CultuCat

MEMORIA

Projecte d'Enginyeria del Software

Q2 2024/2025

FIB

GRUP 11

Pol Boncompte Díez
Max Estradé Pey
Marc Herrera Moliné
Ángel Jiménez Torres
Marcos Martínez Martínez
Shence Zhou Ye

Índice

Índice de figuras

1. Requisitos

1.1. Concepción General del Proyecto

Nuestro proyecto consiste en el desarrollo de una aplicación móvil enfocada a compartir información de eventos culturales en Cataluña. Las actividades se mostrarán en un mapa interactivo para que así el usuario pueda ver las actividades cercanas a él. También tendrá la posibilidad de buscar actividades por nombre o categoría. En la propia aplicación se podrá ver toda la información disponible de las actividades, al igual que otros enlaces del evento.

La innovación principal de nuestra aplicación es que a esta plataforma le añadimos un componente social. En la aplicación se podrá formar grupos con otros usuarios para ir a eventos, se podrán realizar reseñas de eventos a los que el usuario haya asistido, se podrán compartir imágenes de los eventos…

La calidad de la base de datos es suficiente para nuestro objetivo, ofreciendo un conjunto actualizado de todos los proyectos culturales de Cataluña registrados por la Generalitat (fecha de inicio y final, localización, precio de entrada…).
Con estos datos podremos cumplir nuestro objetivo de mostrar las mejores actividades en las cercanías de los usuarios.

Figura 1. Diversos puntos de interés cultural en Cataluña.

1.2. “NOT” list

Para ver el alcance que supondrá nuestro proyecto hemos realizado una NOT list, en la que clasificamos las funcionalidades más importantes que están dentro o fuera de nuestra aplicación. Además, hemos clasificado aquellas funcionalidades que todavía no tenemos claras si formarán parte del software como “puede ser”. Estas últimas están pendientes de análisis para ver si es factible llevarlas a cabo con el tiempo y los medios que tenemos.

DENTRO	**FUERA**
Registro e inicio de sesión (Gestión de cuenta de usuario).	Regulación automática de mensajes de odio o que violen las normas de la aplicación que aseguran el bienestar de la comunidad.
Mapa interactivo.	Ofrecer actividades culturales fuera de Cataluña.
Informar sobre eventos o lugares de interés cultural de Cataluña. (notificaciones según preferencias)	Indicar lo sostenible que son los eventos.
Buscador de actividades con filtro.	Guardar varias sesiones de usuarios diferentes.
Añadir y gestionar amistades.	Informar sobre el estado meteorológico.
Calendario de tus actividades de la propia aplicación.	Garantizar la veracidad y confianza de un perfil.
Crear grupos de amigos para participar juntos en determinados eventos.	Sistema de indicaciones detalladas y asistente de voz indicando las direcciones.
Idiomas de la aplicación: Catalán, Castellano e Inglés.	Compra/Venta de entradas.
Trazar rutas hacia los eventos/lugares indicando el transporte público recomendado para llegar (línea de metro, autobús, tranvía, etc.)	Subir fotos de los eventos o lugares a los que ha asistido el usuario.
Valoraciones y reseñas de actividades.	A partir de tus preferencias, un horario y localización te genera una lista de opciones de actividades que puedes hacer (Planificador).

Compatibilidad en Android y preparado para ser compilado para iOS.	Indicar los puntos de información.
Mensajería instantánea.	Reportar y bloquear usuarios.
Calcular el tránsito en tiempo real en el momento de hacer la ruta hacia el evento.	Sincronizarse con calendarios externos (Google)
Widget que a partir de tu geolocalización te muestre las actividades que tienes cerca.	Acceder a la página web de la actividad.
Comentarios, me gusta y compartir.	
Apartado de eventos o lugares “Trending”.	

PUEDE SER

1.3. Modelo Conceptual

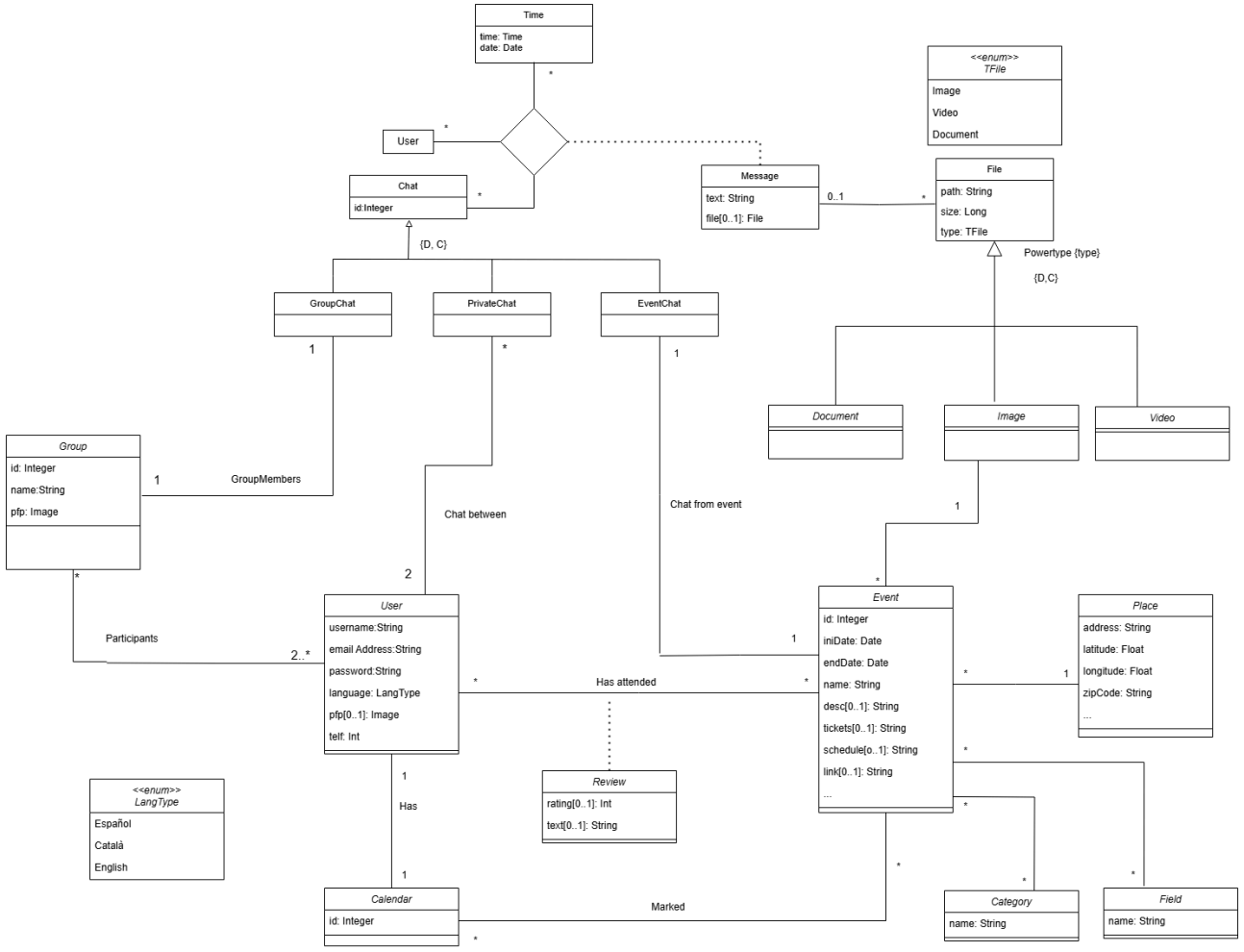

Figura 2. Modelo conceptual de CultuCat

Restricciones Textuales:

R1: Claves primarias: (User, username), (Event, id), (Chat, id), (Group, id), (Time, time + date), (Calendar, id), (Place, latitude + longitude), (File, path), (Category, name), (Field, name).
R2: No puede haber 2 *Users* con el mismo *email.*
R3: No puede haber 2 *Users* con el mismo *telf*
R4: No puede haber 2 *Place* con la misma *address*
R5: El *rating* de una *Review* ha de ser mayor o igual que 0 y menor o igual que 10
R6: La *latitude* de un *Place* debe ser mayor que -90 y menor que 90
R7: La *longitude* de un *Place debe ser mayor que* -180 y menor que 180
R8: En un *Event* la *iniDate* ha de ser anterior a la *endDate*.
R9: El password de un *User* debe tener al menos 8 caracteres, incluyendo mayúsculas y minúsculas, y como mínimo un número y un carácter especial.
R10: Un grupo tiene como mínimo y máximo 1 creador

1.4. Resumen del Product Backlog

Épicas	Historias de usuario	Sprint que empieza	Sprint que termina
GESTIÓN DE USUARIOS	Registro de usuario	1	1
	Login	1	1
	Cerrar sesión	2	2
	Ver perfil	1	2
	Editar perfil	2	2
	Recuperación de contraseña	1	1
	Seleccionar idioma	2	2
	Eliminar cuenta	2	2
	Login Google	2	2
	Mejorar perfil	3	3
	Ver perfil de un amigo	3	3
	Implementar amigos	3	3
GESTIÓN DE BÚSQUEDA	Búsqueda por nombre	1	1
	Búsqueda por categoría	2	2
	Búsqueda por ubicación	2	2
	Búsqueda por fecha	2	2
	Filtrar por rango de fechas	2	2
	Filtrar por varios	2	2
GESTIÓN DEL MAPA INTERACTIVO	Visualización de eventos en el mapa	1	1
	Ubicación en tiempo real	1	1
	Filtro de eventos en el mapa	2	2
	Detalles de eventos en el mapa	1	1
	Indicaciones para llegar al evento	2	3
	Mejorar visualización del evento	3	3
GESTIÓN DE FUNCIONALIDADES SOCIALES	Creación de grupos	2	2
	Unirme a un grupo	2	2

