



**UNIVERSIDAD
DE GRANADA**

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Bot de Telegram para la gestión de bandas

Autor

Daniel Haro Contreras

Directora

Rosana Montes Soldado



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN**

—
Granada, noviembre de 2022



Bot de Telegram para la gestión de bandas

Autor

Daniel Haro Contreras

Directora

Rosana Montes Soldado

Bot de Telegram para la gestión de bandas: Subtítulo del proyecto

Daniel Haro Contreras

Palabras clave: Telegram, bot, gestión, música, banda

Resumen

Poner aquí el resumen.

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3,

Abstract

Write here the abstract in English.

Yo, Daniel Haro Contreras, alumno de la titulación TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada, con DNI 76656133P, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Daniel Haro Contreras

Granada a X de mes de 201 .

D.^a **Rosana Montes Soldado**, profesora del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado *Bot de Telegram para la gestión de bandas*, ha sido realizado bajo su supervisión por **Daniel Haro Contreras**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de noviembre de 2022.

La directora:

Rosana Montes Soldado

Agradecimientos

Poner aquí agradecimientos...

Índice general

1. Introducción	21
1.1. Motivación y justificación del proyecto	21
1.2. Objetivos	23
2. Estado del arte	25
2.1. Dominio del problema a resolver	25
2.2. Aplicaciones similares	27
2.2.1. Glissandoo	27
2.2.2. Agregación de servicios no especializados	29
3. Mordente	31
3.1. Desarrollo de Mordente	31
3.1.1. Diseño Centrado en el Usuario	31
3.1.2. Design Thinking	32
3.2. Personas ficticias	32
3.3. Malla receptora de información	33
3.3.1. Aspectos interesantes o relevantes de la idea	34
3.3.2. Críticas constructivas o posibilidades de mejora	34
3.3.3. Preguntas nuevas a partir de la experiencia del usuario	35
3.3.4. Ideas que surgen	36
3.4. Descripción de la propuesta	36
3.4.1. Propuesta del producto	36
3.4.2. Elección de metodología de desarrollo	38
3.4.3. Planificación	38
3.4.4. Presupuesto	40
4. Diseño de la interfaz de usuario	43
4.0.1. Bocetos de interfaz	44
5. Diseño técnico	47
5.1. Listado de historias de usuario (Backlog)	47
5.2. Historias de usuario	49
5.3. Metodologías y tecnologías de base que podrían usarse	49
5.3.1. Lenguaje de programación	49

5.3.2. Framework para el desarrollo del bot	51
5.3.3. ORM: <i>Object-relational mapper</i>	52
5.3.4. Base de datos	53
5.3.5. Servidor virtual	54
5.3.6. Almacenamiento de archivos	54
5.3.7. Documentación	55
5.4. Modelo de la base de datos	55
5.5. Arquitectura de Mordente	56
5.5.1. Arquitectura entre puntos	56
5.5.2. Arquitectura entre servicios	56
5.6. Seguridad	58
6. Implementación	61
6.1. Exploración de funcionalidades	61
6.1.1. Infraestructura de contenedores	61
6.1.2. Conexión a base de datos	62
6.1.3. Manejo de mensajes de Telegram	63
6.2. Internacionalización	64
6.3. Separación MVC: Plantillas	65
6.4. Unirse a un grupo	66
6.5. Crear, mostrar y eliminar agrupaciones	67
6.6. Optimización de la autenticación en cada mensaje	67
6.7. Crear, mostrar y eliminar eventos	68
6.7.1. Selector de fecha	68
6.7.2. Guardar sesión en la base de datos	68
6.8. Configuración del <i>linter</i>	69
6.9. Limitación de errores para el bot	70
6.10. Mejora del flujo de trabajo para depurar	71
6.11. Respuestas de asistencia prevista	71
6.11.1. Pedir justificación	71
6.11.2. Notificar administradores	71
6.12. Asignación de miembros a eventos	72
6.13. Recordatorio de eventos diarios	72
6.14. Adición y eliminación de administradores	72
6.15. Despliegue de producción	72
6.15.1. Modificaciones previas en el código	73
6.15.2. Creación de servidor virtual	74
6.15.3. Solucionando el uso anormal de la CPU	75
6.16. Obras	76
6.16.1. Almacenando partituras: creación del <i>bucket S3</i>	76
6.16.2. Intermediando entre el chat y Digital Ocean Spaces	77
6.16.3. Lógica común con agrupaciones y eventos	78
6.16.4. Parametrizando los valores del <i>bucket</i>	78
6.17. Copias de seguridad de la base de datos	78

6.18. Despliegue Continuo (CD): Automatizando el despliegue del bot a producción	80
6.19. Configurando la detección automática de vulnerabilidades	81
6.20. La función de editar	82
6.21. Página web: <code>mordente.es</code>	83
6.21.1. Creación del proyecto	83
6.21.2. Creación de contenido	83
6.21.3. Alojar en servidor	84
6.21.4. Redirecciones	85
6.21.5. Asignación de dominio	85
6.21.6. Probando otros proveedores	86
6.22. Creación de dirección de email	87
6.23. Registro remoto de errores	87
7. Pruebas	91
7.1. Diseño de las pruebas	91
7.1.1. Participantes	91
7.1.2. Tareas a realizar	91
7.1.3. Preguntas a realizar	92
7.2. Realización de las pruebas	92
7.3. Informe final de las pruebas	92
7.4. Demostración en vídeo	93
8. Conclusiones y trabajos futuros	95
8.1. Conclusiones	95
8.2. Trabajos futuros	95
8.2.1. Webapp	95
8.2.2. Webapp integrada en el bot	95
Bibliografía	99

Índice de figuras

2.1. Capturas de la aplicación <i>Glissando</i>	28
3.1. Persona: Eugenio Soto	33
3.2. Persona: Clara Villena	34
3.3. Persona: Eugenio Soto	35
3.4. Persona: Inma Medina	36
3.5. Diagrama de Gantt del proyecto	42
4.1. Diseño de la interfaz de lista	45
4.2. Diseño de la interfaz del detalle de un elemento	46
4.3. Diseño de la interfaz de creación de un elemento	46
5.1. Modelo de la base de datos	59
5.2. Arquitectura entre puntos	60
5.3. Arquitectura entre servicios	60
6.1. Separación Modelo-Vista-Controlador	66
6.2. Gráfico de carga de la CPU	76
6.3. Archivos en el <i>bucket S3</i> antes de superar el máximo de días configurado	79
6.4. Archivos en el <i>bucket S3</i> tras superar el máximo de días configurado	80
6.5. Marca de verificación del Despliegue Continuo	81
6.6. Creando el proyecto de Sentry	88
6.7. Código generado por Sentry para incluir en un proyecto	88
6.8. Esperando al primer log de Sentry	88
6.9. Registro de un error en Sentry	89

Índice de tablas

3.1. Planificación temporal	40
3.2. Presupuesto	41
5.1. Comparación de lenguajes de programación	51
5.2. Comparación de frameworks para creación de bots de Telegram	52
6.1. Variables de entorno necesarias para configurar el <i>bucket S3</i>	78
6.2. Variables de entorno necesarias para configurar las copias de seguridad.	79
6.3. Redirecciones en mordente.es	85
7.1. Formulario para la prueba de usabilidad	93

Capítulo 1

Introducción

Este proyecto intenta dar respuesta a las agrupaciones musicales que demandan una plataforma gratuita y libre para la mejora de su gestión.

1.1. Motivación y justificación del proyecto

En 2018 se contabilizaron unas 6197 agrupaciones musicales en España según el INE [1], número que se mantiene estable a lo largo de los años. Cada agrupación musical, definida como “dos o más personas que, a través de la voz o de instrumentos musicales, interpretan obras musicales pertenecientes a diferentes géneros y estilos” según la RAE, se compone de distintos miembros, formando una sociedad. Como en todas las sociedades, se necesita cierto orden para poder alcanzar los objetivos, por lo que algunos de los miembros se tienen que encargar de su gestión: el director musical, el director artístico, el presidente de la asociación, el secretario... Siendo la gestión distinta en cada agrupación.

Los miembros de las agrupaciones musicales realizan ensayos periódicamente con el doble objetivo de mejorar su técnica de interpretación y preparar la celebración de eventos, tales como conciertos, certámenes, concursos, pasacalles... En los ensayos y eventos se interpreta una o varias obras musicales.

El director musical de una agrupación musical se encarga de coordinar la interpretación de las obras musicales, así como planificar y preparar los ensayos y eventos. En ocasiones se sustituye o se complementa con el rol de director artístico, visibilizando el hecho de que los eventos a menudo no son puramente musicales, sino que se añaden teatros o espectáculos visuales como bailes o musicales.

Los administradores de la agrupación (habitualmente, junta directiva) se encargan de las labores de administración no musicales, ayudando también al director. Habitualmente entre estos encontramos un presidente, un secretario y un tesorero.

Para interpretar una obra musical, los miembros de la agrupación leen un papel. El papel puede ser de dos tipos: el que lee el director se denomina *partitura*, y es el papel que recoge la música de todos los instrumentos y voces. Por otra parte, cada músico lee su *particella* (también llamada “parte”), que solo recoge lo que debe interpretar él individualmente.

En los últimos tiempos la gestión de estas agrupaciones se ha realizado de forma manual, consistente en que los administradores, el director y los miembros se comunican mediante un grupo de mensajería sin ningún tipo de automatización.

De esta forma, algunas de las tareas más frecuentes en la administración de la agrupación podrían ser:

- Para la asistencia a eventos y ensayos, cada miembro escribe a través de un grupo, o individualmente al director o un administrador, su intención de asistir o no. Esta comunicación previa le es útil para gestionar qué repertorio es mejor ensayar, en qué orden, o si por el contrario es conveniente aplazar o suspender el evento o ensayo.
- Para el repertorio, los distintos documentos (partitura y partes) que necesitan los músicos se envían a través de alguna plataforma de almacenamiento en la nube (como Google Drive, OneDrive o Dropbox) o se imprimen en papel físico por los administradores, repartiéndose en un ensayo a los miembros presentes.
- Respecto a la comunicación de eventos y ensayos, se realiza por el canal no especializado de comunicación y la responsabilidad de recordarlos debidamente a los miembros antes de su celebración recae en los administradores.

Esta gestión manual y dependiente de aplicaciones de comunicación no especializadas deriva en varios problemas:

- La probabilidad de error humano aumenta en el momento en el que los administradores son responsables de recordar eventos y gestionar asistencia.
- La carga de trabajo de los administradores es mayor dada la responsabilidad de gestión manual que tienen a la hora de organizar un evento.
- Se crea una duplicidad de comunicación, ya que los miembros pueden escribir sobre su asistencia al director o a los administradores, pudiendo darse el caso de que finalmente no todos dispongan de los datos necesarios para organizar un ensayo o evento, o que dispongan de datos contradictorios entre sí.

Por todos estos problemas derivados de la gestión manual se necesita software dedicado específicamente a este propósito, automatizando la gestión, que sea personalizable a las necesidades de cada agrupación, no implique enormes gastos de uso y sea fácilmente usable por todos los usuarios.

De este modo, el proyecto está justificado por la necesidad de alternativas para mejorar la gestión del gran número de agrupaciones musicales que existen en nuestro país y que no han podido automatizar aún su gestión.

1.2. Objetivos

Los objetivos de este trabajo se pueden resumir en tres fundamentales:

- Por un lado, dar respuesta a los requerimientos de numerosas agrupaciones musicales con un software que les permita automatizar la gestión de eventos, miembros, ensayos, repertorio, etc., ayudando a una organización más ágil y efectiva.
- Por otro lado, investigar hasta qué punto la interfaz de usuario proporcionada por un bot de Telegram es suficiente, competitiva y cómoda para el usuario. Esta interfaz está integrada en un chat, por lo que se pretende identificar las limitaciones que puedan darse, y explorar soluciones que sorteen dichas carencias.
- Aportar al conocimiento colectivo aportando guías sobre el desarrollo de un bot de Telegram usando tecnologías actuales.

Para el primer objetivo se cumpla de manera satisfactoria, se establecen varios requisitos básicos:

- Se debe disponer de una página web donde quede claro qué problemas pretende resolver la herramienta, de modo que las agrupaciones puedan ver las ventajas de su uso. Esta información también debe quedar clara en la propia herramienta.
- El uso de la herramienta debe ser lo más sencillo y obvio posible, dada la heterogeneidad de los usuarios, por su edad, profesión, nivel de estudios o grado de implicación en la agrupación.
- En el desarrollo de la herramienta se tienen que tener en cuenta los roles de miembro no administrador y quien sí es administrador, de forma que cada rol deberá tener distintas funcionalidades disponibles. Por ejemplo, solo un administrador podrá eliminar a otros miembros.
- Se debe buscar el mejor diseño que se pueda ofrecer dentro de las limitaciones de la tecnología escogida, de modo que un mal diseño no haga que los usuarios prefieran volver a la gestión manual.

Capítulo 2

Estado del arte

2.1. Dominio del problema a resolver

En la actualidad, la digitalización de empresas, administraciones y organismos públicos avanza a ritmo imparable, y la situación excepcional generada por la pandemia de COVID-19 no ha hecho más que afianzar este avance.

Los datos del Índice de Digitalización de la Economía y la Sociedad (DESI) en 2022[2], índice que monitoriza el rendimiento digital de Europa y el progreso de los países de la Unión Europea en su competitividad digital, muestran que todos los países han progresado en su digitalización. Destaca el hecho de que los países que han empezado en un nivel más bajo de desarrollo digital crecen a un ritmo más rápido, y que numerosos países crecen a un ritmo más elevado del que se les espera, como es el caso de España[3] (con un 0.7 % más de progreso del esperado). En concreto, nuestro país es el séptimo con mayor nivel de digitalización según este índice, y el número de personas con competencias digitales básicas es del 64 %, por encima de la media de la UE, del 54 %. Está por encima de la UE también en implementación de la banda ancha fija de al menos 100 Mbps, usuarios de la administración electrónica o pymes con nivel básico de intensidad digital.

A nivel empresarial otros estudios concluyen que la digitalización está más estancada[4], debido a una baja inversión en TIC.

Por otro lado, a nivel de la administración se están impulsando estrategias para avanzar en la digitalización, como España Digital 2026[5], fomentando programas para acelerar la digitalización de pymes (pequeñas y medianas empresas), mejorar la conectividad o

Pese a todo esto, algunos sectores de la sociedad aún manifiestan reticencias para proceder a su digitalización, por la falta de herramientas, recursos o ayuda. Una de ellas es la que nos ocupa en este trabajo: las asociaciones o agrupaciones musicales. La complejidad de su gestión es suficiente para que los beneficios de la digitalización se puedan manifestar, dada la cantidad de

personas que se pueden encontrar en cada una de ellas y la naturaleza de los procesos que siguen, como la programación de ensayos y eventos, la gestión de repertorio o la gestión de miembros.

Por ejemplo, para la correcta planificación de ensayos y eventos, tanto el director como los administradores necesitan conocer con antelación qué músicos tienen pensado asistir y cuáles estarán ausentes. Esto les permitirá saber si se necesita llamar a un músico externo a la agrupación para que participe en un evento, o si es mejor cancelar o aplazar el evento. Para esta tarea, habitualmente el director avisa del evento por un grupo de mensajería a los miembros, y pide que quien no tenga pensado asistir se lo notifique. De este modo se generan varios problemas:

- La información sobre qué miembros han notificado su *no asistencia* solo la tiene el director, por lo que si otros administradores quieren conocerla se la tienen que pedir individualmente.
- Si algún miembro ha comunicado su “no asistencia” a otro administrador, tiene que poner la información en común con la del director, pudiendo dar lugar a discrepancias.
- Si el director quiere tener la información de asistencia centralizada, tiene que repasar sus conversaciones con los miembros para elaborar una tabla de asistencia, que tiene que ir actualizando manualmente, con la carga de trabajo que ello supone.

Similares problemas se crean para los demás procesos que se llevan a cabo en una agrupación musical.

Es evidente, por tanto, que se necesita una herramienta específica que permita automatizar sus procesos y avanzar en su digitalización para equipararla a la del resto de la sociedad.

Aunque ya existen herramientas como Glissandoo, que se analiza en la sección 2.2.1, son de pago, poco personalizables o están poco integradas en el flujo de trabajo actual de las agrupaciones. Es por ello que este trabajo pretende aportar una alternativa libre que solucione estos inconvenientes, de modo que:

- Cada agrupación pueda montar su propio servidor con la aplicación, personalizándola y adaptándola a sus necesidades.
- Los gastos por uso de la herramienta solo dependan del servidor que use la agrupación, si usa un servidor dedicado.
- Sea usable en todas las plataformas en las que se pueda acceder a un navegador web.

- Esté integrada en una herramienta de mensajería que ya esté en uso como es **Telegram**¹, pero añadiendo funcionalidad específica y automática necesaria para la gestión de la agrupación, eximiendo a los administradores y el director de tareas que se pueden automatizar.

2.2. Aplicaciones similares

Aunque aún la mayoría de agrupaciones usan Whatsapp como herramienta de comunicación sin usar ningún software más que automatice las tareas, en los últimos tiempos ha surgido otra herramienta que da respuesta a sus necesidades:

2.2.1. Glissandoo

Glissando² es un software creado por la empresa Plausible Technologies, con sede en Valencia. En su página web se define como “un software que nace con el objetivo de profesionalizar, modernizar y digitalizar las instituciones musicales”. Incorpora numerosas funcionalidades para digitalizar instituciones musicales, tal y como pretende este trabajo. Entre ellas se encuentran:

- Organización de ensayos y conciertos. Los miembros pueden ver los ensayos y conciertos en la página de inicio.
- Previsión de asistencia y pasar lista: para cada ensayo o concierto, el miembro puede seleccionar si tiene prevista su asistencia o no.
- Distribución de partituras: en la sección de inicio se puede acceder a un listado de todo el repertorio de los ensayos y conciertos programados, y en cada uno de los ensayos se puede ver el repertorio asociado.
- Comunicaciones: los administradores pueden añadir avisos que reciben los miembros, teniendo la capacidad de responder, reaccionar, y adjuntar archivos a los comunicados.

Se pueden ver varias capturas de la aplicación en la figura 2.1.

Algunas ventajas de esta alternativa son:

- Presenta un diseño intuitivo y fácil de usar para todos los usuarios.
- Está disponible para las plataformas móviles iOS y Android, ampliamente usadas por los usuarios.
- Es gratuita si la agrupación tiene menos de 20 músicos.

