   } else if (menuItem.getItemId() == R.id.senado) {
       cambiarFragment(new SenadoFragment());
   } else if (menuItem.getItemId() == R.id.partidas) {
       cambiarFragment(new PartidasFragment());
   } else if (menuItem.getItemId() == R.id.salir) {
       finish();
   }
   return true;
};
```

- **Hilo principal del juego:**

Este hilo se inicia en JuegoActivity y se encarga de inicializar todo lo necesario para el funcionamiento del juego y de actualizar la interfaz cada milisegundo.

```
private void ejecutarHiloJuego() {
    Context context = this;
    // Iniciar un hilo secundario para ejecutar el código
continuamente
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
              aldea.getCabaniaCaza().getTimerPartidaCaza()
                      .getLanzadorEventos().addEventListener(
                              (PartidaCazaEventListener) context);
              ControladorAldea.iniciarAldea();
              while (enEjecucion) {
                 Thread.sleep(1);
                  if (!UtilidadRed.hayInternet(context)) {
                      finish();
                  }
                    // Actualizar la interfaz al final de cada
ciclo
                    actualizarUI();
              }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

- **Compra de recursos:**

Según el botón presionado se determina que recursos se van a comprar y se hacen las comprobaciones necesarias para ver si el usuario puede comprarlo.

```
public void onClick(View v) {
        RecursosEnum recurso = null;
        int precio = 0;
        /* Determina el recurso y el precio en función del botón
        presionado */
        if (v.getId() == R.id.buttonComprarTablones) {
            recurso = RecursosEnum.TABLONES_MADERA;
            precio = Constantes.Mercader.PRECIO_TABLONES;
        } else if (v.getId() == R.id.buttonComprarTroncos) {
            recurso = RecursosEnum.TRONCOS_MADERA;
            precio = Constantes.Mercader.PRECIO_TRONCOS;
        }else if (v.getId() == R.id.buttonComprarComida) {
            recurso = RecursosEnum.COMIDA;
            precio = Constantes.Mercader.PRECIO_COMIDA;
        }else if (v.getId() == R.id.buttonComprarPiedra) {
            recurso = RecursosEnum.PIEDRA;
            precio = Constantes.Mercader.PRECIO_PIEDRA;
        }else if (v.getId() == R.id.buttonComprarHierro) {
            recurso = RecursosEnum.HIERRO;
            precio = Constantes.Mercader.PRECIO_HIERRO;
        } else {
            return;
        }
        // Realiza la compra del recurso si tiene menos del maximo
        if (ControladorRecursos.getCantidadRecurso(
                        aldea.getRecursos(), recurso) <
                        recurso.getMax())
        {
            if

(ControladorRecursos.consumirRecurso(aldea.getRecursos(),
                    RecursosEnum.ORO, precio)) {
                    ControladorRecursos.agregarRecursoSinExcederMax(
                    aldea.getRecursos(),
                    recurso, Constantes.Mercader.CANTIDAD);
            } else {
                if (getActivity() != null) {
                    Toast.makeText(getActivity(),
                    getString(R.string.msj_oro_insuficiente),
                    Toast.LENGTH_SHORT).show();
                }
            }
        }
    }
```

- **Mostrar Ranking:**

Mostrar el top 10 mundial en el callback de la base de datos.

```
@Override
public void onExito(List<UsuarioDTO> ranking) {
    textViewUsuarios.setText("");
    for (int i = 0; i < Math.min(ranking.size(), 10); i++) {
        String email = ranking.get(i).getEmail().split("@")[0];
        String puntos = String.valueOf(ranking.get(i).getPuntos());
        // Usar String.format para formatear el texto con longitud
fija
        String formattedText = String.format("%-20s %s", email,
puntos);
        // Agregar el texto al TextView
        textViewUsuarios.append(formattedText + "\n");
    }
}
```

- **Cargar Ranking de la BBDD:**

Obtiene el top 10 mundial de RealtimeDatabase y notifica al callback.

```
public void getRanking(ObtenerRankingCallback callback) {
   Query onlineUsersQuery = usuariosRef.getRef()
                        .orderByChild(PATH_PUNTOS).limitToLast(10);
   onlineUsersQuery.addListenerForSingleValueEvent(new
     ValueEventListener() {
       @Override
       public void onDataChange(@NonNull DataSnapshot
dataSnapshot) {
           List<UsuarioDTO> ranking = new ArrayList<>();
           for (DataSnapshot snapshot :
dataSnapshot.getChildren()) {
             UsuarioDTO usuarioDTO = new UsuarioDTO();
             usuarioDTO.setEmail(snapshot.child(PATH_EMAIL)
                                 .getValue(String.class));
             usuarioDTO.setPuntos(Optional.ofNullable(
             snapshot.child(PATH_PUNTOS)
                     .getValue(Integer.class)).orElse(0));
             ranking.add(usuarioDTO);
           }

           /* Invertimos la lista porque limitToLast() devuelve
los
           elementos en orden ascendente */
           Collections.reverse(ranking);
           callback.onExito(ranking);
       }

       @Override
       public void onCancelled(@NonNull DatabaseError
databaseError) {
           callback.onError(databaseError);
       }
   });
}
```