	Reseñas de eventos	2	2
	Implementar chat del evento	2	2
	Chatear en grupo	2	2
	Compartir eventos	2	3
	Chatear en privado	2	2
GESTIÓN DE VALORACIONES	Dar like / dislike a reseñas de otros	2	3
	Ver valoraciones destacadas	X	X
	Reportar valoraciones inapropiadas	X	X
	Valoración por criterios	X	X
	Ranking de eventos por valoraciones	3	3
	Historial de valoraciones	3	3
GESTIÓN DE NOTIFICACIONES	Notificaciones de nuevos eventos	X	X
	Notificaciones de invitaciones a grupos	3	3
	Recordatorio de eventos guardados	3	3
	Notificaciones de comentarios	2	3
	Notificaciones de cambios en eventos	3	3
	Configurar notificaciones	2	X
	Notificaciones de eventos en tendencia	X	X
GESTIÓN DE CALENDARIO	Añadir eventos al calendario	2	2
	Sincronizar con otros calendarios	3	X
	Ver mi calendario semanal o mensual…	2	2
	Historial de eventos asistidos	X	X
GESTIÓN DE SERVICIOS	Redireccionar a la web del evento	2	X
	Reserva de plazas en eventos gratuitos	X	X
	Acceso a descuentos y promociones	X	X
	Acceso a servicios de comida	X	X

	Contacto con organizadores	3	X
GESTIÓN DE SEGURIDAD Y PRIVACIDAD	Gestión de permisos	X	X
	Activar modo privado	X	X
	Bloqueo de usuarios	X	X
	Denuncia contenido inapropiado	X	X
	Autenticación en dos pasos	X	X
	Configurar visibilidad	X	X
	Hacer Tests	2	3
GESTIÓN DE SOPORTE Y AYUDA	Centro de ayuda	3	3
	Chat de soporte	X	X
	Soporte por correo electrónico	3	3
	Reporte de fallos	3	3
	Sistema de tickets	X	X
GESTIÓN DEL PROYECTO Y SERVIDOR	Configurar CI/CD para el repositorio front-end	2	2
	Configurar CI/CD para el repositorio back-end	2	2
	Configurar el servidor para ejecutar el proyecto	2	2

1.5. Requisitos No Funcionales

Nº 1	**Facilidad de uso**
Descripción	Este requisito describe las características que ha de tener el sistema para que este resulte fácil de usar a los usuarios, que pueden variar de acuerdo a las habilidades de los usuarios y de la complejidad del sistema.
Clasificación Volere	11a. Requisitos de Facilidad de uso
Justificación	Todos los clientes han de tener una buena experiencia usando el sistema, para ello la aplicación ha de ser fácil de interactuar. De esta manera, se sentirán más satisfechos con la plataforma.
Condición de satisfacción	Los usuarios han de poder usar nuestro sistema sin tener conocimiento previo de cómo se usa. Este ha de ser intuitivo para nuevos usuarios. Haremos una encuesta a gente que no ha probado la aplicación anteriormente y un 90% han de poder usar nuestro sistema sin dificultades.

Nº 2	**Personalización**
Descripción	Este requisito describe las maneras en que el producto puede ser modificado o configurado para tener en cuenta las preferencias personales del usuario. Ha de tener aspectos como el idioma y opciones personales de configuración.
Clasificación Volere	11b. Requisitos de Personalización y Internacionalización
Justificación	Todos los clientes han de tener una buena experiencia usando el sistema, para ello la aplicación se ha de adaptar a los usuarios.
Condición de satisfacción	La aplicación permite la opción de cambiar el idioma del sistema.

Nº 3	Escalabilidad
Descripción	Esto especifica los aumentos esperados en dimensión que el producto debe ser capaz de manejar. Cuando la empresa crezca, nuestro sistema debe incrementar sus capacidades para tratar los nuevos volúmenes.
Clasificación Volere	12g. Requisito de Expansión o Crecimiento
Justificación	El sistema debe permitir el crecimiento a futuro. Es decir, el diseño inicial debe de estar preparado para que el sistema crezca.
Condición de satisfacción	El producto será capaz de procesar a los clientes existentes. El producto debe estar preparado para recibir nuevas funcionalidades.

Nº 4	Soporte
Descripción	Describe el nivel de soporte que el producto requiere.
Clasificación Volere	14b. Requisitos de Soporte
Justificación	La aplicación deberá proporcionar soporte con el objetivo de ayudar a los usuarios a resolver dudas o problemas que puedan tener con el objetivo de que todos los usuarios puedan usar el sistema y tengan una experiencia agradable.
Condición de satisfacción	La aplicación tiene que disponer de un apartado de Ayuda donde se muestren las preguntas frecuentes.

Nº 5	Privacidad
Descripción	Describe lo que el producto tiene que hacer para asegurar la privacidad de las personas sobre quienes almacena información.
Clasificación Volere	15c. Requisitos de Privacidad
Justificación	El sistema tiene que mantener la información protegida y solo disponible para el personal autorizado con el objetivo de evitar filtraciones de datos personales de nuestros usuarios.
Condición de satisfacción	El sistema hará conscientes a sus usuarios sobre sus prácticas de información antes de recolectar datos de ellos. El sistema notificará a los clientes sobre los cambios en su política de información. El sistema revelará información confidencial solamente de acuerdo con la política de información de la organización.

Nº 6	Disponibilidad
Descripción	Describe el tiempo durante el cual el sistema debe estar disponible para su uso, incluyendo tiempos permitidos de caída y mantenimientos planificados.
Clasificación Volere	13b. Requisitos de Disponibilidad
Justificación	El sistema debe estar disponible de forma continua ya que los usuarios pueden acceder en cualquier momento, especialmente durante fines de semana o festivos.
Condición de satisfacción	El sistema estará disponible al menos el 99% del tiempo, con los mantenimientos que se anunciaran previamente.

Nº 7	Rendimiento
Descripción	Describe cuánto de rápido debe responder el sistema bajo determinadas condiciones de carga. Incluye tiempos máximos de respuesta aceptables y capacidad para mantener la calidad del servicio.
Clasificación Volere	13a. Requisitos de Rendimiento
Justificación	Para ofrecer una buena experiencia al usuario, el sistema debe cargar actividades, chats y otras funcionalidades sin demoras perceptibles.
Condición de satisfacción	El sistema debe responder a cualquier solicitud del usuario en menos de 3 segundos en el 95% de los casos bajo condiciones normales de carga.

1.6. Aspectos Transversales

- **Geolocalización:** La aplicación lee la ubicación del usuario para que este pueda ver los eventos que tiene cerca y para recibir notificaciones de eventos cerca de él.
- **Redes sociales:** La aplicación permite el inicio de sesión con una cuenta de Google y compartir eventos mediante un enlace.
- **Gamificación:** Nuestra aplicación permitirá publicar reseñas a los eventos que los usuarios atiendan.
- **Stakeholders reales:** Hemos presentado la aplicación a diferentes usuarios y hemos recogido su feedback mediante una encuesta. Hemos implementado alguno de los cambios sugeridos.
- **Refutación:** La aplicación envía notificaciones al usuario cuando recibe algún mensaje, cuando alguien vote alguna de sus reseñas, cuando esté cerca de un evento en su calendario, etc.
- **Chat:** Creemos que la comunicación es un aspecto importante de nuestra aplicación, por eso incluimos varios chats (personales, de grupo y de evento).
- **Calendario:** Nuestro sistema contará con un calendario personal para cada usuario, para que pueda ver los eventos a los que está apuntado.
- **Web-app admin:** No tenemos pensado implementar una web de administración.

- **Multiidioma:** Queremos que nuestra aplicación también sea útil para gente fuera de Cataluña, para eso incorporaremos una selección de idioma dentro de la aplicación.

Aspecto	Estado actual
Geolocalización	La aplicación puede recibir la ubicación del usuario y mostrarla en el mapa.
Redes sociales	La aplicación permite iniciar sesión con una cuenta de Google y compartir eventos mediante otras redes sociales.
Gamificación	La aplicación permite que se puedan escribir reseñas.
Chat	La aplicación permite tres tipos de chats: personales, de grupo y de evento. Todos los chats se actualizan al momento mediante notificaciones push y se pueden ver los mensajes anteriores.
Stakeholders reales	Se ha hecho una encuesta a 23 personas y se ha implementado parte del feedback.
Refutación	Se reciben notificaciones mediante Firebase Messaging.
Chat	La aplicación cuenta con chats personales, de grupo y de evento.
Calendario	Calendario funcional en la aplicación.
Web-app admin	-
Multiidioma	La interfaz de nuestra aplicación está disponible en catalán, inglés y castellano y se ha implementado de manera que permita la inclusión de nuevos idiomas.

1.7. Servicios Externos

Hemos llegado a un acuerdo con **Aire Lliure** para proporcionarles un servicio exclusivo que les permitirá acceder a una API con información actualizada sobre los diferentes eventos culturales y de entretenimiento que tienen lugar en la ciudad de Barcelona. A través de esta API, podrán consultar en tiempo real la programación de actividades, conciertos, exposiciones y otras experiencias disponibles en la ciudad.

