UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

La venganza de César Borgia. Aplicación piloto para escape rooms urbanos geolocalizados

Autor/es

JERAI FERRADÁS GUTIÉRREZ

Director/es

ELOY JAVIER MATA SOTÉS y Diego Téllez Alarcia

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2019-20

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

La venganza de César Borgia. Aplicación piloto para escape rooms urbanos geolocalizados

Realizado por:

Jerai Ferradás Gutiérrez

Tutelado por:

Eloy Javier Mata Sotés

Diego Téllez Alarcia

Logroño, julio, 2020

Resumen

Los *Escape Rooms* o salas de escape se han popularizado en los últimos años. Salas llenas de acertijos que los jugadores deben resolver para poder salir antes de que se acabe el tiempo. Pero estas actividades tienen limitaciones, ya que el espacio es limitado y, además, los escapes no cambian. Una vez has participado en uno ya no tiene sentido repetir.

En este proyecto se estudia otra alternativa, los *Escape Streets*. Estas actividades consisten en ir resolviendo enigmas por la calle guiados por una aplicación móvil, aunque los actuales *Escape Streets* del mercado tienen un inconveniente en común con los *Escape Rooms*, y es que cada uno de ellos solo ofrece una partida.

En este proyecto creamos una plataforma que permite crear diferentes partidas desde la misma aplicación. Para ello se ha desarrollado una estructura que se adapta a diferentes partidas.

De esta manera el coste de cada partida se puede reducir considerablemente para el usuario, ya que la creación de cada partida no requiere del desarrollo de una aplicación independiente.

Abstract

Escape Rooms had become very popular in recent years. Rooms full of puzzles that players must solve in order to get out before they run out of time. But these activities have limitations, for example, the space is limited and the escapes do not change. Once you have participated in any of them there is no point in coming back again.

This proyect explore another alternative, the Escape Streets. These activities consist of solving puzzles on the street guided by a mobile application, although the current Escape Streets on the market have a common drawback with the Escape Rooms, and that is that each of them only offers one game.

In this project we create a platform that allows you to create different games from the same application. For this, a structure has been developed that adapts to different escapes.

In this way the cost of each escape can be reduced considerably for the user, since the creation of each game does not require the development of a separate application.

Índice

1.0 Introducción

1.1 Contexto

Un *escape room* o *sala de escape* es un juego en el que los jugadores entran en un espacio cerrado y siguiendo una historia, deben resolver una serie de acertijos para poder salir. En los últimos años estas salas de escape se han hecho muy populares es todo el mundo y el fenómeno sigue en alza.

La demanda de estas actividades es muy alta pero el coste económico que conlleva es muy alto, con precios que rondan los 15-20€ por persona.

La alternativa que se propone en este TFG son los *escape street*, una variante de los *escape rooms* menos conocida que consiste en realizar todas las pruebas y acertijos en lugares públicos.

Los *escape streets* nacen a partir de los *escape rooms*, pero su estructura es completamente diferente. Abren muchas posibilidades a los diseñadores de estas actividades al no estar limitados a recintos cerrados, aunque por otro lado están limitados por no controlar los elementos de la calle. Por lo tanto, se puede considerar que son actividades totalmente diferentes a los *escape rooms*.

Cabe destacar el potencial uso turístico que se le puede dar a estas actividades, ya que una historia puede llevarte por los sitios más importantes de una ciudad y explicarte cosas sobre ellos de una forma lúdica. Esto puede hacer el aprendizaje de la cultura de la localidad atractiva a todo tipo de públicos.

Durante este proyecto, la crisis del COVID-19 ha obligado a gran parte de la población a guardar cuarentena en sus casas y mantener un distanciamiento social. Las salas de escape convencionales han tenido que cerrar y como alternativa, algunas de ellas han creado versiones online que han tenido bastante éxito en esta situación excepcional.

Ante esta situación, podemos valorar que los *escape streets* permiten realizar la actividad con mayores niveles de seguridad que las *escape rooms* por el hecho de realizarse al aire libre y permitir mantener la distancia de seguridad.

1.2 Integrantes

Jerai Ferradás

Estudiante de Ingeniería Informática y desarrollador del proyecto.

Eloy Javier Mata

Profesor del Departamento de Matemáticas y Computación de la Universidad de La Rioja. Tutor del TFG

Diego Téllez

Profesor del Departamento de Ciencias de la Educación de la Universidad de La Rioja. Encargado de dirigir el desarrollo de la historia de “LA VENGANZA DE CÉSAR BORGIA” haciendo las funciones de cliente para este proyecto.

1.3 Antecedentes

Durante el curso académico 2018/19 el profesor de Didáctica de las Ciencias Sociales, Diego Téllez, propuso a sus alumnos desarrollar el contenido de un escape street.

De este proyecto surgió una historia centrada en el histórico personaje César Borgia, el cual fue asesinado en Viana (lugar donde se desarrolla el escape street). Los alumnos de la asignatura desarrollaron por parejas diferentes pruebas para descubrir el enigma de quien asesinó a César Borgia. Estas pruebas consistían en recorrer la localidad de Viana obteniendo información sobre su historia hasta obtener la respuesta a cada prueba. Estas respuestas podían ser fechas, monumentos, nombres de calles, etc.

De esta manera quedaron especificados los diálogos, pistas, lugares, acertijos y demás detalles que se usarán en el presente proyecto.

Una vez finalizado el curso, Diego propuso a Eloy Mata (profesor del grado de informática) la realización de una aplicación móvil que implementase el escape street y Eloy propuso este proyecto como trabajo de final de grado de informática.

2.0 Planificación

2.1 Estudio de mercado

La idea de este proyecto surgió a partir de las aplicaciones desarrolladas por la web http://www.escapestreet.es/, donde se pueden encontrar varias aplicaciones para distintas localidades.

Esta web ofrece la creación de escape streets personalizados para una historia y localidad que solicitemos.

Algunos de los escape streets que han creado son:

- Logroño: https://1521elasedio.es/
- Pamplona: https://www.escapebull.com/
- Tudela: https://jugar.escapetudela.com/
- San Sebastián: http://www.escaperoomsansebastian.com/escape-street/

También existen otros escape streets de otras empresas como por ejemplo https://streetskp.com/ que se desarrolla en la ciudad de Bilbao.

Todas estas son aplicaciones independientes que además requieren de un pago relativamente alto por cada partida que se juega. Los precios rondan desde los 5€ (“El secreto de Benjamín” en Tudela) hasta los 59€ (“La leyenda del mercader de relojes” en San Sebastián).

También hay que tener en cuenta que algunos de estos escape streets, como es el caso del de San Sebastián, ofrecen tablets para seguir la aventura, así como otros materiales y personal que guía la partida. Mientras que otros como por ejemplo el de Logroño anteriormente mencionado, son totalmente autónomos: se realiza la reserva y el pago por internet para que el día y hora acordados se active la partida desde los dispositivos de los jugadores.

En cuanto a la estructura de las diferentes partidas, todas estas aplicaciones siguen aproximadamente la misma estructura. Primero se presenta la historia, luego se presenta el primer reto que se suele resolver introduciendo un texto o un número. Si después de un tiempo no se ha resuelto el reto empiezan a aparecer pistas para ayudar a los jugadores.

Una vez resuelto el reto se presenta el siguiente, siguiendo en todo momento una estructura narrativa.

2.2 Planificación temporal

El proyecto comenzó el día 17 de febrero y se planea terminar a principios de junio. A partir de mi horario personal se ha planificado realizar entre 4 y 5 horas diarias de trabajo. Se ha calculado el número de días laborables entre las fechas de inicio y fin del proyecto para aproximar las horas de trabajo a realizar.

	Inicio	Horas planeadas
Semana 1	17-feb	23
Semana 2	24-feb	18
Semana 3	02-mar	23
Semana 4	09-mar	23
Semana 5	16-mar	23
Semana 6	23-mar	23
Semana 7	30-mar	23
Semana 8	06-abr	14
Semana 9	20-abr	23
Semana 10	27-abr	18
Semana 11	04-may	23
Semana 12	11-may	23
Semana 13	18-may	18
Semana 14	25-may	23
Semana 15	01-jun	23

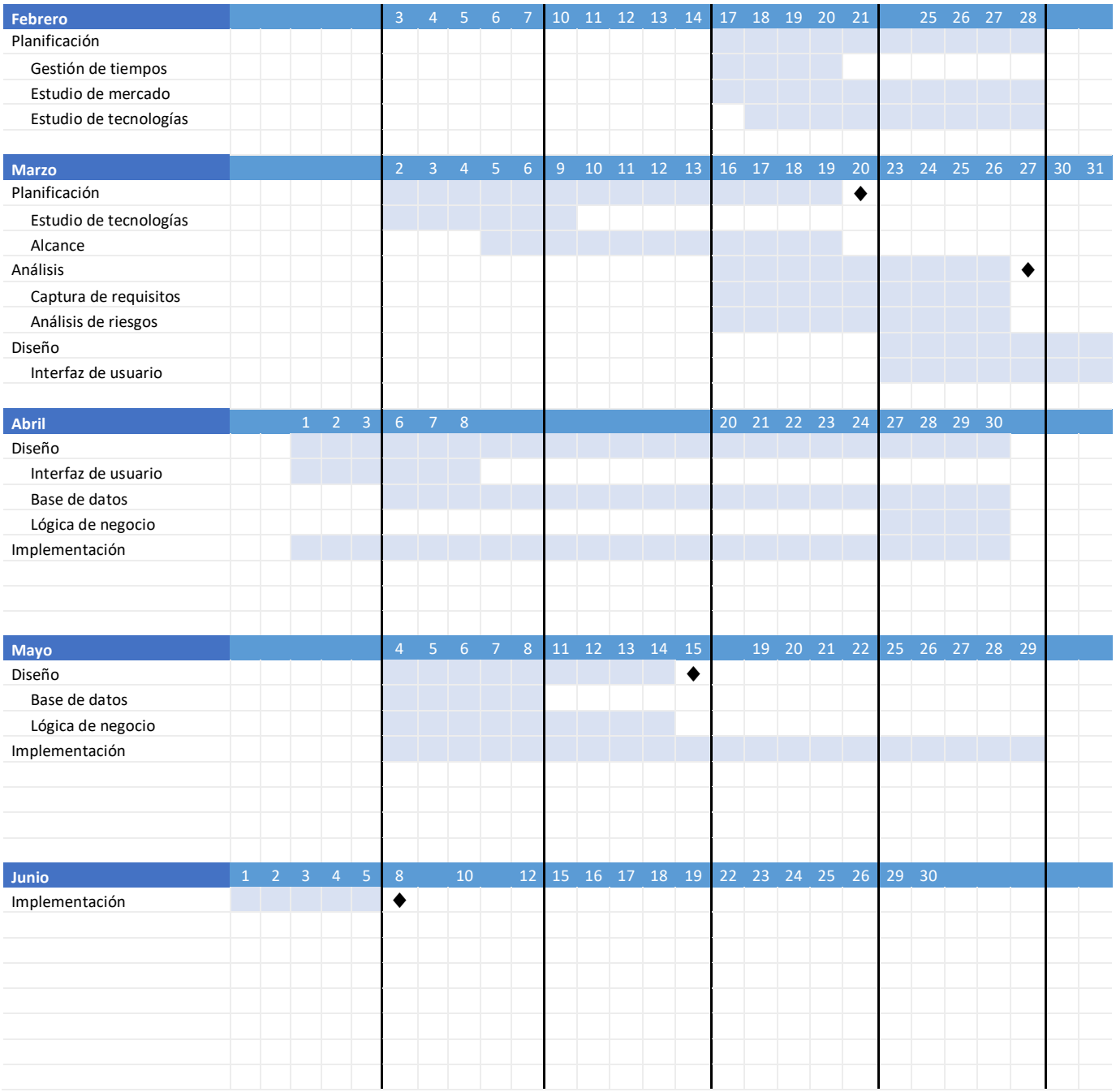
Esto supone un total de 70 días laborables y 321 horas de trabajo un el proyecto.