- **Asignar Aldeano al edificio:**

  Cuando un jugador modifica los aldeanos asignados desde la seekbar del menú, la cantidad que asigne es la cantidad que tendrá ahora el edificio, por eso se calcula la diferencia de aldeanos para asegurar que tanto si la cantidad es mayor o menor a la anterior se asegura que se devuelven o se quitan correctamente los aldeanos de la aldea y se comprueba también que no se pase del máximo de aldeanos del edificio.

```
public synchronized void modificarAldeanosAsignados(int
 aldeanosAsignados) {
   if (aldeanosAsignados >= 0 && aldeanosAsignados <=
aldeanosMaximos) {
       int diferenciaAldeanos;
       if (aldeanosAsignados < this.aldeanosAsignados) {
           diferenciaAldeanos =
this.aldeanosAsignados-aldeanosAsignados;
           devolverAldeanos(diferenciaAldeanos);
           this.aldeanosAsignados = aldeanosAsignados;
           reiniciarProduccion();
       } else if (aldeanosAsignados > this.aldeanosAsignados) {
           diferenciaAldeanos =
aldeanosAsignados-this.aldeanosAsignados;
           if
(ControladorAldea.asignarAldeano(diferenciaAldeanos)) {
               this.aldeanosAsignados = aldeanosAsignados;
               reiniciarProduccion();
           }
       }
   }
}
```

- **Generar Aldeano:**

  La aldea generará un aldeano cada 5 segundos siempre que haya comida y que no sobrepase la población máxima, teniendo en cuenta los aldeanos asignados a edificios.

```
public void generarAldeano() {
   if (poblacion+1 <= aldeanosMaximos &&
(poblacion+aldeanosAsignados) <
       aldeanosMaximos) {
       if (ControladorRecursos.consumirRecurso(recursos,
           RecursosEnum.COMIDA, 1)) {
           poblacion++;
       }
   }
}
```

- **Recursos:**

  El controlador de recursos provee las operaciones necesarias para el manejo de los recursos de forma segura y realizando las correspondientes validaciones.

```
public final class ControladorRecursos {
   public static synchronized int
getCantidadRecurso(Map<RecursosEnum,
   Integer>
   recursos, RecursosEnum recurso) {
       Integer cantidadActual = recursos.get(recurso);
       if (cantidadActual != null) {
           return cantidadActual;
       }
       return 0;
   }
   public static synchronized void
agregarRecurso(Map<RecursosEnum,
   Integer> recursos, RecursosEnum recurso, int cantidad) {
       if (cantidad > 0) {
           recursos.put(recurso, getCantidadRecurso(recursos,
           recurso)+cantidad);
       }
   }
   public static void
agregarRecursoSinExcederMax(Map<RecursosEnum,
   Integer> recursos, RecursosEnum recurso, int cantidad) {
       int cantidadActual =
       getCantidadRecurso(Aldea.getInstance().getRecursos(),
recurso);
       int disponible = recurso.getMax() - cantidadActual;

       agregarRecurso(recursos, recurso, Math.min(cantidad,
disponible));
   }
   public static synchronized boolean
   puedeConsumirRecurso(Map<RecursosEnum, Integer> recursos,
RecursosEnum
   recurso, int cantidad) {
       return cantidad <= getCantidadRecurso(recursos, recurso) &&
                                                          cantidad
> 0;
   }
   public static synchronized boolean
consumirRecurso(Map<RecursosEnum,
   Integer> recursos, RecursosEnum recurso, int cantidad) {
       if (puedeConsumirRecurso(recursos, recurso, cantidad)) {
           recursos.put(recurso, getCantidadRecurso(recursos,
           recurso)-cantidad);
           return true;
       }
       return false;
   }
```

```
}
```

Los generadores de recursos permiten a los edificios generar recursos cada 10 segundos asegurando que no se excedan los máximos permitidos y generando según la cantidad de aldeanos asignados al edificio. Se muestra el generador estándar, pero hay otras implementaciones para edificios que funcionan de manera distinta como la carpintería que por cada tablón consume 2 troncos o la cabaña de caza, en la cual pueden morir aldeanos durante las incursiones.

```
@Data @AllArgsConstructor
public class GeneradorEstandar implements IGeneradorRecursos {
   protected RecursosEnum recurso;
   private static final Random random = new Random();

   @Override
   public void producirRecursos(Map<RecursosEnum, Integer>
recursos,
   RecursosEnum recurso, int aldeanosAsignados) {
       if (aldeanosAsignados > 0) {
           int cantidad =
             calcularCantidadProducida(aldeanosAsignados);

     ControladorRecursos.agregarRecursoSinExcederMax(recursos,
                                        recurso, cantidad);
       }
   }

   protected int calcularCantidadProducida(int aldeanosAsignados)
{
       int cantidadProducida = 0;

       // Calcular la cantidad producida
       for (int i = 0; i < aldeanosAsignados; i++) {
           if (random.nextDouble() < 0.5) {
               cantidadProducida++;
           }
       }
       // Ajustar la cantidad producida por la calidad del recurso
       cantidadProducida = Math.max(cantidadProducida, 1);
       cantidadProducida = cantidadProducida /
                                    recurso.getCALIDAD();

       return cantidadProducida;
   }
}
```

```
@Override
public void run() {
   ListaHilos.add(thread);
   Map<RecursosEnum, Integer> recursosIniciales = new
   HashMap<>(recursos);
   try {
     while (JuegoActivity.enEjecucion) {
       recursosIniciales = new HashMap<>(recursos);
       //Genera recursos, espera x tiempo y despues los pasa a la
aldea *
       for (IGeneradorRecursos generadorRecursos :
generadoresRecursos) {
                generadorRecursos.producirRecursos(recursos,
                     generadorRecursos.getRecurso(),
aldeanosAsignados);
       }
       Thread.sleep(SEGUNDOS_ENTRE_RECURSOS * 1000);
       finalizarGeneracionRecursos();
     }
   } catch (InterruptedException e) {
       /* En caso de interrupcion se vuelve al estado anterior,
para
       evitar que se dupliquen recursos */
       recursos = recursosIniciales;
       ListaHilos.remove(thread);
   }
}
```

