Implementación de una infraestructura REST con privacidad para GTD

TRABAJO FIN DE GRADO

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero, Alejandro Del Río Caballero

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software y Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2023-2024

Implementación de una infraestructura REST con privacidad para GTD

Memoria de Trabajo Fin de Grado

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero, Alejandro Del Río Caballero

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería del Software y Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Curso 2023-2024

Resumen

La metodología de productividad personal conocida como “Getting Things Done” (GTD), creada por David Allen, es uno de los métodos más efectivos para la organización de tareas en la actualidad. Su objetivo es maximizar la productividad a través de la consolidación de todas las tareas, proyectos y actividades en un solo lugar. Aunque existen muchas aplicaciones disponibles para ayudar a poner en práctica la filosofía GTD, la mayoría son propiedad de empresas que pueden tener acceso a la información personal de los usuarios, lo que puede violar su privacidad.

palabras clave: GTD, React Native, REST API, OAuth 2.0, Multiplataforma

Abstract

Blah Blah ...

Autorización de difusión y utilización

Los abajo firmantes, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado "Implementación de una infraestructura REST con privacidad para GTD", realizado durante el curso académico 2023-2024 bajo la dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Pablo Gamo González, Carlos Gómez López, Javier Gil Caballero, Alejandro Del Río Caballero

Juan Carlos Sáez Alcaide

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

La metodología *GTD*(*Getting Things Done*) tiene como objetivo ayudar a las personas a realizar sus tareas del día a día de manera que no dependan de su memoria y se centren en el ahora, sin estar pendiente de futuras tareas. Este método fue creado por David Allen, quien lo recogío en su libro que recibe el mismo nombre [1] y fue traducido al español como "Organízate con eficacia".

El flujo de trabajo que establece *GTD* pasa por diversas etapas cada una con una finalidad específica que da validez a la efectividad del método. También propone definir en cada tarea el lugar físico donde se realizará, con el fin de asociar los distintos lugares que una persona visita durante el día a tus tareas pendientes. En la sección 2.1 se explica la idea con más detalle.

1.1 Motivación

Inmersos en plena era digital, caracterizada por la cultura de la inmediatez, no resulta tarea fácil mantener el enfoque y evitar distracciones en medio de una vorágine de información y estímulos. Sin embargo, es precisamente en ese entorno donde la productividad personal, guiada por la metodología *GTD*, se vuelve una herramienta fundamental para poder combatir ésta problemática.

El sistema *GTD* (*Getting Things Done*), es conocido por su eficacia tanto a nivel personal como profesional en la organización, planificación y administración de tareas y proyectos. Sin embargo, algunas de las aplicaciones que lo implementan tienen varios inconvenientes.

En primer lugar, muchas de estas aplicaciones carecen de un enfoque en materia de protección de datos, ya sea por falta de transparencia de los mismos, pues no especifican cómo es el tratamiento y procesamiento de los mismos. O porque las medidas que han llegado a poner en práctica, no son lo suficientemente robustas para proteger la

información que alojan sus usuarios (p.e: los datos son alojados en servidores de terceros y no sabemos los protocolos de seguridad que tienen éstos)

Por otro lado, muchas de estas aplicaciones tienen una alta curva de aprendizaje o son compatibles con un parque de dispositivos limitado. Condicionando de esta forma el acceso de esta metodología a un público más generalista.

Por ello, motivados por poner solución a esta problemática, buscamos desarrollar una aplicación multiplataforma que además de implementar la metodología *GTD*, destaque por su interfaz intuitiva y amigable, guiada por una infraestructura *REST*, con el objetivo de permitir al usuario tener el control de sus datos, garantizando la privacidad y la transparencia de los mismos.

1.2 Objetivos

El objetivo central de este proyecto es desarrollar una aplicación multiplataforma *GTD*. Para ello hemos utilizado el framework *React Native* que permite crear una interfaz de usuario coherente que funcione en todos los sistemas operativos principales, como *Android*, *iOS*, *MacOS*, *GNU/Linux* y *Windows*, y garantiza que los usuarios puedan gestionar sus tareas y proyectos de manera eficiente desde cualquier dispositivo, sin importar la plataforma que utilicen.