¹<https://telegram.org/>

²<https://glissandoo.com/>

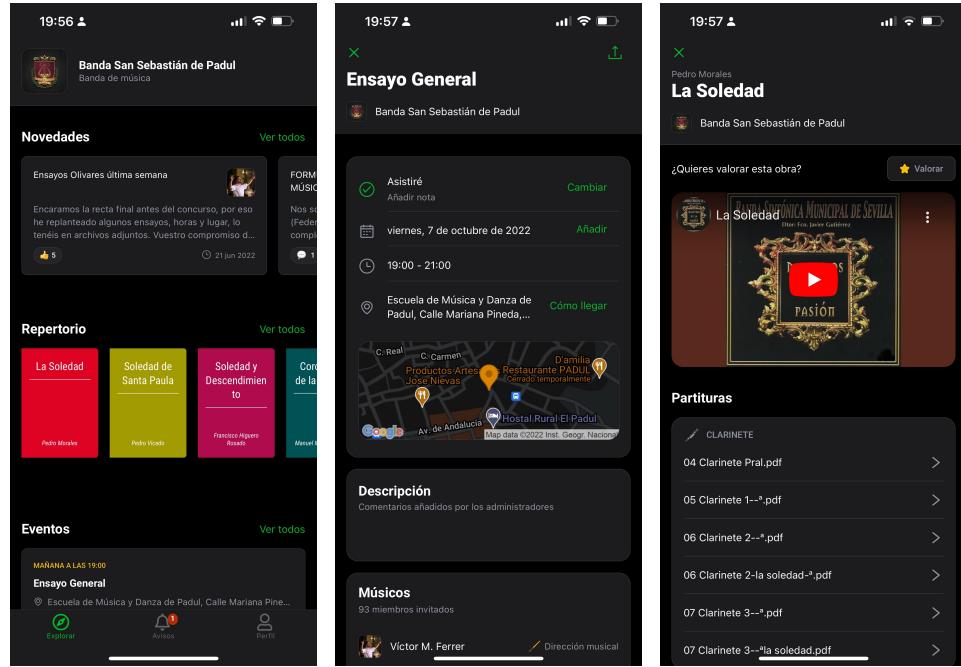


Figura 2.1: Capturas de la aplicación *Glissandoo*

- Es elaborada en por una empresa valenciana, lo cual da seguridad de que se adaptará a las necesidades de las agrupaciones musicales españolas.
- Incorpora un módulo de comunicaciones para poder centralizar todos los procesos de la agrupación en la aplicación, sin dejar los comunicados en otra aplicación de mensajería como WhatsApp.

Sin embargo, esta herramienta tiene varias desventajas:

- El código es privado, y una agrupación no puede montar su propio servidor con la aplicación en cuestión.
- Es de pago para uso en agrupaciones con más de 20 miembros.
- Solo se puede usar en móvil, ya que no dispone de versión web ni de escritorio. (Esto ha cambiado durante la realización de este trabajo, ya que han implementado una versión web).
- Sigue existiendo duplicidad, ya que aunque los miembros usen esta aplicación, la comunicación bidireccional ocasional entre administradores y miembros de la banda se sigue realizando a través de una herramienta distinta de comunicación no especializada.

2.2.2. Agregación de servicios no especializados

Mediante *agregación de servicios no especializados* nos referimos al uso de varias herramientas que no están especializadas para este caso de uso, pero que de cierta forma pueden suplir las necesidades:

- Como almacenamiento de partituras, existen servicios como **Google Drive**³, **Dropbox**⁴ o **Microsoft OneDrive**⁵.
- Para la planificación de eventos y conciertos se puede usar un calendario web como **Google Calendar**⁶ o **Microsoft Outlook**⁷.
- Para estimar la asistencia se pueden usar herramientas que permiten encuestas de este tipo como **Doodle**⁸.
- Para controlar la asistencia real se puede usar una hoja de cálculo, por ejemplo en **Google Sheets**⁹.
- Como medio de comunicación se utiliza otro servicio como **WhatsApp**¹⁰ o **Telegram**¹¹.

La ventaja de esta aproximación (la mayormente utilizada en la actualidad) es el mayor grado de personalización, ya que los servicios se pueden intercambiar si la funcionalidad o confiabilidad no es la esperada.

No obstante, las desventajas son numerosas, como ya se ha comentado anteriormente:

- Gran carga de trabajo para los administradores.
- Redundancia de comunicaciones entre los miembros y los administradores.
- Muchas de estas herramientas requieren de suscripción *premium* para usar toda la funcionalidad.

³<https://drive.google.com/>

⁴<https://www.dropbox.com/>

⁵<https://onedrive.live.com/>

⁶<https://calendar.google.com/>

⁷<https://outlook.live.com/>

⁸<https://doodle.com/>

⁹<https://docs.google.com/spreadsheets/>

¹⁰<https://www.whatsapp.com/>

¹¹<https://telegram.org/>

Capítulo 3

Mordente

Tras estudiar la motivación de este trabajo y habiendo justificado la necesidad de una alternativa software para la gestión de agrupaciones musicales, pasamos a desarrollar el producto. Para esta fase se va a hacer uso de la llamada metodología de Diseño Centrado en el Usuario (UCD por sus siglas en inglés), descrita en el siguiente apartado.

3.1. Desarrollo de Mordente

En este punto se van a establecer las metodologías de diseño a seguir durante el desarrollo de la herramienta, de modo que tengamos disponibles unas guías a seguir durante el proceso y unas herramientas que nos permitan definir la funcionalidad esperada y concretar las necesidades de los futuros usuarios.

3.1.1. Diseño Centrado en el Usuario

El Diseño Centrado en el Usuario está estrechamente relacionado con el Diseño Centrado en el Humano[6]. Este último “es una aproximación al desarrollo de sistemas interactivos que se centra específicamente en hacer sistemas usables. Es una actividad multi-disciplinar” [7].

Los principios[8] en los que se fundamenta se pueden resumir en:

1. **Aproximación temprana a usuarios y tareas:** la recogida de información es estructurada y sistemática.
2. **Medida empírica y testeo del uso del producto:** se centra en la facilidad de aprendizaje y de uso, y la prueba de prototipos con usuarios reales.
3. **Diseño iterativo:** el producto se diseña, modifica y prueba de forma repetida, permitiendo repensar completamente la idea gracias a la prueba de modelos conceptuales tempranos.

3.1.2. Design Thinking

El Design Thinking es un proceso de diseño descrito en cinco fases[9]: definir el problema, buscar las necesidades, idear, construir y probar. Este proceso nos ayuda a aplicar el Diseño Centrado en el Usuario mediante una serie de procesos.

Procesos

Otros autores hablan de que el Design Thinking no se compone de fases sino de varios espacios que no son secuenciales: se solapan y se forma un bucle entre ellos, repitiéndolo más de una vez y explorando distintas aproximaciones [10].

1. **Inspiración:** se observa cómo funcionan las personas para encontrar problemas y oportunidades.
2. **Empatía:** se trata de entender los deseos y necesidades de los potenciales clientes, no solo de forma técnica sino sabiendo por qué hacen las cosas de la forma actual y qué les resulta más importante.
3. **Ideación:** se trata de una combinación de pensamiento divergente y convergente. El pensamiento divergente genera nuevas ideas de forma creativa, mientras que el convergente trata de sintetizar las ideas y elegir las mejores. Teniendo diversas personas en el equipo, la técnica más utilizada es el *brainstorming*, en el que todos los miembros dan ideas espontáneas que se apuntan de forma que cada nueva idea puede fomentar la creatividad de los demás.
4. **Implementación y prototipado:** en este proceso las ideas se convierten en productos concretos. Durante este proceso se van generando diversos prototipos que se prueban con usuarios reales para seguir mejorando la idea.

3.2. Personas ficticias

Las personas ficticias se crean dentro de una técnica perteneciente al *Design Thinking*. Se tratan de representaciones ficticias de los usuarios, con metas, motivaciones, características y comportamientos de un grupo real de usuarios [11].

Esta técnica nos permite diseñar para personas reales y no para “usuarios” abstractos.

En el caso del proyecto que nos ocupa, se han creado cuatro personas ficticias: dos que cumplirían el rol de administrador en el sistema y otras dos que lo usarían únicamente como miembros.

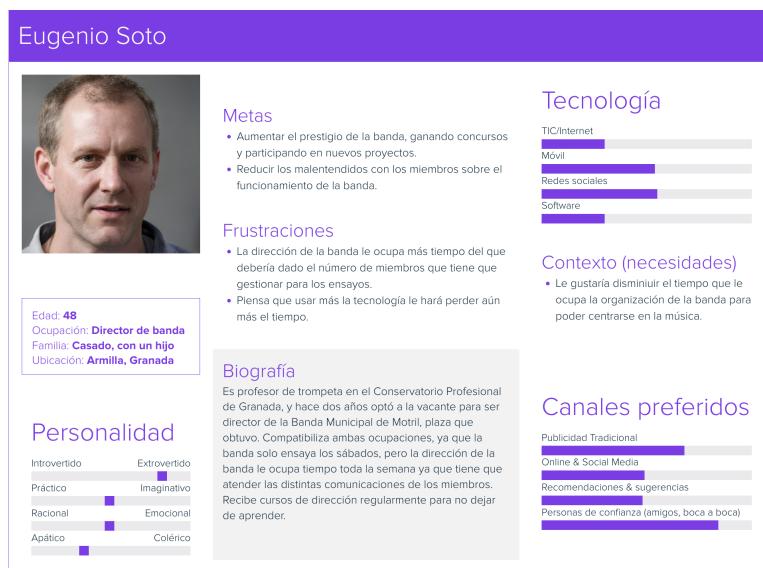


Figura 3.1: Persona: Eugenio Soto

Las figuras 3.1 y 3.2 muestran dos personas que se dedican a la organización de la banda, Eugenio más desde el punto de vista musical y Clara desde el logístico.

Mientras Clara está claramente más familiarizada con las nuevas tecnologías, Eugenio muestra algunas reticencias que deberemos tratar durante el desarrollo, de forma que el producto final sea fácilmente usable y accesible. Igualmente, ya que Clara sí usa las nuevas tecnologías habitualmente, el producto tiene que ser atractivo para ella de forma que no suponga un retroceso con respecto a las herramientas actuales.

Los perfiles que vemos en las figuras 3.3 y 3.4 se refieren a personas que acuden como miembros a la banda de su respectivo pueblo. A Inma se aplican las mismas restricciones que a Eugenio, además de que debemos asegurarnos de que realmente facilitamos que los miembros recuerden los eventos diarios para hacerles la vida más fácil. Con respecto a David, cabe reseñar que, ya que la música no es su mayor afición, preferirá usar la app durante el menor tiempo posible, por lo que se debe intentar que las notificaciones que le lleguen le permitan responder de forma rápida.

3.3. Malla receptora de información

La malla receptora de información (*Feedback Capture Grid*) es un método para organizar feedback de manera estructurada en el cual se divide una hoja de papel en cuatro cuadrantes, cada uno representado con un símbolo diferente. El superior izquierdo se representa con un símbolo "+", y en él

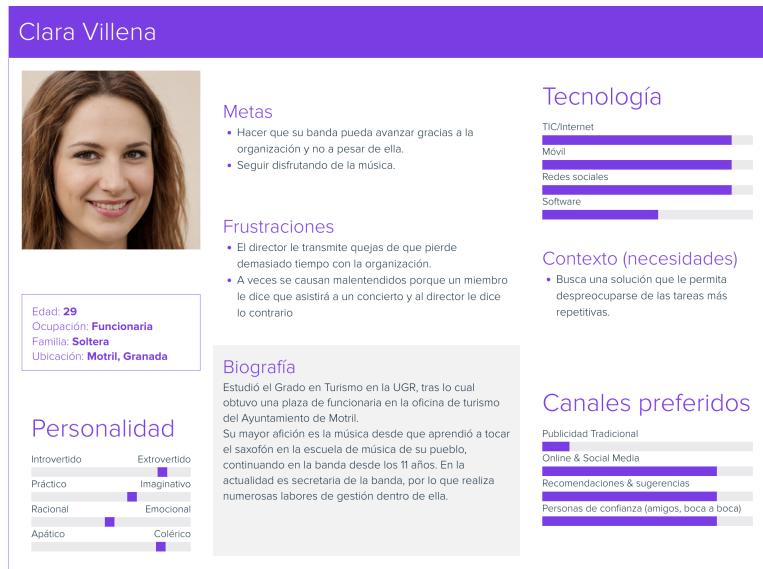


Figura 3.2: Persona: Clara Villena

anota el feedback positivo. En el superior derecho, con símbolo de triángulo, se recogen las críticas. En el inferior izquierdo, con una interrogación, se añaden las preguntas que han surgido y en el inferior derecho, con una bombilla, las nuevas ideas que surgen [12].

A continuación se resumen los puntos recogidos para cada uno de los apartados.

3.3.1. Aspectos interesantes o relevantes de la idea

- Está integrado en una app de mensajería.
- Funciona en todos los dispositivos móviles y de escritorio.
- El software es de código abierto.
- Lo puede utilizar cualquier agrupación.
- Se reciben notificaciones a través de la misma app de comunicación.
- Se pueden crear instancias del bot configurables a las necesidades de distintas agrupaciones.

3.3.2. Críticas constructivas o posibilidades de mejora

- Depende del funcionamiento de Telegram.
- La interfaz es más limitada que una aplicación web.



Figura 3.3: Persona: Eugenio Soto

- Se requiere tener cuenta de Telegram.
- Se requiere un servidor propio en el que esté alojada la aplicación para funcionar, o pagar el uso de uno en la nube.
- La interfaz en ordenador requiere de demasiados clics para llevar a cabo las tareas.

3.3.3. Preguntas nuevas a partir de la experiencia del usuario

- ¿Cuál es el coste de mantener la aplicación?
- ¿Se podrá pedir algo al bot desde un grupo?
- ¿Se podrán elegir qué notificaciones y recordatorios queremos recibir?
- ¿Los administradores podrán ver informes de la asistencia a ensayos?
- ¿La aplicación fomentará la asistencia de los miembros a los ensayos?
- ¿Los usuarios más reacios a la tecnología sabrán usar el bot con soltura?
- ¿Se va a poder usar el bot en varios idiomas?



Figura 3.4: Persona: Inma Medina

3.3.4. Ideas que surgen

- Hacer también una aplicación web para no depender de Telegram.
- Implementar un sistema de premios para los usuarios que cumplan objetivos de asistencia.
- Tener una instancia del bot funcionando para las agrupaciones que no lo quieran tener en un servidor propio.
- Crear una página web con guías para el uso del bot.
- Implementar internacionalización con varios idiomas.

3.4. Descripción de la propuesta

Teniendo en cuenta el análisis que se ha hecho hasta ahora, se puede dar una propuesta de solución que responda a las preguntas y necesidades expuestas, tanto en las personas ficticias como en el *Feedback Capture Grid*.

3.4.1. Propuesta del producto

Primeramente se determinan unas características y funcionalidades mínimas para tener un Producto Mínimo Viable, o MVP por sus siglas en inglés. El Producto Mínimo Viable es una técnica que se basa en disponer

de suficientes características para satisfacer a los primeros usuarios, de manera que las siguientes funcionalidades se pueden desarrollar teniendo en cuenta el *feedback* de estos primeros usuarios [13]. En nuestro caso, las funcionalidades incorporadas en esta primera versión serán:

1. Agrupaciones:
 - a) [Administrador] Crear nuevas agrupaciones
 - b) [Administrador] Eliminar agrupaciones
 - c) [Administrador] Invitar miembros a la agrupación
 - d) [Administrador] Eliminar miembros de la agrupación
 - e) [Usuario] Unirse a una agrupación existente
 - f) [Usuario] Salirse de una agrupación
 - g) [Usuario] Ver todas las agrupaciones en las que estoy inscrito
 - h) [Usuario] Ver el detalle de una agrupación
2. Eventos:
 - a) [Administrador] Crear nuevos eventos en mi agrupación
 - b) [Administrador] Eliminar eventos
 - c) [Administrador] Invitar miembros a un evento
 - d) [Administrador] Eliminar miembros de un evento
 - e) [Usuario] Ver los eventos próximos
 - f) [Usuario] Ver el detalle de un evento
3. Asistencia:
 - a) [Administrador] Ver qué usuarios han respondido su asistencia a un evento
 - b) [Usuario] Responder la asistencia a un evento
 - c) [Usuario] Registrar la razón por la que no puedo asistir a un evento
4. Recordatorios:
 - a) [Usuario] Recibir recordatorios diarios con los eventos de ese día
5. Repertorio:
 - a) [Administrador] Añadir una obra a la agrupación
 - b) [Administrador] Eliminar una obra la agrupación
 - c) [Administrador] Subir partituras de las obras

- d) [Administrador] Eliminar una obra de la agrupación
- e) [Usuario] Ver las obras de mi agrupación
- f) [Usuario] Descargar las partituras de mi agrupación

6. Administración:

- a) [Administrador] Añadir administradores
- b) [Administrador] Quitar administradores

7. Notificaciones:

- a) [Administrador] Recibir notificación cuando un miembro responde sobre su asistencia
- b) [Usuario] Recibir notificación cuando se asigna un evento

3.4.2. Elección de metodología de desarrollo

Dadas las características del proyecto, se va a apostar por una metodología de desarrollo ágil como es SCRUM, aunque con ciertas adaptaciones ya que el equipo de desarrollo va a estar compuesto por una sola persona.

Esta metodología ágil nos permitirá ir mejorando el producto a lo largo del desarrollo, pudiendo probarlo desde etapas tempranas. SCRUM es un método de desarrollo e innovación que enfatiza un conjunto de valores y prácticas para la gestión de proyectos, más que tratar requerimientos e implementación. Trata de hacer el proceso de la gestión más **empírico** que **definido** [14].

La naturaleza flexible del marco de trabajo SCRUM se adapta a proyectos como el que nos ocupa, dado que no se conocen todos los requisitos concretamente con antelación y nos ayuda a que el usuario sea parte del desarrollo, algo útil si queremos hacer un desarrollo centrado en el usuario.

3.4.3. Planificación

Para poder tener un producto que cumpla con los objetivos reseñados anteriormente, es necesario establecer en qué orden se van a realizar las distintas tareas.

Planificación en sprints

Dado que la metodología de desarrollo a usar será SCRUM, para la planificación del proyecto se va a usar uno de sus elementos más importantes como es la planificación de los sprints.

Los sprints son eventos de una longitud fija, generalmente menor a un mes. Durante un sprint se intenta realizar todo el trabajo necesario para alcanzar el objetivo del producto.

Los sprints corresponden a objetivos concretos del producto, por lo que el resto de tareas (como la toma de decisiones o el aprendizaje) se van a incluir en la planificación temporal pero no como sprints. Los sprints planificados para este proyecto son:

1. **Primer sprint:** Se va a implementar un bot de prueba con funcionalidad de actualización de una base de datos de palabras. Este bot servirá como exploración de las distintas funcionalidades que se usarán posteriormente.
2. **Segundo sprint:** Se va a implementar la lógica para responder a los comandos de los usuarios, así como la funcionalidad para crear agrupaciones y unirse a una agrupación.
3. **Tercer sprint:** Se van a crear diversos plugins, uno para permitir guardar la sesión del usuario en la base de datos y otro para usar un menú de calendario que sirva para fijar las fechas de eventos. Estos plugins serán publicados.
4. **Cuarto sprint:** Se añadirá la funcionalidad básica relacionada con los eventos, como la creación y eliminación. También permitir a un miembro salirse de una agrupación, así como habilitar enlaces de invitación.
5. **Quinto sprint:** Se podrán añadir y eliminar obras a las agrupaciones. Función de edición de agrupaciones, eventos y obras.
6. **Sexto sprint:** Notificaciones de asignación de eventos y adición de miembros a las agrupaciones además de recordatorios de los eventos diarios.