- **Partida de Caza:**

  Cuando se inicia la partida de caza se inicia un Timer de 10 segundos que es el encargado de llamar al generador de recursos y calcular las muertes aleatorias en sus distintos eventos.

```
public void iniciarPartidaCaza(int numAldeanos, int tiempoTotal) {
    if (numAldeanos <= aldeanosMaximos) {
        if (ControladorAldea.asignarAldeano(numAldeanos)) {
            partidaActiva = true;
            aldeanosAsignados = numAldeanos;
            /* El timer se encarga de llamar a producir recursos
cada
            segundo */
            timerPartidaCaza = new TimerPartidaCaza(tiempoTotal,
this);
            timerPartidaCaza.start();
        }
    }
}
public void finalizarPartidaCaza() {
    /* Si se ha generado mas del maxmio se añade solo lo necesario
para
    llegar al maximo */
    int comidaAldea = ControladorRecursos.getCantidadRecurso(
                      aldea.getRecursos(), RecursosEnum.COMIDA);
    int comidaGenerada = ControladorRecursos.getCantidadRecurso(
                                      recursos, RecursosEnum.COMIDA);
    if ((comidaAldea+comidaGenerada) >
RecursosEnum.COMIDA.getMax()) {
        recursos.put(RecursosEnum.COMIDA,
                     RecursosEnum.COMIDA.getMax()-comidaAldea);
    }
    if (aldeanosAsignados > 0){
        transferirRecursoAldea(RecursosEnum.COMIDA);
    }
    devolverAldeanos(aldeanosAsignados);
    aldeanosAsignados = 0;
    aldeanosMuertosEnPartida = 0;
    partidaActiva = false;
}
```

- **Incursión:**

  Cuando un usuario ataca a otro, si este tiene más atacantes que defensores la víctima ganará el ataque y viceversa, en caso de empate se decide el ganador de manera aleatoria, se calculará cuántos recursos y aldeanos perderá o ganará cada usuario y sus puntuaciones del ranking, y se desencadenaran varios callbacks y eventos encargados de notificar a la víctima y de actualizar la interfaz del atacante.

```
public void gestionarAtaque(UsuarioDTO victima, int
soldadosEnviados, AtaqueCallback callback) {
   FirebaseFirestore.getInstance()
     .collection(Constantes.BaseDatos.COLECCION_USUARIOS)
     .document(victima.getEmail()).get()
     .addOnSuccessListener(document -> {
       RecursosDTO recursosDTO = document.get(
       Constantes.BaseDatos.RECURSOS, RecursosDTO.class);
       EdificioDTO castilloDTO = document.get(
           Constantes.BaseDatos.CASTILLO, EdificioDTO.class);
       if (castilloDTO != null && recursosDTO != null) {
         int defensas = castilloDTO.getAldeanosAsignados();
         boolean victoria;
         if (defensas > soldadosEnviados) victoria = false;
         else if (defensas < soldadosEnviados) victoria = true;
         else {
           // Si hay empate se decide aleatoriamente
           int random = Utilidades.generarIntRandom(0, 1);
           victoria = random != 0;
         }
         if (victoria) {
           // Calcular cuantos recursos se roban
           Map<RecursosEnum, Integer> recursosRobables =
           getRecurosRobables(recursosDTO);
           quitarRecursosVictima(recursosRobables,recursosDTO,
                                       castilloDTO, victima);
           darRecursosAtacante(recursosRobables);
           // Aumentar puntos al ganar

gestorRealTimeDatabase.modificarPuntuacionUsuarioActual(
                 Constantes.Castillo.PUNTOS_VICTORIA);
         } else {
           /* Si el atacante pierde los soldados enviados
           mueren y bajan los puntos */
           aldea.setPoblacion(
                  aldea.getPoblacion()-soldadosEnviados);
            gestorRealTimeDatabase
                    .modificarPuntuacionUsuarioActual(
                         Constantes.Castillo.PUNTOS_DERROTA);
          }
          gestorRealTimeDatabase.guardarUltimoAtaque(
                                    System.currentTimeMillis());
              callback.onAtaqueTerminado(victima, victoria);
       } else callback.onError(new FirebaseFirestoreException(
         "Error al obtener los datos necesarios para el ataque",
                   FirebaseFirestoreException.Code.NOT_FOUND));
     }).addOnFailureListener(callback::onError);
}
```

- **Sincronización de datos locales con datos del servidor:**
  Cuando se inicia y cuando se cierra JuegoActivity se realizan ciertas acciones para asegurar que siempre se guarde la partida en el servidor o en local para no perder los datos, y de sincronizar estas dos bases de datos cuando sea necesario para que el jugador no pierda su progreso.