La utilización de *React Native* simplifica el desarrollo, mantenimiento y escalabilidad del proyecto, lo que resulta en una aplicación ágil y adaptable a futuras actualizaciones y cambios en las plataformas de destino.

Nuestra aplicación *GTD* destaca por su modo offline, que permite a los usuarios gestionar tareas incluso sin conexión a Internet (como puede darse el caso en dispositivos móviles). A diferencia de otras alternativas, nuestra app garantiza una experiencia ininterrumpida al almacenar datos localmente y sincronizar automáticamente con el servidor cuando se recupera la conexión, asegurando la disponibilidad constante de la información en todos los dispositivos del usuario.

En el desarrollo de nuestra aplicación *GTD*, la *API REST* desempeña un papel crucial al proporcionar *endpoints* para la comunicación cliente-servidor, permitiendo operaciones *CRUD* en los datos, como por ejemplo, tener un *endpoint* '/tareas' para la creación y lectura de tareas. De esta manera, la *API REST* proporciona una interfaz estandarizada y eficiente para la manipulación de datos en nuestra plataforma *GTD*. Además, la implementación de sólidas prácticas de seguridad, como autenticación y cifrado de datos, asegura la integridad y confidencialidad de la información, garantizando una experiencia segura para nuestros usuarios.

Al desarrollar nuestra aplicación perseguimos una serie de objetivos específicos que quedaron representados mediantes los siguientes requisitos de alto nivel:

- **Gestión de tareas centralizadas:**

La aplicación debe permitir a los usuarios crear nuevas tareas, las cuales por defecto se agregarán al *Inbox* (almacén de tareas sin organizar dentro de la metodología *GTD*). Desde allí, los usuarios podrán asignarles etiquetas, vincularlas a proyectos o áreas específicas, y moverlas entre diferentes secciones.

- **Flexibilidad en la Organización:**

  Los usuarios deben tener la capacidad de organizar sus tareas de manera flexible, añadiéndolas a proyectos o áreas relevantes, y asignándoles etiquetas para una clasificación más detallada. Además, la aplicación debe permitir la edición y eliminación de tareas según sea necesario

- **Seguimiento y Priorización:**

  La aplicación debe proporcionar una serie de funcionalidades para el seguimiento y la priorización de las tareas de manera eficiente y efectiva. Los usuarios deben tener la capacidad de completar tareas y asignarles una importancia relativa. Además, la aplicación debe facilitar el acceso a información detallada sobre cada tarea, incluyendo su estado actual, fecha de vencimiento y cualquier nota asociada. Es crucial que las tareas se prioricen automáticamente según su relevancia para el usuario.

- **Seguridad:**

  Nuestra aplicación GTD debe implementar un robusto sistema de autenticación y autorización para garantizar que solo usuarios autorizados puedan acceder y manipular los datos a través de la *API REST*. Además, se requiere cifrado de extremo a extremo para proteger la confidencialidad de la información durante su transmisión y almacenamiento, asegurando así una experiencia segura para todos los usuarios.

- **Funcionalidades de Búsqueda y Filtrado:**

  Los usuarios deben poder buscar y filtrar sus tareas según etiquetas, áreas, proyectos u otros criterios relevantes, lo que facilita la gestión y visualización de las tareas.

- **Usabilidad:**

  La aplicación proporciona un modo offline que permite gestionar tareas sin conexión almacenando los datos localmente en el dispositivo del usuario y asegurando la continuidad del trabajo sin importar la disponibilidad de conexión.

- **Integración con agentes conversacionales:**

  La aplicación se integra con agentes conversacionales como *Alexa* y *Google Assistant* ofreciendo una experiencia más versátil, permitiendo así a los usuarios interactuar con la aplicación mediante comandos de voz, simplificando aún más la entrada y gestión de tareas de forma intuitiva y sin esfuerzo.

Estos requisitos proporcionan una base sólida para el desarrollo de la aplicación *GTD*, garantizando una experiencia integral que cumpla con los principios fundamentales de la metodología *GTD* y satisfaga las necesidades de los usuarios en la gestión efectiva de sus tareas y proyectos.