La planificación consta de varios hitos para cada parte del proyecto:

- Planificación → 20 de marzo
- Análisis → 27 de marzo
- Diseño → 15 de mayo
- Implementación → 8 de junio

Aunque el hito más importante es el de finalización del proyecto, que se ha definido para el 8 de junio.

Estos hitos además de la repartición del tiempo para cada parte del proyecto se pueden ver en el siguiente diagrama de gant:



2.3 Metodología

Para decidir la metodología a utilizar en este proyecto se han tenido en cuenta las siguientes características:

- Los requisitos están muy definidos desde el principio
- No se prevén cambios sustanciales durante su desarrollo
- No existe un cliente que quiera realizar cambios durante el proceso de desarrollo.
- El equipo de desarrollo consta de un único miembro.

Por esto se ha decidido realizar un desarrollo en cascada.

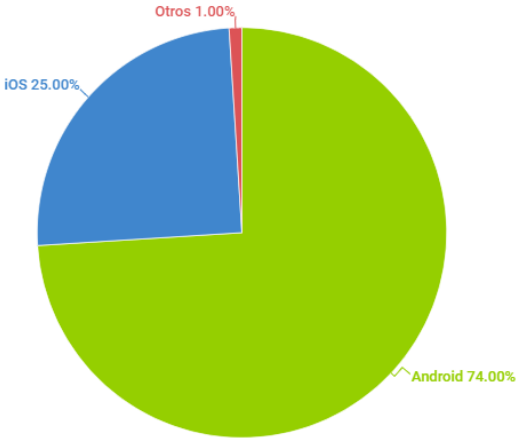
Este tipo de desarrollo se caracteriza por seguir una serie de fases de manera secuencial, de manera que no se pasa a la siguiente sin haber terminado la anterior. Estas fases son:

1. Definición de requisitos
2. Análisis
3. Implementación
4. Pruebas
5. Mantenimiento

La metodología en cascada pura prohíbe comenzar con una fase sin haber finalizado completamente la anterior, aunque en este caso seremos más flexibles en este aspecto, ya que por muy bien definidos que estén los requisitos del proyecto al principio, éstos pueden variar en algunos aspectos durante el desarrollo de este.

2.4 Tecnologías

Para decidir qué tecnologías utilizar en este proyecto primero debemos analizar en qué dispositivos se va a desarrollar la aplicación. Para ello analizaremos las estadísticas de uso de los diferentes sistemas operativos para dispositivos móviles.



Según varias estadísticas encontradas en internet, Android es el sistema operativo más utilizado en el mundo con una cuota de mercado aproximada del 74%.

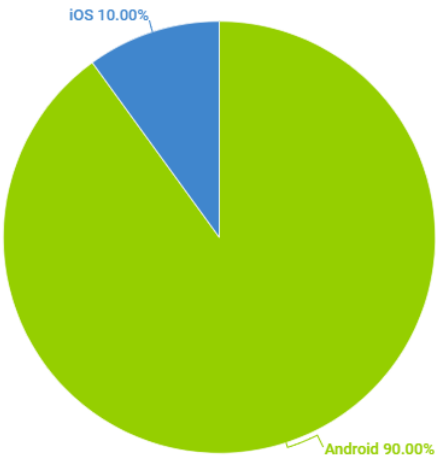
Por otra parte, el sistema operativo de Apple ocupa alrededor del 25%.

Tan solo un 1% de los dispositivos utilizan otro sistema operativo distinto.

Esto supone que, si se desarrollase la aplicación únicamente para Android, no estaría disponible para alrededor de una cuarta parte de la población mundial.

Sin embargo, si centramos el público objetivo en España, los resultados cambian:

Mientras que en Estados Unidos la batalla entre los 2 principales sistemas operativos está bastante igualada, en España tan solo 1 de cada 10 usuarios usan iOS.



A partir de estos datos podemos deducir que lo más lógico es desarrollar la aplicación para Android, pero existe la posibilidad de desarrollarla para los 2 sistemas operativos sin tener que realizar dos aplicaciones distintas: las aplicaciones Híbridas.

Las aplicaciones híbridas son aplicaciones desarrolladas desde un framework que adapta la aplicación a distintas plataformas.

Existen a su vez dos tipos de herramientas: las que utilizan un elemento de tipo webview en la que se incrusta una aplicación web y las que utilizan directamente los elementos nativos del dispositivo.

Algunos ejemplos del primer tipo son las basadas en Apache Cordoba, como por ejemplo Phonegap e Ionic. Estas herramientas permiten desarrollar aplicaciones móviles como si se tratase de una web, utilizando generalmente javascript como lenguaje de programación que a su vez puede utilizar otro framework de desarrollo como Angular.

Es una forma fácil y rápida de crear aplicaciones simples, que no requieran un alto rendimiento y que no necesiten usar intensivamente los elementos del dispositivo ya que el acceso al hardware y a los APIs nativos es muy limitado.

La otra alternativa híbrida son frameworks como Xamarin o ReactiveNative, que, a diferencia de las anteriores herramientas, estas permiten el acceso a los APIs y elementos visuales de los distintos sistemas operativos.

Tras analizar las distintas herramientas disponibles se ha decidido utilizar el SDK Flutter para este proyecto.

Flutter

Flutter es un SDK desarrollado por Google que permite desarrollar aplicaciones iOS y Android utilizando el lenguaje de programación Dart, aunque puede combinarse con código nativo de cada SO. Es una buena opción para este proyecto, ya que aporta muchas herramientas para la implementación de la interfaz de usuario y el lenguaje de programación es muy similar a Java.

Por un lado, la interfaz gráfica se desarrolla a través del mismo lenguaje de programación que las funcionalidades de la aplicación.

Flutter utiliza Widgets, que son la unidad básica para la construcción de interfaces de usuario. Desde un texto hasta un contenedor con scroll o incluso combinaciones de Widgets, todos los elementos que componen la interfaz son objetos heredados de la clase Widget.

De esta manera se concadenan unos Widgets dentro de otros creando un árbol de widgets. Por ejemplo:

```
Container(
height: 56.0, // altura del contenedor
child: Row(
  // Row es un contendor que muestra una
  // lista de Widgets de forma horizontal
  children: <Widget>[
    Text("Hola"),
    Text("mundo"),
  ],
),
```

En este ejemplo podemos ver un contenedor con dos atributos. Un atributo *height* para la altura y un atributo *child* en el que se añade un Widget *Row*, que es un contenedor de varios elementos mostrados en fila, en este caso dos textos.

Se pueden llegar a crear árboles muy complejos de forma muy sencilla, ya que todos estos elementos se pueden generar de forma dinámica al ser creados en un lenguaje de programación. Esto resulta más complicado en sistemas como la programación web con HTML, donde se mezcla el HTML con otro lenguaje de programación en el mismo documento.

Base de datos

En cuanto al sistema de persistencia de la aplicación, existen varias opciones, entre las que se encuentran la clásica base de datos relacional u otra alternativa muy relacionada con las aplicaciones móviles hoy en día: los servicios de Google Firebase.

Google Firebase Database es una base de datos NoSQL que almacena los datos en formato JSON y permite la sincronización en tiempo real con cada cliente conectado. Es un sistema muy dirigido a aplicaciones multiplataforma, ya que dispone de apis para distintos entornos, como por ejemplo Android, iOS, Java, Javascript, Python, etc.

Uno de los problemas de esta base de datos es que, al estar estructurada en forma de árbol, no permite relacionar elementos de distintas ramas y ello puede complicar la estructura de la base de datos. Sin embargo, tiene otros puntos buenos como la sincronización en tiempo real con cada usuario o su facilidad de instalación, ya que solo hay que conectarse con una cuenta de Google en la web de Firebase y conectar el servicio con nuestra aplicación. Desde esta web podemos gestionar la base de datos completa y crear reglas de acceso personalizadas.

Además, Firebase dispone del servicio Firebase Storage que permite almacenar archivos como imágenes, videos y demás archivos que utilice la aplicación. Te permite crear reglas de acceso a estos archivos y el API se encarga de gestionar las descargas y subidas de archivos de forma que, si el usuario pierde la conexión a internet durante la transferencia, esta se reanudará cuando recupere la conexión.