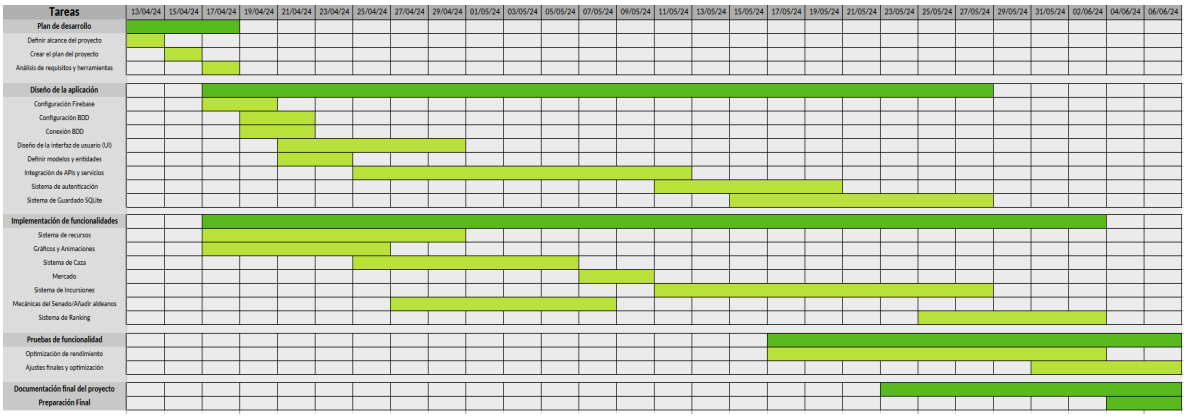
```
@Override
protected void onStart() {
    super.onStart();
    // Aqui hay mas codigo irrelevante para esta funcionalidad
    if (UtilidadRed.hayInternet(this)) {
        try {
            // Sincronizar partida local
            if (!gestorSqlite.estaSincronizado()) {
                gestorSqlite.cargarDatos();
                gestorFirestore.guardarDatos(emailUsuario, this);
            }
        } catch (RuntimeException e) {
            e.printStackTrace();
        }
        gestorFirestore.cargarDatos(emailUsuario, this);
    }
}
@Override
protected void onDestroy() {
    super.onDestroy();
    try {
        gestorFirestore.guardarDatos(emailUsuario, this);
    } catch (RuntimeException e) {
        gestorSqlite.guardarDatos();
    }
    // Aqui hay mas codigo irrelevante para esta funcionalidad
}
```

8.- Planificación del proyecto

8.1.- Acciones

- **Toma de requisitos**
  - Discusiones para definir los objetivos del proyecto.
  - Documentación de los requisitos funcionales y no funcionales acordados.
- **Análisis de los requisitos**
  - Evaluación de la viabilidad técnica.
  - Identificación de las tecnologías a utilizar.
  - Elaboración de un plan de proyecto detallado.
- **Diseño**
  - Interfaces: Prototipos de interfaces de usuario.
  - Procesos: Diagramas de flujo de los procesos del sistema.
  - Base de datos: Modelado de la base de datos utilizando anotaciones JPA para definir las entidades y relaciones.
- **Codificación**
  - Configuraciones necesarias del entorno.
  - Desarrollo de controladores y herramientas.
- **Pruebas**
  - Pruebas unitarias para verificar la funcionalidad de cada componente.
  - Pruebas de integración para asegurar que todos los componentes trabajan juntos correctamente.
  - Pruebas de rendimiento.
- **Documentación**
  - Documentación técnica del código y arquitectura del sistema.
  - Guías de uso y manuales para los usuarios finales.

8.2.- Temporalización y secuenciación



- **Plan de desarrollo**
  - Definir alcance del proyecto: Determinar los objetivos y límites del proyecto, especificando qué se incluirá y qué no.
  - Plan del proyecto: Elaborar un cronograma con todas las tareas y sus plazos.
  - Análisis de requisitos y herramientas: Identificar las necesidades del proyecto y seleccionar las herramientas y tecnologías adecuadas.
- **Diseño de la aplicación**
  - Configuración Firebase: Preparar Firebase para gestionar la autenticación, la base de datos y otros servicios.
  - Configuración BDD: Establecer la estructura inicial de la base de datos.
  - Diseño de la interfaz de usuario (UI): Crear prototipos y diseños de las interfaces que usarán los usuarios.
  - Definir modelos y entidades: Crear los modelos de datos y las entidades que representarán la información en la base de datos.
  - Integración de APIs y servicios: Conectar la aplicación con servicios externos mediante APIs.
  - Sistema de autenticación: Implementar un sistema para gestionar el acceso de los usuarios.
  - Sistema de Guardado SQLite: Configurar el almacenamiento local de datos usando SQLite.
- **Implementación de funcionalidades**
  - Sistema de recursos: Desarrollar la gestión de recursos dentro de la aplicación.
  - Gráficos y Animaciones: Crear elementos visuales y animaciones para mejorar la experiencia del usuario.
  - Sistema de Caza: Implementar funcionalidades relacionadas con la caza.
  - Mercado: Desarrollar un sistema de compra de objetos.
  - Sistema de Incursiones: Crear mecánicas para las incursiones entre jugadores.
  - Mecánicas del Senado/Añadir aldeanos: Implementar funcionalidades del senado y la gestión de aldeanos.
  - Sistema de Ranking: Crear un sistema para clasificar a los jugadores según su puntuación.
- **Pruebas de funcionalidad**
  - Optimización de rendimiento: Mejorar la eficiencia y velocidad de la aplicación.
  - Ajustes finales y optimización: Realizar los ajustes necesarios y optimizar la aplicación.
- **Documentación del proyecto**
- **Preparación Final:** Completar y revisar toda la documentación del proyecto, asegurando que esté lista para su presentación final.