1.2.1 Arquitectura del sistema

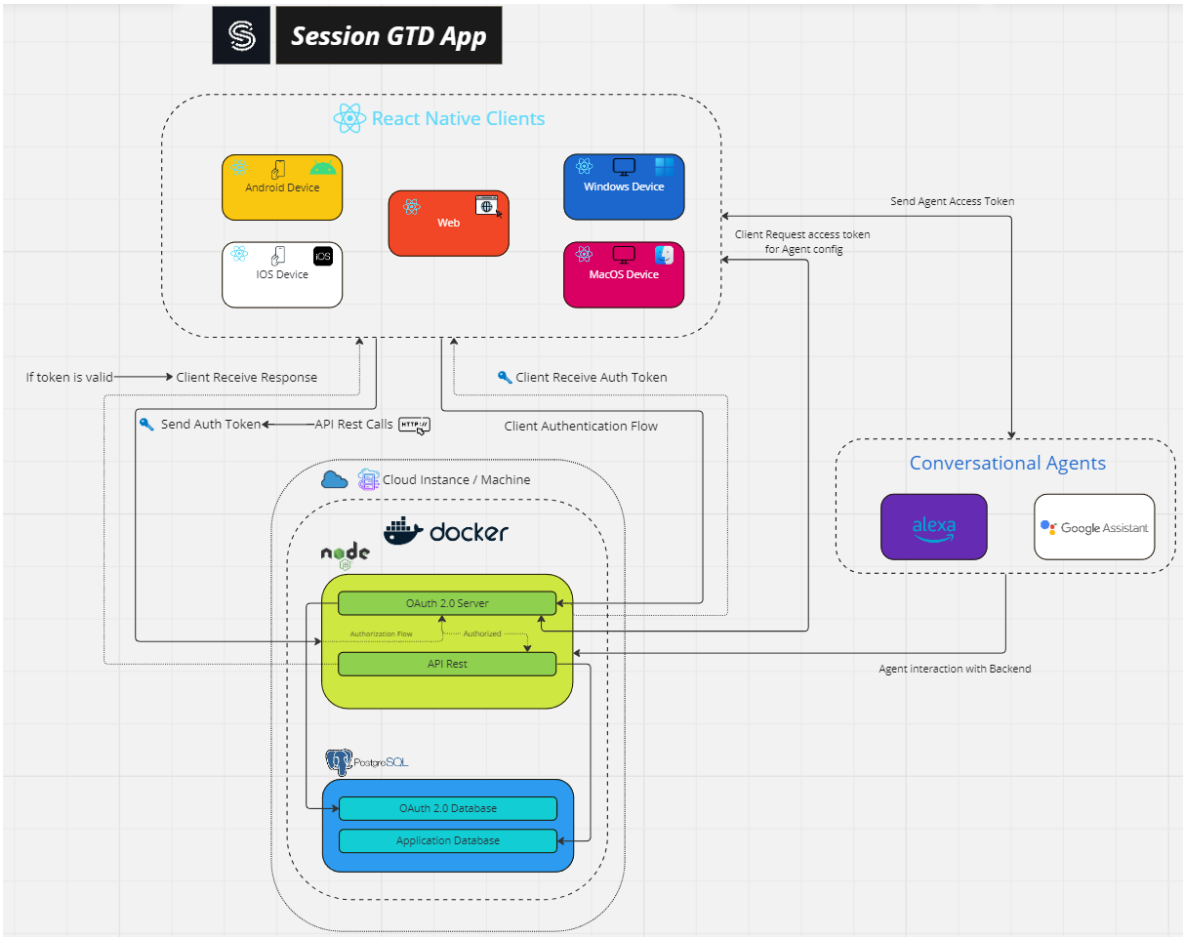

Figura 1.1: Diagrama de bloques del sistema

La figura 1.1 muestra el diagrama de bloques del sistema, que está compuesto principalmente por 2 componentes principales, una aplicación cliente, disponible para diversos dispositivos y un *backend* el cual cuenta con una *API Rest* con un sistema de autorización seguro.

La aplicación cliente está implementada con el framework multiplataforma *React Native*, pudiendo ser ejecutada en diversos dispositivos. Los clientes interactúan mediante *REST* sobre *HTTPS* con el *backend*.