Figura 3. Logo de la aplicación Aire Lliure

Además, hemos establecido una colaboración con **E-MoveBCN**, quienes nos proporcionarán una API especializada en cálculo de rutas y movilidad urbana. Gracias a esta integración, podremos ofrecer a los usuarios la posibilidad de planificar sus desplazamientos de manera eficiente, permitiéndoles visualizar el recorrido óptimo desde su ubicación hasta el evento de su interés. Esto no solo mejorará la experiencia del usuario, sino que además hará que este no salga del entorno de CultuCat.

La comunicación con ambos equipos hasta el momento ha sido mayoritariamente vía oral en el aula, aunque hemos usado correo electrónico para comunicarnos fuera del área de trabajo.

2. Metodología

2.1. Gestión del Proyecto

Este proyecto se desarrolla con prácticas Agile, más en concreto aplicamos la metodología Scrum.
Scrum se basa en la entrega incremental de productos mediante iteraciones cortas llamadas sprints, fomentando la colaboración, la adaptabilidad y la mejora continua. Hemos decidido usar esta metodología, ya que somos un equipo pequeño (6 personas) en el que no hay una jerarquía marcada. Scrum nos permite distribuir el trabajo de manera autoorganizada, y evaluar el progreso mediante entregas al final de cada sprint.

2.1.1. Reuniones del Equipo

Siguiendo la metodología Scrum, nuestro equipo realiza diversos tipos de reuniones para informar del estado actual del proyecto y para organizar el trabajo a futuro:

- Sprint Planning: Antes de comenzar un sprint, todo el equipo se reúne junto con el Product Owner para determinar qué historias de usuario se realizarán en el sprint. El Sprint Master será el encargado de dirigir esta reunión.

- Daily: En un Scrum tradicional esto se haría cada día del sprint, pero como nuestro equipo no trabaja diariamente en este proyecto, hemos decidido hacerlo dos veces por semana. Esta reunión destaca por ser muy corta, y en ella, cada miembro del equipo tiene que responder a las mismas tres preguntas: ¿qué has hecho desde la última reunión?, ¿qué harás hasta la próxima? Y ¿qué problemas te has encontrado? Para agilizar la reunión, todos los miembros del equipo tendrán que estar de pie mientras se realice.

- Sprint Retrospective: Esta reunión se hace al acabar un sprint, en ella todos los miembros del equipo hablan de que les ha gustado, que han aprendido y que ha faltado durante el último sprint. Esta reunión se hace para que el equipo pueda ver en qué puntos tiene que mejorar y en cuáles se ven más seguros. Para esta reunión usamos como herramienta la página de RetroTool[1].

- Sprint Review: Al igual que el Sprint Retrospective, el Sprint Review se hace al final del sprint y consiste en presentar el producto al Product Owner para que este ofrezca comentarios de cómo está avanzando la aplicación. También se analiza el cómo ha ido el sprint y se explican las desviaciones que han ocurrido. De la misma manera, los miembros del equipo harán preguntas al Product Owner para definir mejor el alcance del proyecto.

[1] RetroTool de CultuCat (Link en 4. Referencias)

2.1.2. Gestión del Proyecto con Taiga

Para poder aplicar la metodología Scrum usamos Taiga, que nos permite crear un backlog para nuestras historias de usuario, clasificarlas por épicas, asignarlas a miembros del grupo, etc.
Nuestro Taiga está organizado en 3 partes:

- Épicas: Se pueden ver todas las épicas, sus historias que las componen, su estado y su progreso.
- Scrum: Contiene el backlog de las historias de usuario y los 3 sprints. Dentro de cada sprint se puede ver las tareas de cada historia de usuario. Las historias de usuario se ordenan según su prioridad.
- Issues: En este apartado reportamos los bugs que encontramos en la aplicación.

2.1.3. Definition of Done

Al principio de cada *Sprint* se deciden qué historias de usuario se implementan en esta iteración (*Sprint Backlog)*, teniendo en cuenta su coste, el cual es decidido previamente por los miembros del grupo mediante un Planning Poker (con valores de la secuencia de Fibonacci). Para poder tener una mejor organización de las tareas hemos definido un ***Definition of Done*** (Definición de Listo), que define qué tiene que tener una tarea para que se la considere finalizada:

- Se ha realizado una prueba para verificar el cambio.
- Las pruebas funcionales, asociadas a la funcionalidad o cambio, pasan sin errores.
- El código ha sido revisado y aprobado por otro miembro del equipo.
- Los cambios han sido completamente integrados y subidos al repositorio principal.
- Se han cumplido todos los criterios de aceptación definidos en la historia de usuario.
- Se han contemplado los casos límite y la gestión de errores correspondiente.
- La documentación técnica del diseño se ha actualizado.

2.1.4. Definición de Historias de Usuario

Para aplicar la metodología Scrum, utilizamos Taiga como herramienta de gestión. Esta nos permite crear un backlog con nuestras historias de usuario, clasificarlas por épicas y asignarlas a los miembros del equipo.

En Taiga definimos las historias de usuario siguiendo el formato:

“Como [tipo de usuario], yo quiero [funcionalidad] para poder [objetivo]”.

Además, especificamos los criterios de aceptación correspondientes.

Hemos estructurado las historias de usuario agrupándolas por épicas, y dentro de cada historia hemos creado tareas más concretas para facilitar la implementación y el seguimiento. Las tareas han sido estimadas en horas según su complejidad y el esfuerzo esperado, basándonos en la experiencia del equipo y en la planificación durante las

reuniones de *Sprint Planning*. Cada historia de usuario también está valorada con *Story Points*.

Para mejorar la organización, hemos añadido etiquetas a cada tarea, indicando si pertenece a back-end, front-end o servicios.

2.1.5. Evolución de las Historias de Usuario

Una vez empezado el *Sprint* (cada sprint de unas 4 semanas), cada historia de usuario se divide en tareas, y cada tarea pertenece a una categoría dependiendo de su progreso. Al principio empiezan en la categoría *To Do* (Por Empezar). Cuando alguien comienza a trabajar en una funcionalidad, esta pasa a *In Progress* (En Proceso). Una vez que el código está implementado, se revisa para asegurarse de que no tenga errores, lo que se indica en la fase *Ready for Test* (Listo para probar). Finalmente, cuando se cumple con todos los requisitos, la tarea se da por completada y se mueve a *Done* (Finalizado).

Ya que nuestros sprints están limitados a 4 semanas, es posible que no se finalicen todas las tareas. Si se da el caso, las tareas pendientes se incluirán en el *backlog* del *sprint* siguiente y se continuarán desde donde se dejaron.

2.2. Gestión de Versiones

Para organizar el trabajo en equipo, y gestionar las diferentes versiones del código de la aplicación, utilizamos **Git** como sistema de control de versiones y **GitHub** como plataforma colaborativa basada en Git.

2.2.1 Git i GitFlow

Git es un sistema de control de versiones distribuido que permite registrar todos los cambios realizados en el código de forma ordenada y segura. Gracias a Git, varios de nosotros podemos trabajar en paralelo sin interferir entre nosotros, y se puede mantener un historial completo del proyecto, facilitando la colaboración y el control de calidad.

En nuestro proyecto seguimos la metodología GitFlow, una estrategia de ramificación (branching model) que permite organizar el trabajo de desarrollo de manera estructurada. Esta estrategia se representa visualmente en la siguiente imagen:

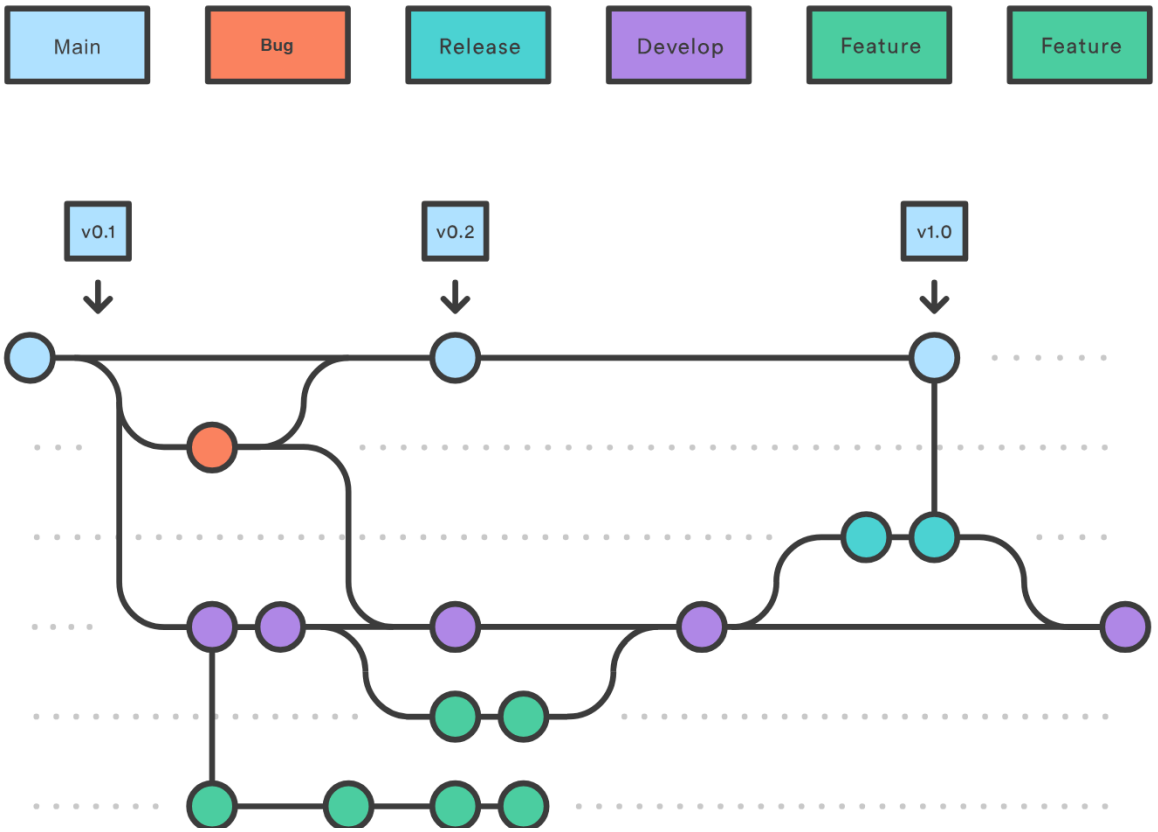

Figura 4. Ejemplo de ramas en Git

Las ramas principales y su uso son los siguientes:

- Rama *Main*: Rama preparada para hacer las *releases* (entregas) del código. Al hacer un *commit* en esta rama, el comentario debe de empezar con “/main:”.