Planificación temporal

La planificación del desarrollo del bot en el tiempo se sucederá de la siguiente forma:

- Primeramente se tomarán las decisiones oportunas sobre las herramientas a utilizar, requisitos concretos del software y análisis de la competencia.
- Después habrá un periodo de formación en las herramientas necesarias para el desarrollo de la aplicación. Esta formación también se extenderá a lo largo de todo el desarrollo.
- Los sprints de desarrollo vendrán tras el periodo inicial de formación.
- Finalmente tendremos un periodo de pruebas y de finalización de la memoria.

Toma de decisiones	10 horas
Formación	100 horas
Implementación	150 horas
Pruebas	10 horas
Memoria	150 horas
Total	420 horas

Tabla 3.1: Planificación temporal

En la tabla 3.1 se muestra el tiempo estimado que requerirá cada tarea del desarrollo. Dado que el tiempo requerido para cada crédito ECTS es de entre 25 y 30 horas y esta asignatura es de 12 créditos, hablaríamos de 300-360 horas, sin embargo el tiempo para el desarrollo se estima mayor por la formación previa requerida en las nuevas tecnologías utilizadas.

Este exceso de tiempo no se estima perjudicial dada la naturaleza formativa de este trabajo, ya que los conocimientos adquiridos podrán seguir siendo utilizados posteriormente a la finalización.

Diagrama de Gantt

El diagrama de Gantt que se puede ver en la figura 3.5 se ha construido teniendo en cuenta que cada día se van a trabajar 2 horas, por lo tanto se van a trabajar 210 días, o lo que es lo mismo, 42 semanas.

Así mismo, no se concretan las fechas reales ya que pueden cambiar por factores externos durante el desarrollo del proyecto.

3.4.4. Presupuesto

El coste de infraestructura es nulo en esta primera versión, puesto que el bot puede estar alojado en un servidor propio o usando el plan gratuito de cualquier servicio de alojamiento de servidores virtuales en la nube. Por tanto, el coste a considerar es el del tiempo empleado por el desarrollador.

Según el Anexo I del XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública¹ el salario mínimo a partir del 31 de diciembre de 2019 para un Programador Junior es de 15860,56 euros al año. La tabla 3.2 muestra el presupuesto estimado teniendo en cuenta el siguiente cálculo y la suposición de 12 meses el año:

$$12 \text{ meses} \times 4 \frac{\text{semanas}}{\text{mes}} \times 40 \frac{\text{horas}}{\text{semana}} = 1920 \text{ horas}$$

$$\text{coste por hora: } \frac{15860,56}{1920} = 8,26 \text{ €}$$

¹https://www.boe.es/diario_boe/txt.php?id=BOE-A-2018-3156

Concepto	Unidades	€/unidad	Total
Hora de Programador Junior	420	8,26	3469,2
Total			3469,2

Tabla 3.2: Presupuesto

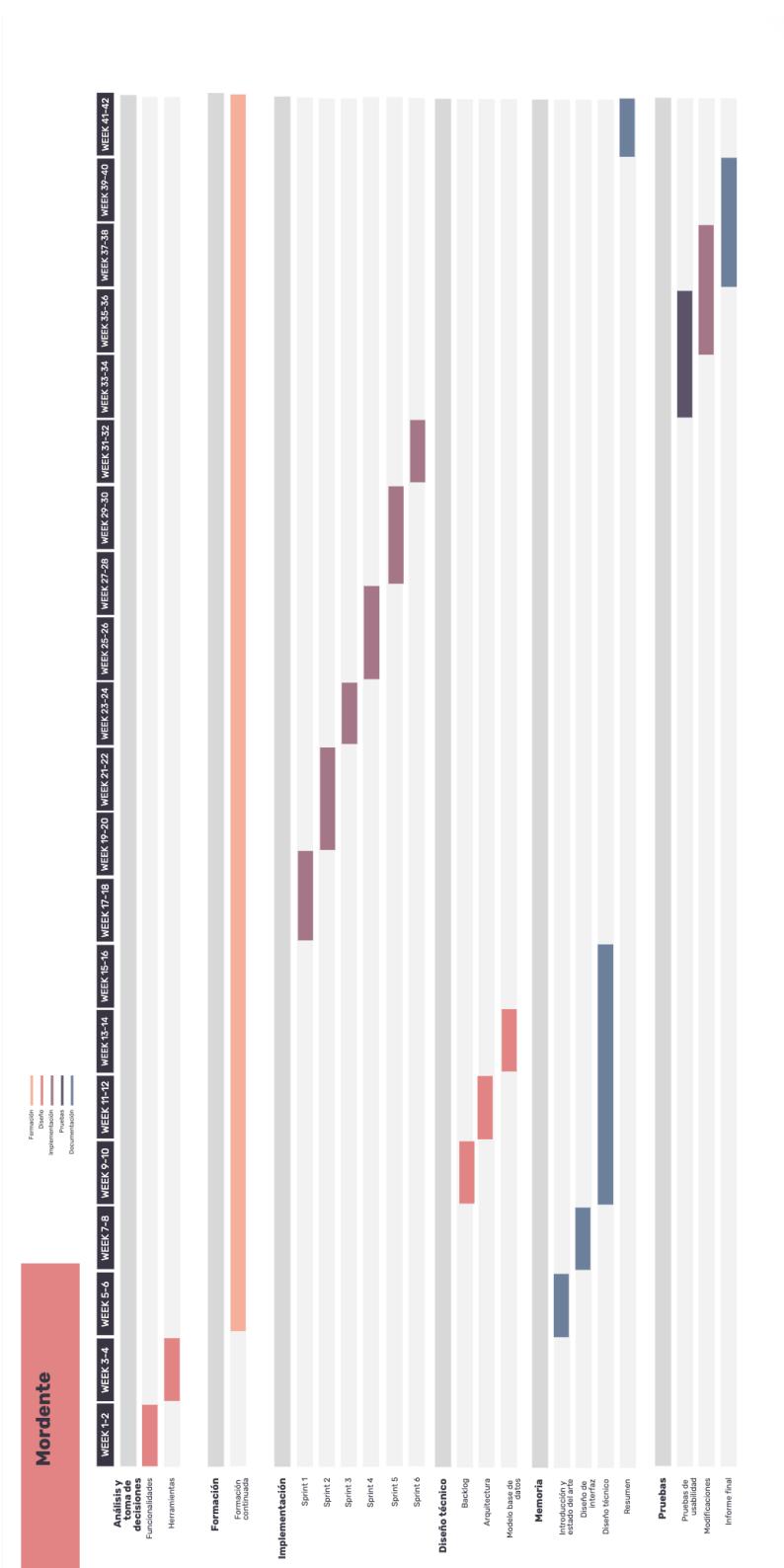


Figura 3.5: Diagrama de Gantt del proyecto

Capítulo 4

Diseño de la interfaz de usuario

A nivel de diseño de la interfaz, lo principal es reseñar que nos encontramos con una gran limitación: no podemos desarrollar una interfaz¹ totalmente a nuestro gusto, sino que debemos usar la API que nos proporciona Telegram.

Es decir, nuestro bot va a operar dentro de un chat, y como todo chat, la interfaz estará compuesta sola y únicamente por mensajes, que podrán ser de distinto tipo (texto, imágenes, documentos, stickers...). La única adición que aporta Telegram a esta interfaz es que los mensajes que envía el bot pueden adjuntar un menú de opciones que el usuario puede pulsar para pedir al bot una determinada acción, además de tener un menú de comandos disponible continuamente junto al cuadro de introducción de texto.

Esta rigidez de la interfaz puede ser vista de manera positiva, ya que aporta las siguientes ventajas:

- Las decisiones de diseño son mucho más simples, puesto que no tenemos todo un elenco de estilos, formatos e interacciones que elegir. Por ejemplo, si el usuario quiere acceder a la información general sobre la aplicación, la mejor solución dentro del bot es que el usuario use un comando /about al cual el bot responde con un texto explicativo. En el caso de una aplicación web, la interacción por parte del usuario podría ser desde pulsar un botón hasta usar un atajo de teclado, y la visualización podría ser desde sobreponer un cuadro con la información hasta enlazar a una página distinta.
- La personalización ya está implementada para nosotros: esta característica se delega a la propia aplicación de Telegram, que permite se-

¹El 16 de abril de 2022, la API para bots Telegram fue actualizada[15] con una nueva función para abrir una aplicación web sin salir del bot como respuesta a varios eventos. Sin embargo, esa opción se ha descartado para este trabajo ya que la implementación del bot se encontraba en estado avanzado y se ha analizado que las complicaciones técnicas de implementar esta nueva interfaz extenderían el tiempo de desarrollo considerablemente. Se propone como mejora futura.

lecciónar el tema claro u oscuro o cambiar el tamaño del texto.

- La accesibilidad está garantizada, ya que la interfaz de mensajes ya está totalmente adaptada a personas con discapacidad auditiva o visual.
- La familiarización del usuario con la interfaz es mayor de antemano, ya que con toda probabilidad habrá usado un chat antes y la interfaz del bot no es más que una extensión de un chat corriente. Esto le supondrá una adaptación más sencilla y más centrada en saber cuál es la funcionalidad que en cómo está dispuesta.
- El usuario interacciona de forma privada en un chat con el bot, de modo que el tono de respuesta del bot y el intercambio de mensajes hacen que el usuario perciba la interfaz como más amigable y cercana.
- Los entornos que más necesitan la comunicación se benefician de esta integración. Por ejemplo, en el caso que nos ocupa siempre se suele usar un grupo de mensajería, y un bot puede ser integrado en un grupo de Telegram.

Decimos que la limitación es autoimpuesta porque es tarea de este trabajo también analizar si las ventajas expuestas superan o igualan a las desventajas de operar dentro de un chat, es decir, si un bot de Telegram de este tipo puede operar sin la ayuda de una aplicación web o móvil que la acompañe.

4.0.1. Bocetos de interfaz

Con todo lo anterior en mente, pasamos a crear varios bocetos de lo que debería ser la interfaz entre el usuario y el bot:

Lista de elementos

Aunque se ha expuesto que la interfaz está limitada a la API disponible, en el caso de las listas tenemos 3 opciones si queremos que el usuario pueda acceder al detalle de cada elemento:

1. Enviar un mensaje muy corto acompañado de un menú, cada botón del menú muestra el título de un elemento y al pulsar en él se recibe el detalle.
2. Igual que la opción 1, pero añadiendo en el texto una lista de los elementos con un resumen de cada uno.
3. Nueva opción explorada en este trabajo como variación de la opción 2: omitir el menú para no repetir los nombres, y en su lugar acompañar

cada elemento de la lista con un comando del estilo /event_N de manera que al pulsar y enviar el comando el bot responda con el detalle del elemento.

En todo caso la lista debe ir acompañada de un botón para añadir elementos.

El diseño se ve en la figura 4.1.

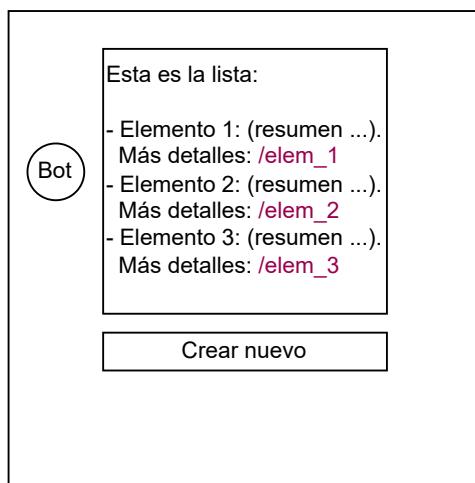


Figura 4.1: Diseño de la interfaz de lista

Detalle de elemento

En este caso, toda la información sobre el elemento se muestra en el mensaje de respuesta, que va acompañado de un menú con las acciones que se pueden realizar con el elemento. Ver figura 4.2.

Creación de elemento

La interfaz más sencilla y equivalente a un formulario para crear un elemento es una conversación: el bot va preguntando cada campo requerido al usuario, que puede saltar los campos opcionales mediante un botón de "Saltar". Idealmente, para la introducción de fechas se debería mostrar un menú con forma de calendario. Ver figura 4.3.

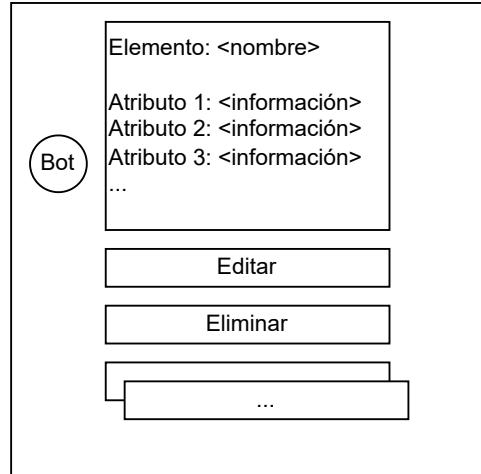


Figura 4.2: Diseño de la interfaz del detalle de un elemento

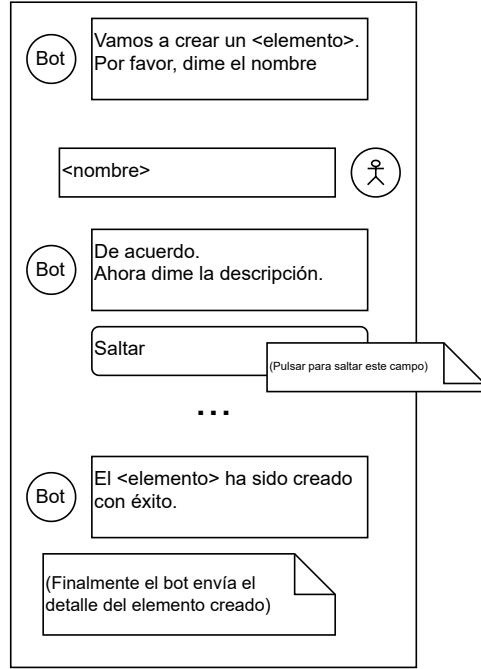


Figura 4.3: Diseño de la interfaz de creación de un elemento

Capítulo 5

Diseño técnico

5.1. Listado de historias de usuario (Backlog)

Para extraer las historias de usuario se ha usado la aplicación **Jira**¹, de la compañía **Atlassian**. Por consistencia y aunque puedan parecer desordenados, los identificadores de las épicas y de las historias que aparecen en esta sección son los que se han obtenido en dicha herramienta.

MTB-1. Gestión de agrupaciones.

MTB-8. Como administrador de una agrupación, quiero crear una nueva agrupación.

MTB-9. Como administrador de una agrupación, quiero eliminar mi agrupación.

MTB-10. Como administrador de una agrupación, quiero editar mi agrupación.

MTB-2. Gestión de membresías.

MTB-11. Como administrador de una agrupación, quiero añadir miembros a mi agrupación.

MTB-13. Como usuario, quiero unirme a una agrupación.

MTB-12. Como administrador de una agrupación, quiero eliminar miembros de la agrupación.

MTB-14. Como miembro, quiero salirme de una agrupación.

MTB-15. Como miembro, quiero seleccionar qué instrumento toco en la agrupación.

MTB-4. Gestión de eventos.

¹<https://www.atlassian.com/software/jira>

- MTB-16. Como administrador de una agrupación, quiero añadir un evento.
- MTB-35. Como administrador de una agrupación, quiero configurar qué miembros están invitados a un evento.
- MTB-17. Como administrador de una agrupación, quiero editar los datos de un evento.
- MTB-18. Como administrador de una agrupación, quiero eliminar un evento.
- MTB-19. Como miembro, quiero ver la lista de eventos a los que estoy invitado.
- MTB-20. Como miembro, quiero ver los detalles de un evento.
- MTB-36. Como administrador, quiero publicar un evento.
- MTB-37. Como miembro, quiero posponer un evento.
- MTB-5. Notificaciones.
 - MTB-58. Como administrador, quiero ser notificado cuando un miembro registra su asistencia prevista a un evento.
 - MTB-26. Como miembro, quiero ser notificado cuando soy invitado a un evento.
 - MTB-27. Como miembro, quiero ser notificado cuando se modifica un evento.
 - MTB-28. Como miembro, quiero ser notificado cuando se pospone un evento.
 - MTB-55. Como miembro, quiero ser notificado cuando se añade nuevo repertorio.
 - MTB-47. Como miembro, quiero seleccionar qué notificaciones recibo.
- MTB-21. Gestión de asistencia.
 - MTB-22. Como miembro, quiero registrar mi asistencia prevista a un evento.
 - MTB-23. Como miembro, quiero justificar mi ausencia a un evento.
 - MTB-24. Como administrador, quiero ver qué miembros asistirán a un evento.
 - MTB-25. Como administrador, quiero registrar la asistencia real a un evento.
 - MTB-48. Como administrador, quiero ver un resumen de la asistencia a un evento.
- MTB-6. Recordatorios.

MTB-32. Como miembro, quiero recibir recordatorios de los eventos próximos.

MTB-29. Gestión de administradores.

MTB-30. Como administrador, quiero hacer que un miembro sea administrador.

MTB-31. Como administrador, quiero quitarle a un miembro los permisos de administrador.

MTB-33. Como administrador, quiero especificar mi rol como administrador.

MTB-3. Gestión de repertorio.

MTB-34. Como administrador, quiero añadir repertorio a mi agrupación.

MTB-45. Como administrador, quiero editar una obra.

MTB-46. Como administrador, quiero eliminar una obra.

MTB-43. Como administrador, quiero añadir la partitura de una obra.

MTB-44. Como administrador, quiero eliminar la partitura de una obra.

MTB-49. Como administrador, quiero añadir un *tag* a una obra.

MTB-50. Como administrador, quiero eliminar un *tag* de una obra.

MTB-56. Como miembro, quiero descargar las partituras.

MTB-59. Registro de errores.

MTB-60. Como administrador de sistemas, quiero tener una vista centralizada de los errores que se hayan registrado en la aplicación.

5.2. Historias de usuario

5.3. Metodologías y tecnologías de base que podrían usarse

En esta sección haremos una reflexión y análisis sobre las herramientas que usaremos durante el desarrollo, de forma que pueda existir una formación en el uso de estas tecnologías previa al inicio del desarrollo del proyecto.

5.3.1. Lenguaje de programación

Se plantean los siguientes lenguajes de alto nivel, que disponen de frameworks implementados para crear un bot de Telegram:

- **JavaScript**: es un lenguaje compilado en tiempo de ejecución, con tipado dinámico y multi-paradigma, soportando programación orientada a objetos, funcional, imperativa y dirigida por eventos[16]. Se conforma al estándar “ECMAScript”, y es una de las tecnologías centrales de la World Wide Web: el 98 % de los sitios web lo usan en el lado del cliente[17], y desde el surgimiento de Node.js es una tecnología en auge para servidores.
- **TypeScript** (<https://www.typescriptlang.org/>): es un superconjunto de JavaScript que añade sintaxis para tipos estáticos, y a través de un compilador genera código JavaScript[18]. El añadido de tipos estáticos permite detectar errores de forma más temprana y agilizar la escritura de código gracias al autocompletado. Por otro lado, la adición de tipos es un tiempo extra empleado por el programador, por lo que debe analizarse si es conveniente su uso o no.
- **Python** (<https://www.python.org/>): es un lenguaje interpretado, interactivo y principalmente orientado a objetos, aunque soporta otros paradigmas como el funcional o el procedimental[19]. Es muy usado en el campo de la computación científica y la inteligencia artificial. Su uso en el desarrollo web se ha extendido para el lado del servidor con la aparición de frameworks como Django (<https://www.djangoproject.com/>) o Flask (<https://flask.palletsprojects.com/>).
- **PHP** (<https://www.php.net/>): es un lenguaje interpretado usado principalmente para desarrollo web en el lado del servidor, cuya principal característica es que puede ser embebido en HTML, de forma que cada “trozo de PHP” se ejecuta para intercambiarse por HTML. Sin embargo su uso ha cambiado en los últimos años, dando lugar a frameworks como Symphony (<https://symfony.com/>) o Laravel (<https://laravel.com/>) basados en la arquitectura Modelo-vista-controlador.