El *backend* implementa 2 componentes principales e incluye una base de datos. Por una

parte está el módulo *OAuth 2.0* el cual se encarga de gestionar el flujo de autenticación y autorización, es decir gestiona el acceso de los usuarios de las aplicaciones cliente a la información y a los servicios que proporciona el *backend*. Por otra parte, el *backend* también está compuesto por una *API* que sigue la arquitectura *REST* e implementa y expone mediante diversos *endpoint*s los diferentes servicios de la aplicación. Por último el *backend* contiene también la base de datos de la aplicación, la cual contiene tanto las tablas que utiliza el módulo *API Rest* como el módulo *OAuth 2.0*. Todos los servicios que contiene el *backend* están gestionados mediante contenedores *Docker*, de esta manera es posible arrancar, conectar y configurar los diversos módulos de manera sencilla y ágil, con el fin de poder desplegar dichos servicios en cualquier máquina sin necesidad de más configuración particular a cada entorno.

Por último, el sistema contiene un tercer componente el cual permite conectar algunos agentes conversacionales con la aplicación. Desde los clientes es posible realizar dicha configuración de agentes mediante la generación de una clave especial para estos, de manera que los agentes una vez configurados puedan acceder a los servicios del *backend* mediante comandos de voz.

1.3 Plan de Trabajo

1.3.1 Tareas

Para llevar a cabo el desarrollo del proyecto hemos dividido las tareas a realizar en varias fases que se comentan a continuación.

En primer lugar y de manera individual, realizamos una labor de búsqueda comparando nuestro modelo de proyecto con otras aplicaciones existentes en el mercado para posteriormente poner en común las distintas ideas. Cada miembro del equipo se instaló una de estas aplicaciones y fue apuntando las posibles mejoras que podríamos implementar para aportar más valor a nuestro producto. Gracias a este ejercicio, llegamos a muchas de las conclusiones explicadas en la **sección 1.1**.

Posteriormente, hubo una fase de diseño en la cual realizamos un *mockup* de como nos gustaría que fuese nuestra aplicación.

A continuación, tuvimos una fase de aprendizaje en la cual buscamos información y experimentamos con las distintas herramientas y componentes necesarios para montar nuestra aplicación. Entre estos componentes buscamos familiarizarnos con *express*, *docker*, *AWS*, *React Native*, *firebase* entre otros para realizar una comparación y escoger las tecnologías que mejor se adaptaban a nuestras necesidades para el futuro desarrollo de nuestra aplicación.

Una vez terminadas las distintas tareas previas al desarrollo comentadas anteriormente, comenzamos con la programación del *backend* que nos llevó aproximadamente cuatro meses, durante este tiempo estuvimos implementando los servicios principales de la

aplicación además del sistema de autenticación y autorización. Una vez terminada la parte esencial del *backend*, comenzamos con el desarrollo del *frontend*, que ha sido la fase más costosa en tiempo y esfuerzo, ya que hemos dedicado un gran trabajo a crear una interfaz amigable y usable además de funcionalidad extensa (filtrado, búsqueda, edición…). Es por ello que hemos ido realizando actualizaciones necesarias en *backend* para el correcto funcionamiento de la aplicación.