- Rama *Development*: En esta rama el código debe de ser estable, aunque esté en entorno de desarrollo. Es la rama a la que se le aplican los cambios que aportan las implementaciones de funcionalidades. Al hacer un *commit* en esta rama el comentario debe de empezar con “/dev:”.

- Ramas *Feature:* Se creará una rama por cada funcionalidad que se quiera implementar. Todo el desarrollo de la funcionalidad, incluido el testing, se hace dentro de su propia rama. El nombre de cada rama depende de la funcionalidad que se implemente:“*feature_*nombre-historia” (ejemplo: *feature_login*), decidimos no especificar el número de la rama por si tenemos dos historias similares (ejemplo: *feature_login* y *feature_loginGoogle*).
  Aunque han de ser tan independientes como puedan, es posible que una rama *feature* surja de otra rama *feature*. Si se da el caso, para poder hacer *merge* de una rama, todas sus ramas hijas se tienen que haber unido a la principal. Al hacer un *commit* en esta rama el comentario debe de empezar con “/feat:”.

- Ramas *Bug*: Cuando aparezcan errores en las ramas *Develop* o *Main* se creará una rama *Bug* si es necesario para solucionarlo, y si se encontrase un *bug* en una rama *Feature* se arreglaría en la misma rama. El nombre de la rama será “*bug_#*issue” (ejemplo: *bug_#12-Error-al-enviar-mensaje*).

2.2.2 GitHub

GitHub es la plataforma que usamos para alojar nuestros repositorios Git en la nube, facilitando el trabajo colaborativo. A través de GitHub podemos compartir el código entre todos los miembros del equipo, crear ramas y gestionar Pull Requests.

En nuestro caso, hemos dividido el proyecto en dos repositorios independientes, uno para cada parte de la aplicación:

- **CultuCat_Back:** Contiene el back-end del proyecto (API y servidor).

- **CultuCat_Front:** Contiene el front-end, es decir, la aplicación móvil.

Aunque están separados, ambos repositorios siguen exactamente la misma estructura de ramas GitFlow, lo que garantiza coherencia y organización en el trabajo del equipo.

2.3. Comunicación dentro del Equipo

Fuera de las reuniones presenciales que se hacen en horario lectivo, nos comunicamos usando:

- WhatsApp: Hay un grupo donde están todos los miembros del equipo. Comunicación de tipo informal donde se resuelven dudas que puedan surgir, que sean fáciles de explicar, distribuir trabajo, programar reuniones y hacer recordatorios.
- Discord. Vía de comunicación primaria. Contamos con un servidor de Discord cuya principal función es albergar reuniones. El servidor cuenta con 3 canales de texto y 3 canales de texto, los canales están separados por General, Front-end y Back-end. Los canales de voz son el servicio que utilizamos para resolver dudas que sean más complejas de explicar.

Hemos decidido que nuestro canal de comunicación principal sea Discord, ya que ahí tenemos todo organizado por secciones y vinculado con nuestro proyecto de Taiga.

Contamos con un swagger para mostrar las llamadas API del servidor. Al empezar una historia de usuario, el equipo de Front pedirá al equipo de Back las funciones que necesita y, una vez implementadas, el equipo de Back indicará al equipo de Front cómo se ha de llamar a estas funciones y lo incluirá en el swagger.

2.4. Gestión de Calidad

2.4.1. Integración y Despliegue Continuo

Para garantizar la calidad del *software* desarrollado, hemos decidido usar herramientas como uso de integración y despliegue continuo (CI/CD). Al no haber problemas en el repositorio GitHub respecto a CI/CD, nos aseguramos de que el código entregado es de calidad. Todo código implementado por un miembro del equipo es revisado por otro miembro del equipo.

Hemos usado linters en nuestros proyectos para identificar fallos o problemas en el sistema. Podemos generar informes de calidad con estos linters siempre que queramos, pero los hemos implementado en nuestros pipelines de CI/CD para que se ejecuten automáticamente al hacer commit en la rama main o en la rama de desarrollo. En front usamos los linters propios de Flutter y en back usamos el linter **flake8**.

2.4.2. Codigo de Calidad

Aparte de esto, aprovechamos las herramientas que ofrecen los IDEs, que marcan errores de sintaxis, variables que no se usan, errores de ortografía, etc.
Al poder detectar errores durante la parte de implementación, se pueden solucionar de forma mucho más sencilla que si se detectaran en ejecución.

Para mantener un código limpio y fácil de leer, seguimos una guías de estilo al programar:

- Front-end: Effective Dart
- Back-end: PEP8

Estas guías nos indican cómo estructurar nuestro código y qué convención de nombres debemos de seguir.

2.5. Plan de Tests

Para garantizar la calidad del sistema, realizamos pruebas tanto unitarias como de integración utilizando las herramientas que proporcionan Django (para el back-end) y Flutter (para el front-end).

En el back-end, utilizamos UnitTest y Django REST Framework mediante APIClient para simular peticiones a nuestros endpoints. Nos centramos principalmente en testear las vistas (views.py), ya que es donde se encuentra la lógica de negocio. Las pruebas incluyen tanto casos exitosos como escenarios de error (por ejemplo, credenciales incorrectas o campos faltantes).

En el front-end, empleamos el paquete Flutter_test para realizar pruebas de widgets. Por ejemplo, se comprueba que los elementos de la interfaz cambian correctamente su estado al interactuar con ellos.

Las pruebas unitarias permiten comprobar que las clases funcionan en un entorno aislado, mientras que las pruebas de integración verifican que las interacciones entre clases se comportan de la manera esperada. Esto también ayuda a garantizar que los cambios recientes en el código no introduzcan errores en funcionalidades ya implementadas.

Calculamos la cobertura de los tests del back-end (se encuentra en la carpeta de coverage), no lo hacemos en el front-end ya que solo hacemos un test de UI de nuestra *homescreen* (la pantalla del mapa). No usamos herramientas como SonarQube, pero sí aprovechamos los linters y los avisos del IDE para detectar errores de sintaxis, advertencias y malas prácticas de desarrollo.

En cuanto a la integración continua (CI), utilizamos la sección Actions de nuestros repositorios en GitHub. Esta nos permite, en cada push, ejecutar las pruebas unitarias y de integración previamente programadas, tanto para el front-end como para el back-end.

Antes de realizar un merge a la rama dev, se debe comprobar que el código supere correctamente todos los tests implementados hasta ese momento.

2.6. Gestión de Configuraciones

Para llevar a cabo el despliegue de nuestras aplicaciones, utilizamos GitHub Actions, que permite crear scripts de automatización que se ejecutan cuando se realiza un push a la rama seleccionada.

Este script incluirá las órdenes necesarias para ejecutar las pruebas de integración continua, la revisión de código y el despliegue en el servidor de **Virtech.**

Habrá un único script de despliegue que, dependiendo de si el commit se realiza en la rama *dev* o en *main*, tomará las variables de entorno correspondientes y desplegará en la cuenta la configuración adecuada. Estas variables de entorno contienen información privada para autenticarse en ciertos servicios externos.

Para trabajar en local se necesita lo siguiente:

- IDE para Front (Android Studio) e IDE para Back (VS Code)
- Python 3
- Virtualenv
- Variables de entorno env
- Flutter SDK
- Emulador Android
- Código que se encuentra en los repositorios
- Variables secretas (ejemplo: google api key)

2.7. Interacción con otros Grupos

La interacción con los otros equipos sobre los servicios que vamos a ofrecer y recibir se gestiona principalmente a través de reuniones en clase. Si es necesario, la comunicación también puede llevarse a cabo por correo electrónico o WhatsApp.

Las negociaciones con cada grupo ya se han realizado. Hemos acordado con **E-MoveBCN** que nos proporcionarán una API para calcular rutas entre dos ubicaciones. Por otro lado, hemos llegado a un acuerdo con **Aire Lliure** para ofrecerles una API que les permita consultar los diferentes eventos culturales en Barcelona.

Para gestionar estas APIs, utilizaremos Swagger, una herramienta que nos permitirá visualizar, probar, diseñar y documentar las APIs de manera eficiente.