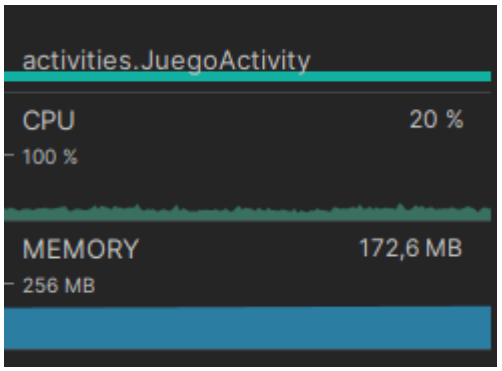
9.- Pruebas y validación

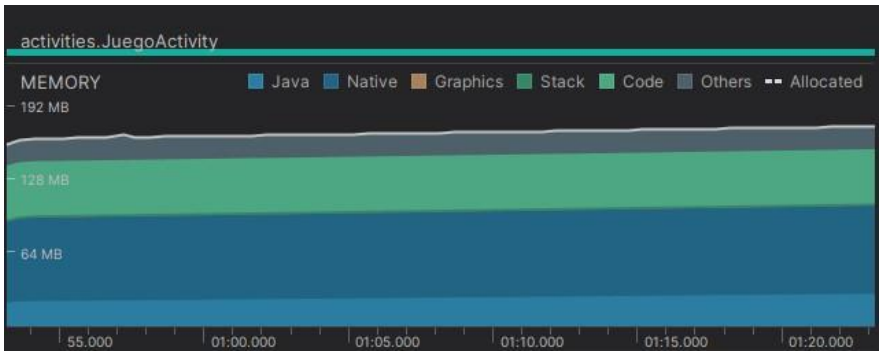
- **Criterios de Aceptación**:
  - Requisitos Funcionales:
    - Registrar nuevo jugador
    - Iniciar sesión
    - Cerrar sesión
    - Cargar partida
    - Guardar partida
    - Construir Edificios
    - Mejorar Edificios
    - Asignar Aldeanos
    - Mercado
    - Partidas de Caza
    - Incursiones
    - Mejorar Senado
    - Asignar Tareas desde el Senado
    - Generación de recursos
    - Establecer Defensas
    - Ranking
    - Navegar por las Pantallas
    - Configuración
    - Tutorial
  - Requisitos No Funcionales:
    - Rendimiento
    - Seguridad
    - Usabilidad
    - Disponibilidad
    - Escalabilidad
    - Mantenimiento
    - Compatibilidad
- Planificación de las Pruebas
  - Casos de Uso:
    - Registrar usuario
    - Iniciar sesión
    - Cerrar sesión
    - Cargar partida en servidor
    - Guardar partida en servidor
    - Cargar partida en local
    - Guardar partida en local
    - Mejorar Senado
    - Mejorar Edificio
    - Asignar Aldeano
    - Partida de Caza
    - Generación de Aldeanos
    - Comprar Recurso
    - Incursión