1.3.2 Diagramas de Gantt

1.3.2.0.1 Backend

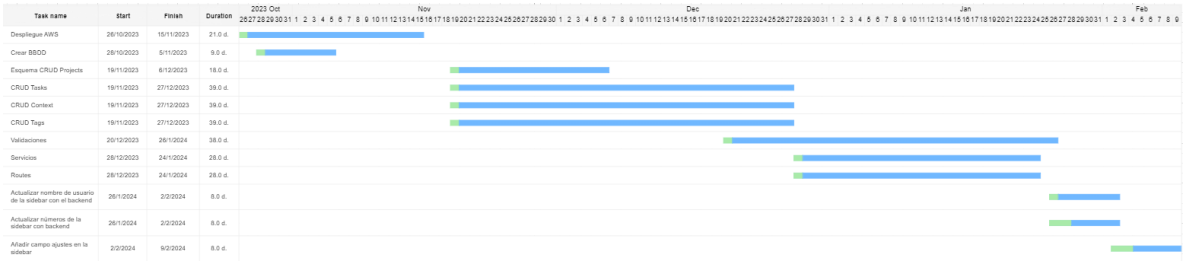

Figura 1.2: Diagrama de Gantt backend

1.3.2.0.2 Frontend

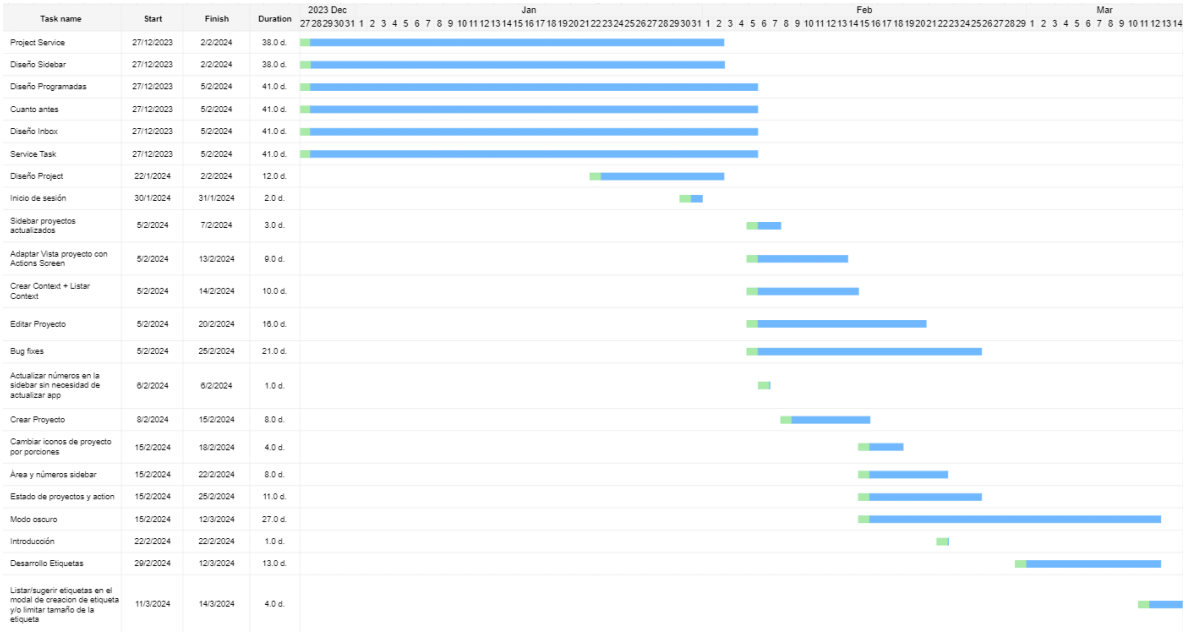

Figura 1.3: Diagrama de Gantt frontend

Capítulo 2

Planificación del proyecto

A la hora de gestionar y planificar un proyecto existen dos tipos de enfoque, el predictivo y el adaptativo. El enfoque predictivo se utiliza cuando se tiene claro cómo va a ser el proyecto y se conocen las variables y resultados del mismo, mientras que el enfoque adaptativo es más flexible y permite modificar el alcance del proyecto conforme a las necesidades que van surgiendo a lo largo del desarrollo del propio proyecto. Este segundo enfoque conduce a una mayor calidad y productividad aumentando el compromiso del equipo. **REFERENCIAR**

En nuestro caso, hemos seguido un enfoque adaptativo ya que nuestro proyecto requiere flexibilidad y velocidad. Para ello, hemos seguido una metodología *agile* para desarrollar nuestro proyecto. Esta metodología consiste en trocear el proyecto en pequeñas partes que se tienen que ir completando en cortos periodos de tiempo y que podemos observar sus ciclos en la figura 2.1. Siguiendo esta idea, hemos dividido las tareas semanalmente, de tal manera que en cada reunión definimos las tareas a realizar la próxima semana.