Se proporciona una tabla comparativa entre los lenguajes, la tabla 5.1.

Decisión

Por el menor tiempo necesario para la formación del desarrollador en los lenguajes **JavaScript** y **TypeScript**, se elegirá uno de estos.

TypeScript es de gran ayuda cuando se desarrolla una aplicación compleja y con un solo desarrollador: puede acelerar el desarrollo gracias al autocompletado y evita la mayor parte de los errores en tiempo de ejecución, reduciendo el tiempo necesario para pruebas. Es el lenguaje que usaremos en el proyecto.

	JavaScript	TypeScript	Python	PHP
Tipo	Compilado en tiempo de ejecución	Compilado a JavaScript	Interpretado	Interpretado
Tipado estático	No	Sí	Poco estricto, y poco uso	Poco estricto, y poco uso
Coste ^a	Nulo	Bajo	Medio	Medio
Popularidad ^b	65.36 %	34.83 %	48.07 %	20.87 %
Comunidad ^c	2432281	196707	2035248	1447104

^aAprendizaje necesario por parte del autor para poder implementar el proyecto

^b<https://survey.stackoverflow.co/2022/>

^cPreguntas totales en stackoverflow: <https://stackoverflow.com/tags>

Tabla 5.1: Comparación de lenguajes de programación

5.3.2. Framework para el desarrollo del bot

Existen diversas bibliotecas que ayudan a crear un bot, de forma que no haya que escribir todo el código desde cero para interaccionar con la API de Telegram, siguiendo el principio “Don’t Reinvent the Wheel” (“No reinventes la rueda”) para también ahorrar presupuesto.

Las opciones que se analizan son:

- **Telegraf** (<https://telegraf.js.org/>): biblioteca desarrollada inicialmente para JavaScript, migrada en la versión 4 dando soporte a TypeScript. No dispone de documentación guiada, y la migración a la versión 4 trajo consigo más complejidad de uso. Está inspirada el sistema de middleware de Express.²
- **grammY** (<https://grammy.dev/>): biblioteca desarrollada desde un inicio para TypeScript (aunque se puede usar con JavaScript). Está inspirada en telegraf y fue creada desde cero por uno de los antiguos mantenedores de Telegraf como única forma de resolver sus mayores inconvenientes³. Dispone de una completa documentación con guías para cada uno de los conceptos importantes.
- **node-telegram-bot-api** (<https://github.com/yagop/node-telegram-bot-api>): biblioteca ligera para interactuar con la API de Telegram, diseñada para Node.js, y no implementa un sistema de middleware.

²<https://expressjs.com/es/guide/using-middleware.html>

³<https://github.com/telegraf/telegraf/discussions/1526>

	telegraf	grammY	node-telegram-bot-api	python-telegram-bot
Lenguaje	JavaScript	TypeScript	JavaScript	Python
Documentación	Autogenerada, solo API	Completa, numerosos tutoriales	Básica	Autogenerada, solo API
Versión API Telegram	6.2	6.2	6.2	6.2
Popularidad ^a	6.1k	663	6.5k	19.8k

^aEstrellas en GitHub a 7 de octubre de 2022

Tabla 5.2: Comparación de frameworks para creación de bots de Telegram

- **python-telegram-bot** (<https://python-telegram-bot.org/>): biblioteca para Python que provee clases de alto nivel como capa de abstracción sobre la API de Telegram.

La tabla 5.2 compara los distintos frameworks.

Decisión

En base a la decisión anterior sobre el lenguaje de programación, elegiremos **telegraf**, **grammY** o **node-telegram-bot-api**.

Lo primero que consideraremos es el nivel de abstracción de estas librerías. **Telegraf** y **grammY** aportan un mayor nivel de abstracción gracias a implementar una arquitectura basada en **middleware**, muy conveniente para bots complejos.

Una vez sopesado esto, el punto que más pesa es la documentación: mientras **grammY** aporta una documentación completa y con guías para cada uno de los problemas que se pueden encontrar desarrollando un bot de Telegram, **telegraf** **no tiene** ninguna documentación más allá de la autogenerada por los comentarios en su código. Dado que no tenemos ninguna experiencia con ninguna de las herramientas, utilizaremos **grammY** para asegurarnos que la formación pueda llevarse a cabo sin problemas.

5.3.3. ORM: *Object-relational mapper*

Un **ORM** es una herramienta que nos permite transformar los registros de la base de datos en objetos del lenguaje de programación que estamos usando y viceversa.

Las más famosas para **TypeScript** son **TypeORM**, **Sequelize** y **Prisma**. Sin embargo, en este aspecto la decisión es sencilla dado que **Prisma** es el único los tres con la característica de ser totalmente *type-safe*: ofrece tipado

estático para todas las entradas y salidas consultas de la base de datos. Por este motivo **prisma** se ha erguido como un estándar para el lenguaje **TypeScript**.

5.3.4. Base de datos

La base de datos es “una colección compartida de datos lógicamente relacionados, junto con una descripción de estos datos, que están diseñados para satisfacer las necesidades de información de una organización”[20].

Por otra parte, el sistema gestor de base de datos (SGBD) es “un sistema software que permite a los usuarios definir, mantener, crear y controlar el acceso a la base de datos[20].

Necesitaremos estas herramientas ya que buscamos un sistema que guarda datos de manera persistente y poder recuperarlos y actualizarlos en cualquier momento.

Se analizan alternativas para bases de datos relacionales, basadas en el concepto matemático de relación, representado por tablas[20]; y también para bases de datos no relacionales documentales.

- Para bases de datos **relacionales**:

- **PostgreSQL** (<https://www.postgresql.org/>): es un SGBD de código abierto y centrado en la escalabilidad.
- **MySQL** (<https://www.mysql.com/>): es un SGDB de código abierto desarrollado por Oracle.
- **MariaDB** (<https://mariadb.org/>): es un SGDB de código abierto creado por desarrolladores de MySQL como respuesta a la adquisición por parte de Oracle, y con características muy parecidas.

- Para bases de datos **no relacionales**:

- **MongoDB** (<https://www.mongodb.com/>): SGDB de código disponible (no necesariamente de código abierto, por usar una licencia SSPL[21]).
- **Firestore** (<https://firebase.google.com/docs/firestore>): incluido en la suite de servicios “Firebase” de Google destinada a la gestión rápida y centralizada de aplicaciones.

Decisión

El modelo de base de datos que se propone es puramente relacional, por lo que usaremos una base de datos relacional.

Puesto que las diferencias son pequeñas entre las distintas opciones de base de datos relacional y la documentación de **Prisma** está mayormente basada en PostgreSQL, se escoge esta opción.

5.3.5. Servidor virtual

Entre los proveedores que ofrecen servicios gratuitos de computación en la nube se encuentran:

- **DigitalOcean** (<https://www.digitalocean.com/>) ofrece 200\$ gratuitos a estudiantes para desplegar servidores privados y servicios de almacenamiento. Además, dispone de servidores preconfigurados, como el que incluye docker preinstalado⁴.
- **Heroku** (<https://www.heroku.com/>) tiene la desventaja de que no permite desplegar directamente infraestructura configurada con docker compose. Ofrece 14\$ al mes gratuitos para estudiantes.
- **Microsoft Azure** (<https://azure.microsoft.com/>), **Amazon Web Services** (<https://aws.amazon.com/>) o **Google Cloud** (<https://cloud.google.com/>) son plataformas usadas por grandes empresas, con configuraciones muy completas y con gran variedad de servicios. Solo **Azure** ofrece un plan gratuito para estudiantes.

Decisión

Hemos probado a configurar el plan gratuito de **Microsoft Azure** para crear un servidor, pero hemos encontrado numerosos errores y un panel de configuración caótico y lleno de opciones que no necesitamos, por lo que optaremos por **DigitalOcean**.

5.3.6. Almacenamiento de archivos

Para guardar las partituras de los usuarios necesitaremos un servicio donde almacenarlas. Los servicios más extendidos para guardar archivos de usuarios se denominan de **almacenamiento de objetos**: “es una tecnología que almacena y administra datos en un formato no estructurado denominado objetos. [...] Los sistemas de almacenamiento de objetos en la nube distribuyen estos datos a través de varios dispositivos físicos, pero permiten a los usuarios acceder al contenido de forma eficiente desde un único repositorio de almacenamiento virtual”[22].

Existen varias alternativas de distintos proveedores como **Amazon S3**⁵, **DigitalOcean Spaces**⁶ o **Google Cloud Storage**⁷.

⁴<https://marketplace.digitalocean.com/apps/docker>

⁵<https://aws.amazon.com/s3>

⁶<https://www.digitalocean.com/products/spaces>

⁷<https://cloud.google.com/storage>

Decisión

Escogeremos la alternativa de **DigitalOcean** ya que utiliza la misma API que **Amazon S3** y nos permitirá tener la administración del servidor y del almacenamiento de archivos centralizada en un mismo panel de control además de utilizar los créditos gratuitos.

5.3.7. Documentación

Existen numerosas herramientas para generar páginas web de documentación.

- **GitBook** (<https://www.gitbook.com/>): añade estilos a los README.md de **GitHub**. No permite por tanto crear una *Landing Page* personalizada sino que solo genera páginas de documentación. Un ejemplo creado con **GitBook** es <https://zod.dev/>.
- **Docusaurus** (<https://docusaurus.io/>): se basa en la biblioteca para interfaces de usuario react, y permite crear múltiples páginas además de la documentación. Un ejemplo es <https://trpc.io/>.
- **VuePress** (<https://vuepress.vuejs.org/>): Muy parecido a **Docusaurus**, pero basado en la biblioteca vue en lugar de react. La documentación de **grammY** (<https://grammy.dev/>) está creada con esta biblioteca.

Decisión

Solo podremos desarrollar una *landing page* personalizada con **Docusaurus** o **VuePress**. Dado que el desarrollador tiene experiencia con react, escogeremos **Docusaurus** para disminuir el tiempo necesario de formación.

5.4. Modelo de la base de datos

Teniendo claros los requisitos del sistema, podemos analizar la información que necesitaremos representar en la base de datos.

Se propone el modelo relacional de la figura 5.1, con un total de 11 tablas. La tabla Session se queda aislada ya que será utilizada para guardar las sesiones de los usuarios, y la tabla debe tener un esquema específico para poder usarla automáticamente con el adaptador como se explicará en la sección 6.7.2.

Este modelo es el necesario para implementar todas las historias de usuario que se han extraído, incluidas las que se plantean como tareas futuras a la entrega de este trabajo.

El diagrama se ha realizado con la herramienta <https://dbdiagram.io/>.

5.5. Arquitectura de Mordente

Expliquemos la arquitectura del sistema en dos pasos:

5.5.1. Arquitectura entre puntos

Con la arquitectura entre puntos nos referimos a la visualización a alto nivel de las distintas partes físicas necesarias para que nuestra herramienta funcione.

En nuestro caso, existen tres partes claramente diferenciadas:

- **El dispositivo del usuario**, o también llamado cliente. Tiene instalada la aplicación de Telegram, y se comunica bidireccionalmente con los servidores de Telegram para enviar y recibir mensajes.
- **Los servidores de Telegram**. Actúan como intermediarios entre distintos usuarios, así como entre los usuarios y nuestro bot.
- **El bot**. Es un programa que se comunica con los servidores de Telegram para enviar y recibir mensajes, tal y como los clientes, aunque con ciertas diferencias de funcionamiento con los clientes. Este programa puede estar alojado en cualquier ordenador, y en nuestro caso estará alojado en un servidor privado virtual.

Esta arquitectura está representada en la figura 5.2.

Es importante entender que la única responsabilidad que nos corresponde a nosotros es la de desarrollar la última parte: el bot. El cliente y los servidores de Telegram son partes externas fuera de nuestra responsabilidad, lo cual es una de las ventajas de desarrollar un servicio de este tipo.

5.5.2. Arquitectura entre servicios

Para la funcionalidad que necesitamos, el bot debe estar compuesto por varios microservicios a su vez:

- **El propio bot:** es el microservicio donde implementaremos la comunicación con los servidores de Telegram para recibir mensajes, procesarlos y responderlos adecuadamente.
- El bot realiza peticiones a una **base de datos** donde almacenaremos de forma persistente los datos de los usuarios. Esto garantizará que no perdamos información cuando el bot se reinicie.
- Opcionalmente, podremos añadir un microservicio encargado de realizar una **copia de seguridad** periódica de la base de datos.

- Por otro lado tendremos un servicio de **almacenamiento de archivos** para guardar las copias de seguridad y otros archivos (en nuestro caso, las partituras).

En la figura 5.3 se puede visualizar la arquitectura entre los distintos servicios.

Localización de los servicios

Con respecto a dónde disponer los distintos servicios, existen al menos dos opciones: una es alojarlos en el propio servidor que gestionaremos nosotros y cuyos microservicios podremos levantar en un solo paso usando *Docker Compose*⁸, y otra es utilizar servicios auto-gestionados por plataformas externas. Hagamos una reflexión para cada uno de los servicios:

- El **bot** puede estar en un servidor propio o en un servicio gestionado, ambas opciones son válidas. La ventaja de tenerlo en el servidor propio será la mayor cercanía posible con la base de datos, resultando en una mayor velocidad de respuesta. La ventaja de utilizar un servicio gestionado es la mayor facilidad de escalado (mantener la capacidad de respuesta independientemente del número de usuarios), sin embargo esto no es algo prioritario en la primera fase de desarrollo.
- Casi todos los proveedores de servicios en la nube ofrecen **bases de datos** gestionadas⁹, aunque siempre es conveniente que el bot y la base de datos se encuentren físicamente cercanos para reducir la latencia de las frecuentes consultas. Es por ello y porque no se prevé una gran cantidad de información almacenada durante el periodo inicial que optaremos por alojar la base de datos como un contenedor dentro de nuestro servidor, al igual que el bot.
- El **almacenamiento de archivos** siempre se suele delegar a un servicio externo de almacenamiento de objetos gestionado¹⁰. La mayor ventaja de esta aproximación es la disponibilidad de una CDN (*Content Delivery Network*, o Red de Entrega de Contenidos)[23]. Gracias a esta red tendremos un conjunto de servidores distribuidos geográficamente que pueden entregar rápidamente el contenido. Además, usar un servicio fuera de nuestro servidor garantiza que podamos realizar copias de seguridad de la base de datos que no se pierdan si el servidor se rompe por cualquier causa.

⁸<https://docs.docker.com/compose/>

⁹Ejemplos: Digital Ocean Managed Databases (<https://www.digitalocean.com/products/managed-databases>), Heroku Postgres (<https://www.heroku.com/postgres>), Google Cloud Firestore (<https://cloud.google.com/firestore>)

¹⁰Ejemplos: Amazon S3 (<https://aws.amazon.com/es/s3/>), Digital Ocean Spaces (<https://www.digitalocean.com/products/spaces>) o Google Cloud Storage (<https://cloud.google.com/storage>)

Entorno de producción y de desarrollo

Es importante tener en cuenta también que esta configuración puede variar dependiendo de si nos encontramos en el entorno de desarrollo o de producción. En nuestro caso, el servicio de copias de seguridad no existirá en el entorno de desarrollo.

Además, tendremos dos réplicas (una de producción y otra de desarrollo) del bot, la base de datos y el almacenamiento de archivos para que las pruebas no afecten en ningún caso a los datos de los usuarios reales.

5.6. Seguridad

Se intentará maximizar la seguridad de la aplicación gracias a los siguientes puntos:

- Los contenedores de Docker utilizados en el servidor están completamente aislados de la red excepto en los puertos que configuremos manualmente. En nuestro caso, el bot podrá conectarse a la base de datos pero ninguno de los contenedores tendrá puertos expuestos al exterior que pudieran ser usados por atacantes.
- Gracias al punto anterior, los ataques de denegación de servicio (DDoS) no son un riesgo. Además el servidor tendrá un Firewall configurado.
- El uso de herramientas *open-source* como Prisma para hacer las consultas a la base de datos en lugar de realizarlas manualmente con código SQL disminuye al máximo la posibilidad de Inyecciones SQL.
- La inyección de código al servidor no es posible ya que no se ejecuta ningún código enviado por el cliente, y por otro lado el cliente es responsabilidad de Telegram.
- Se añadirán métodos que permitan detectar automáticamente las vulnerabilidades introducidas por dependencias externas. Como el repositorio estará alojado en GitHub, se propone el uso de *Dependabot*¹¹.