Existen otros servicios de Google como Firebase Authentication o Google Analytics que, aunque podrían utilizarse en un futuro para la autentificación de usuarios y la recogida de estadísticas, no se encuentran dentro del alcance de nuestro proyecto. Estos servicios se podrían añadir a la aplicación de manera muy sencilla al estar implementados otros servicios similares.

2.5 Alcance

En este apartado se definen todas las características y funciones de la aplicación que se va a desarrollar divididas en dos tipos de requisitos, que son los funcionales y los no funcionales.

Requisitos funcionales

1. La aplicación mostrará una lista con las diferentes partidas disponibles, indicando al menos el nombre de la partida, una breve descripción, la dirección de inicio y la duración aproximada de la partida
2. Se mostrará la información de la partida antes de empezarla.
3. Al comenzar una partida se mostrará un diálogo en el que se darán las primeras explicaciones de la partida.
4. Durante los diálogos, a medida que se va obteniendo información, se van desbloqueando misiones, personajes de la historia e información sobre estos.
5. Tras finalizar el diálogo inicial se mostrará un mapa centrado en la posición geográfica del jugador y con marcas en lugares relevantes para la historia.
6. Las misiones podrán iniciarse tras desbloquearse durante un diálogo o al llegar a un lugar relevante para la historia.
7. Excepto durante los diálogos, se podrán ver toda la información sobre las misiones activas, así como la de los personajes conocidos.
8. La información de los personajes se irá desbloqueando a medida que avance la historia
9. Habrá misiones de distintos tipos en función de la forma en que se resuelvan: Unas se resolverán escribiendo un texto, otras mediante un código numérico y otras llegando a una localización específica.

Requisitos no funcionales

1. La aplicación será desarrollada para Android
2. Aunque la aplicación no se desarrolle para el sistema operativo iOS, su adaptación a este SO será sencillo
3. Toda la información de las partidas se almacenará en una base de datos y no se guardará esta información en la propia aplicación hasta que se inicie la partida.

2.6 Análisis de riesgos

Para realizar el análisis de riesgos utilizaremos las siguientes medidas en función de la probabilidad de que el riesgo llegue a ocurrir y el impacto que supondría:

Probabilidad	Impacto
Alta	Catastrófico
Media	Serio
Baja	Tolerable

Se dividirá el análisis en dos tablas dependiendo del tipo de riesgo, ya sean riesgos técnicos o riesgos de proyecto.

Estos son los riesgos técnicos detectados, así como las medidas que se tomarán para reducirlo:

Riesgo	Probabilidad	Impacto	Medidas
Perdida de acceso a la base de datos	Media	Medio	Tener un registro de toda la información de la base de datos en local.
Pérdida del código de la aplicación	Baja	Catastrófico	Realización de una copia de seguridad en un disco duro externo de forma regular.
Pérdida de la memoria	Baja	Catastrófico	La memoria se guardará en la nube mediante OneDrive y se harán copias de seguridad en un disco duro externo de forma regular
Pérdida de acceso a internet	Baja	Medio	
La tecnología utilizada no cumple con los requisitos	Alta	Medio	Se hará un estudio exhaustivo de las diferentes tecnologías disponibles para que si alguna falla se conozcan las alternativas.

Y en esta tabla podemos ver los riesgos de proyecto:

Riesgo	Probabilidad	Impacto	Medidas
Retraso en la finalización del proyecto	Baja	Catastrófico	Se realizará una correcta planificación temporal, así como un adecuado proceso de control durante el desarrollo.
Pérdida de contacto con los tutores	Baja	Medio	

3.0 Análisis y Diseño

3.1 Diseño de interfaz

Para el diseño de la interfaz estableceremos el público objetivo en un grupo de personas de entre 15 y 40 años con un conocimiento de las tecnologías móviles moderado.

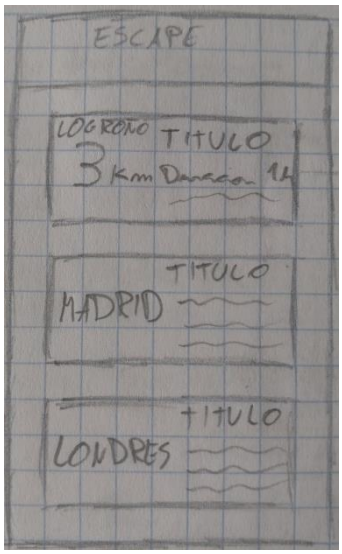
La aplicación también debe de ser accesible para usuarios de fuera de este rango, pero consideraremos que en este rango se encuentra la mayor parte de los usuarios de la aplicación.

En primer lugar, definiremos las funcionalidades que debe implementar la interfaz:

1. Los usuarios deben poder seleccionar la partida que deseen jugar a partir de una lista de las disponibles. La información de cada escape debe de ser accesible antes de iniciar la partida.
2. Una vez iniciada la partida, los usuarios deben tener a su disposición:
    a. Una lista de las misiones activas. Incluyendo toda la información de las mismas y la opción de completarlas que corresponda con el tipo de misión.
    b. Una lista de personajes conocidos con toda la información desbloqueada que se tenga.
    c. Un mapa del lugar en el que se encuentre indicando la posición del jugador y los lugares de interés para la partida.
3. Al iniciar la partida y al finalizar cada misión se puede iniciar un diálogo en el que los personajes expliquen al jugador los siguientes pasos a seguir.

A partir de estos datos se diseñan unos prototipos en papel para las distintas vistas que se deducen de los requisitos.

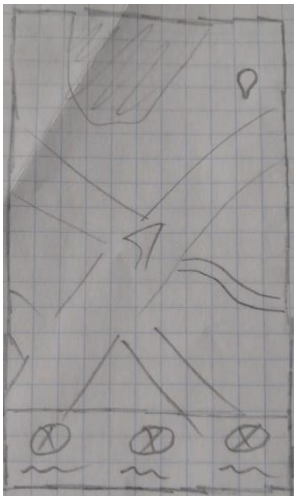
En primer lugar, tenemos una lista de las diferentes partidas disponibles, donde podemos encontrar la información más importante sobre cada una. Tras seleccionar cualquiera de estas se abriría una segunda pantalla con toda la información detallada y la opción de comenzar la partida.

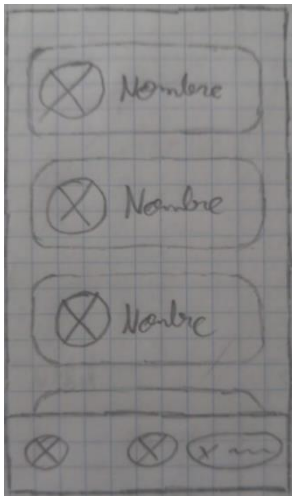


Una vez comenzada la partida se mostrará el mapa centrado en la localización del usuario con un menú inferior que identifique las otras dos ventanas: la lista de misiones y la lista de personajes.

La lista de misiones mostrará para cada misión el título, la descripción y un icono que identificará el tipo de misión en función de su color.

La lista de personajes mostrará la imagen de perfil de cada uno junto con su nombre.



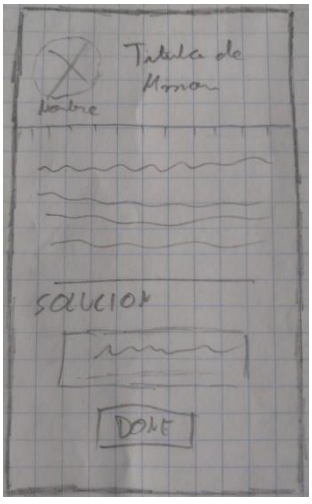


Al pulsar sobre algún elemento de la lista de misiones o la lista de personajes, se abrirá una ventana con la información detallada del mismo.

En el caso de las misiones se verá la misma información que en la lista además de la información necesaria para resolverla.

- Para misiones de tipo texto se mostrará un *textfield* donde el usuario podrá introducir la solución.
- Para las misiones de tipo localización se indicará que para resolver la misión el usuario debe ir al lugar indicado en el mapa.

En el caso de los personajes se mostrará en la cabecera la misma información que en la tarjeta de la lista y debajo la información desbloqueada sobre el mismo.



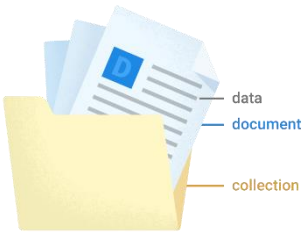
3.2 Diseño de la Base de Datos

Como ya se ha mencionado en el apartado de Tecnologías, el servicio de base de datos que se utiliza en este proyecto es de Firebase, en concreto Cloud Firestore.

El propio servicio nos explica su estructura de funcionamiento de la siguiente manera:

> *A partir del modelo de datos NoSQL de Cloud Firestore, almacenas los datos en documentos que contienen campos que se asignan a valores. Estos documentos se almacenan en colecciones, que son contenedores para tus documentos y que puedes usar para organizar tus datos y compilar consultas.*

Una forma sencilla de entender esto es imaginarse a las colecciones como carpetas que contienen documentos que a su vez contienen datos.



Tipos de datos

A continuación, se mostrará la estructura de la base de datos utilizada para este proyecto, pero antes definiremos los diferentes tipos de datos que utilizamos.

Los tres primeros tipos de datos son propios de Firebase, pero el resto se han definido para nuestro proyecto concreto. Se han definido aquí para explicar que son datos que siguen una estructura definida y que no son simples cadenas de texto.

1. String
   Cadena de caracteres.
2. Número
   Número entero.
3. Geopoint
   Dato que se compone de dos números correspondientes a la latitud y longitud de una coordenada geográfica.
4. Referencia
   String que hace referencia a otros documentos siguiendo el siguiente patrón: *Colección/Identificador de documento*. Por ejemplo:
   a. Character/personaje3
   b. Mission/mission4
   c. Character/personaje7/Info/3
5. TipoDeSolucion
   String que identifica el tipo de solución necesaria para completar la misión. Puede ser: text, number, location
6. <type>
   El tipo de dato del elemento solution depende del tipo de Mission siguiendo el siguiente esquema:
   - text → String
   - number → number
   - location → GeoPoint
7. URL
   Enlace a una imagen almacenada en Firebase Storage.