Adicionalmente, hemos empleado herramientas ágiles como el tablero *Kanban*, el cual nos ayuda a gestionar el proyecto de una manera más visual, viendo en qué estado se encuentra cada tarea que hemos definido. Para ello hemos utilizado la herramienta *Notion* como podemos ver en la figura 2.2, la cual muestra un ejemplo de cómo hemos ido organizando las distintas tareas a lo largo de este proyecto. Notion también nos permite crear un tablero distinguiendo el estado actual de las tareas en "sin empezar", "en curso" o "terminadas" y también ofrece la posibilidad de asignar cada tarea a uno o varios miembros del equipo que serán los encargados de completarlas.

Nuestro proyecto ha seguido un ciclo de vida iterativo e incremental, de tal manera que en cada iteración se ha revisado y mejorado el producto, el cual ha sido desarrollado por partes que se han ido integrando para construir el producto final.

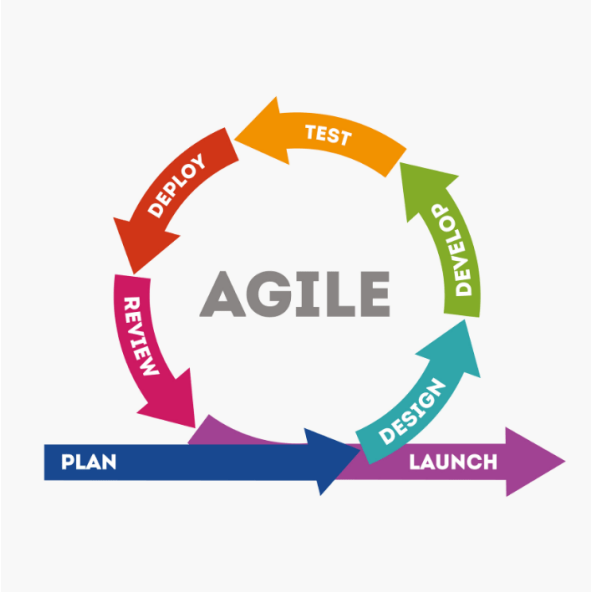

Figura 2.1: Ciclo metodologías agile

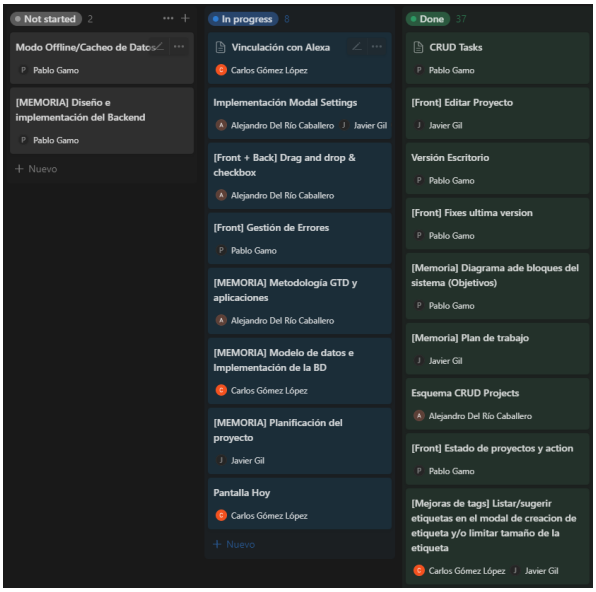

Figura 2.2: Tablero de las tareas en Notion

2.1 Gestión de configuración del Software

2.1.1 Sistema de control de versiones

Para una correcta gestión y desarrollo de cualquier proyecto software un sistema de control de versiones es esencial e imprescindible para un equipo de desarrollo. Para nuestro trabajo hemos utilizado el ampliamente extendido *Git*. Este sistema nos permite sincronizar las distintas versiones garantizando la gestión de conflictos entre los cambios realizados por el equipo de desarrollo. También guarda los datos de las distintas versiones para que sean recuperables en cualquier momento si fuese necesario. Para el alojamiento del repositorio del código hemos utilizado la plataforma *Github*.