2.8. Gestión de Bugs

Para gestionar los *bugs* (errores) usamos la herramienta de *issues* de Taiga. Cuando un miembro del equipo descubra un *bug*, el miembro deberá abrir una *issue* con la etiqueta *bug* en el repositorio relacionado.No contamos con una plantilla para estos bugs pero en el *issue* del bug se tiene que indicar cuál es el error, cómo se puede reproducir y qué miembro deberá solucionarlo.

Por otra parte, también se tendrá que reportar el error encontrado en alguno de los canales de comunicación del equipo.

Si el bug se encuentra en una rama de feature, entonces se arreglara en la misma rama. Si se encuentra en alguna otra, y solo se necesita de un commit para solucionarlo, no se creará una nueva rama, y simplemente se hará un commit con el hot fix donde se indicará qué ha provocado el error y cómo se ha resuelto el bug. Finalmente, si se encuentra en una rama que no es feature y necesita más de un commit, se creará una rama Bug donde se solucionará. En la rama main no debería de haber ningún bug, pero si se diera el caso, se debe arreglar con un commit con el hotfix.

2.9. Tratamiento de Requisitos No Funcionales

Los requisitos no funcionales (NFRs) son fundamentales para garantizar la calidad global de la aplicación. En nuestro proyecto, los hemos definido siguiendo criterios que aseguran su claridad, viabilidad y verificabilidad.

Cada requisito no funcional se ha redactado en un formato **concreto y medible**, especificando los **criterios de satisfacción** de manera que permitan su comprobación durante las fases de desarrollo y pruebas. Nos hemos asegurado de que cumplan características de calidad como:

- **Claridad:** redactados de forma comprensible y sin ambigüedades.
- **Realismo:** ajustados a las capacidades del equipo y los recursos disponibles.
- **Medibilidad:** incluyen métricas o condiciones observables que permiten evaluar si se cumplen.
- **Verificabilidad:** es posible comprobar su cumplimiento mediante pruebas o revisiones objetivas.

Los requisitos no funcionales han sido documentados en un apartado específico del proyecto y categorizados según sus objetivos: usabilidad, rendimiento, escalabilidad, privacidad, disponibilidad, soporte, entre otros.

Además, se han asociado mecanismos de validación generales para garantizar su cumplimiento, com per exemple: proves d'usabilitat, eines de monitoratge de rendiment i disponibilitat, revisió de polítiques de privacitat, i testejos automatitzats o manuals segons el cas.

2.10. Asistentes de Código

Para mejorar la eficiencia de desarrollo usaremos asistentes de código como:

- **Gemini AI**: Integrado en la aplicación Android Studio, nos ayudará a automatizar la creación de código Flutter.
- **GitHub Copilot:** Integrado en editores de código como Visual Studio, nos permitirá arreglar errores en el código y facilitará la creación del mismo.

Como no podemos estar seguros de que el código generado por IA es correcto, este será revisado dos veces por dos miembros del equipo diferentes; el primero será el miembro que ha hecho la petición a la inteligencia artificial, que hará la revisión al recibir el código, y el segundo miembro revisará el código justo antes de cerrar la historia de usuario.

Con estos asistentes de código se pretende aumentar la velocidad a la que se implementan funcionalidades, para así poder tener un mayor alcance de proyecto con el mismo tiempo.

3. Descripción Técnica

3.1. Concepción General de la Arquitectura

Usaremos un sistema con un único servidor. Este servidor se encargará de alojar el front-end de la aplicación, base de datos (usaremos solo una) y el servidor back-end de la aplicación.
Hemos decidido usar un solo servidor en vez de un número superior porque pensamos que nuestra aplicación tendrá un flujo reducido de peticiones al servidor, por tanto, como no necesitamos más potencia, preferimos usar un único servidor para que así sea más fácil de implementar.

3.1.1. Arquitectura Física

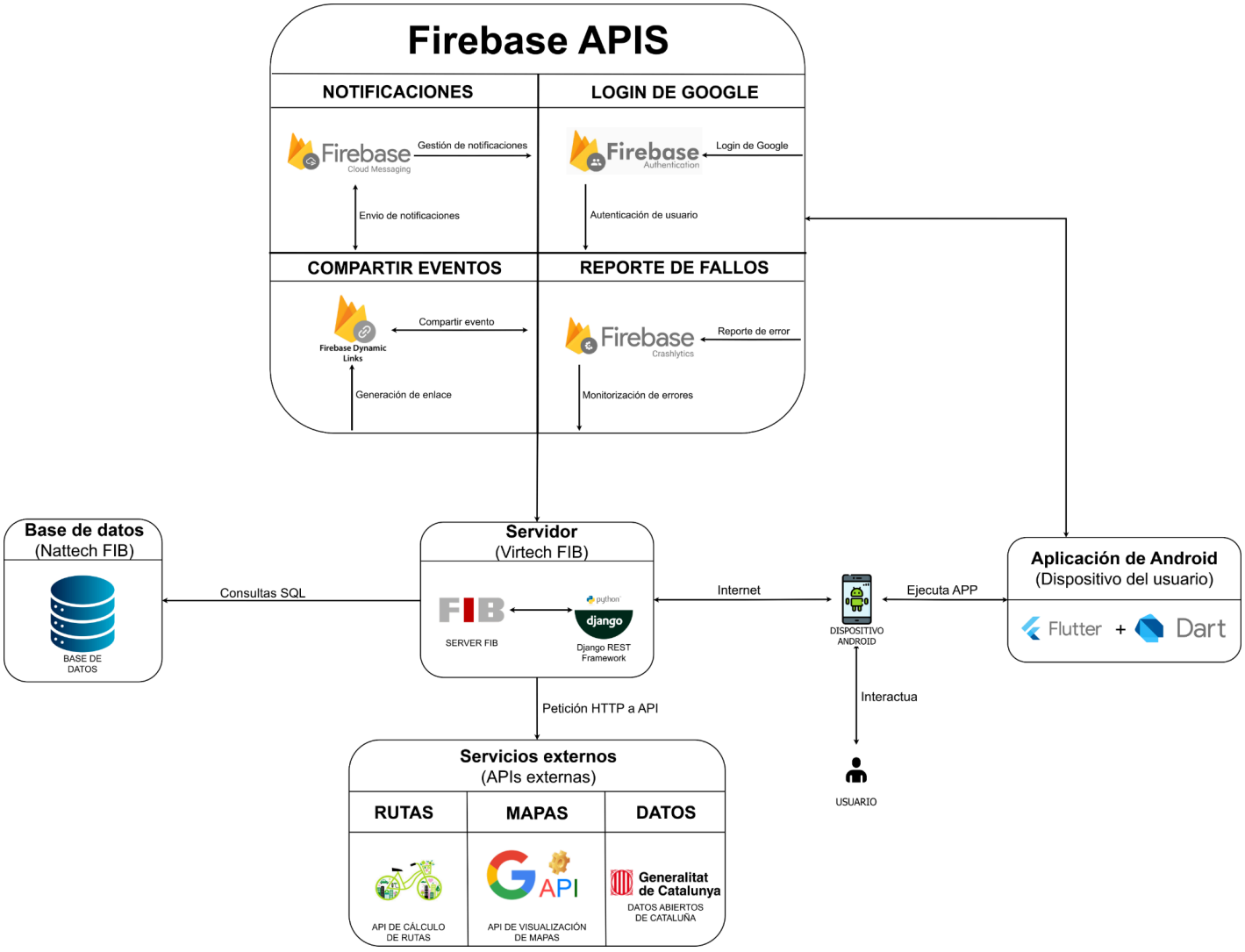

Figura 5. Diagrama de Arquitectura Física

3.1.2. Patrones Arquitectónicos

En nuestro proyecto hemos adoptado diferentes patrones arquitectónicos para cada uno de los principales componentes del sistema: utilizamos Flutter para el front-end, Django para el back-end y PostgreSQL como base de datos. Esta separación permite adaptar la arquitectura de cada componente a sus necesidades específicas, garantizando así una mayor eficiencia, escalabilidad y claridad en el diseño.

Arquitectura en el Front-end (Flutter)

Para la aplicación cliente, utilizamos una arquitectura basada en funcionalidades (**feature-based architecture**). La estructura del proyecto se organiza en torno a carpetas dentro del directorio screens, donde cada pantalla representa una funcionalidad independiente del sistema (por ejemplo, login, chats, eventos, perfil, etc.). Esto permite una mejor organización del código, escalabilidad y separación de responsabilidades.

Para la gestión de la creación de objetos dentro de la aplicación, empleamos el patrón **Factoría**. Este patrón permite delegar la creación de instancias a una clase especializada, lo que facilita la gestión y extensión de objetos complejos sin acoplar directamente el código cliente a las clases concretas. Así se mejora la flexibilidad y la mantenibilidad del sistema. Usamos este patrón para gestionar nuestras conexiones con Firebase.