Estructura general

- Escapes
    - name: String
    - synopsis: String
    - city: String
    - startPoint: Geopoint
    - duration: [Número, Número]
    - Characters
        - name: String
        - bigImageURL: URL
        - smallImageURL: URL
        - Info
            - text: String
    - Dialog
        - text: String
        - character: Referencia
        - unlocks: [Referencias]
    - Missions
        - title: String
        - summary: String
        - type: TipoDeSolucion
        - Solutions
            - solution: <type>
            - Dialog
                - text: String
                - character: Referencia
                - unlocks: [Referencias]

Descripción de cada Colección

A continuación, se explica lo que representa cada uno de los campos mencionados en la estructura general. También se especifica cuáles de estos campos son obligatorios. Si no se menciona se consideran optativos.

Escapes

Es el conjunto de las diferentes partidas creadas para los usuarios. Cada una tiene su propia historia y no están relacionadas entre sí.

Name: Título del escape. [Obligatorio]

Synopsis: Pequeña descripción del escape que busca dar una imagen general de la partida sin explicar ningún punto importante de la historia. Busca atraer a los jugadores. [Obligatorio]

City: Nombre de la localidad donde se desarrolla el escape. [Obligatorio]

StartPoint: Lugar desde donde se inicia el escape. [Obligatorio]

Duration: Las horas y minutos que se prevé que dure la partida. Es un dato aproximado que se calcula una vez terminado el escape. [Obligatorio]

Characters

Lista de personajes que aparecen en la historia. Cada uno se representa con los siguientes datos:

Name: Nombre completo del personaje. [Obligatorio]

bigImageURL: Enlace a una imagen completa del personaje. Generalmente será de cuerpo completo y mirando hacia la derecha.

smallImageURL: Enlace a una imagen más pequeña que se usa como imagen de perfil, por lo que generalmente será la cara del personaje

Info: Colección de textos de información sobre el personaje que se van desbloqueando a lo largo de la partida. [Al menos 1 es obligatorio]

Dialog

Un dialogo es un conjunto de frases que representan una conversación, ya sea entre personajes o con el jugador.

Cada una de estas frases se compone de:

Text: La frase [Obligatorio]

Character: Referencia al personaje que dice la frase. Si no se indica se considera que lo dice un narrador o voz en off.

Unlocks: Lista de referencias a los elementos que se desbloquean en el momento de mostrar esta frase. Estos elementos pueden ser Personajes, misiones o información de algún personaje.

Missions

Colección de misiones.

Title: Título de la misión.

Summary: Resumen la misión. Contiene toda la información recabada en los diálogos anteriores y todo lo necesario para poder resolver la misión.

Type: Texto que define el tipo de dato necesario para resolver la misión. Puede ser:

- “text” – Dato de tipo String.
- “number” – Dato numérico.
- “location” – Geopoint que representa una localización a la que ir.

Solutions

Colección de las distintas soluciones posibles para una misión. Es obligatorio que exista al menos una solución para cada misión, pero podría haber más. Cada una de ellas tiene un diálogo asociado de forma que en función de la solución que se dé, se mostrará un diálogo distinto con distintos desbloqueos. De esta forma se puede, por ejemplo, dividir la historia en función de las decisiones del jugador.

> Solution: Dato que representa la solución de la misión. El tipo depende de lo indicado en la misión.

Firebase Storage

Firebase Storage es un servicio de firebase que permite almacenar archivos. Éstos pueden ser gestionados desde las aplicaciones que se conecten al servicio.

En nuestro caso se guardan las imágenes de los personajes y no se utiliza la funcionalidad completa del sistema ya que lo único que necesita la aplicación es descargar las imágenes para mostrarlas. Esto se puede conseguir fácilmente en Flutter mediante un Widget que se encarga de mostrar imágenes de internet a partir de una URL.

Firebase Storage nos proporciona la URL de acceso una vez subida cada imagen, por lo que sólo hay que copiar ese enlace y añadirlo a la base de datos tal y como está indicado en el apartado correspondiente.

La estructura de almacenamiento elegida consiste en guardar las imágenes organizadas por carpetas en función del escape al que pertenezcan. Las carpetas tienen como nombre el identificador del escape y dentro se encuentran 2 imágenes por cada personaje. Los nombres de estas imágenes también siguen un patrón. Ambas imágenes comienzan con el identificador del personaje, pero la pequeña termina con "_small". Todas las imágenes son png dado que es un formato que permite transparencias.

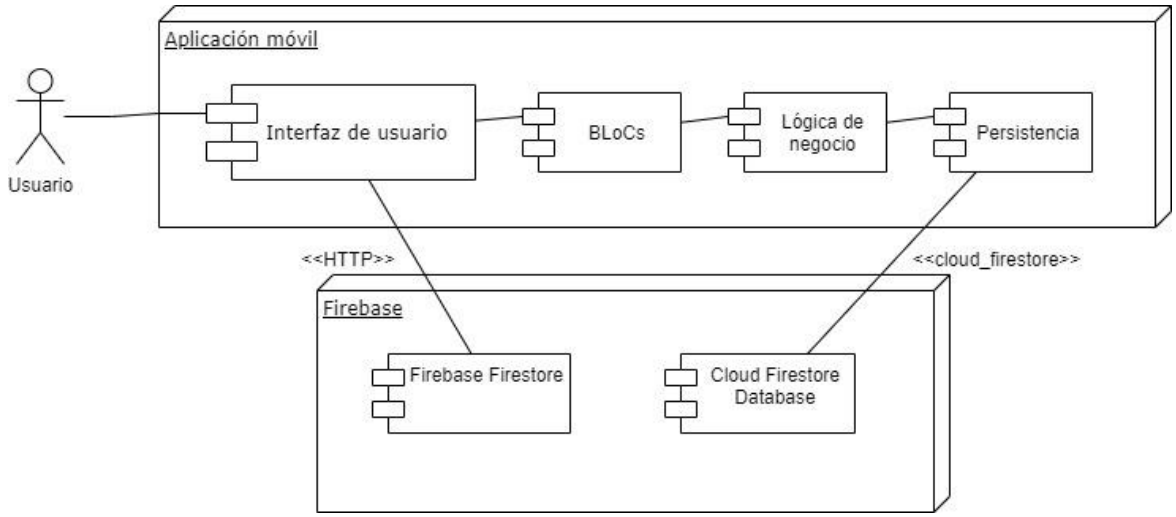
Por ejemplo, suponiendo que el escape tiene como identificador "historiaPorValencia" y que uno de los personajes tiene como identificador "agenteRamirez", sus imágenes estarían localizadas en:

> /historiaPorValencia/agenteRamirez.png
>
> /historiaPorValencia/agenteRamirez_small.png

Por motivos de seguridad se han definido unas reglas de acceso que impiden que la aplicación modifique de alguna manera los archivos guardados, pero que cualquiera pueda acceder a ellos.

3.3 Diseño de la lógica de negocio

La estructura del sistema sería el siguiente:



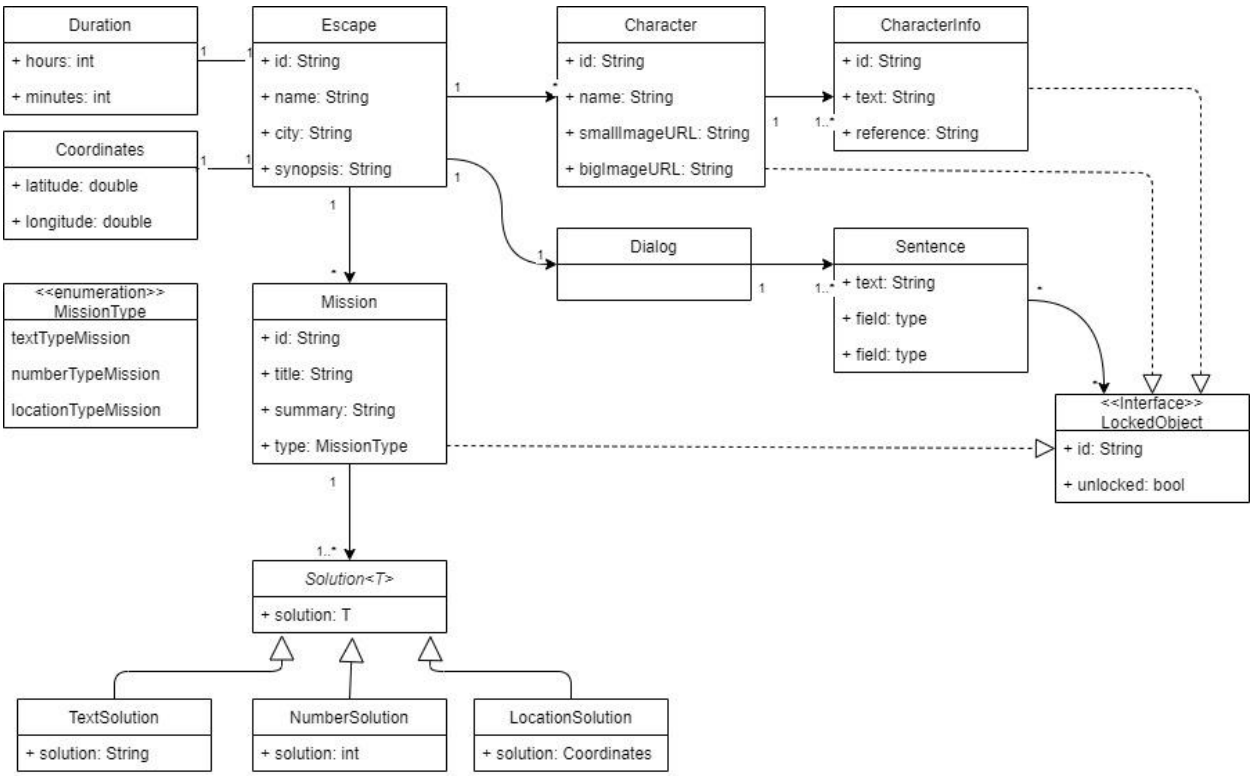
Se compone de dos partes. La aplicación móvil con la que interactúan los usuarios y un servidor de firebase en el que se almacenan los *escapes* con sus imágenes correspondientes.