Además hemos establecido un procedimiento de gestión de ramas en nuestro repositorio, con el fin de tener controladas las distintas funcionalidades desarrolladas. El repositorio contiene 2 ramas principales, la rama "dev" que es la que utilizamos diariamente para desarrollar nuestro producto, y que es donde se encuentra todo el codigo que está siendo desarrollado en el momento y que por lo tanto no está acabado. Por otro lado rama "main", que es la principal y la que contiene las funcionalidad terminadas y probadas, esta rama contiene la vesión mas actualizada de la aplicación. Cuando se ha terminado de desarrollar una versión y esta está probada en la rama dev, se procede a hacer merge hacia la rama main.

Por último, tenemos ramas auxiliares que utilizamos para realizar funcionalidades más específicas, de tal manera que si los cambios realizados hacen que otras partes de la aplicación puedan no funcionar, no afecte al resto y se pueda realizar un *backup* de manera sencilla. Un ejemplo de estas ultimas pueden ser las ramas llamadas *OAuth2.0* o *modoOfline* cuyo nombre hace referencia al desarrollo de de dicha funcionalidad que se decidió hacer por separado a las ramas principales. Una vez implementadas dichas funcionalidades, estos cambios pasan a la rama dev y posteriormente a main.

2.1.2 Entornos de desarrollo e Integración

Amazon Web Services (*AWS*) ha sido una parte clave para nuestro proyecto, este famoso proveedor de servicios en la nube dispone de gran cantidad de recursos de computación. Para nuestro proyecto hemos utilizado el servicio *EC2* bajo la prueba gratuita de este mismo. Este servicio te permite crear instancias de maquinas en los servidores de *AWS* donde puedes ejecutar cualquier tipo de aplicación o proceso. La prueba gratuita te da la posibilidad de crear y tener encendida una de estas instancias sin ningun coste durante un año siempre y cuando no sobrepases los límites, que de darse el caso procederán a cobrarte las tasas estandar, sin embargo para el desarrollo del trabajo no hemos llegado ni por asomo a superarlos.

Para el desarrollo del trabajo creamos una instancia con un sistema operativo *Linux* que en este caso era una distribución de *Amazon*. El principal uso que hemos hecho de esta instancia ha sido la ejecución de nuestro *backend* en la nube, lo cual nos ha facilitado

bastante el proceso de desarrollo ya que hemos podido ejecutar nuestra aplicación en un entorno de servidor y realizar las correspondientes integraciones con el *frontend*. Además tener el proyecto siendo ejecutado en la nube nos ha permitido que todos los miembros del equipo pudieran probar con facilidad los cambios realizados por los demás así como compartir el contexto de datos de la aplicación, lo que ha facilitado las pruebas.

En la instancia previamente mencionada hemos instalado la el sistema de gestión de contenedores *Docker*, que es el sistema con el que completamente el *backend*. Mediante la la herramienta de dicho sistema llamada *Docker-compose* podemos ejecutar varios contenedores y que estos sean gestionados de manera conjunta por *Docker*. Para ello definimos en un fichero *.yaml* el conjunto de contenedores que sera ejecutados dentro del mismo entorno así como la configuración de cada uno. Esta configuración coincide con la modelada en apartado de arquitectura de la introducció(incluir ref). En nuestro caso tenemos 2 contenedores, uno formado a partir de una imagen de *Node.js* donde se ejecuta la *API Rest* y otro donde se ejecuta la base de datos el cual está generado a partir de una imagen de *Postgresql*. La principal ventaja de esta herramienta es la posibilidad de definir todos tus servicios a ejecutar dentro de un solo fichero con el fin de luego desplegar o ejecutar en tu máquina todos ellos con un solo comando en vez de tener que gestionar por separado todos estos servicios. Además *Docker* crea un red interna para los contenedores de manera que puedan conectarse entre si, si la situación lo requiere, como por ejemplo en nuestro caso, donde tenemos por separado un contenedor con la *BBDD* y otro con la *API REST*.