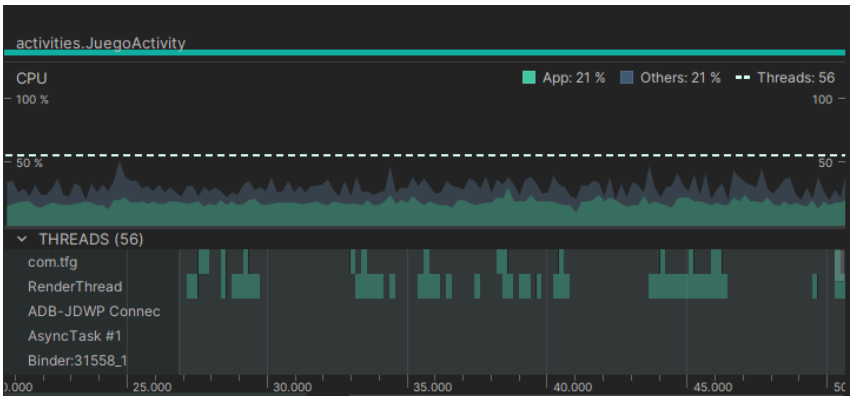
- Preparación del Entorno de Pruebas
  - Configuración del Entorno:
    - Hemos configurado dispositivos (emulados) con diferentes versiones de Android y distintas versiones de la api del sistema para garantizar compatibilidad de una mayor cantidad de dispositivos.
    - Se han simulado diversas condiciones de red para pruebas de conectividad con las distintas bases de datos y verificar el correcto funcionamiento del sistema online de ataques entre jugadores.
  - Datos de Prueba:
    - Hemos creado distintos perfiles de jugadores con distintos niveles de progreso y diferentes configuraciones de aldea para garantizar que el juego gestiona bien todos los datos y no se produce ningún bloqueo que impida el avance del jugador.
- Ejecución de las Pruebas
  - Pruebas Funcionales:
    - Sesion de los usuarios: Se ha validado que los usuarios pueden registrarse, iniciar sesión cerrar sesión y que la aplicación gestione todo de manera correcta, verificando que no haya ningún problema de seguridad y se ha comprobado que el sistema de notificaciones se actualiza de manera correcta según los cambios en la sesión de usuario para asegurar que solo se reciben las notificaciones del ultimo usuario que guardó sus sesión en el dispositivo.
    - Guardado y carga de datos: Se han realizado diversas pruebas para asegurar que los datos siempre se cargan y guardan de manera correcta, manejando posibles cambios de cuenta de usuario en tiempo de ejecución de la aplicación para garantizar la integridad de los datos, y también se ha comprobado que en caso de cualquier fallo se actualice de manera correcta los datos de conexión del usuario de lado del servidor y que se guarde una copia de la partida de manera local en el cliente para su posterior sincronización.
    - Generación de Recursos: Se han hecho pruebas para verificar el correcto funcionamiento de todo el sistema de recursos del juego para asegurar que todo funcione de manera correcta según los límites de nivel del senado y de los edificios y también se ha comprobado que en casos de interrupciones de procesos concurrentes los recursos se guarden de manera correcta y no se generen datos erróneos o condiciones de carrera.
    - Desbloqueo de Edificios: Hemos testeado que todos los edificios se desbloquean de manera correcta según las mejoras del senado y que la interfaz y los datos del juego respondan correctamente a estos cambios.
    - Ataques a otros jugadores: Se ha comprobado que el sistema online de ataques entre usuarios funciona de manera correcta validando bien las cantidades de recursos y aldeanos que se deben quitar o añadir a cada usuario y se comprobado que el sistema de notificaciones relacionado a los ataques muestre los mensajes correctos a los usuarios.

- Partidas de caza: Se han realizado pruebas para garantizar la correcta generación de recursos de una partida de caza garantizando que un aldeano que ha muerto no genere comida tras su muerte y que si mueren todos ninguno traiga comida.
- Asignación de aldeanos: Se ha probado que la asignación de los aldeanos no genere problemas con los máximos de población en procesos concurrentes y que la generación de recursos responda correctamente a la asignación de los aldeanos

- Pruebas de Usabilidad:
  - Interfaz de Usuario: Se ha comprobado que la interfaz de usuario utilice buenas prácticas de usabilidad como que todos los elementos tengan un tamaño suficientemente grande y que se muestren mensajes de tutorial para ayudar al usuario a entender mejor lo que está viendo .
  - Feedback del Usuario: Hemos realizado pruebas con familiares y amigos para obtener feedback y realizar ciertos cambios en la interfaz y ciertas mecánicas del juego.
- Pruebas de Rendimiento:
  - Carga y Consumo de Recursos: Se ha medido los recursos consumidos por el juego en distintas situaciones y el consumo de batería en sesiones largas.
    - Consumo medio de ram y cpu:







- Consumo de batería: En sesiones de 3 horas el juego ha llegado a consumir entre un 25% y un 40% de la batería del dispositivo, variando según las características del dispositivo.

- Estabilidad: Se han realizado pruebas de estrés para verificar la estabilidad del juego bajo condiciones intensas.

- Pruebas de Seguridad:
  - Protección de Datos: Se ha comprobado que las reglas de seguridad para lectura y escritura de datos establecidas en los distintos sistemas de Firebase funcionan correctamente.
- Pruebas Unitarias:
  - Se han realizado distintos tests con JUnit para asegurar que el código funciona de manera correcta. Se muestra como ejemplo los tests de las distintas operaciones que se pueden realizar con los recursos:

```
@Test
public void testGetCantidadRecurso() {
    recursos.put(RecursosEnum.TRONCOS_MADERA, 100);
    Assert.assertEquals(100,
            ControladorRecursos.getCantidadRecurso(
             recursos, RecursosEnum.TRONCOS_MADERA));
    Assert.assertEquals(0,
     ControladorRecursos.getCantidadRecurso(
                    recursos, RecursosEnum.PIEDRA));
}
```