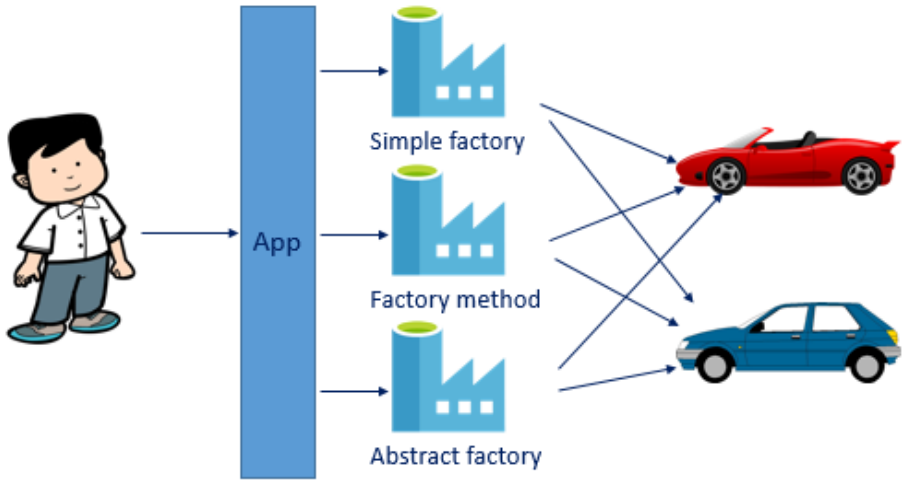

Figura 6. Diagrama simplificado del patrón Factoría

También utilizamos el patrón **Módulo** para organizar y encapsular funcionalidades relacionadas en unidades cohesivas. Las pantallas más complejas son separadas en diversos ficheros para los diversos widgets (módulos), luego se juntan para formar la pantalla completa. Cada módulo agrupa componentes, servicios y lógica específica, proporcionando una estructura clara y facilitando la reutilización y el aislamiento del código.

Además del patrón Factoría y Módulo, también hacemos uso del patrón **Singleton**, útil cuando se necesita garantizar que una clase tenga una única instancia a lo largo de toda la aplicación. El ejemplo más claro en nuestra aplicación es la clase que gestiona la ubicación de los servicios que usamos, al referirnos siempre a una misma instancia, cambiar la ubicación de los servicios de toda la aplicación es muy sencillo.

Arquitectura en el Back-end (Django)

En el back-end, seguimos la arquitectura modular típica de Django (**monolítica**), basada en aplicaciones independientes (apps). Cada funcionalidad principal del sistema (usuarios, eventos, chats, etc.) está encapsulada dentro de una app con sus propios modelos (models.py), vistas (views.py) y pruebas (tests.py). Una peculiaridad de este proyecto es que todas las apps comparten el archivo de rutas (urls.py), ya que al no ser un proyecto tan grande, no hay una cantidad de rutas que masifiquen el archivo, y permite a los desarrolladores ver de manera más visual las funcionalidades de back-end.
Este enfoque favorece la escalabilidad y el mantenimiento, ya que permite trabajar de forma aislada sobre diferentes partes del sistema.

La capa de almacenamiento está implementada con **PostgreSQL**, donde gestionamos y persistimos toda la información de la aplicación. Esta base de datos relacional nos proporciona un rendimiento óptimo, soporte para operaciones complejas y una alta fiabilidad en el manejo de datos estructurados.

El flujo de comunicación se basa en peticiones HTTP entre el cliente (Flutter) y el servidor (Django). Cuando un usuario interactúa con la aplicación, esta envía una solicitud al back-end, el cual procesa la lógica correspondiente, consulta o actualiza los datos en la base de datos, y devuelve una respuesta que luego se muestra en la interfaz del usuario.

Este enfoque modular y desacoplado ofrece múltiples ventajas: facilita el desarrollo y mantenimiento de cada parte del sistema, permite reutilizar componentes, asegura la integridad de los datos mediante una base de datos robusta, y abre la posibilidad de escalar la aplicación distribuyendo los componentes en distintos entornos o servicios en la nube según se necesite.

3.2. Capa de Dominio

3.2.1. Diseño de Back-end

En primer lugar, respecto a los patrones del back-end, hemos usado un patrón propio de Django: Perfil de usuario extendido.

Este patrón consiste en tener la lógica de manejo de los usuarios repartida.
Por una parte, tenemos las tablas creadas por Django en nuestra base de datos, que son las que empiezan por “auth_”. Por ejemplo: auth_user, auth_permission, auth_group… Estas tablas sirven para usar las funciones proporcionadas por Django para manejar el registro de usuario y autenticación del usuario al hacer login. Por otro lado, tenemos la tabla users, la cual contiene información relacionada con nuestra aplicación y que está conectada con el resto de tablas de la base de datos. Lo que hacemos con este patrón es crear un atributo OneToOne entre users y auth_users. De esta manera, mantenemos la lógica de nuestra aplicación separada de la lógica de registro y login, manteniendo relacionadas ambas tablas de tal forma que podamos acceder desde cada una a la otra.

3.2.2. Estructura del código del Back-end

La estructura de nuestro código consiste básicamente en que creamos una aplicación de Django (carpeta con models.py, views.py…) para cada tabla de nuestra base de datos. En el archivo models.py especificamos como es la tabla para enlazar Django con la base de datos, y en views.py implementamos las funciones de respuesta a las peticiones (endpoints). De esta forma, cada tabla de nuestra base de datos tiene su propia carpeta, aislando las funcionalidades de cada tabla para que sean independientes y estén bien localizadas.

3.2.3. Diseño de Front-end

En cuanto a la estructura del front-end, hemos seguido la organización básica que proporciona Flutter al crear un nuevo proyecto, adaptándola a nuestras necesidades para mantener un código claro, modular y escalable. Todo el código fuente generado por nosotros se encuentra dentro de la carpeta principal **/lib**.

Dentro de **/lib**, hemos creado varias subcarpetas que permiten estructurar correctamente los distintos componentes de la aplicación:

- **/screens**: Contiene las diferentes pantallas de la aplicación. Cada pantalla se encuentra dentro de su propia carpeta, como por ejemplo **/login**, **/register**, **/home**, etc. En cada una de estas carpetas se encuentra un archivo .dart con el código de la interfaz y la lógica específica de esa pantalla.
    - **/widgets**: Dentro de algunas carpetas, como por ejemplo **/profile**, tenemos la carpeta **/widget** para componentes visuales reutilizables como botones personalizados, campos de texto, cabeceras, etc.

- **/utils**: Carpeta que contiene utilidades generales que pueden ser reutilizadas en diferentes partes de la aplicación. Por ejemplo, el archivo user_preferences.dart nos permite guardar localmente información del usuario como su ID, token de autenticación y otros datos relevantes mediante el uso de SharedPreferences.

En resumen, la estructura del front-end está pensada para ser simple pero sólida, y nos ha permitido trabajar de forma organizada, colaborativa y eficiente.

3.3. Diagrama de la Base de Datos (UML)

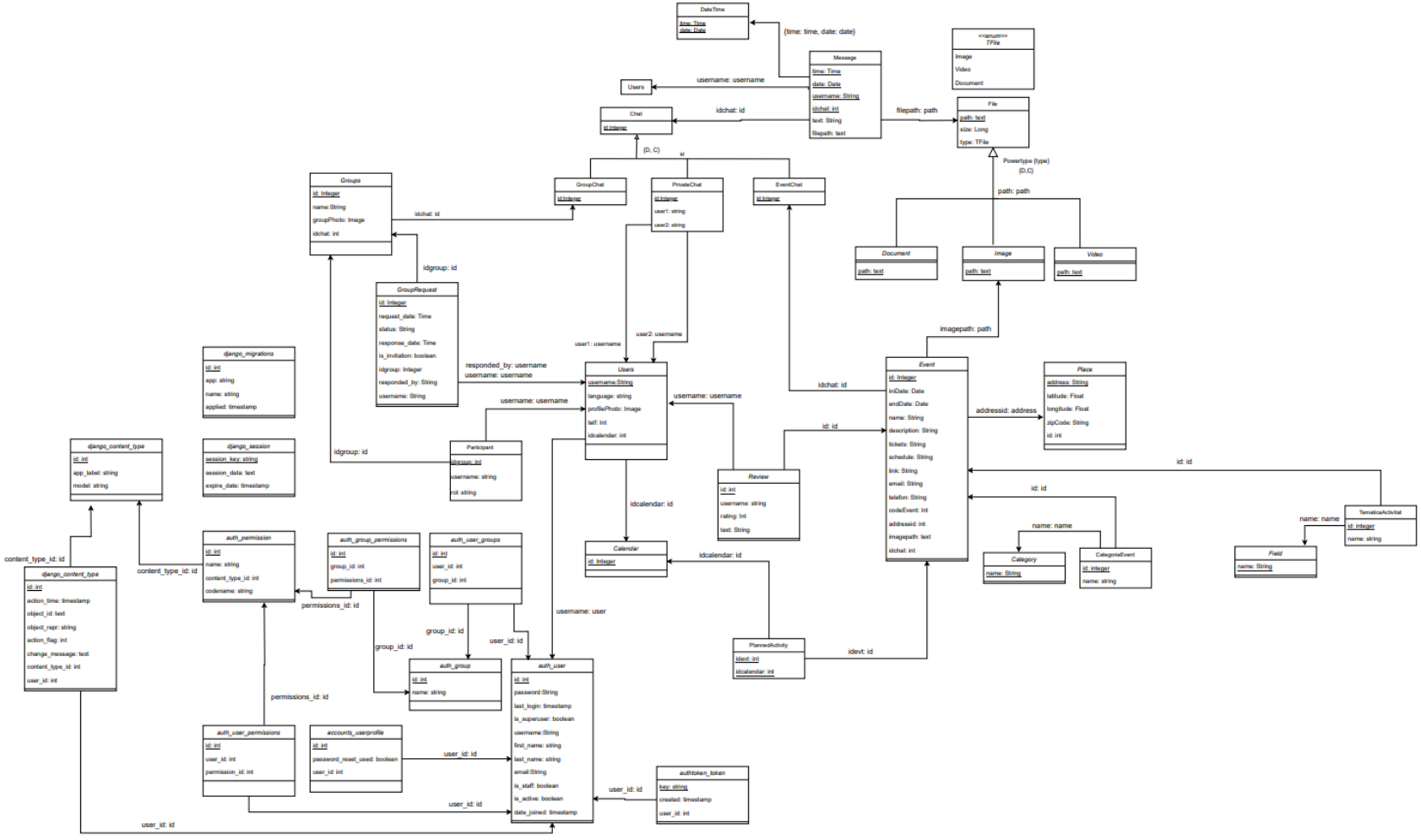

Figura 7. Diagrama de la Base de Datos de CultuCat

Para ver el diagrama en mayor detalle se puede ver mejor en Diagrama de la BD CultuCat[2].