La interfaz de usuario se comunica con Firebase Firestore mediante http para obtener las imágenes de cada escape.

Los BLoCs son los elementos encargados de detectar eventos en la aplicación y, en función de ellos, modificar la interfaz o notificar a la lógica de negocio.

La capa de persistencia solo se encarga de descargar la información de los escapes.

La lógica de negocio se compone del siguiente modelado de clases:



3.4 Patrón BLoC

El nombre BLoC viene de *Business Logic Components*. Se trata de un patrón de programación que, en nuestro caso, nos ayuda a gestionar el flujo de información a través de nuestros árboles de Widgets.

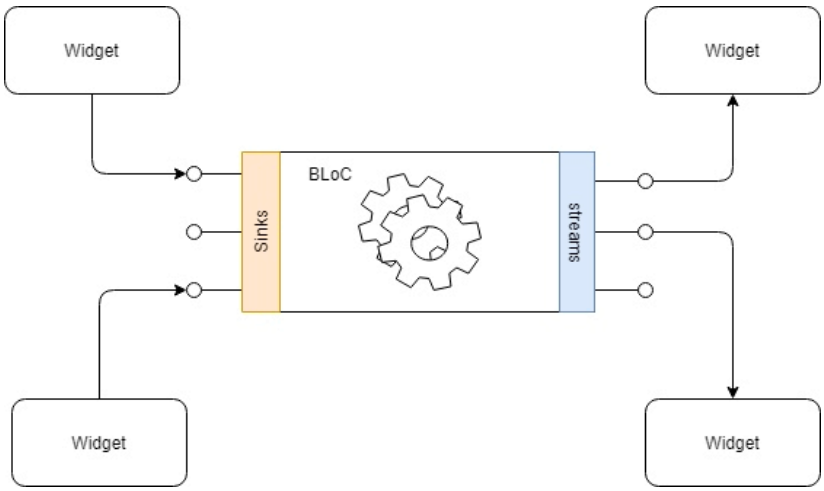
Es muy común que en nuestra aplicación necesitemos compartir información entre Widgets, por ejemplo, que un botón añada un elemento en un Widget de otra zona del árbol. En principio para poder compartir esta información, el dato debería viajar por todos los Widgets intermedios hasta llegar al adecuado. El patrón BLoC nos permite compartir información entre distintos Widgets sin importar dónde se encuentren.

El patrón funciona de la siguiente manera:

Se crea un objeto BLoC que será el encargado de gestionar los datos. A este BLoC se subscriben mediante *streams* los widgets que tengan que recibir actualizaciones. A través de estos streams se mandan las notificaciones de cambio de forma asíncrona a todos los Widgets que estén escuchando. Por otra parte, los widgets que tengan que enviar alguna información se la enviarán al bloc mediante *sinks*. Los *sinks* son una abstracción de objetos que reciben un dato.

Tanto las notificaciones que entran por los sinks como las que salen por los streams pueden contener información, de modo que cuando el objeto BLoC recibe una notificación de algún Widget, lo procesa y decide en función de la notificación recibida las notificaciones que debe mandar.

De esta manera cualquier widget puede mandar un evento al BLoC, este lo procesará y si corresponde mandará una notificación de cambio por los streams a los Widgets que estén subscritos.



Para simplificar la implementación de este patrón en nuestro proyecto utilizaremos el paquete *flutter_bloc*. Este paquete crea una capa de abstracción con respecto al envío de evento, la subscripción a los streams y el acceso a los BLoCs.

En primer lugar, nos da el Widget *BlocProvider*, a través del cual crearemos el BLoC y lo hará accesible desde sus Widgets inferiores.

```
BlocProvider(
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

Y por otro nos da dos opciones para detectar cambios en el estado del BLoC. Una de ellas es el Widget *BlocBuilder*. Este detecta los cambios en el estado del BLoC y reconstruye la interfaz con los nuevos datos.

```
BlocBuilder<BlocA, BlocAState>(
  builder: (context, state) {
    // Construcción del widget a partir del estado del bloc BlocA
  }
)
```

La otra opción para detectar los cambios es el Widget *BlocListener*. Es muy parecido al anterior, pero en vez de reconstruir la interfaz, nos permite ejecutar una función.

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // hacer cosas con el estado del bloc BlocA
  },
  child: Container(),
)
```

Para utilizar estas herramientas necesitamos un objeto Bloc y para ello necesitamos tres clases distintas: el propio BLoC, un evento y un estado.

La estructura que seguiremos en este proyecto es crear una clase abstracta para el estado y otra para el evento. De esta manera implementaremos los distintos eventos y estados del BLoC.

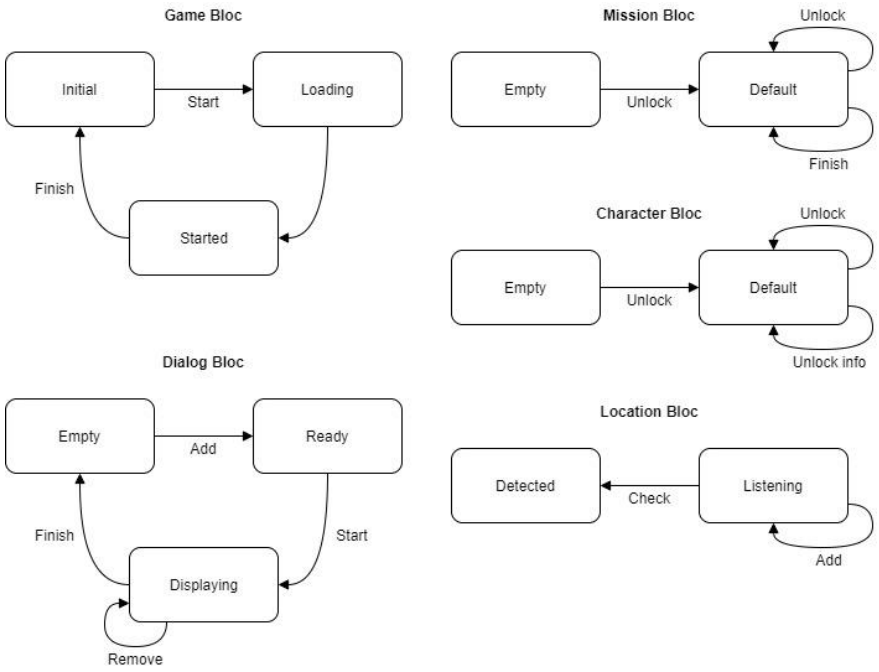
```
//BLOC
class CustomBloc extends Bloc<CustomEvent, CustomState>{
  @override
  LocationState get initialState => EmptyCustomState();

  @override
  Stream<CustomState> mapEventToState(CustomEvent event) async*{
    // Aqui se actualiza el estado del bloc a partir de un evento
  }
}

//ESTADOS
abstract class CustomState{}

class EmptyCustomState extends CustomState{}
class FullCustomState extends CustomState{}


//EVENTOS
abstract class CustomEvent{}

class AddCustomEvent extends CustomEvent{}
class RemoveCustomEvent extends CustomEvent{}
```

Para esta aplicación necesitamos 5 BLoCs:

1. Game. Este BLoC gestiona el estado general de una partida. Definiendo si se ha comenzado una partida, si no o si se está cargando. Dependiendo del estado se mostrará al usuario la pantalla de selección de escapes, la de carga o la de la partida.
2. Mission. Gestiona las misiones activas guardando la lista de las mismas. Sólo tiene dos estados, uno inicial en el que está vacío y otro en el que se guardan las misiones activas.
3. Character. Exactamente lo mismo que el anterior, pero para los personajes desbloqueados. También guarda la información desbloqueada de cada personaje.
4. Location. Guarda las coordenadas que deben lanzar un evento cuando el usuario llega a ellas. Recibe las coordenadas del usuario mediante eventos y si la localización coincide pasa a estado *Detected* para que la lógica de negocio se encargue del suceso.
5. Dialog. Como su nombre indica se encarga de los diálogos. En un primer momento se encuentra a la espera de nuevos diálogos. Cuando le mandan alguno cambia de estado para que la lógica de negocio se encargue de abrir la ventana adecuada, recoja el diálogo, lo muestre, etc. De esta manera pasa a al estado *Displaying*, desde donde también puede recibir más diálogos. Tras mostrar todos los diálogos, la lógica de negocio le notifica mediante un evento que puede volver a su estado inicial.

En los siguientes diagramas de estados se pueden ver los diferentes estados por los que pasa cada BLoC y los eventos que provocan los cambios de estado. Los diagramas representan los cambios que pueden surgir, pero el hecho de que reciba un evento no significa necesariamente que se cambie de estado. Los eventos y estados pueden contener información y en función de esta información, el BLoC analiza si tiene que cambiar de estado o no.

Por ejemplo, en el diagrama del BLoC “Location Bloc”, aparece un evento “Check” que cambiaría al estado “Detected”, pero esto sólo sucede si las coordenadas que se pasan en el evento coinciden con alguna de las que se tienen guardadas en el estado “Listening”.



4.0 Implementación

En base al estudio de herramientas realizado, en este proyecto se han utilizado las siguientes herramientas y tecnologías:

- Google Firebase Cloud Firestore como base de datos.
- Google Firebase Cloud Storage como servidor de imágenes.
- Android Studio como entorno de desarrollo.
- Flutter como SDK.
- Dart como lenguaje de programación.
- Android Emulator como simulador de dispositivos.

Flutter permite ampliar las funcionalidades de su sistema utilizando paquetes creados por ellos o por usuarios de la comunidad. Estos paquetes van desde Widgets hasta funcionalidades más internas.