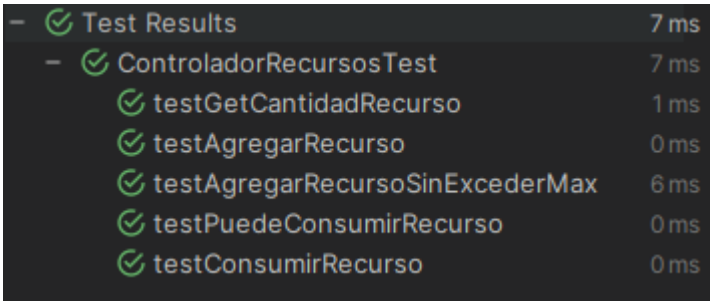
```
@Test
    public void testAgregarRecurso() {
        ControladorRecursos.agregarRecurso(recursos,
            RecursosEnum.TRONCOS_MADERA, 50);
        Assert.assertEquals(50,

        ControladorRecursos.getCantidadRecurso(
          recursos, RecursosEnum.TRONCOS_MADERA));
        ControladorRecursos.agregarRecurso(
          recursos, RecursosEnum.TRONCOS_MADERA, 25);
        Assert.assertEquals(75,
                ControladorRecursos.getCantidadRecurso(
                 recursos,
RecursosEnum.TRONCOS_MADERA));
    }
    @Test
    public void testAgregarRecursoSinExcederMax() {
        /* Se pasa el hashmap de la aldea, ya que se
        comrpueba siempre que el maximo que no se excede
        sea el de la aldea y no el del hashmap local del
        edificio */
        Aldea aldea = Aldea.getInstance();
        aldea.setRecursos(new HashMap<>());
        aldea.getRecursos()
                .put(RecursosEnum.TRONCOS_MADERA, 0);
        RecursosEnum.TRONCOS_MADERA.setMax(30);
        aldea.getRecursos()
                .put(RecursosEnum.TRONCOS_MADERA, 10);
        ControladorRecursos.agregarRecursoSinExcederMax(
           aldea.getRecursos(),
           RecursosEnum.TRONCOS_MADERA, 10);
        Assert.assertEquals(20,
            ControladorRecursos.getCantidadRecurso(
               aldea.getRecursos(),
               RecursosEnum.TRONCOS_MADERA));
        ControladorRecursos
          .agregarRecursoSinExcederMax(
               aldea.getRecursos(),
               RecursosEnum.TRONCOS_MADERA, 20);
        Assert.assertEquals(30, ControladorRecursos
           .getCantidadRecurso(aldea.getRecursos(),
                       RecursosEnum.TRONCOS_MADERA));
    }
```

```
@Test
    public void testPuedeConsumirRecurso() {
        recursos.put(RecursosEnum.PIEDRA, 50);
        Assert.assertTrue(
            ControladorRecursos.puedeConsumirRecurso(
                recursos, RecursosEnum.PIEDRA, 25));
        Assert.assertFalse(
            ControladorRecursos.puedeConsumirRecurso(
                recursos, RecursosEnum.PIEDRA, 60));
    }
    @Test
    public void testConsumirRecurso() {
        recursos.put(RecursosEnum.PIEDRA, 50);
        Assert.assertTrue(

ControladorRecursos.consumirRecurso(recursos,
                                RecursosEnum.PIEDRA,
25));
        Assert.assertEquals(25,
            ControladorRecursos.getCantidadRecurso(
                            recursos,
RecursosEnum.PIEDRA));
        Assert.assertFalse(

ControladorRecursos.consumirRecurso(recursos,
                                RecursosEnum.PIEDRA, 30));
        Assert.assertEquals(25,
            ControladorRecursos.getCantidadRecurso(
                        recursos, RecursosEnum.PIEDRA));
    }
```



- Análisis de Resultados
    - Análisis de Resultados: Hemos identificado que la mayoría de bugs y problemas han surgido en los cambios de aldeanos y recursos durante procesos concurrentes y problemas en los procesos asíncronos de la base de datos, pero se han solucionado gracias a la realización de las pruebas comentadas anteriormente.

- Acción Correctiva
    - Resolución de Problemas: Se han solucionado todos los problemas existentes conocidos.
    - Retesting: Hemos vuelto a realizar pruebas tras corregir errores.

- Herramientas utilizadas en las pruebas:
    - Profiler de Android Studio para las pruebas de rendimiento.
    - JUnit para los tests unitarios.
    - Crashlytics de Firebase para ver métricas del servidor.

10.- Relación del proyecto con los módulos del ciclo

- **Sistemas informáticos**
    - Configuración del entorno de desarrollo.
- **Bases de Datos**
    - Diseño e implementación de la base de datos.
    - Creación de diagramas entidad / relación y de tablas de la base de datos.
- **Programación**
    - Desarrollo de la aplicación en Java, creación de controladores y servicios.
- **Lenguajes de marcas y sistemas de gestión de información**
    - Uso de archivos de configuración (XML, JSON).
- **Entornos de desarrollo**
    - Configuración del entorno de desarrollo de Android Studio(Paquetes, carpetas, etc).
    - Creación de diagramas de clases y de casos de uso.
- **Acceso a datos**
    - Desarrollo de controladores y servicios que interactúan con la base de datos.
    - Mapeo de clases a las tablas de una base de datos utilizando un ORM.
    - Procesamiento de ficheros JSON para cargar datos mediante la librería Jackson.
- **Desarrollo de interfaces**
    - Implementación de interfaces de usuario (Activities).
    - Diseñar la interfaz de usuario utilizando buenas prácticas de usabilidad.
- **Programación multimedia y dispositivos móviles**
    - Desarrollo principal en Android Studio.
    - Diseño responsive de la interfaz de usuario utilizando los distintos tipos de layouts de android.
    - Desarrollo de videojuegos.
- **Programación de servicios y procesos**
    - Programación concurrente basada en hilos para el sistema de generación de recursos.
    - Ejecución de un proceso en segundo plano de manera activa para recibir notificaciones con la aplicación cerrada.