3.3.1. Justificación del diagrama de la base de datos

En primer lugar, en nuestra base de datos encontramos algunas tablas que empiezan por la palabra “django”. Algunos ejemplos son: django_migrations, django_content_type, django_session… Estas son tablas propias de django, el framework que utilizamos para implementar el back-end de nuestra aplicación, por lo que estamos obligados a incorporarlas al esquema de nuestra base de datos si queremos que django reconozca nuestra estructura de base de datos.

A continuación, encontramos el “puente” entre django y el dominio de nuestra aplicación. Este “puente” se encarga de manejar la lógica de usuarios de la app. Encontramos tablas que empiezan por el prefijo “auth”, como por ejemplo: auth_permission, auth_group_permissions, auth_user_groups… Estas tablas también son propias de django y son necesarias en nuestro esquema para poder usar las funciones propias de django que gestionan funcionalidades como el login y la autenticación. Sin estas tablas no se podrían usar las funciones de login de django ni su módulo *django.contrib.auth* que permite gestionar la autenticación de usuarios. Por otra parte, la parte de la lógica de usuarios correspondiente a nuestra app, la encontramos en la tabla Users. Esta tabla maneja la información de los usuarios más relacionada con el dominio de nuestra aplicación, como su calendario o su foto de perfil. La razón de separar la lógica de los usuarios en una tabla nuestra y varias de django es poder tener información personalizada de los usuarios adaptada a nuestro dominio, al mismo tiempo que aprovechamos las funciones proporcionadas por django para gestionar el login y la autenticación de usuarios de forma fácil, eficiente y correcta. Además, la modificación de las tablas de django suponía la generación de posibles problemas de compatibilidad de cara al futuro, por lo que hemos preferido escoger la solución actual.

Posteriormente, ya encontramos todas las tablas encargadas de formar el dominio de nuestra aplicación. Encontramos que cada usuario tiene su propio calendario, el cual contiene distintos eventos a los que se ha apuntado el usuario (tabla Planned activity). Estos eventos tienen información propia (tabla Event) y se realizan en un sitio concreto (tabla Place). Hemos separado el lugar en tablas distintas, ya que un lugar puede albergar distintos eventos, y de esta forma se evita guardar información repetida en la base de datos. Finalmente, siguiendo el mismo criterio, hemos optado por separar las categorías de los eventos y sus temas en distintas tablas (tablas Category y Field, respectivamente). Por tanto, cada evento tiene unas categorías y temas asociados (tablas CategoriaEvent y TematicaActivitat, respectivamente). Además, cada evento tiene sus reseñas (tabla Review), y cada usuario puede publicar una reseña en cada evento.

Respecto a la parte más social, existen grupos (tabla Groups). Un usuario puede pertenecer a varios grupos, en el cual tiene un determinado rol (tabla Participant). Estos grupos tienen su propio chat (tabla GroupChat). Los usuarios pueden recibir invitaciones a grupos o solicitar unirse a ellos (tabla GroupRequest).
Los chats forman una jerarquía, ya que no solo existen chats grupales, sino que también existen chats privados (tabla PrivateChat) y de eventos (tabla EventChat). Estos chats

[2] Diagrama de la BD CultuCat (Link en 4. Referencias)

contienen distintos mensajes (tabla Message), los cuales son enviados por un usuario concreto, en un día y hora concretos y pueden tener o no un archivo asociado. Este archivo (tabla Field) puede ser de distintos tipos, lo que da lugar a otra jerarquía. Puede ser un documento (tabla Document), una imagen (tabla Image) o un vídeo (tabla Video).

Así pues, la lógica de nuestra aplicación está formada por distintas tablas, las cuales se agrupan en tres grandes grupos: usuarios, eventos y chats. La razón de hacer el diseño así ha sido la de poder tener todo nuestro dominio relacionado, pero al mismo tiempo con el reparto de funciones más exacto posible, de manera que si en algún momento surgen problemas o se quiere ampliar funcionalidades, sea muy fácil saber qué parte de la base de datos hay que modificar o dónde reside el problema. Además, esto también ayuda a facilitar la implementación de código porque hace todo mucho más fácil e intuitivo.

3.4. Instrumentación y Tecnologías

3.4.1 Tecnologías de Back-end

3.4.1.1 Infraestructura del Back-end

Usaremos un servidor virtual que nos ofrece la FIB, llamado **Virtech**, con sistema operativo Ubuntu. Hemos decidido usar este servidor, ya que es gratuito y ha sido recomendado por otros usuarios que cursan esta asignatura. Consideramos que es un servidor que ofrece un servicio al nivel de los planes gratuitos de otras compañías como Amazon Web Services, Google Cloud Services o Microsoft Azure, y tiene el aliciente de que es más cercano a nosotros.

3.4.1.2 Tecnologías del Back-end

Para el back-end usaremos el *framework* **Django** (lenguaje Python). Teníamos otras opciones como Node.js, pero finalmente nos hemos decantado por Django por todas las ventajas que ofrece este framework de Python. En primer lugar, tiene una gran comunidad y documentación, lo que facilita su aprendizaje. En esto coincide con Node.js, sin embargo, en cuanto a velocidad de desarrollo, Django es superior, ya que cuenta con librerías como Django Rest Framework, que nos permiten desarrollar APIs fácilmente. En este punto Django es superior a Node.js, pese a que este cuente con frameworks como Express.js. Además, cabe destacar que Django cuenta con un ORM altamente compatible con bases de datos PostgreSQL (entre muchas otras), lo que nos permitirá trabajar con nuestra base de datos de forma mucho más cómoda. Otro factor importante es la robustez y seguridad que ofrece Django. Así pues, todos estos factores han acabado decantando la balanza a favor de este popular framework.

3.4.2 Tecnologías de Front-end

Para el desarrollo del front-end usaremos el *framework* **Flutter**, que es un *framework* desarrollado por Google que utiliza el lenguaje Dart. La principal razón de esta elección ha sido su capacidad de poder generar aplicaciones nativas tanto para Android como para iOS a partir de una única base de código. Aunque nosotros solo compilamos la aplicación para Android, creemos que es una buena opción, ya que nos da versatilidad y escalabilidad por si queremos ampliarla a iOS en un futuro.

Híbrido (Flutter, React Native, etc.)	**Nativo (Swift, Kotlin, etc.)**
Código único para Android e iOS	Código separado para Android e iOS
Desarrollo más rápido	Desarrollo más lento
Mantenimiento más sencillo	Mantenimiento más complejo
Rendimiento muy bueno (pero no máximo)	Rendimiento óptimo
Acceso limitado a APIs del dispositivo	Acceso completo a APIs del dispositivo
Tamaño de app más grande	Tamaño de app más optimizado
Curva de aprendizaje más fácil	Curva de aprendizaje más pronunciada
Posibilidad de escalar a Web	No aplicable para Web

Además, Flutter permite compilar en caliente (hot reload), lo que permite ver los cambios en la interfaz de forma inmediata sin necesidad de reiniciar la aplicación. Esta funcionalidad acelera considerablemente el proceso de desarrollo y facilita mucho el trabajo. Es por eso que Flutter nos ha parecido la opción más cómoda y adecuada para el front-end por todas las características mencionadas.

3.4.3 Tecnologías Base de Datos

Para la base de datos usaremos una de tipo relacional, concretamente PostgreSQL, por su compatibilidad con Django y la experiencia previa del equipo. Esta tecnología destaca frente a otras opciones por su soporte avanzado de transacciones, tipos de datos complejos (como JSONB), alta conformidad con SQL y un motor robusto y extensible. En comparación con bases no relacionales como MongoDB, PostgreSQL ofrece mayor consistencia y control de integridad, ideal para la estructura relacional de nuestra aplicación.

Somos conscientes de que, si se quisiera almacenar todos los datos tal como se obtienen desde la API, MongoDB podría ser una opción más adecuada, sin embargo, como nuestra intención es seleccionar únicamente cierta información antes de guardarla, consideramos que la opción que hemos escogido es la más apropiada para nuestro caso.

Finalmente, hemos decidido usar Firebase en nuestra aplicación para facilitar funcionalidades que nos serían complejas sin esta plataforma, como el "*Login con Google*" y "*Notificaciones*", ya que Firebase contiene funcionalidades específicas que simplifican su desarrollo.

3.5. APIs

3.5.1. APIs Desarrolladas

Proporcionamos a **Aire Lliure** un servicio exclusivo que les permitirá acceder a una API con información actualizada sobre los diferentes eventos culturales y de entretenimiento que tienen lugar en la ciudad de Barcelona. A través de esta API, podrán consultar en tiempo real la programación de actividades, conciertos, exposiciones y otras experiencias disponibles en la ciudad.