Los que se han utilizado en este proyecto son:

```
dependencies:
  google_maps_flutter: ^0.5.28+1
  location: ^2.5.3
  geolocator: ^5.3.1
  url_launcher: ^5.4.2
  firebase_core: ^0.4.4+3
  cloud_firestore: ^0.13.4+2
  bubbled_navigation_bar: ^0.0.4
  flutter_bloc: ^4.0.0
  equatable: ^1.1.1
  hydrated_bloc: ^4.0.0
  json_annotation: ^3.0.1
  flutter:
    sdk: flutter

dev_dependencies:
  flutter_test:
    sdk: flutter
  build_runner:
  json_serializable: ^3.3.0
  flutter_launcher_icons: ^0.7.5
```

1. google_maps_flutter → https://pub.dev/packages/google_maps_flutter

Este paquete nos proporciona el Widget para mostrar un mapa de Google Maps. Permite controlar la posición de la cámara, zoom, agregar marcadores, definir qué controles aparecen, etc.

2. location → https://pub.dev/packages/location

Con este paquete se controla la posición del dispositivo. Con él podemos detectar la posición geográfica del jugador.

3. geolocator → https://pub.dev/packages/geolocator

Este paquete tiene varias funcionalidades, pero la que utilizamos en el proyecto es la de obtener información de un lugar a partir de unas coordenadas. Se usa para obtener la dirección (calle, número, localidad…) de las coordenadas de inicio de los escapes.

4. url_launcher → https://pub.dev/packages/url_launcher

Nos permite ir a una dirección web utilizando otras aplicaciones del dispositivo. Lo utilizamos para llamar a la aplicación de Google maps al pulsar en la tarjeta de dirección que podemos ver en la información de un escape.

5. cloud_firestore → https://pub.dev/packages/cloud_firestore

Este paquete nos permite acceder al servidor de Firebase.

6. bubbled_navigation_bar → https://pub.dev/packages/bubbled_navigation_bar

Se trata de un Widget que aporta un menú inferior con el que se puede navegar por diferentes vistas. Se puede ver en las pantallas principales durante las partidas.

7. flutter_bloc → https://pub.dev/packages/flutter_bloc

Como ya se había mencionado anteriormente, este paquete ayuda a implementar el patrón BLoC en Flutter.

8. equatable → https://pub.dev/packages/equatable

Genera la clase abstarcta *Equatable* que permite implementar la comparación entre objetos de la misma clase sin tener que generar las funciones correspondientes de forma manual.

9. hydrated_bloc → https://pub.dev/packages/hydrated_bloc

Se verá este paquete en profundidad en el apartado 4.2 Persistencia de datos

10. json_annotation → https://pub.dev/packages/json_annotation

Utilizando las etiquetas que nos aporta este paquete, podemos serializar los objetos en cadenas de texto con formato json:

```
part 'ExampleClass.g.dart';
@JsonSerializable()
class ExampleClass{
    int number;
    ExampleClass({this.number});
    factory ExampleClass.fromJson(Map<String, dynamic> json) =>
          _$ExampleClassFromJson(json);
    Map<String, dynamic> toJson() => _$ExampleClassToJson(this);
}
```

Una vez configuradas las clases ejecutamos el comando

```
flutter packages pub run build_runner build
```

Y se genera el archivo ExampleClass.g.dart con la implementación de los métodos *_$ExampleClassFromJson()* y *_$ExampleClassToJson()*.

4.1 Acceso al servidor

Hemos visto que el paquete *cloud_firestore* nos permite acceder al servidor de firebase, pero para que funcione debemos configurarlo adecuadamente.

Desde la consola de Firebase debemos agregar la aplicación Android siguiendo una serie de pasos. Después de esto podremos descargar un archivo json con las claves de acceso, identificadores, etc. Agregamos el archivo en el directorio Android/app.

Si quisiésemos configurarlo para iOS tendríamos que repetir los mismos pasos para esta plataforma, ya que no se puede hacer para ambas a la vez.

A partir de aquí ya podemos hacer consultas a la base de datos. Aquí podemos ver cómo se obtiene la lista de escapes:

```
Firestore db = Firestore.instance;
db.collection("Escapes")
        .getDocuments()
        .then((QuerySnapshot querySnapshot){
      List<Escape> lista = List();
      querySnapshot.documents.forEach((doc) => lista.add(
          Escape(
            id: doc.documentID,
            name: doc.data['name'],
            synopsis: doc.data['synopsis'],
            city: doc.data['city'],
            duration: Duration(
              hours: doc.data['duration'][0],
              minutes: doc.data['duration'][1],
            ),
            startPoint: Coordinates(
              latitude: (doc.data['startPoint'] as GeoPoint).latitude,
              longitude: (doc.data['startPoint'] as GeoPoint).longitude,
            ),
            dialog: doc.data['dialog'],
          )
      ));});
```

Desde *Firestore.instance* seleccionamos la colección que queremos explorar, luego indicamos que queremos todos los documentos y una vez recibida la información, la procesamos para crear una lista de objetos Escape.

4.2 Persistencia de datos

Ya hemos visto cómo se almacena la información de los escapes en la base de datos de firestore, pero durante el desarrollo de la aplicación surgió un problema: Android, el sistema operativo del dispositivo móvil, cierra la aplicación cuando no se está usando para ahorrar recursos. Esto hace que los datos referentes al progreso de la partida se puedan perder y el usuario tenga que volver a empezar.

Para solucionar este problema se analizaron las posibilidades de mantener la aplicación funcionando en segundo plano, pero para esto haría falta realizar llamadas al sistema operativo

anfitrión. Habría que comunicar Flutter con el sistema Android para generar un servicio que se mantuviese activo y del que obtener la información al recargar la aplicación.

Dado que esto añadía una alta complejidad al proyecto y que habría que diseñar un sistema independiente para cada sistema operativo, se descartó la idea y en su lugar se encontró el paquete *hydrated_bloc*.

Este paquete se encarga de almacenar y recargar los estados de los BLoCs. Para ello, cada vez que se actualiza el estado, se crea un json y se almacena. Cuando se recarga la aplicación, el BLoC en vez de empezar en su estado inicial, busca en el almacén si hay algún json y de ser así genera el estado correspondiente a la información almacenada.

Para implementar esto, el BLoC hereda de la clase *HydratedBloc* en vez de heredar de *Bloc*. Esto nos añade 2 funciones adiccionales: *fromJson* y *toJson*.

> *toJson* es la función que se llama cada vez que se actualiza el estado del BLoC y se encarga de convertir el estado actual en un *Map<String, dynamic>*.
>
> *fromJson* hace la función contraria, recibe lo producido por la anterior función y lo convierte en el estado que corresponda.

A continuación, podemos ver un ejemplo de un BLoC que almacena un número entero y tiene dos eventos, uno para aumentarlo y otro para disminuirlo.

```
class CounterBloc extends HydratedBloc<CounterEvent, CounterState> {
  @override
  CounterState get initialState {
    return super.initialState ?? CounterState(0);
  }

  @override
  CounterState fromJson(Map<String, dynamic> json) {
    try {
      return CounterState(json['value'] as int);
    } catch (_) {
      return null;
    }
  }

  @override
  Map<String, int> toJson(CounterState state) {
    try {
      return { 'value': state.value };
    } catch (_) {
      return null;
    }
  }

  @override
  Stream<CounterState> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield CounterState(state.value - 1);
        break;
      case CounterEvent.increment:
        yield CounterState(state.value + 1);
```

```
        break;
    }
  }
}

enum CounterEvent { increment, decrement }
class CounterState {
  int value;
  CounterState(this.value);
}
```

Podemos ver cómo al obtener el estado inicial comprueba el estado inicial de la clase *HydratedBloc* y en caso de devolver null, se define como *CounterState(0)*.

4.3 Detección de localizaciones

Una de las partes más importantes de la aplicación es el modo en el que se detecta cuando un jugador resuelve una misión de tipo localización. Para lograr esta funcionalidad se utilizan varias partes del código que veremos a continuación.

En primer lugar, vamos a ver cómo se construye la base de la que cuelgan todos los widgets de una partida.

```
Widget build(BuildContext context) {
  return BlocBuilder<GameBloc, GameState>(
    builder: (context, state){
      if(state is InitialGameState)
        return MainLayout(
          body: Center(
            child: CircularProgressIndicator(),
          ),
        );
        return MultiBlocProvider(
          providers: [
            BlocProvider.value(
              value: gameState.missionBloc,
            ),
            BlocProvider.value(
              value: gameState.characterBloc,
            ),
            BlocProvider.value(
              value: gameState.dialogBloc,
            ),
            BlocProvider.value(
              value: gameState.locationBloc,
            ),
          ],
          child: BlocListener<DialogBloc, DialogState>(
            listener: _checkDialogBlocChange,
            child: BlocListener<LocationBloc, LocationState>(
              listener: _checkLocationBlocChange,
              child: [...],
            ),),);},);}
```

Se puede ver cómo se recoge el estado del BLoC que gestiona si la partida está en marcha y en función de este, se muestra la pantalla de carga o la partida en sí.

Si la partida ya ha cargado, se añaden el resto de BLoCs al árbol para que sean accesibles a los Widgets inferiores y se detectan los cambios de estado en los BLoCs de diálogos y de localizaciones.

```
Widget build(BuildContext context) {
  return Scaffold(
    body: GoogleMap(
      onMapCreated: (GoogleMapController controller) {
        _locationTracker.onLocationChanged().listen((locationData) {
          LocationBloc locationBloc =
            BlocProvider.of<LocationBloc>(context);
          locationBloc.add(CheckLocationEvent(
            Coordinates(
              latitude: locationData.latitude,
              longitude: locationData.longitude)
            )
          );
        });},),);}
```

En este fragmento de código simplificado tenemos la construcción del mapa de Google y cómo una vez cargado, se crea un *listener* con el que se detecta cada actualización en la posición geográfica del jugador.

Al detectar este cambio, recoge el BLoC *LocationBloc* que se creó en el primer fragmento de código y le envía la nueva localización detectada para que compruebe si es solución de alguna de las misiones activas.