¹¹<https://docs.github.com/es/code-security/dependabot>

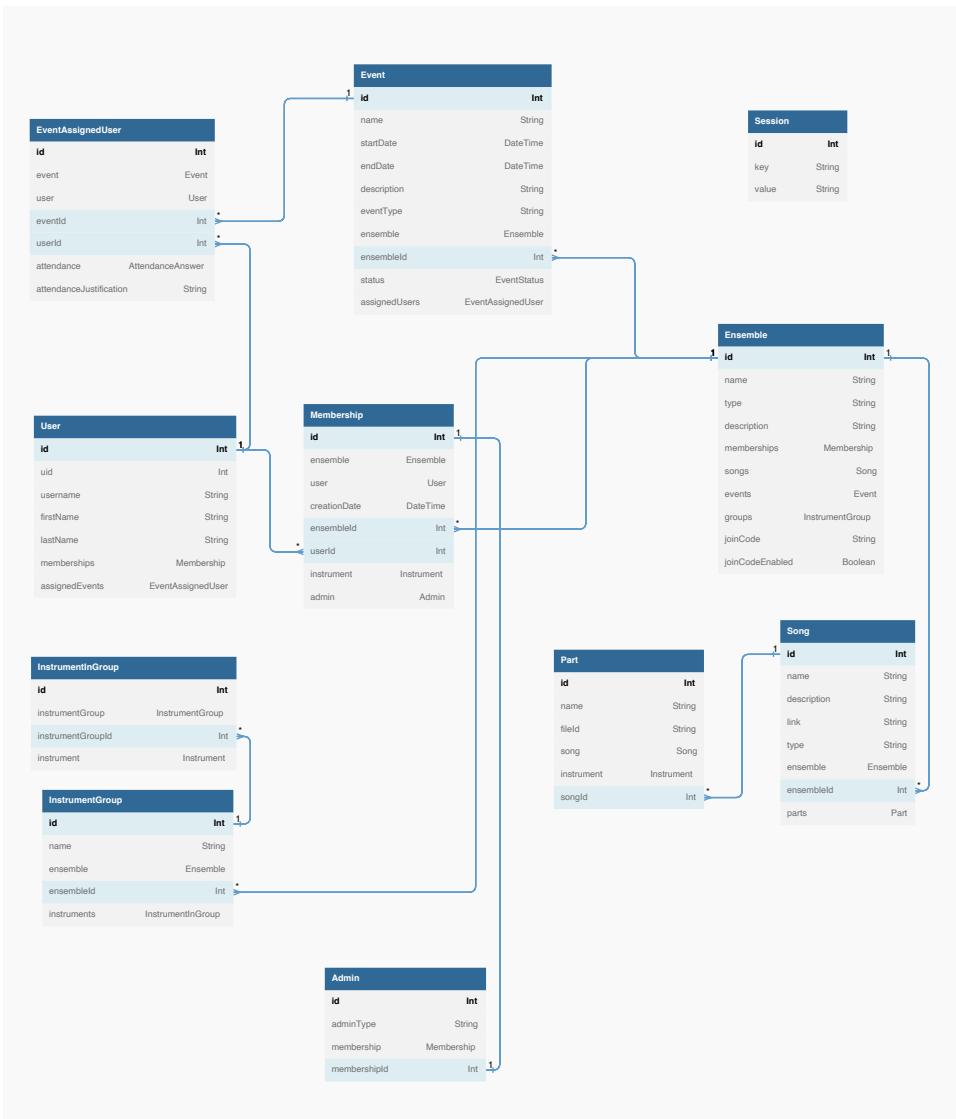


Figura 5.1: Modelo de la base de datos

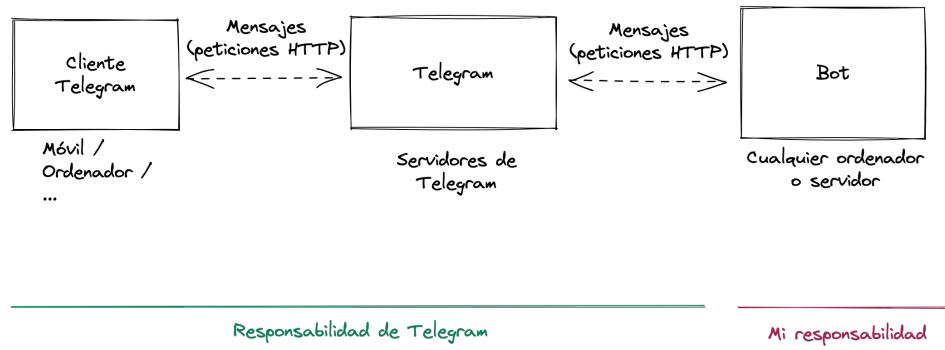


Figura 5.2: Arquitectura entre puntos

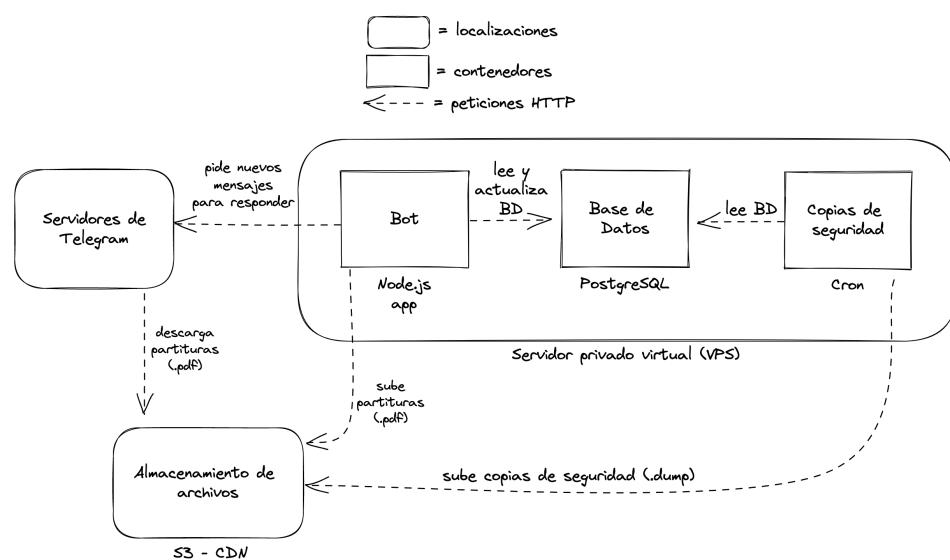


Figura 5.3: Arquitectura entre servicios

Capítulo 6

Implementación

En esta sección vamos a ahondar en ciertos detalles sobre **cómo se ha construido** este proyecto.

Las secciones están ordenadas de forma cronológica y a modo de relato, de forma que empezaremos explorando las funcionalidades que podemos aprovechar de las herramientas a usar y terminaremos creando la página web de promoción cuando la aplicación ya esté funcionando.

6.1. Exploración de funcionalidades

Comenzaremos la implementación del bot con un pequeño bot de prueba que nos permita descubrir cuál será el flujo de trabajo con las herramientas de programación que hemos escogido en la sección 5.3. Estos son los pasos que daremos con este primer bot:

1. Responder de manera básica básica a los mensajes recibidos, con un ¡Hola! Mensaje recibido.
2. Conectarse a una base de datos sencilla de una sola tabla, que guarda palabras.
3. Listar palabras y eliminar palabras en la base de datos de forma global.
4. Asignar las palabras a usuarios concretos para que solo puedan ver y eliminar las suyas.

6.1.1. Infraestructura de contenedores

Para comenzar a desarrollar, primero crearemos los archivos que configuren los contenedores de Docker. Estos archivos nos permitirán levantar el bot en cualquier dispositivo fácilmente, solo ejecutando el comando `docker compose up`. Tal y como hemos explicado en la sección 5.5.2, en el entorno de desarrollo tendremos solo dos contenedores:

- El contenedor `app` será el encargado de ejecutar el bot en `Node.js`.
- El contenedor `postgres` contendrá la base de datos de PostgreSQL. Haremos que este contenedor exponga el puerto 5432 al sistema para poder depurar la base de datos.

Tendremos que añadir también dos volúmenes¹:

- Uno que hará que el código que estamos desarrollando se encuentre actualice en tiempo real dentro del contenedor sin tener que reconstruir la imagen.
- El otro estará asociado a la base de datos de forma que los datos guardados se almacenen de forma persistente.

El archivo `docker-compose.yml` resultante con contenedores y volúmenes se puede consultar en GitHub².

6.1.2. Conexión a base de datos

El **ORM** (*Object-relational mapper*) `prisma` nos ayudará a conectar con la base de datos de manera segura y aprovechando las ventajas del tipado estático que proporciona **TypeScript**: convierte los registros de la base de datos en objetos del lenguaje de programación que estamos usando y viceversa.

La configuración de `prisma` se realiza a través del esquema de la base de datos en el archivo `schema.prisma`³. El esquema se define en un lenguaje propio en un nivel de abstracción superior sobre SQL y que permite usar el mismo modelo intercambiando diferentes proveedores (PostgreSQL, MySQL o incluso proveedores No-SQL como MongoDB).

En esta primera parte, el esquema solo contendrá una tabla `Word`, de modo que en `schema.prisma` solo tendremos que configurar esa tabla, indicar que el proveedor es PostgreSQL y la URL de conexión⁴.

Ejecutando el siguiente comando, `prisma` crea el código necesario para hacer todas las consultas a la base de datos, es decir, el **Cliente Prisma**:

```
prisma generate
```

Ejecutaremos este comando cada vez que hagamos cambios en el esquema para generar un nuevo cliente.

Para usarlo desde **TypeScript**, solo necesitamos crear un archivo `PrismaClient.ts` con el siguiente código:

¹<https://docs.docker.com/storage/volumes/>

²<https://github.com/daniharo/mi-banda-bot/blob/98a0662b/docker-compose.yml>

³<https://github.com/daniharo/mi-banda-bot/blob/main/prisma/schema.prisma>

⁴Archivo `schema.prisma`: <https://github.com/daniharo/mi-banda-bot/blob/main/prisma/schema.prisma>

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

export default prisma;
```

El objeto `prisma` contiene una clave para cada tabla de la base de datos, de manera que las consultas a la base de datos serían tan sencillas como:

```
// Consultar todas las palabras:
const words = await prisma.word.findAll();

// Eliminar palabras que contengan "palabra":
const deletedWord = await prisma.word.deleteMany({
    where: { word: { contains: "palabra" } }
});

// Modificar todas las palabras que comienzan por "foo":
const updatedWord = await prisma.word.updateMany({
    where: { word: { startsWith: "foo" } },
    data: { word: "nuevaPalabra" }
});
```

La mayor ventaja que tenemos es que `prisma` es totalmente *type-safe*: se obtiene un error directamente en el editor de código ante cualquier operación inválida. Por ejemplo, si intentamos crear un registro sin añadir un campo obligatorio, veremos esa línea subrayada de color rojo y **TypeScript** nos pedirá solucionarlo antes de poder ejecutar el programa.

6.1.3. Manejo de mensajes de Telegram

La lógica que nos permite implementar `grammY`, la biblioteca que hemos escogido en la sección 5.3.2 para comunicarnos con la API de Telegram, es la siguiente:

- Cada mensaje entrante pasa por una cadena de funciones llamadas **middleware**. A cada **middleware** le podemos dar una responsabilidad distinta:
 - **Modificar** el objeto `ctx` que va recorriendo todos los `middleware` y que contiene toda la información del mensaje entrante para añadir información adicional. En este caso, debe llamar a `next()` para que el mensaje siga recorriendo el resto de `middleware`.
 - **Manejar** el mensaje respondiendo, actualizando la base de datos, o con cualquier tarea que se necesite. En este caso no se llama a `next()` ya que el recorrido del mensaje terminaría ahí.

- El middleware se instala en el bot mediante métodos como `.use`, `.on` o `.command`:
 - `.use` nos permite instalar middleware que se ejecuta siempre.
 - `.on` nos permite usar filtros sencillos como `.on("message")`, en este caso para que ese middleware solo maneje mensajes y no otros eventos como pulsaciones en menús.
 - `.command` maneja mensajes que incluyan el comando especificado como primer argumento. Por ejemplo, `.command("/about")` se encargaría de responder al comando `/about`.

Para esta primera prueba, vamos a instalar solo estas funciones de middleware:

- `.command("start")` responderá al inicio del bot con un ‘Hey!’’.
- `.command("list")` responderá con la lista actual de palabras.
- `.command("add")` añadirá la palabra especificada en el mensaje.
- `.command("delete")` eliminará la palabra que se concrete.
- `.on("message")` estará al final y responderá a todos los mensajes que no sean manejados por el middleware anterior con un ‘Lo siento, no sé de qué me hablas’’.

El código de este primer bot de prueba está disponible en GitHub⁵.

6.2. Internacionalización

Si la aplicación crece, es probable que la terminen usando usuarios que hablan otro idioma, ya sea otra lengua cooficial de nuestro país (como el catalán o el gallego) o una lengua foránea.

Vamos a dejar el código preparado para que, si esto ocurre, solo tengamos que añadir las traducciones para el idioma correspondiente.

Para ello, añadimos el paquete `@grammyjs/fluent`⁶. Este paquete nos permitirá usar la sintaxis del **Proyecto Fluent**⁷ de Mozilla para crear las traducciones. Para añadirlo, solo tenemos que ejecutar siguiente comando, el mismo que utilizaremos con todas las dependencias que añadamos:

```
yarn add @grammyjs/fluent
```

⁵<https://github.com/daniharo/mi-banda-bot/blob/main/src/app.ts>

⁶<https://www.npmjs.com/package/@grammyjs/fluent>

⁷<https://projectfluent.org/>

Incluiremos en el archivo `src/locales/es.ftl`⁸ las cadenas de texto que necesitemos usar en el bot.

Para agilizar el desarrollo, dejaremos en línea dentro del código las cadenas de texto cortas y que no ocupen más de una línea. Convertir estas cadenas en traducciones es una tarea mecánica que se puede posponer al momento en el que tengamos un prototipo funcional con el que hacer las pruebas de usabilidad.

6.3. Separación MVC: Plantillas

El patrón arquitectónico **Modelo-Vista-Controlador** se usa comúnmente para desarrollar interfaces de usuario dividiendo la lógica en tres elementos conectados[24]. Esto permite hacer una mejor **división del código**.

Dado que no existe ningún *framework opinionado*⁹ para desarrollar bots de Telegram que siga este patrón (sí existen para desarrollo web, por ejemplo **Symphony**¹⁰ para PHP), vamos a estructurar el código de la siguiente forma:

- **Modelo:** En la ruta `src/models/` implementaremos toda la comunicación directa con la base de datos: consultas, eliminaciones y modificaciones, en un archivo distinto para cada modelo.
- **Vista:** Las vistas estarán compuestas por:
 - **Menús** en línea que aparecen debajo de un mensaje enviado por el bot, se encontrarán en la ruta `src/menus/`.
 - **Conversaciones** que mantiene el bot con un usuario, en la ruta `src/conversations/`.
 - **Plantillas** complejas de mensajes que debe enviar el bot sustituyendo ciertas variables o recorriendo un vector de elementos. Usaremos el motor de plantillas `Pug.js`¹¹ y guardaremos las plantillas en la ruta `src/templates/`.
- **Controlador:** Los manejadores de mensajes serán las funciones que actúen de controlador. Estos se pueden escribir a modo de **Composer** o de **Middleware**, por lo que tendremos las carpetas `/src/composers/` y `/src/middleware/`. El archivo `app.ts` se encargará de iniciar la cadena de llamadas a todos los controladores.

⁸<https://github.com/daniharo/mordente/blob/main/src/locales/es.ftl>

⁹Con *opinionado* nos referimos a aquel marco de trabajo en el que se manifiestan claramente qué técnicas y procesos se deben seguir para su uso: su documentación específica **cómo** proceder y no solo **qué** se puede hacer.

¹⁰<https://symfony.com/>

¹¹<https://pugjs.org/api/getting-started.html>

En la figura 6.1 se representa visualmente la separación de responsabilidades.

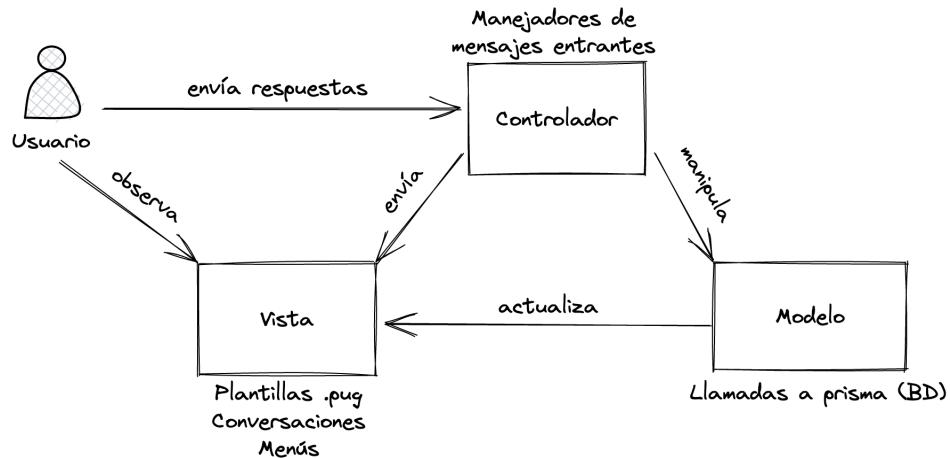


Figura 6.1: Separación Modelo-Vista-Controlador

Hasta ahora tenemos todos los elementos que se han explicado, excepto las plantillas. Tras instalar el motor de plantillas con `yarn add pug`, deberemos asegurarnos de que las plantillas se compilan cuando se inicia el bot¹².

6.4. Unirse a un grupo

Cuando el bot se inicia, presentará a los usuarios dos opciones rápidas: crear una nueva agrupación o unirse a una ya existente.

Dado que la creación de la agrupación requiere iniciar una conversación con el usuario, implementaremos primero la funcionalidad de unirse, que solo requerirá de un comando `/start` por parte del usuario acompañado de un código de invitación, que de momento será el ID de la agrupación.

Para implementar esta funcionalidad, vamos a usar un truco: por ahora crearemos las agrupaciones en la base de datos de forma manual, mediante la interfaz que nos proporciona `prisma studio`¹³.

Añadimos por tanto el menú `startMenu` que da la bienvenida a los usuarios y une al usuario a una agrupación si el comando viene acompañado de un código¹⁴. Esto es posible gracias a la característica **Deep Linking**¹⁵ de la **API para Bots de Telegram**.

¹²Commit: <https://github.com/daniharo/mordente/commit/107f7e50>

¹³<https://www.prisma.io/studio>

¹⁴Commit: <https://github.com/daniharo/mordente/commit/5ded357c>

¹⁵<https://core.telegram.org/bots/features#deep-linking>

6.5. Crear, mostrar y eliminar agrupaciones

Todas las funcionalidades del bot dependen del hecho de que existan agrupaciones creadas en la base de datos por administradores, a las cuales se podrán unir los miembros.

Por ello esta será la próxima tanda de historias de usuario a implementar.

Permitiremos crear una agrupación utilizando el comando `/create16` o desde el menú de inicio del bot, que se obtiene con el comando `/start17`.

Además, añadimos la vista (plantilla) necesaria para ver el detalle de una agrupación, `ensemble-detail.pug18`. Permitiremos también al usuario cancelar la conversación para crear la agrupación en cualquier momento, mediante un comando `/cancel19`.

6.6. Optimización de la autenticación en cada mensaje

En la práctica totalidad de los mensajes que recibe el bot necesitamos comprobar cuál es el ID de ese usuario en nuestra base de datos. Esto es así porque, en vistas a futuro, hemos decidido que el ID del usuario en Telegram no sea el ID del usuario en nuestra base de datos, de manera que nuestra aplicación no dependa totalmente de Telegram.

Hacer una consulta a la base de datos en todas las funciones que manejan mensajes es repetitivo e ineficiente, por lo que se ha decidido implementar un *middleware* encargado de:

- Consultar si tenemos el ID de ese usuario en la sesión²⁰ actual.
- Si no lo tenemos, comprobar si existe en la base de datos.
 - Si existe en la base de datos pero hace demasiado tiempo de la última actualización de nuestra base de datos, actualizamos y guardamos el ID en la sesión.
 - Si existe en la base de datos y está actualizado, solo guardamos el ID en la sesión.
 - Si no existe en la base de datos, lo añadimos (creamos una cuenta) guardando el ID obtenido en la sesión.
- Finalmente, ejecutamos `ctx.userId = ctx.session.userId` para que todos los manejadores de mensajes puedan acceder al ID fácilmente.

¹⁶Commit: <https://github.com/daniharo/mordente/commit/82fa3897>

¹⁷Commit: <https://github.com/daniharo/mordente/commit/8cf020f>

¹⁸Commit: <https://github.com/daniharo/mordente/commit/b65de583>

¹⁹Commit: <https://github.com/daniharo/mordente/commit/90ddedfb>

²⁰<https://grammy.dev/plugins/session.html>

Una vez está implementado, eliminamos todas las consultas a la base de datos en el código para sustituirlas por `ctx.userId`²¹.

6.7. Crear, mostrar y eliminar eventos

Una de las funcionalidades más centrales de este proyecto es la de gestionar **eventos**.

Teniendo la funcionalidad de crear, ver y eliminar agrupaciones implementada, la funcionalidad relacionada con los eventos solo requiere adaptar el código anterior a las características de los eventos. Necesitaremos una conversación para crear los eventos (`createEventConversation`²²), las plantillas correspondientes al detalle del evento (`event-detail.pug`) y a la lista de eventos (`events-summary.pug`) y los menús para la lista de eventos (`eventListMenu`) y el detalle de evento (`eventMenu`)²³.

6.7.1. Selector de fecha²⁴

Dado que uno de los atributos de cada evento es la fecha y hora de inicio y la fecha y hora de fin, debemos otorgar al usuario la posibilidad de responder al bot fácilmente cuando se les pide estos datos.

Una forma muy intuitiva de introducir una fecha es proporcionar al usuario un calendario donde pulsar para seleccionarla.

Implementar un calendario dentro de Telegram presenta numerosas complicaciones técnicas, sin embargo hemos conseguido implementarlo y publicarlo para que el resto de la comunidad pueda mejorar este código y usarlo en sus proyectos²⁵.

En la URL <https://mordente.es/video/calendario.mp4> se puede ver un vídeo corto mostrando el funcionamiento de este calendario.

6.7.2. Guardar sesión en la base de datos

Uno de los problemas que hemos detectado durante la implementación de esta funcionalidad es que hasta ahora la **sesión del usuario** se está guardando en la memoria principal del sistema. La **sesión del usuario** guarda información como:

²¹Commits: <https://github.com/daniharo/mordente/compare/be269acf...331168c6>

²²Commit: <https://github.com/daniharo/mordente/commit/65979e02>

²³Commits: <https://github.com/daniharo/mordente/compare/77b0fb44...d820211d>

²⁴Posteriormente durante el desarrollo del proyecto se ha detectado que este selector de fecha interfirió con una nueva versión de la dependencia `grammy-conversations`, la cual necesitábamos actualizar. Solucionar este problema en el selector de fecha demoraría demasiado el desarrollo, por lo que se ha optado por sustituirlo temporalmente por mensajes de texto del formato DD/MM/YYYY para la fecha y HH:MM para la hora.

²⁵<https://www.npmjs.com/package/grammy-calendar>

- ID del usuario que nos ha enviado un mensaje, por la implementación de la sección 6.6.
- Paso en el que se encuentra dentro de una conversación para crear o eliminar una agrupación, un evento...
- ID del elemento que se encuentra editando en un determinado momento.

En el caso de que se reinicie el bot momentáneamente, no queremos que el bot pierda todo el contexto que se encontraba manejando. Por ejemplo, mientras un usuario crea un evento, si ha llegado a la mitad de la conversación pero reiniciamos el bot, debe volver a empezar. Por esto, vamos a hacer uso de una de las posibilidades que nos ofrece **grammY**: guardar la sesión en la base de datos.

El *framework* **grammY** ofrece diversos **adaptadores** sesión-base de datos que permiten guardar la sesión en la base de datos de forma transparente para el desarrollador. Sin embargo, no hay ningún adaptador para el ORM que estamos utilizando, **prisma**, tal y como hemos encontrado en un *issue* de GitHub²⁶.

La solución pasa por contribuir a la comunidad del software libre con la implementación de este adaptador. Por ello, nos ponemos manos a la obra y, una vez el trabajo está hecho, hacemos la petición de integrar el adaptador en el repositorio de adaptadores²⁷, no sin encontrar dificultades durante el proceso.

Los mayores problemas en este paso han sido para hacer funcionar los tests, tal y como se puede ver en el *Pull Request*, aunque finalmente hemos conseguido hacerlos funcionar.

Tras publicar el adaptador, el creador de **grammY** ha compartido con la comunidad un agradecimiento al autor de este proyecto por esta contribución²⁸.

6.8. Configuración del *linter*

El motivo por el que estamos usando el lenguaje **TypeScript** en lugar de **JavaScript** en este proyecto es porque el **tipado estático** evita gran parte de los errores que se puedan dar al ejecutar el programa por despistes del programador, además de mejorar el autocompletado que proporciona el IDE.

Sin embargo, se puede dar un paso más en la dirección de evitar errores que se dan durante la ejecución: un *linter* es un programa que “revisa y

²⁶<https://github.com/grammyjs/storages/issues/80>

²⁷*Pull Request*: <https://github.com/grammyjs/storages/pull/108>

²⁸Mensaje en canal de Telegram: https://t.me/grammyjs_news/35

observa el código en busca de errores que le puedan afectar”[25]. Algunos de los errores que puede detectar son[25]:

- Código poco intuitivo o **difícil de mantener**.
- Uso de **malas prácticas**.
- **Estilos de código inconsistentes**.

Además, nos proporciona la posibilidad de solucionar automáticamente muchos de los errores que detecta.

El *linter* más conocido para **JavaScript** y **TypeScript** es **ESLint**²⁹. Por ello hemos decidido configurar **ESLint** en el proyecto³⁰ de manera que esté integrado con **TypeScript** y solucionar todos los problemas que nos ha detectado tras la configuración³¹.

6.9. Limitación de errores para el bot

Hemos detectado que cuando el programa lanza una excepción en cualquier punto, la ejecución termina por lo que el bot deja completamente de funcionar.

El *framework* usado para crear el bot, **grammY**, nos permite *vallar* los errores, lo que se conoce como **error boundary**. Mediante esta técnica, podemos asignar para cada tipo de mensaje que le pueda llegar al bot qué queremos que pase cuando se lance una excepción

En nuestro caso aplicaremos esta técnica en el nivel más alto de la aplicación, de manera que todos los errores pasen por este manejador de errores. Solo tenemos que añadir el código³²:

```
bot.catch(error => {
    // Qué queremos hacer con el error.
    // En nuestro caso, mostraremos el mensaje de error
    // en la consola.
    ...
    console.error('Error handling update ${update_id}:');
    ...
})
```

²⁹<https://eslint.org/>

³⁰Cambios en el código: <https://github.com/daniharo/mordente/compare/a1f07349...0e8f3aaa>

³¹Commit: <https://github.com/daniharo/mordente/commit/100b60b2>

³²Commit: <https://github.com/daniharo/mordente/commit/0299185c>

6.10. Mejora del flujo de trabajo para depurar

Durante el desarrollo, es muy frecuente encontrar problemas y errores cuyo origen resulta difícil de averiguar. Por ello habitualmente recurrimos a escribir una salida en la consola que indique el valor de ciertas variables cuando el bot recibe un mensaje. Sin embargo esta es una forma poco eficiente de depuración ya que nos proporciona información limitada.

Node.js proporciona una forma de depurar que expone en un puerto toda la información de la ejecución actual del programa para que, utilizando una herramienta externa (como IntelliJ[26] o Chrome[27]), podamos depurar de forma eficiente la ejecución del programa.

Para ello, modificamos el archivo `docker-compose.yml` convenientemente para exponer el puerto 9200 al sistema y configuramos node para que registre la información de depuración en ese puerto³³.

6.11. Respuestas de asistencia prevista

Una de las funcionalidades más interesantes de nuestro bot será la gestión de la asistencia prevista: los miembros avisan si podrán ir o no a los eventos y los administradores obtienen esta información.

Para ello, lo primero que haremos será permitir a los usuarios responder si asistirán o no a un evento, añadiendo los correspondientes botones al menú del detalle de un evento³⁴.

6.11.1. Pedir justificación

También es conveniente que los miembros opcionalmente puedan justificar su ausencia a un determinado evento, por lo cual si expresan que no asistirán, iniciaremos una conversación (`attendanceConversation`) para preguntar al miembro por qué no podrá asistir, acompañando la pregunta de un botón de Saltar dado que esta justificación será opcional³⁵.

6.11.2. Notificar administradores

Los administradores deben ser notificados inmediatamente cuando un miembro responde sobre su asistencia.

Tan pronto como el bot reciba la respuesta de un miembro, se comunicará con todos los administradores de la agrupación para informar de ello³⁶.

³³Commit: <https://github.com/daniharo/mordente/commit/5a64a851>

³⁴Commit: <https://github.com/daniharo/mordente/commit/9d205960>

³⁵Commit: <https://github.com/daniharo/mordente/commit/06efba49>

³⁶Commit: <https://github.com/daniharo/mordente/commit/fb7e33e4>

6.12. Asignación de miembros a eventos

Ahora haremos que los eventos puedan ser asignados a usuarios concretos.

Durante la creación de un evento, preguntaremos a un usuario si quiere asignarlo a todos los usuarios o si asignará manualmente más tarde³⁷. Haremos también que los usuarios solo puedan ver los eventos que tienen asignados³⁸.

Por último, implementaremos un menú paginado que permita a los administradores elegir exactamente qué miembros estarán asignados a cada evento³⁹. Para la paginación guardaremos en la sesión del usuario en qué página se encuentra en cada momento.

6.13. Recordatorio de eventos diarios

La historia de usuario relativa a los recordatorios diarios de eventos es la única que no dispara una petición al bot sino que dispara el bot por sí mismo cada cierto tiempo.

Es por ello que debemos configurar una tarea cron que se ejecute cada día para enviar los recordatorios. En esta tarea, se consultan a la base de datos los eventos que tienen lugar hoy y se envía un mensaje a cada usuario que tiene eventos hoy con la información de los eventos correspondientes⁴⁰.

6.14. Adición y eliminación de administradores

Para añadir y eliminar administradores, ofreceremos a los ya administradores, en el menú de membresía de los demás usuarios, la opción **Hacer admin** o **Quitar de admin**.

Para ello implementamos las funciones correspondientes al modelo y los botones correspondientes en el menú reseñado⁴¹.

6.15. Despliegue de producción

Nos encontramos en un punto donde faltan muy pocas historias de usuario por implementar, por lo cual va siendo conveniente que el bot esté disponible de forma continua en un entorno de producción.

³⁷Commit: <https://github.com/daniharo/mordente/commit/9e951c77>

³⁸Commit: <https://github.com/daniharo/mordente/commit/77ad84b1>

³⁹Commit: <https://github.com/daniharo/mordente/commit/b111f68e>

⁴⁰Commit: <https://github.com/daniharo/mordente/commit/33859f05>

⁴¹Commit: <https://github.com/daniharo/mordente/commit/60db0538>

6.15.1. Modificaciones previas en el código

Antes de proceder a configurar un servidor de producción con nuestro código, es necesario modificar ciertas partes del código.

PM2: evitando caídas

Aunque en la sección 6.9 hayamos implementado un método que capta los errores de forma que se responde al usuario con un error en lugar lanzar una excepción que pare la ejecución del bot, en el entorno producción será necesario dar un paso más.

PM2⁴² es un gestor de procesos para Node.js que, entre otras funcionalidades, permite configurar nuestro programa de Node.js como un demonio. De esta forma, si en algún momento el bot se cae por alguna excepción, se volverá a levantar automáticamente.

Es por ello que se ha decidido añadir **PM2** como dependencia e implementar su respectiva configuración⁴³.

Modificaciones en los contenedores de docker

En el entorno de desarrollo, hay varios contenedores funcionando que tienen puertos TCP expuestos hacia el exterior:

- La base de datos tiene expuesto el puerto 5432 para poder depurar en tiempo real la base de datos de manera gráfica con prismastudio⁴⁴.
- El bot expone el puerto 9200 para poder depurar la ejecución del bot⁴⁵.

En un entorno de producción queremos que los contenedores estén totalmente aislados. Para ello se han modificado los archivos docker-compose, de manera que ahora tenemos tres⁴⁶:

- docker-compose.yml: Es el archivo base que usarán todos los entornos.
- docker-compose.override.yml: Añade los puertos a exponer en el entorno de desarrollo.
- docker-compose.prod.yml: Configura el bot para el entorno de producción ajustando la variable de entorno NODE_ENV a "production".

⁴²<https://pm2.keymetrics.io/>

⁴³Commit: <https://github.com/daniharo/mordente/commit/0783b132>

⁴⁴<https://www.prisma.io/studio>

⁴⁵<https://nodejs.org/en/docs/guides/debugging-getting-started/>

⁴⁶Commit: <https://github.com/daniharo/mordente/commit/8f5739ae>

Migraciones de la base de datos

Hasta ahora las migraciones de la base de datos⁴⁷ se realizaban de forma automática cada vez que se encendía el bot con dockercomposeup.

A partir de ahora, querremos que las migraciones se apliquen de forma manual cuando se despliegue código a producción que incluya cambios en el esquema. Para ello, añadimos como *scripts* los comandos necesarios para realizar las migraciones⁴⁸.

Compilación del código TypeScript a JavaScript

El código que hemos escrito en el lenguaje **TypeScript** debe ser **transpilado**⁴⁹ al lenguaje **JavaScript** para que Node.js pueda leerlo.

Idealmente debemos efectuar esta transpilación mientras construimos la imagen de docker, mientras actualmente lo estamos haciendo al iniciar la ejecución del bot. Por ello realizaremos los cambios oportunos en el código para corregirlo⁵⁰.

6.15.2. Creación de servidor virtual

Ya tenemos el código listo para desplegarlo en un servidor de producción que esté encendido las 24 horas del día.

Tal y como hemos decidido durante el **Diseño Técnico** en la sección 5.3.5, procedemos a crear un servidor virtual en la plataforma de **DigitalOcean**. Para ello empezamos registrándonos en su página web⁵¹ y creando un nuevo proyecto llamado **mordente**.

Los pasos para crear el servidor han sido los siguientes:

1. Pulsar **Create** y **Droplets**. Accederemos a la pantalla de creación de Droplets.
2. Seleccionamos la pestaña **Marketplace**, donde podremos encontrar múltiples plantillas preconfiguradas para necesidades comunes.
3. Buscamos Docker dentro del **Marketplace**, y seleccionamos el resultado principal.
4. Ajustamos la configuración para usar el plan más económico y ubicar el servidor en Frankfurt, la región con menor latencia desde Granada.

⁴⁷Las **migraciones** ajustan la base de datos a cambios en el esquema SQL.

⁴⁸Commit: <https://github.com/daniharo/mordente/commit/98f3186d>

⁴⁹Transpilar código fuente significa transformarlo desde un lenguaje de programación a otro con un nivel similar de abstracción[28].

⁵⁰Commit: <https://github.com/daniharo/mordente/commit/3be387e6>

⁵¹<https://cloud.digitalocean.com>

5. Añadimos la clave pública SSH de nuestro equipo para que la autenticación solo pueda realizarse mediante esta.

Una vez tenemos el servidor creado, los pasos a ejecutar en el servidor para encender el bot son:

```
# Generar clave privada y pública
ssh-keygen

# Abrir la clave pública para añadirla en GitHub
# (https://github.com/settings/keys)
cat .ssh/id_ed25519.pub

# Crear carpeta para repositorios y abrirla
mkdir repos && cd repos

# Clonar el repositorio de GitHub
git clone git@github.com:daniharo/mordente.git

# Ejecutar el bot
docker compose -f docker-compose.yml -f docker-compose.prod.yml up

# Ejecutar las migraciones de la base de datos
docker compose exec app yarn run migrate:prod
```

Tras ejecutar estos pasos, ya tenemos el bot funcionando en el entorno de producción.

6.15.3. Solucionando el uso anormal de la CPU

Tras una de las modificaciones en el código, se ha comprobado que el la carga mínima de la CPU es constantemente entre el 16 y 17 %. Aunque no sea una carga muy alta, es un valor que no tiene sentido ya que la carga mientras no hay peticiones de los usuarios debe ser muy próxima a 0.

Tras investigar el problema, se ha descubierto que tiene origen en una comprobación de estado (HEALTHCHECK) de la base de datos que se estaba haciendo de forma periódica (cada 0.5 segundos) cuando en realidad solo queríamos que se ejecutara una sola vez al encender el bot. Sin embargo, actualmente **Docker** no permite configurar esta comprobación de manera que se ejecute una vez inmediatamente al ejecutar el bot y después cada minuto, como se ha comprobado por un *issue* abierto en **GitHub**⁵².

⁵²<https://github.com/moby/moby/issues/33410>

Por esto se ha decidido eliminar la comprobación completamente hasta que los cambios que el equipo de Docker ya tiene preparados⁵³ sean publicados.

En la figura 6.2 se puede observar cómo el uso de la CPU era cercano a 0 antes de realizar el cambio en el código, aumentó al 16 % cuando hicimos incluirnos el `HEALTHCHECK` y volvió a normalizarse cuando revertimos el cambio.

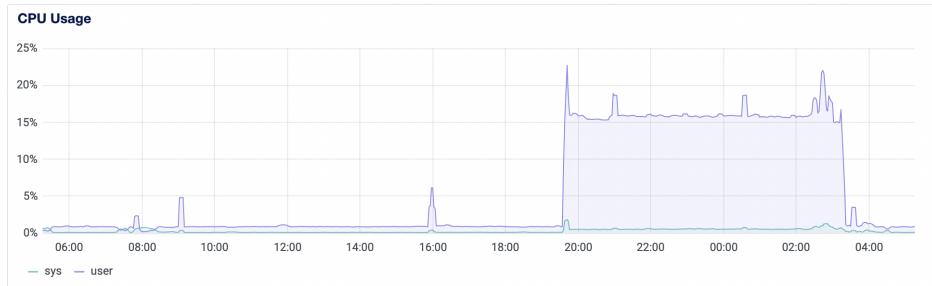


Figura 6.2: Gráfico de carga de la CPU

6.16. Obras

La última **épica** de historias de usuario que nos queda por comenzar a implementar es la relacionada con las **obras**: creación, visualización y eliminación. Sin embargo, hay una historia de usuario especial: se requiere gestionar archivos PDF de los usuarios, las partituras.

6.16.1. Almacenando partituras: creación del *bucket S3*

Con respecto al almacenamiento de las partituras, vamos a usar una solución estándar en la industria como son los *buckets S3*. Un servicio de este tipo nos permite almacenar gran cantidad de ficheros de forma simple y referenciados por una clave: esto se conoce como **almacenamiento de objetos**.

Como estamos usando el proveedor **Digital Ocean** que nos proporciona crédito gratuito, crearemos el *bucket* en su propio servicio de almacenamiento de objetos, llamado **Digital Ocean Spaces**. Este proveedor llama a los *buckets espacios*, pero al ser totalmente equivalentes y compartir la misma API, en este trabajo los denominaremos con el nombre estandarizado de **bucket**.

El proceso de creación del *bucket* para este proveedor es muy sencillo:

1. Accedemos al panel de administración de **Digital Ocean**.

⁵³<https://github.com/moby/moby/pull/40894>

2. Hacemos clic en **Create** y en **Spaces**.
3. Dejamos la configuración por defecto ya que es adecuada a nuestro proyecto.
4. Recibimos la confirmación de que el *bucket* ya ha sido creado.

6.16.2. Intermediando entre el chat y Digital Ocean Spaces

Una vez que tenemos el *bucket* creado, solo necesitamos implementar la intermediación entre los usuarios y el *bucket* para que puedan recibir y enviar las partituras dentro del mismo chat. Para ello, añadimos los paquetes `@aws-sdk/client-s3` y `@aws-sdk/s3-request-presigner` como dependencia, creados por **Amazon Web Services** y totalmente compatibles con cualquier servicio de almacenamiento de objetos S3 como el que usamos nosotros, **Digital Ocean Spaces**. También añadimos la dependencia `@grammyjs/files` que nos ayudará a manejar archivos que el bot envía o recibe.

Pidiendo la partitura

Para crear la obra en nuestra base de datos, implementamos la conversación `createSongConversation` que, tras preguntar por el nombre, pedirá al usuario que envíe el archivo de la partitura.

Recibiendo partituras

Cuando el bot recibe la partitura que ha enviado el usuario, la descarga en una ruta temporal⁵⁴ y conecta con el *bucket* para enviarle el archivo a la ruta `<id_agrupaciÃasn>/<id_obra>/<nombre_archivo>.pdf` con la función `uploadFile`. Esta ruta nos permitirá asegurar que podemos mantener los nombres de los archivos sin que puedan colisionar entre sí.

Enviando partituras

Por defecto, todos los objetos que tenemos almacenados tienen acceso privado, es decir, solo el dueño del *bucket* puede acceder a ellos. Por tanto, cuando un usuario pide una partitura, primero debemos generar una URL que está firmada por el dueño del *bucket* y que permite el acceso durante un tiempo especificado a otros usuarios⁵⁵. Esta URL temporal se enviará a la API de Telegram para enviar el archivo dentro del chat, mediante el método `replyWithDocument`⁵⁶.

⁵⁴<https://grammy.dev/guide/files.html#receiving-files>

⁵⁵<https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html>

⁵⁶<https://grammy.dev/guide/files.html#sending-files>

Variable de entorno	Significado
S3_ENDPOINT	Dominio en el que se encuentra el <i>bucket</i>
S3_REGION	Región en la que se aloja el <i>bucket</i>
S3_BUCKET	Nombre del <i>bucket</i> que hemos creado
S3_KEY	Clave de acceso al <i>bucket</i>
S3_SECRET	Clave secreta de acceso al <i>bucket</i>

Tabla 6.1: Variables de entorno necesarias para configurar el *bucket* S3.

Los cambios en el código que implementan la creación de partituras y la comunicación con el *bucket* están disponibles en [GitHub](#)⁵⁷.

6.16.3. Lógica común con agrupaciones y eventos

Ya se pueden crear obras, por lo que ahora necesitamos implementar la plantillas necesarias para visualizar la lista de obras (`song-list.pug`), el menú de la lista de obras (`songListMenu`) y el menú para el detalle de una obra (`songMenu`)⁵⁸. No se ha implementado una plantilla para el detalle de una obra ya que solo vamos a utilizar su nombre, por lo que una cadena de texto como respuesta es suficiente.

La implementación de estas funcionalidades se ha realizado siguiendo los pasos que se dieron tanto con las agrupaciones como con los eventos.

6.16.4. Parametrizando los valores del *bucket*

Para que el código se pueda usar con cualquier proveedor de servicios de almacenamiento de objetos, es necesario hacer que los datos de conexión al proveedor sean variables fáciles de cambiar.

Por ello vamos a añadir estos datos de conexión a las variables de entorno. En concreto, los datos que se necesita obtener del proveedor están explicados en la tabla 6.1.

Si en el futuro se optara por otro proveedor, simplemente tendríamos que cambiar estas variables de entorno.

6.17. Copias de seguridad de la base de datos

Todo sistema que pretenda ser **fiable** para los usuarios debe implementar un sistema de modo que si los datos se corrompen accidentalmente, la pérdida de datos esté controlada y no sea ilimitada: por ejemplo, que se pueda recuperar una versión anterior de una fecha concreta.

⁵⁷Commit: <https://github.com/daniharo/mordente/compare/1f448ef1...782e57b7>

⁵⁸Commit: <https://github.com/daniharo/mordente/commit/da63c3b0>

Variable de entorno	Significado
SCHEDULE	Cada cuánto tiempo se realizará la copia
BACKUP_KEEP_DAYS	Cuántos días queremos que se mantengan las copias de seguridad
S3_PREFIX	Prefijo para guardar las copias en el <i>bucket</i>

Tabla 6.2: Variables de entorno necesarias para configurar las copias de seguridad.

Esto lo podemos solucionar fácilmente añadiendo un nuevo servicio a la infraestructura montada en docker compose.

Haciendo una búsqueda, encontramos que la imagen de docker eeshugerman/postgres-backup-s3 tiene lo que necesitamos: realiza una copia de seguridad en un *bucket* de S3 (recordemos que hemos creado uno en la sección 6.16.1) con la configuración dada.

Las variables de configuración que le tenemos que proporcionar a este servicio son todas las relacionadas con el *bucket* S3, explicadas en la tabla 6.1, las relacionadas con el acceso a la base de datos, y las relativas a la copia de seguridad que se detallan en la tabla 6.2.

Tras añadirlo⁵⁹ comprobamos que siempre se mantienen 8 días de copias de seguridad, tal y como se puede comprobar en las figuras 6.3 y 6.4. Es por esto que comprobamos que a la variable de entorno BACKUP_KEEP_DAYS realmente hay que sumarle 1 ya que especifica cuántas copias **anteriores** a la que se está creando en ese momento queremos mantener.

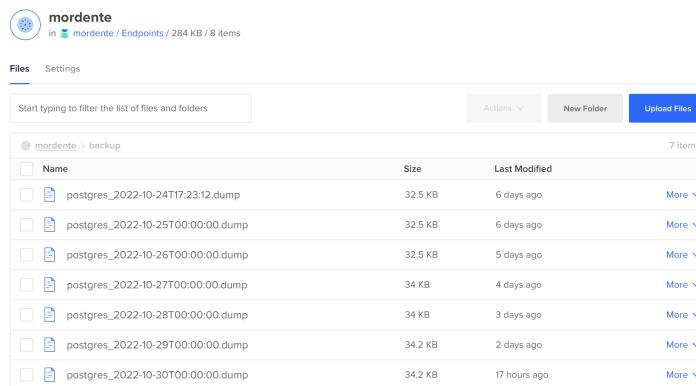


Figura 6.3: Archivos en el *bucket* S3 antes de superar el máximo de días configurado

⁵⁹<https://github.com/daniharo/mordente/compare/c4387aa8...2940907d>

6.18. Despliegue Continuo (CD): Automatizando el despliegue del bot a producción

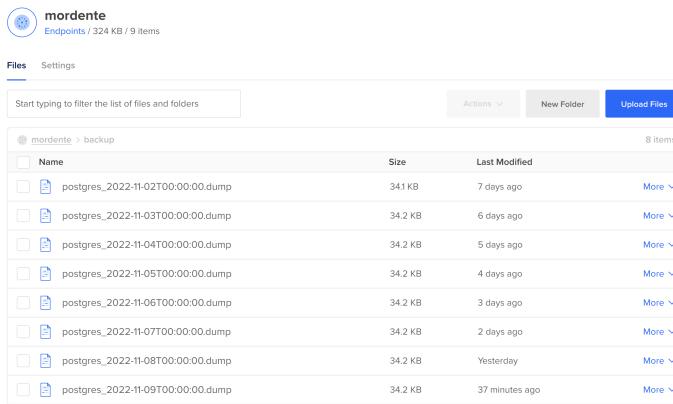


Figura 6.4: Archivos en el *bucket S3* tras superar el máximo de días configurado

6.18. Despliegue Continuo (CD): Automatizando el despliegue del bot a producción

Cuando cambiamos el código del proyecto, para actualizar el código que tenemos subido en el servidor de producción tenemos que seguir los siguientes pasos:

1. Acceder mediante ssh al servidor.
2. Actualizar el código con `git pull`.
3. Reconstruir la imagen de docker usando el comando `docker compose up -build`.

Dado que estos pasos se van a seguir de forma repetitiva cada vez que

Es aquí cuando entra en juego el **Despliegue Continuo (CD)**: es una técnica que consiste en desplegar automáticamente la aplicación a producción cuando se realizan cambios (*commits*) en el código[29].

GitHub permite emplear esta técnica fácilmente mediante el uso de las llamadas **Github Actions**⁶⁰: mediante el añadido de ciertos archivos especiales al repositorio, **GitHub** ejecuta las acciones especificadas en los supuestos que configuremos.

En nuestro caso, queremos que cuando haya nuevo código en *GitHub* para la rama `main` se ejecuten automáticamente los pasos reseñados en el comienzo de esta sección. Para ello, añadimos un archivo en la ruta `.github/workflows/deploy.yml`, en el cual especificamos cuándo queremos que se

⁶⁰<https://github.com/features/actions>

ejecute (en los `git push a main`), qué pasos hay que seguir, además de otras configuraciones que se pueden ver en el *commit* correspondiente⁶¹.

A partir de este momento, tal y como se puede ver en la figura 6.5, a la derecha de cada *commit* en GitHub aparece un símbolo a modo de marca de verificación en color verde, indicando que el despliegue se ha completado sin problemas. Al hacer clic aparece un botón de **Detalles** que nos permite consultar cómo se ha llevado a cabo el despliegue⁶².

Los datos para el acceso mediante ssh al servidor se almacenan cifrados como `Environment secrets`⁶³, de modo que en ningún momento nadie más que el creador del proyecto tiene acceso a ellos. Además, solo los usuarios con permisos de escritura en la rama `main` pueden disparar un nuevo despliegue.

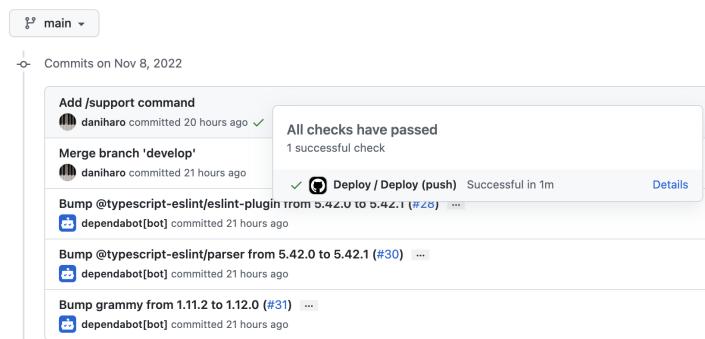


Figura 6.5: Marca de verificación del Despliegue Continuo

Más tarde se ha implementado una configuración para que al modificar ciertos archivos como el `README.md` no se provoque un despliegue⁶⁴.

6.19. Configurando la detección automática de vulnerabilidades

Se ha detectado que una tarea frecuente durante el desarrollo de este proyecto ha sido revisar periódicamente de forma manual si alguna de las dependencias⁶⁵ usadas ha sido actualizada por haberse detectado alguna vulnerabilidad o por incorporar nuevas funcionalidades.

⁶¹<https://github.com/daniharo/mordente/commit/8965e0e8>

⁶²Ejemplo: <https://github.com/daniharo/mordente/actions/runs/3416143998/jobs/5685987409>

⁶³<https://docs.github.com/en/actions/security-guides/encrypted-secrets>

⁶⁴<https://github.com/daniharo/mordente/commit/6ade5d35>

⁶⁵Las dependencias del bot se especifican en el archivo `package.json`: <https://github.com/daniharo/mordente/blob/main/package.json>

Es por ello que se ha configurado una herramienta automática que comprueba diariamente las dependencias y crea un *Pull Request* para cada actualización. Esta herramienta la proporciona *GitHub*, y su nombre es *Dependabot*.

Para configurarla se ha añadido un archivo en la ruta `/ .github/dependabot.yml`⁶⁶ especificando el intervalo entre detecciones (en nuestro caso cada día o `daily`), la ruta donde se encuentra el `package.json` con las dependencias y en qué rama queremos que se hagan los *Pull Request* (`develop` ya que `main` es la rama de producción).

Una vez configurado, comprobamos que cuando hay actualizaciones disponibles obtenemos las notificaciones correspondientes indicándolo⁶⁷.

6.20. La función de editar

La última funcionalidad que falta para que el bot sea usable es la de **editar** elementos, ya sean agrupaciones, eventos u obras.

Dado que el tiempo restante para terminar el proyecto es limitado, se ha decidido implementar solo la edición de agrupaciones de forma que la edición de eventos y obras estaría preparada, solo pendiente de readaptar el código ya implementado.

El flujo del usuario será:

1. Abrir el detalle de la agrupación.
2. Pulsar el botón de “Editar” debajo del detalle, que solo aparecerá si es usuario administrador.
3. Seleccionar en el menú qué atributo quiere editar.
4. Responder al bot con el nuevo valor.
5. Finalmente el usuario recibe una confirmación de que la agrupación ha sido modificada con éxito.

Para ello, implementamos el menú para seleccionar qué atributo vamos a editar (`editEnsembleMenu`) y una nueva conversación para cada atributo que se puede editar: `editEnsembleNameConversation`, `editEnsembleDescriptionConversation`...

Los cambios en el código se pueden ver en el repositorio⁶⁸.

⁶⁶<https://github.com/daniharo/mordente/commit/8754135>

⁶⁷Ejemplo: <https://github.com/daniharo/mordente/pull/25>

⁶⁸<https://github.com/daniharo/mordente/commit/3e630b1>

6.21. Página web: mordente.es

Una vez tenemos la aplicación funcionando y lista para poder hacer pruebas con usuarios, es conveniente desarrollar un sitio web donde explicar qué es Mordente, qué hace y cómo funciona. En esta sección se va a describir el proceso que se ha seguido para conseguirlo.

6.21.1. Creación del proyecto

Como venimos haciendo a lo largo del proyecto, hemos creado un repositorio público vacío en GitHub con el nombre de `mordente-docs`⁶⁹. Seguidamente se ha clonado en el equipo de desarrollo usando el siguiente comando de git:

```
git clone git@github.com:daniharo/mordente-docs.git
```

En la sección 5.3.7 se ha decidido usar la herramienta `docusaurus` para desarrollar la página web, por lo tanto seguiremos sus instrucciones⁷⁰ para iniciar el desarrollo. Tras seguir las instrucciones, ya tenemos la estructura preparada para crear el contenido⁷¹.

6.21.2. Creación de contenido

La estructura de código que nos proporciona `docusaurus` nos permite configurar la estructura, apariencia y contenido de la página web partiendo de una plantilla *responsive* (adaptada a todos los tamaños de pantalla) y accesible.

Aprovecharemos las posibilidades de configuración que nos ofrece `docusaurus` para cambiar los colores a nuestra paleta, añadir el logotipo y el contenido personalizado.

Algunas de las herramientas usadas durante este paso han sido:

- `undraw`⁷²: es una extensa biblioteca de ilustraciones SVG a las cuales se les puede personalizar fácilmente el color, y que representan distintas situaciones o ideas.
- `favicon-generator`⁷³: el `favicon` es el ícono que aparece a la izquierda del título de la página en el navegador. El archivo debe ubicarse en la raíz del directorio de la página y llamarse `favicon.ico`. Esta herramienta nos permite convertir cualquier imagen a un `favicon` con la medida adecuada.

⁶⁹<https://github.com/daniharo/mordente-docs>

⁷⁰<https://docusaurus.io/docs/installation>

⁷¹<https://github.com/daniharo/mordente-docs/commit/1c25316>

⁷²<https://undraw.co/>

⁷³<https://www.favicon-generator.org/>

- Se han usado algunos conocimientos previos de `react`⁷⁴, ya que `docusaurus` está basado en esta biblioteca de interfaces de usuario.

En GitHub se pueden observar los cambios exactos realizados en el código para añadir el contenido de la página web⁷⁵.

6.21.3. Alojar en servidor

Para que la página web esté accesible públicamente, es necesario alojarla en un servidor público. Existen múltiples plataformas que permiten alojar fácilmente contenido estático⁷⁶ como el que genera `docusaurus`, pero las más interesantes permiten actualizar automáticamente el contenido del servidor cada vez que hay un `commit` en la rama principal del repositorio. Algunas de las más conocidas son:

- **Vercel**⁷⁷: Es la empresa que desarrolla el *framework* `Next.js`, una de las herramientas más usadas⁷⁸ por su flexibilidad para crear páginas web requieren de lógica en el servidor o no. Su servicio de alojamiento de páginas web destaca por su facilidad de uso, con una configuración automática e inmediata.
- **Netlify**⁷⁹: Principal competidor de Vercel, ofrece más posibilidades de configuración pero un peor rendimiento.
- **Github Pages**⁸⁰: Es un servicio prestado por la forja de repositorios **GitHub**. Requiere de una configuración más complicada para automatizar la compilación del código⁸¹.
- **Cloudflare Pages**⁸²: Nuevo servicio equivalente a Netlify o Vercel, con una configuración igual de sencilla. Es provisto por Cloudflare, una compañía especializada en prestar servicios de *caché de contenido* y de seguridad.

Se optará por **Vercel** por la familiaridad del desarrollador de este proyecto con este proveedor por proyectos anteriores y por su facilidad de uso.

Para alojar nuestra página en **Vercel**, simplemente seguimos la documentación⁸³, tarea que no nos lleva más de unos minutos. Una vez hemos

⁷⁴<https://reactjs.org/>

⁷⁵<https://github.com/daniharo/mordente-docs/compare/fd28973...e8980b0>

⁷⁶Con *contenido estático* nos referimos a archivos HTML, CSS, imágenes, etc. que se envían directamente al cliente sin necesidad de que haya una lógica en el servidor para calcularlos.

⁷⁷<https://vercel.com/>

⁷⁸<https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe>

⁷⁹<https://www.netlify.com/>

⁸⁰<https://pages.github.com/>

⁸¹<https://docusaurus.io/docs/deployment#deploying-to-github-pages>

⁸²<https://pages.cloudflare.com/>

⁸³<https://docusaurus.io/docs/deployment#deploying-to-vercel>

Ruta origen	Redirige permanentemente a
/repo	Código del bot
/source	Código de la página web
/source/docs	Código L ^A T _E X de la memoria
/source/memoria	Memoria en formato PDF
/memoria	Bot en Telegram
/try	

Tabla 6.3: Redirecciones en mordente.es

terminado, tenemos la página web en una dirección asignada automáticamente: <https://mordente.vercel.app>.

6.21.4. Redirecciones

Para facilitar el acceso al contenido de este trabajo, se va a configurar el servidor de manera que al acceder a ciertas rutas se abra una determinada página de un dominio externo. Lo haremos siguiendo las instrucciones de Vercel para ello⁸⁴. Las rutas que redireccionaremos están descritas en la tabla 6.3.

6.21.5. Asignación de dominio

Registrar un dominio personalizado nos permitirá acceder a la página con una dirección sencilla y fácil de recordar.

Las empresas encargadas de registrar un dominio son llamadas **regidores de dominios**. Tras comparar precios entre diversos registradores para dominios .es, en nuestro caso se opta por **IONOS**⁸⁵, que ofrece el dominio mordente.es por 1,21€ el primer año, la mejor oferta entre las encontradas.

Tras hacer el proceso de compra, seguimos las instrucciones⁸⁶ de **Vercel** para asignar el dominio al proyecto. Esperamos un tiempo aproximado de una hora mientras se propagan los nuevos registros DNS, tras el cual Vercel genera automáticamente el certificado TLS para nuestro dominio⁸⁷ y podemos acceder sin problemas a <https://mordente.es>.

El último paso es modificar la configuración de docusaurus para ajustar correctamente el nuevo dominio principal⁸⁸.

⁸⁴<https://vercel.com/docs/project-configuration#project-configuration/redirects>

⁸⁵<https://www.ionos.es/dominios/>

⁸⁶<https://vercel.com/docs/concepts/projects/domains/add-a-domain>

⁸⁷<https://vercel.com/blog/automatic-ssl-with-vercel-lets-encrypt>

⁸⁸<https://github.com/daniharo/mordente-docs/commit/c2e2f8e>

6.21.6. Probando otros proveedores

Durante la realización de un proyecto personal paralelo⁸⁹ se ha comprobado cómo **Cloudflare Pages**, siendo un proveedor muy parecido a **Vercel**, proporciona un menor tiempo de respuesta (aproximadamente la mitad) a la hora de alojar un servidor Next.js⁹⁰ que incluye *Rutas API Edge*⁹¹.

Es por esto que se ha querido comprobar si para sitios puramente estáticos como el que hemos creado en esta sección, **Cloudflare Pages** es capaz también de dar un mayor rendimiento.

Para ello, se han seguido las instrucciones correspondientes, primero para alojar el contenido⁹² y después para asignar el dominio que creamos en la sección 6.21.5⁹³. Por último se han añadido las redirecciones en el formato requerido por Cloudflare⁹⁴ (y que coincide con **Netlify**⁹⁵).

Conclusiones del cambio de proveedor

Tras seguir las instrucciones, hemos podido comprobar que aunque por culpa de una configuración por defecto incorrecta⁹⁶ parecía que habíamos empeorado el rendimiento, tras solucionar la configuración sí que se ha mejorado el tiempo de respuesta con respecto a Vercel, aunque en una proporción despreciable (unos 120ms en Cloudflare frente a 140ms en Vercel).

Sin embargo, estas mediciones se hicieron por la noche, mientras la mayoría mediciones hechas durante el día indican que **Cloudflare** necesita un mayor tiempo para entregar la web: unos 250ms, frente a 140ms en Vercel.

Se ha detectado que el motivo es que, para reducir la carga de algunos centros de datos, algunas peticiones a páginas que están en el plan gratuito se enrutan a centros de datos lejanos⁹⁷. En nuestro caso, hemos visto peticiones dirigidas a centros de datos en India o Estados Unidos, mientras que **Vercel** enruta todas las peticiones a Frankfurt (Alemania).

Es por esto que se ha decidido revertir el cambio de proveedor.

⁸⁹Aplicación usando ingeniería inversa para consultar en tiempo real los tiempos de paso del Metropolitano de Granada: <https://metrogranada.pages.dev>. Código en <https://github.com/daniharo/Metro-Granada-Webapp>.

⁹⁰<https://nextjs.org/>

⁹¹<https://nextjs.org/docs/api-routes/edge-api-routes>

⁹²<https://developers.cloudflare.com/pages/framework-guides/deploy-a-docusaurus-site/>

⁹³<https://developers.cloudflare.com/pages/platform/custom-domains/>

⁹⁴<https://developers.cloudflare.com/pages/platform/redirects/>

⁹⁵<https://docs.netlify.com/routing/redirects/>

⁹⁶<https://stackoverflow.com/a/74341851/12210701>

⁹⁷<https://community.cloudflare.com/t/peering-why-dont-i-reach-the-closest-datacenter-to-me/76479>

6.22. Creación de dirección de email

El proveedor de DNS utilizado para mordente.es nos permite obtener una dirección de correo electrónico gratuita con 2GB de almacenamiento. Por tanto, se ha creado una dirección de soporte (`soporte@mordente.es`) para los usuarios que puedan tener alguna consulta.

6.23. Registro remoto de errores

Por último, vamos a configurar el servicio **Sentry** para poder obtener en tiempo real registros de los errores que se estén produciendo en la aplicación.

Seguimos los siguientes pasos:

1. Creamos una cuenta gratuita en <https://sentry.io/signup/>.
2. Creamos un nuevo proyecto para la plataforma Node.js (figura 6.6).
3. Añadimos las dependencias `@sentry/node` y `@sentry/tracing` a nuestro proyecto con `yarn add`.
4. Copiamos el código generado en la página web (figura 6.7) a nuestro código en el archivo `src/Sentry.ts`.
5. Configuramos **Sentry** para que en los registros aparezca información de qué usuario ha manifestado el error y qué mensaje estábamos manejando. Crearemos un nuevo middleware llamado `useSentry` para ello.

Tras la configuación⁹⁸, la página de **Sentry** nos indica que está esperando al primer error, como se puede ver en la figura 6.8. Introducimos una llamada a una función que no existe, `foo()`, para comprobar que el error se registra correctamente.

En la página del proyecto en **Sentry** podemos observar que los registros se guardan y podemos ver la información correspondiente, como se muestra en la figura 6.9.

⁹⁸Commit: <https://github.com/daniharo/mordente/commit/1753061f>

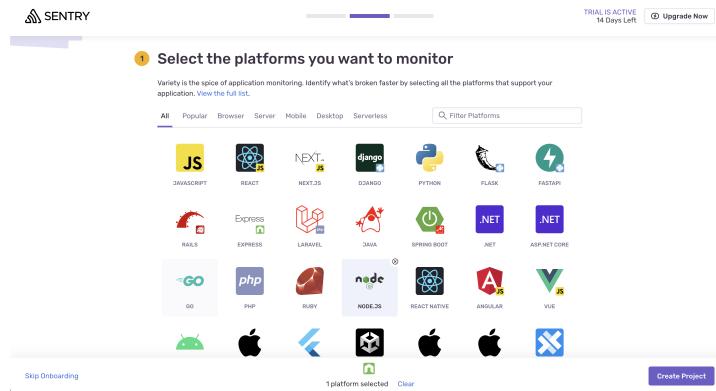


Figura 6.6: Creando el proyecto de Sentry

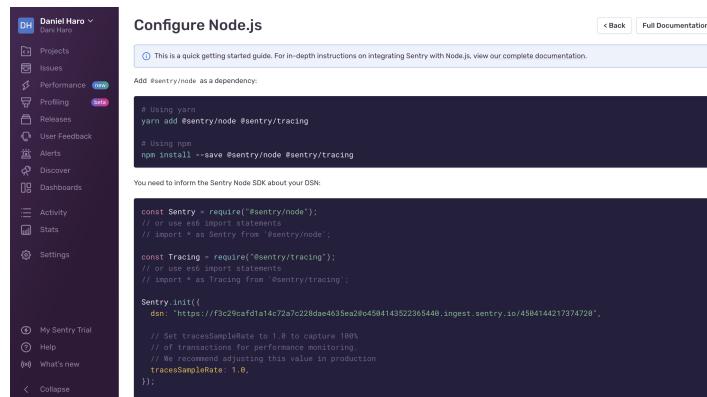


Figura 6.7: Código generado por Sentry para incluir en un proyecto

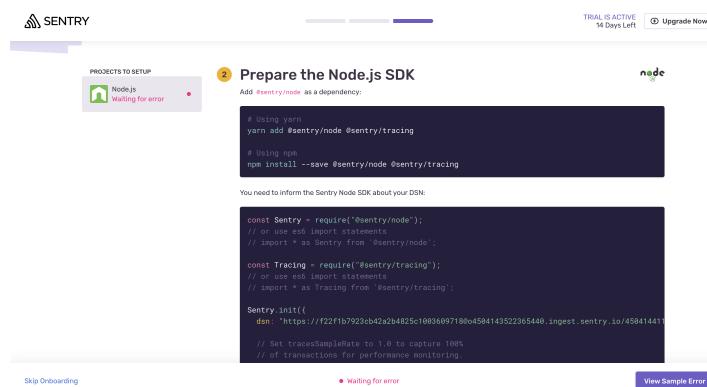


Figura 6.8: Esperando al primer log de Sentry

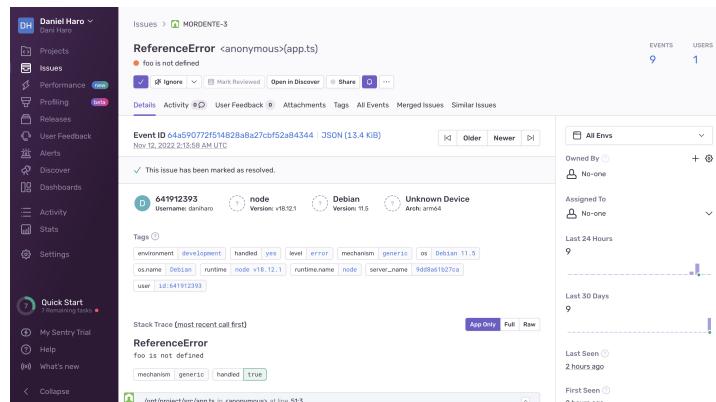


Figura 6.9: Registro de un error en Sentry

Capítulo 7

Pruebas

Tras el capítulo anterior, ya tenemos una implementación de nuestra herramienta que los usuarios pueden probar para darnos retroalimentación de cara a mejorar la usabilidad de la plataforma.

Las pruebas que se van a realizar en este trabajo, dado que la metodología usada ha sido la del *Design Thinking* y el Diseño Centrado en el Usuario, serán pruebas de usabilidad con usuarios reales.

7.1. Diseño de las pruebas

En esta sección se va a determinar cuál será el diseño de las pruebas de usabilidad. Para ello debemos concretar quién participará en las pruebas, qué tendrán que hacer y qué se les va a preguntar.

7.1.1. Participantes

Para esta primera versión se harán pruebas con 10 personas.

Dado que hay dos roles claramente distinguibles en la aplicación, se seleccionarán 3 administradores y 7 miembros para completar la prueba de usabilidad.

7.1.2. Tareas a realizar

Para comprobar la usabilidad los flujos de trabajo distintos de administradores y miembros, se van a proponer tareas distintas para cada uno de los dos roles.

En el caso de los administradores:

1. Crea una agrupación.
2. Crea un evento.
3. Edita la descripción de la agrupación.

4. Sube una obra a la agrupación.
5. Invita a un miembro de tu banda.
6. Haz que no pueda unirse nadie más a la agrupación.
7. Haz administrador al miembro que se ha unido.
8. Quítale los permisos de administrador al miembro que se ha unido.
9. Elimina el evento que creaste.

En el caso de los miembros:

1. Únete a una agrupación.
2. Mira los eventos de la agrupación.
3. Descarga alguna obra de la agrupación.
4. Sal de la agrupación.

7.1.3. Preguntas a realizar

Las preguntas que se van a realizar a los usuarios son las descritas en la tabla 7.1.

Las relacionadas con tareas concretas nos ayudarán a realizar una mejora de la usabilidad antes de la entrega, solucionando los problemas que manifiesten los usuarios y que se puedan solucionar rápidamente, y planificando como trabajos futuros las soluciones más costosas.

Por otro lado, la primera pregunta genérica nos ayudará a calcular el *NET Promoter Score* o NPS. Según esta herramienta, los encuestados que respondan entre 0 y 6 son detractores, entre 7 y 8 son pasivos y entre 9 y 10 son promotores. De esta forma, se obtendrá un índice realizando el siguiente cálculo:

$$\text{NET} = \frac{\text{Promotores} - \text{Detractores}}{\text{Encuestados}} \times 100$$

Este índice puede resultar entre -100 y 100. Diremos que tiene un resultado positivo si es mayor a 0, y excelente si es mayor a 50.

7.2. Realización de las pruebas

7.3. Informe final de las pruebas

Uno de los problemas detectados durante las pruebas viene derivado por el hecho de que Telegram no obliga a los usuarios a asignarse un nombre de usuario, por lo que el campo `username` de nuestra base de datos se queda

Preguntas por tarea	
Pregunta	Tipo de respuesta
1. ¿Has podido realizar la tarea?	Sí/No
2. ¿Ha sido fácil realizar la tarea?	Entre 1 y 5.
Preguntas genéricas	
Pregunta	Tipo de respuesta
1. ¿Algún comentario sobre las tareas? ¿Errores?	Respuesta libre.
2. ¿En qué dispositivo has probado el bot?	Móvil / Ordenador con aplicación / Ordenador en Telegram Web / Otro
3. ¿Opinión sobre la web mordente.es?	Entre 0 y 5 para información, diseño adaptativo, apariencia y velocidad
4. ¿Recomendarías Mordente a un amigo?	Entre 0 y 10
5. ¿Tienes algún comentario genérico?	Respuesta libre.

Tabla 7.1: Formulario para la prueba de usabilidad

vacío. Esto resultaba en una excepción controlada a la hora de actualizar sus datos que se ha corregido¹.

Una de las peticiones más populares ha sido la de incorporar la funcionalidad de **calendario** ya que la alternativa que se ha analizado en la sección 2.2.1 no la implementa. Dado que requiere de más esfuerzo para su implementación, se ha programado como trabajo futuro.

7.4. Demostración en vídeo

Se puede visualizar el funcionamiento del bot en el vídeo disponible en este enlace:

¹<https://github.com/daniharo/mordente/commit/70b3b54>

Capítulo 8

Conclusiones y trabajos futuros

8.1. Conclusiones

8.2. Trabajos futuros

8.2.1. Webapp

Webapp utilizando el código existente.

8.2.2. Webapp integrada en el bot

Bibliografía

- [1] INE, "Anuario Estadístico de España 2020." https://www.ine.es/prodyser/pubweb/anuario20/anu20_04cultu.pdf. [Online; Recuperado: 2021-09-27].
- [2] European Comission, "Digital Economy and Society Index (DESI) 2022." <https://ec.europa.eu/newsroom/dae/redirection/document/88764>. [Online; Recuperado: 2021-09-27].
- [3] European Comission, "Índice de la Economía y la Sociedad Digitales (DESI) 2022 - España." <https://ec.europa.eu/newsroom/dae/redirection/document/88760>. [Online; Recuperado: 2021-09-27].
- [4] UGT - Servicio de estudios, "Digitalización de la empresa española." <https://servicioestudiosugt.com/digitalizacion-de-la-empresa-espanola-3-edicion/>. [Online; Recuperado: 2021-09-27].
- [5] Ministerio de Asuntos Económicos y Transformación Digital, "España Digital 2026." https://espanadigital.gob.es/sites/espanadigital/files/2022-07/Espa%C3%B1a_Digital_2026.pdf. [Online; Recuperado: 2021-09-27].
- [6] W3C - Web Accessibility Initiative, "Notes on user centered design process (ucd)." <https://web.archive.org/web/20220901034811/https://www.w3.org/WAI/redesign/ucd>, 2004. [Online; recuperado: 2022-09-03].
- [7] ISO 13407, "Human centred design processes for interactive systems," 1999.
- [8] J. Rubin, "Handbook of usability testing: How to plan, design, and conduct effective tests," 1984.
- [9] L. J. L. Hasso Plattner, Christoph Meinel, "Design thinking: understand, improve, apply," 2011.
- [10] J. Brown, T. Wyatt, "Design thinking for social innovation. stanford social innovation review," 2010.

- [11] N. Turner, "Getting the most out of personas." <https://web.archive.org/web/20220722145021/https://www.uxforthemasses.com/personas/>, 2010. [Online; recuperado: 2022-07-22].
- [12] R. F. Dam and T. Y. Siang, "Test your prototypes: How to gather feedback and maximise learning." <https://web.archive.org/web/20220124203828/https://www.interaction-design.org/literature/article/test-your-prototypes-how-to-gather-feedback-and-maximise-learning>, 2021. [Online; recuperado: 2022-01-24].
- [13] Techopedia, "What is a minimum viable product (mvp)?" <https://web.archive.org/web/20220914143342/https://www.techopedia.com/definition/27809/minimum-viable-product-mvp>, 2020. [Online; recuperado: 2022-09-14].
- [14] C. Larman, *Agile and iterative development : a manager's guide / Craig Larman.* [electronic resource]. Agile software development series, Boston: Addison-Wesley, 1st edition ed., 2004.
- [15] Telegram, "Notification Sounds, Bot Revolution and More." <https://web.archive.org/web/20220905222430/https://telegram.org/blog/notifications-bots>, 2022. [Online; recuperado: 2022-09-05].
- [16] Wikipedia, "JavaScript — Wikipedia, the free encyclopedia." <http://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1114045733>, 2022. [Online; recuperado: 2022-10-07].
- [17] w3techs.com, "Usage statistics of javascript as client-side programming language on websites." <https://wayback.archive-it.org/20220213043439/https://w3techs.com/technologies/details/cp-javascript>, 2022. [Online; recuperado: 2022-02-13].
- [18] "JavaScript with syntax for types.." <https://web.archive.org/web/20221004155521/https://www.typescriptlang.org/>, 2022. [Online; recuperado: 2022-10-04].
- [19] Python Software Foundation, "General Python FAQ — Python 3.10.7 documentation." <https://web.archive.org/web/20220722225638/https://docs.python.org/3.10/faq/general.html>. [Online; recuperado: 2022-07-22].
- [20] T. M. Connolly, *Sistemas de bases de datos [Recurso electrónico] : un enfoque práctico para diseño, implementación y gestión / Thomas M. Connolly, Carolyn E. Begg.* Madrid: Pearson Educación, 4^a ed. ed., 2006.

- [21] OSI Board of Directors, "The sspl is not an open source license." <https://web.archive.org/web/20220930105202/https://opensource.org/node/1099>, 2021. [Online; recuperado: 2022-10-06].
- [22] Amazon Web Services, "What is Object Storage?." <https://web.archive.org/web/20221029202035/https://aws.amazon.com/what-is/object-storage/>, 2022. [Online; archivado: 2022-10-29].
- [23] Cloudflare, "¿Qué es una CDN?." <https://web.archive.org/web/20221011192027/https://www.cloudflare.com/es-es/learning/cdn/what-is-a-cdn/>, 2022. [Online; recuperado: 2022-10-11].
- [24] Reenskaug, Trygve; Coplien, James O., "The DCI Architecture: A New Vision of Object-Oriented Programming." https://web.archive.org/web/20090323032904/https://www.artima.com/articles/dci_vision.html, 2009. [Online; archivado: 2009-03-23].
- [25] Matías Hernandez, "¿Qué es Linting y ESLint? ¿Cómo empezar?." <https://web.archive.org/web/20220517050909/https://www.freecodecamp.org/espanol/news/que-es-linting-y-eslint/>, 2021. [Online; recuperado: 2022-05-17].
- [26] JetBrains, "Running and debugging Node.js." <https://web.archive.org/web/20220603104430/https://www.jetbrains.com/help/idea/running-and-debugging-node-js.html>. [Online; recuperado: 2022-06-03].
- [27] OpenJS Foundation, "Debugging Node.js." <https://nodejs.org/en/docs/guides/debugging-getting-started/#chrome-devtools-55-microsoft-edge>. [Online; recuperado: 2022-08-06].
- [28] Devopedia, "Transpiler." <https://web.archive.org/web/20220806001256/https://devopedia.org/transpiler>, 2019. [Online; recuperado: 2022-08-06].
- [29] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