11.- Conclusiones

- **Estado del Proyecto**

  El proyecto está en un estado funcional y cumple con la mayoría de las funcionalidades previstas. El rendimiento es decente y la interfaz de usuario es adecuada.

- Cobertura de Funcionalidades
- Implementadas:
  - Registro y autenticación.
  - Guardado y carga de datos con Firestore y SQLite.
  - Notificaciones.
  - Mejoras de edificios, asignación de aldeanos, mercado, caza e incursiones.
- Reflexiones sobre el Proceso
  - Clarificar los requisitos antes de codificar fue esencial.
  - Tener una visión general facilitó una implementación estructurada.
  - La integración de Firebase y SQLite facilitó la sincronización de datos.
- Futuro del Proyecto
  - Escalabilidad: Mejorar la capacidad para manejar más usuarios.
  - Optimización: Refinar la eficiencia del sistema.
  - Interfaz de Usuario: Mejorar la usabilidad y diseño.
  - Nuevas Funcionalidades: Incorporar características basadas en el feedback de los usuarios.

  En resumen, el proyecto ha cumplido con la mayoría de sus objetivos y el proceso ha sido una valiosa experiencia de aprendizaje.

12.- Proyectos futuros

- **Ampliaciones y Mejoras**
  - **Sistema de Comercio Avanzado:**
    - Permitir a los jugadores establecer precios para los productos.
    - Precios dinámicos según oferta y demanda.
  - **Incursiones Más Complejas:**
    - Introducir diferentes tipos de tropas y estrategias de combate.
    - Diversificar las recompensas y ajustar los costos de creación de tropas.
  - **Mejoras en la Interfaz de Usuario:**
    - Hacer la interfaz más intuitiva y atractiva.
    - Agregar tutoriales interactivos detallados.
    - Efectos de sonido diferentes por acción.
- **Nuevos Proyectos**
  - **Juego de Estrategia Avanzado:**
    - Desarrollar un juego más complejo basado en este proyecto, con comercio e incursiones mejorados.

- **Plataforma de Comercio Virtual:**
  - Crear una plataforma web de comercio dinámico independiente con fluctuaciones de precios y subastas.
- **Juego de Estrategia en Tiempo Real:**
  - Desarrollar un juego de estrategia en tiempo real estilo Age of Empires usando como base el sistema de edificios y recursos del juego ya desarrollado.

13.- Bibliografía/Webgrafía

- Páginas web:
  - OpenAI (2022), "ChatGPT", https://chatgpt.com
  - hxkl (2013), "StackOverflow Questions", https://stackoverflow.com/questions/
  - Aswan (2010), "StackOverflow Questions", https://stackoverflow.com/questions/
  - conmadoi (2011), "StackOverflow Questions", https://stackoverflow.com/questions/
  - user573536 (2011), "StackOverflow Questions", https://stackoverflow.com/questions/
  - Dalmas (2011), "StackOverflow Questions", https://stackoverflow.com/questions/
  - MoureDev (2020), "Tutorial Firebase para Android", https://www.youtube.com/
  - Coding with Mukund (2024), "How to get Access Token in the new Firebase Cloud Messaging V1 HTTP", https://www.youtube.com/
  - Martin Kiperszmid (2023), "Enviar Notificaciones Remotas con Firebase Cloud Messaging (FCM) en Android", https://www.youtube.com/
  - Daily Coding (2022), "Android Custom Seekbar | How to Customize a SeekBar", https://www.youtube.com/
  - Foxandroid (2023), "Bottom Navigation Bar - Android Studio", https://www.youtube.com/
  - Tech Harvest BD (2022), "Bottom Navigation Bar - Android Studio", https://www.youtube.com/
  - Google (2024), "Cómo guardar datos en una base de datos local usando Room", https://developer.android.com/training/data-storage/room?hl=es-419
  - Google (2024), "Migrar desde las API de FCM heredadas a HTTP v1", https://firebase.google.com/docs/cloud-messaging/migrate-v1?hl=es
  - Google (2024), "Configurar una aplicación cliente de Firebase Cloud Messaging en Android", https://firebase.google.com/docs/cloud-messaging/android/client?hl=es#java
- Herramientas utilizadas:
  - Android Studio: https://developer.android.com
  - LibreOffice: https://es.libreoffice.org
  - Notepad++: https://notepad-plus-plus.org
  - Carbon: https://carbon.now.sh
  - draw.io: https://app.diagrams.net
  - hackolade: https://hackolade.com

14.- Anexos

- Repositorio de GitHub con el código e imágenes de mayor calidad de algunos de los diagramas.
- Pack de assets utilizado para los gráficos: Tiny Swords (licencia de uso libre)
- Página de iconos sin derechos de autor: Icons8
- Música y efectos de sonido utilizados (obtenidos de pixabay sin derechos de autor):
  - https://pixabay.com/music/acoustic-group-the-britons-127687/
  - https://pixabay.com/es/music/gente-floralia-194076/
  - https://pixabay.com/music/folk-medieval-background-196571/