La estructura del BLoC sería la siguiente:

```
class LocationBloc extends Bloc<LocationEvent, LocationState>{

  Stream<LocationState> mapEventToState(LocationEvent event) async*{
    if (event is CheckLocationEvent)
      yield* _checkLocation(event);
  }

  Stream<LocationState> _checkLocation(CheckLocationEvent event) async*{
    if(state is ListeningLocationState) {
      for (var obj in state.listeningList.entries) {
        if(_checkDistance(obj.key, event.coordinates)<MAX_DISTANCE) {
          state.listeningList.remove(obj.key);
          Solution solution = obj.value.solutions.firstWhere(
            (Solution s) => s is LocationSolution && s.solution==obj.key
          );
          yield DetectedLocationState(
            mission: obj.value,
            solution: solution,
            listeningList: state.listeningList
          );
        }}}}}
```

Aquí podemos ver cómo, en primer lugar, detecta si el evento que recibe es para comprobar una nueva localización y en ese caso recorre la lista de localizaciones activas comprobando que la distancia entre ellas y la posición del jugador es menor que un parámetro establecido.

En ese caso cambia el estado del BLoC a *DetectedLocationState* para que el primer fragmento de código que hemos visto lo detecte y lo gestione de la siguiente manera

```
void _checkLocationBlocChange(BuildContext context, LocationState state)
{
    if (state is DetectedLocationState) {
      gameState.missionBloc.add(FinishMissionEvent(state.mission));
      gameState.dialogBloc.add(AddDialogEvent(state.solution.dialog));
      gameState.locationBloc.add(SetListeningEvent());
    }
}
```

Lo primero que hace es finalizar la misión correspondiente a esa localización, añade el diálogo correspondiente a la lista de diálogos para mostrar y notifica al BLoC de localización que ya ha gestionado la localización detectada.

4.4 La venganza de César Borgia

“La venganza de César Borgia” es el título del escape creado a partir de la historia diseñada por los alumnos de Innovación educativa en Didáctica de las Ciencias Sociales.

La adaptación de la historia a la aplicación se realizó junto con el profesor Diego Téllez. Si bien la historia que se quería contar estaba definida, había huecos que debíamos cubrir y modificaciones que hacer para que la historia se ajustase a la aplicación. Algunos de estas tareas son:

- Redactar muchos diálogos, ya que algunos alumnos los definieron y otros no. Algunos alumnos explicaban las misiones definiendo lo que encontrarían los jugadores y cómo se resolverían, sin embargo, en la aplicación esto se hace mediante diálogos.
- Hubo que cambiar algunas misiones dado que las funcionalidades que ofrece la aplicación no permitían resolverlas como estaba diseñado en un primer momento. Un ejemplo de esto es una misión en la que se debía mostrar una imagen desde la cual se obtenía la clave para resolver el enigma.
- También tuvimos que redactar los resúmenes de todas las misiones, que es donde se muestra toda la información de la misión que se ha obtenido de los diálogos.
- Se redactó la información que daríamos de cada personaje. Este escape tiene una importante carga cultural, por lo que resultaba muy importante que se mostrasen datos de los personajes históricos y que esa información fuese correcta.
- Por último, se planificaron los desbloqueos de misiones, personajes y la información de estos.

El resultado de este trabajo fue recogido en un documento antes de pasar a la base de datos. Este documento puede verse en el anexo 3.

4.4 Pruebas

Para realizar las pruebas del sistema durante la implementación se creó un escape simple con los siguientes elementos:

Dos personajes, uno de ellos desbloqueado desde el principio y con información adicional disponible para desbloquear.

Una misión principal desde la que, dependiendo de la solución que se daba, se testeaban diferentes funcionalidades de la aplicación. Estas posibles soluciones serían:

1. Text. Desbloquea una misión con solución de tipo texto.
2. Number. Desbloquea una misión con solución de tipo numérico.
3. Location. Desbloquea una misión con solución de tipo localización.
4. Info. Desbloquea información adicional del primer personaje.
5. Dialog. Muestra un diálogo entre varios personajes.
6. Character. Desbloquea el segundo personaje.

Todas estas soluciones vuelven a desbloquear la misión principal, lo que resulta en un bucle.

Para realizar una prueba final del sistema y detectar posibles errores en el escape “La venganza de César Borgia”, nos desplazamos el día 4 de julio a Viana. Allí probamos la aplicación y detectamos varios errores.

Estos fueron los errores detectados:

1. No aparecen los diálogos que deberían al completar una misión de tipo localización. La aplicación solo detecta correctamente la primera localización y hay que reiniciar la aplicación para que carguen los diálogos de las siguientes misiones.
2. Aparecen diálogos repetidos, lo que provoca que se desbloqueen misiones que ya habían sido resueltas.
3. En algunos dispositivos no se pueden leer los diálogos más largos.
4. Una errata en un título de la aplicación.
5. Localizaciones imprecisas en la historia.

Algunos de estos errores eran de menor importancia, como la errata y las localizaciones erróneas que se modificaron posteriormente sin mayor dificultad. Pero también se detectaron 2 errores importantes, en primer lugar, la aplicación no detectaba cuándo el jugador llegaba a la localización que le indicaba la prueba, por lo que debíamos reiniciar la aplicación para poder avanzar en la historia. Por otro lado, en determinadas circunstancias se reproducían diálogos que no debían, lo que provocaba que se abriesen misiones que ya habían sido resueltas.

Además de detectar estos errores se propusieron algunas mejoras:

1. Poder avanzar y retroceder en los diálogos.
2. Incorporar un *scroll* en los textos de los diálogos para asegurar que se puedan leer desde todos los dispositivos.
3. En las misiones de tipo localización, añadir un botón que te lleve a la ventana del mapa.

Todos los errores fueron solventados la semana siguiente y se añadieron las dos primeras propuestas.

5.0 Seguimiento y Control

En este apartado se muestran los objetivos iniciales cumplidos, el trabajo realizado y los problemas surgidos durante el desarrollo del proyecto.

5.1 Objetivos alcanzados

Por lo general los objetivos iniciales se han cumplido, aunque algunos de ellos han sufrido ligeras matizaciones:

1. Tras finalizar el diálogo inicial se mostrará un mapa centrado en la posición geográfica del jugador y con marcas en lugares relevantes para la historia.

   En el mapa se marca en todo momento la posición del jugador (siempre que tenga el gps activado y nos haya dado permisos para acceder a su posición) y también las localizaciones a las que las misiones de localización les dirija.

2. Las misiones podrán iniciarse tras desbloquearse durante un diálogo o al llegar a un lugar relevante para la historia.

   Las misiones, personajes e información de los personajes solo se pueden desbloquear mediante diálogos. Este objetivo se alcanza iniciando un diálogo al llegar a una localización marcada por una misión.

5.2 Desviaciones

Durante el mes de marzo comenzó el periodo de cuarentena por el brote de COVID-19 en todo el país. El miércoles 11 de marzo de 2020 se cancelaron las clases presenciales de la universidad y la planificación de horario personal prevista para el desarrollo del proyecto cambió. En un primer momento el desarrollo se paró por completo hasta la normalización de la situación. Esto duró dos semanas y después se retomó el proyecto.

Dada la situación de no poder salir de casa más de lo necesario, la única tarea durante la cuarentena era realizar este proyecto, por lo que la dedicación de horas de trabajo aumentó con respecto a lo planificado.

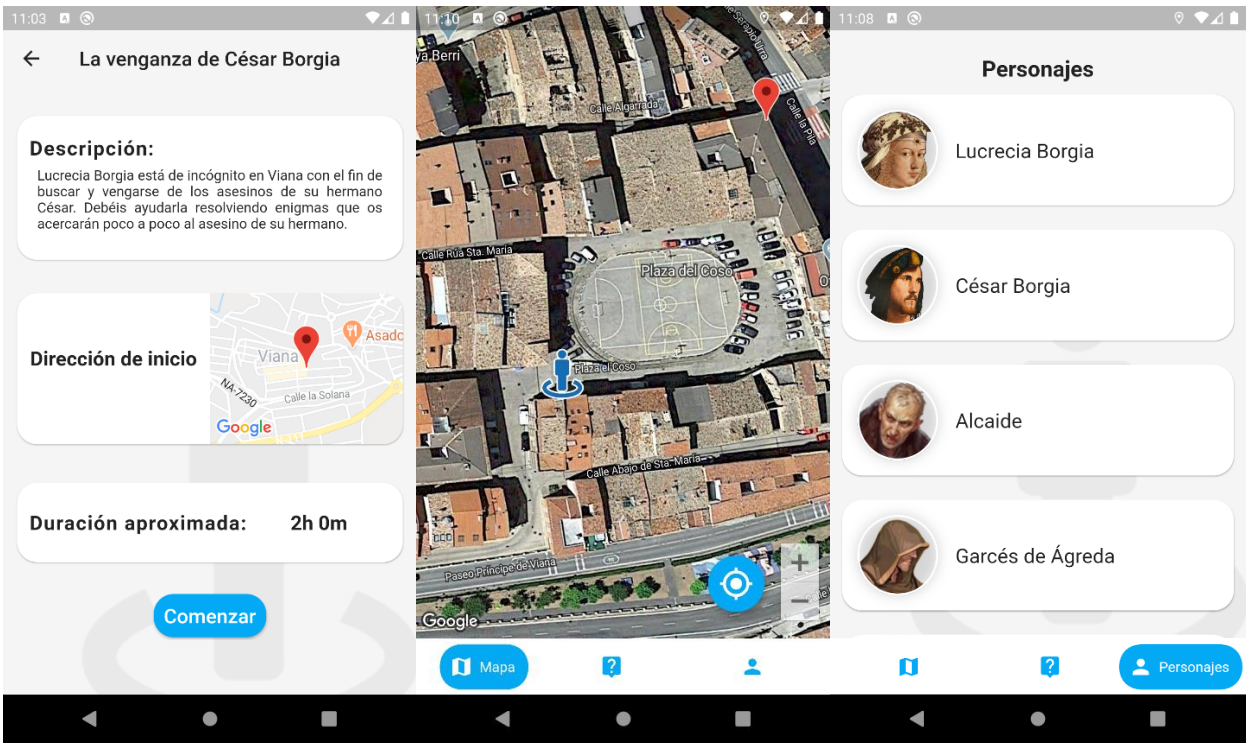
Esto compensó el retraso de las primeras semanas de cuarentena, por lo que el tiempo total de trabajo no cambió mucho, aunque sí las fechas de la realización de cada parte del proyecto.

Por otra parte, las pruebas finales se tuvieron que posponer debido a la imposibilidad de desplazarnos a Viana. Se pudieron realizar el 4 de julio y los problemas detectados se corrigieron la semana siguiente.