Este servicio se basa en proporcionar un par de endpoints los cuales al acceder a ellos estos devuelven un archivo JSON con toda la información que **Aire Lliure** necesita de determinados eventos como su nombre, la fecha o duración, su descripción, su categoría, su temática, etc.

El primer endpoint se llama "**Obtener todos los eventos**", este responde con toda la información de todos los eventos disponibles.

El segundo endpoint se llama "**Obtener eventos según un filtro**". Este también responde con la información de todos los eventos, pero que cumplan con determinadas condiciones. Al introducir unas coordenadas (latitud y longitud) y una palabra o frase, el endpoint devuelve un JSON ordenado por cercanía a las coordenadas introducidas y por similitud a la palabra o frase introducida.

La documentación de estas APIs están en la carpeta "API Obtener-Eventos CultuCat" (Sprint 3).

3.5.2. APIs que Usamos

Para ofrecer una experiencia más completa y eficiente a nuestros usuarios, integramos diversas APIs externas que nos aportan funcionalidades clave en nuestra aplicación.

API de E-MoveBCN

Hemos establecido una colaboración con **E-MoveBCN**, que nos proporciona una API especializada en cálculo de rutas y movilidad urbana. Esta API permite a los usuarios planificar sus desplazamientos de manera eficiente, visualizando el recorrido óptimo desde su ubicación hasta el evento deseado, lo que mejora la experiencia del usuario, fomenta el uso de transporte sostenible y optimiza los tiempos de traslado.

API de Google Maps

Utilizamos la API de Google Maps que nos ofrece un mapa interactivo.

En este mapa mostramos la ubicación de los eventos de Cataluña, a su vez nos permite obtener más información de estos eventos. También, gracias al servicio que nos ofrece **E-MoveBCN**, calculamos las posibles rutas hacia estas con diferentes tipos de transporte y las mostramos en dicho mapa.

API de Dades Obertes de Catalunya

Utilizamos la API de la Generalitat de Catalunya, que ofrece datos actualizados sobre la agenda cultural por localizaciones.

El dataset se actualiza prácticamente a diario, pero para gestionar esta información de manera eficiente, hemos desarrollado un script propio que realiza varias peticiones a la API y almacena en nuestra base de datos únicamente los eventos a partir del 1 de enero de 2025. Este script se ejecuta cada dos o tres días, lo que permite actualizar nuestro dataset interno unas tres veces por semana. De este modo, aunque la Generalitat actualice los datos diariamente, nuestra aplicación trabaja con la base de datos local y ofrece a los usuarios la información actualizada con esta frecuencia.

API de Firebase

Utilizamos Firebase para diversas funcionalidades clave en la aplicación:

- **Authentication:** Autenticación segura y sencilla para los usuarios mediante sus cuentas de Google.

- **Cloud Messaging:** Comunicación directa y en tiempo real con los usuarios mediante notificaciones.

- **Dynamic Links:** Facilita la compartición de eventos con enlaces inteligentes que redirigen a la app o a la web según corresponda.

- **Crashlytics:** Monitorización y reporte automático de fallos y errores para mejorar la estabilidad de la aplicación.

3.6. Herramientas de Desarrollo y Entorno de Trabajo

3.6.1. Herramientas de desarrollo y entorno de trabajo

Para el desarrollo colaborativo del proyecto, hemos utilizado diferentes herramientas que nos permiten mantener una buena organización y eficiencia:

- Visual Studio Code como editor principal para el desarrollo del back-end en Python.

- Android Studio como entorno de desarrollo para la aplicación Flutter, que incluye herramientas para la simulación y prueba en dispositivos virtuales.

- Git como sistema de control de versiones, y GitHub como plataforma centralizada para alojar el repositorio, revisar código y realizar integraciones continuas mediante GitHub Actions.

Para poder trabajar correctamente en local, cada miembro del equipo necesita configurar su entorno con lo siguiente:

- Instalación de Python 3, uso de Virtualenv para gestionar dependencias back-end, e instalación de paquetes mediante requirements.txt.

- Instalación del Flutter SDK, configuración del entorno y de un emulador Android.

- Configuración de variables de entorno (.env) tanto para el front-end como para el back-end, incluyendo claves de API y credenciales de base de datos.

- Acceso a una base de datos PostgreSQL, ya sea local o remota, con las credenciales adecuadas definidas en el entorno.

3.6.2. Uso de frameworks, integración y despliegue

El back-end está desarrollado con Django, un framework robusto para aplicaciones web, mientras que el front-end se ha construido con Flutter, permitiendo un desarrollo multiplataforma. En el front-end, la gestión de dependencias y compilación se ha realizado con Gradle. No hemos implementado herramientas de Integración Continua (CI/CD) ni utilizado Docker, ya que el flujo de trabajo se ha centrado en un desarrollo más tradicional, con despliegues y pruebas manuales.

Actualmente, disponemos de un sistema de Integración y Despliegue Continuo (CI/CD) implementado mediante GitHub Actions. Este sistema ejecuta automáticamente los tests tanto de front-end como de back-end en cada push al repositorio, y válida que el código cumpla con los estándares definidos antes de ser integrado a las ramas principales. También hemos configurado scripts para facilitar el despliegue en el servidor **Virtech**, con el uso de variables de entorno seguras y configuración automática según el entorno (desarrollo o producción).

4. Referencias

[1] Planning Poker Online, “Planning Poker – Estimation Tool for Agile Teams,” *PlanningPokerOnline.com*. [En línia]. Disponible: https://planningpokeronline.com/. [Data de consulta: 28-març-2025].

[2] RetroTool, “Collaborative retrospective tool for agile teams,” *RetroTool.io*. [En línia]. Disponible: https://retrotool.io/. [Data de consulta: 28-març-2025].

[3] GitHub, “CultuCat_Back,” *GitHub - pes2425q2-m-gei-upc*. [En línia]. Disponible: https://github.com/pes2425q2-m-gei-upc/CultuCat_Back. [Data de consulta: 28-març-2025].

[4] GitHub, “CultuCat_Front,” *GitHub - pes2425q2-m-gei-upc*. [En línia]. Disponible: https://github.com/pes2425q2-m-gei-upc/CultuCat_Front. [Data de consulta: 28-març-2025].

[5] Atlassian, “Gitflow Workflow,” *Atlassian*. [En línia]. Disponible: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow. [Data de consulta: 28-març-2025].

[6] Python Software Foundation, “PEP 8 – Style Guide for Python Code,” *Python.org*. [En línia]. Disponible: https://peps.python.org/pep-0008/. [Data de consulta: 28-març-2025].

[7] Dart Team, “Effective Dart: Style,” *dart.dev*. [En línia]. Disponible: https://dart.dev/guides/language/effective-dart/style. [Data de consulta: 28-març-2025].

[8] Patterns.dev, “Design Patterns, JavaScript & TypeScript,” Patterns.dev. [En línia]. Disponible: https://www.patterns.dev/. [Data de consulta: 28-maig-2025].

Referencias para complementar la memoria

1. RetroTool del segundo sprint

2. Diagrama de la BD CultuCat

Anexo

En el drive se encuentra un documento con todas las respuestas de la encuesta en el documento *resultados_encuesta_cultucat.pdf*. La siguiente imagen muestra el porcentaje de los usuarios que opinan que la aplicación es intuitiva de usar.

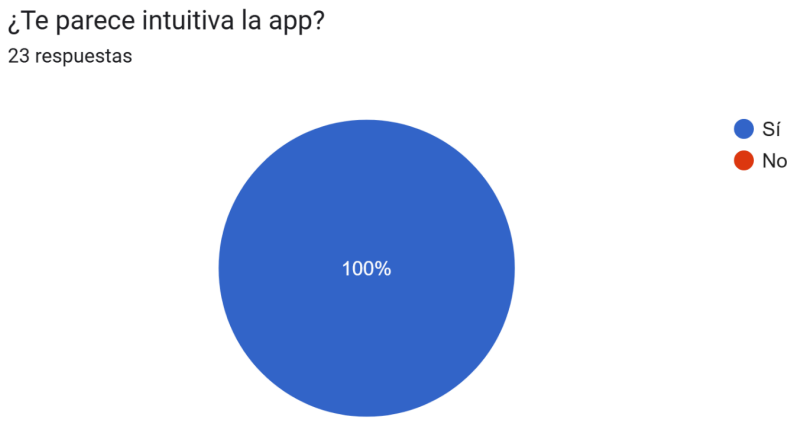

Figura 8. Piechart de la encuesta

Para comprobar la NFR nº 6 hemos decidido mirar cuánto tiempo lleva corriendo nuestro servidor. Para ello usamos el comando *uptime* de ubuntu.

```
alumne@wolverine:~/CultuCat_Back_Clonat/CultuCat_Back/CultuCat_Django$ uptime
 21:02:43 up 86 days,  9:14,  1 user,  load average: 0.00, 0.02, 0.00
alumne@wolverine:~/CultuCat_Back_Clonat/CultuCat_Back/CultuCat_Django$ _
```

Figura 9. Resultado uptime

Podemos ver que el servidor ha estado corriendo por 86 días consecutivos y como nuestra aplicación está testeada para comprobar que no se colgará nunca, podemos asegurar que el servidor estará disponible la gran parte del tiempo.

Para comprobar la NFR nº 7 hemos decidido ejecutar la peticion que tarda más en ejecutarse (GET url/events) 10 veces y hacer su media.

Ejecuciones en segundos: 2.12, 2.22, 2.12, 2.10, 2.13, 2.12, 2.13, 2.12, 2.12, 2.20

Media = 2.1380