Por ello la fecha de finalización del proyecto se atrasó a la tercera semana de julio.

5.3 Interfaz

A continuación, se puede ver el resultado final de la interfaz de la aplicación:



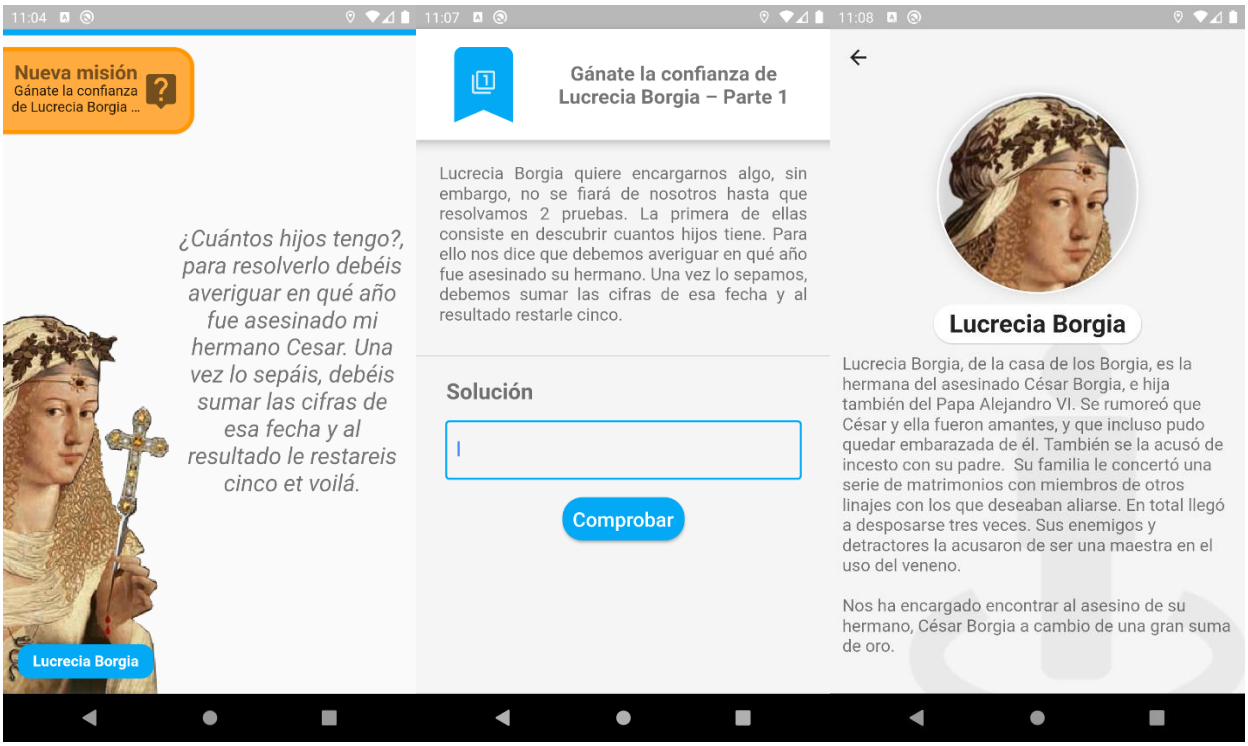
En la primera imagen se pueden ver las especificaciones del Escape "La venganza de César Borgia". Incluye la descripción, La localización donde debe iniciarse la partida y la duración aproximada del Escape. Al pulsar en la tarjeta de la Dirección de inicio se abre la aplicación de Google Maps con la ruta desde nuestra posición hasta la localización de inicio.

En la segunda imagen podemos ver en la parte inferior el menú principal durante la partida. Desde este se puede cambiar la vista entre el mapa (como en este caso), las misiones disponibles y los personajes (tercera imagen).

En el mapa encontramos el icono de una persona, que representa la posición del jugador y también vemos un marcador que nos indica el lugar al que nos dirige alguna misión activa. Al pulsar sobre el marcador aparece el nombre de la misión a la que pertenece.

En la parte inferior derecha tenemos varios controles. En primer lugar, un + y un – que nos permiten hacer zoom en el mapa. Además, podemos movernos por el mapa como haríamos en la aplicación de Google Maps, permitiéndonos hacer zoom, girar la cámara, etc.

El segundo control del mapa nos permite anclar la cámara a la posición del usuario, es decir, la cámara seguiría al jugador. Por defecto esta opción está activada y no saldría el control, sino que saldría al alejar la cámara de la posición del jugador.



La primera imagen nos muestra cómo se ven los diálogos. El texto aparece a la derecha y a la izquierda una imagen del personaje que nos habla.

En la parte superior izquierda nos aparecen notificaciones cuando se desbloquean misiones, personajes e información de personajes. Además, en la parte superior encontramos una barra de progreso que nos indica el avance del diálogo, ya que podemos avanzar y retroceder dependiendo de si pulsamos la parte de la izquierda o la de la derecha de la pantalla.

La siguiente imagen pertenece a la descripción de una misión activa. En este caso se trata de una de tipo numérico, por lo que se nos abre un teclado numérico al introducir la solución.

Por último, vemos la información de un personaje. Aquí se recoge toda la información de un personaje desbloqueada durante la historia.

6.0 Conclusiones

Este proyecto ha establecido una base para algo mucho más ambicioso. Las aplicaciones actuales que implementan este tipo de juegos suelen ser de una única historia y cobran un alto precio por cada partida. Sin embargo, con esta aplicación el desarrollo de los Escapes es menos costoso que crear una aplicación para cada uno.

En este momento las funcionalidades son muy limitadas, pero se ha estructurado todo el proyecto para que se puedan seguir añadiendo funcionalidades al sistema y que la aplicación se amolde a las necesidades de cada Escape.

En el caso del Escape “La venganza de César Borgia”, la narrativa de la historia es lineal. Cuando el jugador termina una misión se le abre otra y así sucesivamente. No utiliza las herramientas de la aplicación para crear varias líneas narrativas. Pero con el estado actual de la aplicación se podría también crear un Escape que ni siquiera requiriese del servicio de geolocalización.

Ofreciendo muchas opciones y haciendo que todas ellas sean opcionales se consigue que la variedad de Escapes que se puedan crear sea muy amplia.

Hay muchas funcionalidades que se pueden implementar para mejorar el estado actual de la aplicación y aquí recogemos algunas de ellas:

1. Determinar alguna misión como final de forma que al completarla se termine la partida.
2. Enviar notificaciones a los usuarios cuando se publique un nuevo escape.
3. Establecer un límite de tiempo o un contador durante la partida.
4. Crear un sistema de pistas que se desbloqueen de alguna de las siguientes maneras:
    a. En función del tiempo. Si el jugador tarda mucho en descifrar un enigma se le desbloquea una pista, si después de un tiempo sigue sin conseguirlo se desbloquea la siguiente y así sucesivamente.
    b. Comprándolas con algún tipo de moneda virtual.
    c. Consumiendo parte del tiempo del contador.
5. Determinar la localización de los personajes en el mapa. Se podrían mover durante la partida.
6. Nuevos tipos de misiones:
    a. Tipo año. Como el tipo numérico pero que representa un año, por lo que tendría 4 dígitos.
    b. Tipo opción. Te da a elegir entre 2 o más opciones.
    c. Tipo localización oculta. Como la localización actual pero no se muestra la localización ni en el mapa ni en las misiones. El jugador se puede encontrar con el diálogo al pasar por algún lugar.
7. Modificadores para las misiones.
    a. El jugador debe estar en una localización concreta para poder resolver el enigma. Por ejemplo, hay que darle un código a un personaje, por lo que el jugador no podrá escribir el código hasta que esté en la localización en la que se encuentre el personaje.
    b. Las misiones de localización no se muestran en el mapa. El jugador debe descubrir el lugar al que le dirige la historia.
8. Dependencia entre misiones. Al finalizar alguna misión, algunas de las activas pueden dejar de tener sentido por lo que tendrían que desactivarse.

Pero ante todo la primera y principal mejora que se debería de implementar a la aplicación es la posibilidad de que los usuarios creen sus propios Escapes. De esta manera se impulsaría el uso de la aplicación y se podría crear una cantidad de contenido importante.

Se podría hacer de varias formas, pero a mi parecer, las más importantes serían desde la propia aplicación o desde un navegador web.

Si se hace desde la propia aplicación, cualquier usuario podrá hacer su propio Escape, pero si se quiere hacer un Escape relativamente complejo, hacerlo desde el dispositivo móvil puede ser un inconveniente. En este caso sería mejor crearlo desde un ordenador mediante una página web.

Bibliografía

Aplicaciones hibridas contra nativas

https://stackoverflow.com/questions/14065610/struggling-between-native-and-phonegap-simple-app-requirements

Estadísticas uso de sistemas operativos en móviles general:

https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

https://gs.statcounter.com/os-market-share/mobile/worldwide

Estadísticas uso de sistemas operativos en móviles en España:

https://es.statista.com/temas/4086/consumo-y-uso-de-smartphones-en-espana/#dossierSummary__chapter2

https://es.kantar.com/tech/m%C3%B3vil/2019/abril-2019-cuota-de-mercado-de-smartphones/

Comparativa de uso de sistemas operativos en móviles EEUU vs España:

https://www.kantarworldpanel.com/global/smartphone-os-market-share/

General: 74% Android 25% iOS

España: 90% Android 10% iOS

Xamarin:

https://dotnet.microsoft.com/apps/xamarin

https://softwarecrafters.io/xamarin/xamarin-forms-apps-nativas-introduccion

Phonegap:

https://phonegap.com/

https://www.arsys.es/blog/programacion/diseno-web/que-es-phonegap/

Ionic

https://ionicframework.com/

https://openwebinars.net/blog/ionic-framework-que-es/

Flutter

https://flutter.dev/

https://flutter-es.io/docs

https://pub.dev/

Comparativa de bases de datos Firebase, Realtime Database contra Cloud Firestore:

https://firebase.google.com/docs/database/rtdb-vs-firestore

Precios y límites de Firebase:

https://firebase.google.com/pricing