2.1.2.1 Sistema de despligue

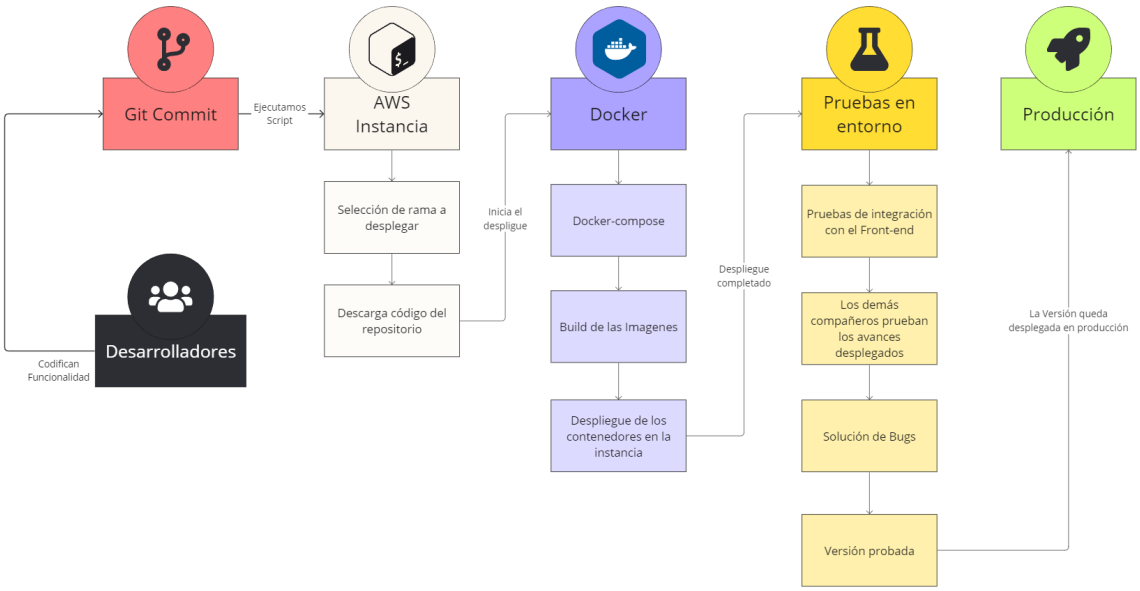

Figura 2.3: Diagrama del sistema de despliegue

Uno de los aspectos que más nos han ayudado a la hora de desarrollar nueva funcionalidad y gestionar la calidad de esta ha sido la puesta en marcha de un sistema de despliegue de vesiones en el entorno de la nube mencionado anteriormente. Este sistema consiste en un *script* de bash que hemos escrito para poder realizar con agilidad despliegues en el entorno y facilitar las pruebas de integración con el frontend asi como la puesta en producción de funcionalidad.

El *script* es lanzado cuando cuando queremos desplegar una nueva versión para probar los cambios desarrollados en local en el entorno del servidor. El proceso que sigue el *script* puede verse modelado en la figura 2.3 donde se observa el flujo que sigue nuestro código cuando realizamos el despliegue. En primer lugar lanzamos el *script* precedido por un commit que hayamos realizado y que queramos probar, después se procede a clonar el código de nuestro repositorio en la máquina, a continuación nos pregunta qué rama queremos desplegar, casi siempre suele ser la rama dev ya que es donde se desarrolla la funcionalidad principal, aunque en ocasiones utilizamos las ramas auxiliares que hemos comentado en el apartado anterior. Tras seleccionar la rama el *script* inicia el proceso de despliegue de los contenedores, ejecutando la herramienta *Docker-compose*. El *scrip*t primero detiene los contenedores en ejecución y a continuación inicia el *build* y el *run* de los nuevos, generados a partir de los cambios descargados desde el repositorio. Tras este ultimo paso los cambios realizados sobre el repositorio se encuentran ya en el entorno.

Cada vez que se realiza un despliegue comenzamos con las pruebas en entorno donde volvemos a comprobar que los cambios también funcionan en este, para ello seguimos el mismo procedimiento que en local pero apuntando a la IP del servidor, para realizar está función utilizamos la herramienta *Postman* para lanzar peticiones a nuestra API y ejecutar sus operaciones. Si observamos que el resultado de la ejecución de las llamadas al servidor no es correcto y detectamos algún fallo procedemos a mirar los logs del contenedor que ejecuta la API, para este aspecto hemos incluido en el código métodos de *logging* y trazado para facilitar la detección de errores así como para depurar el funcionamiento del código. Entre estos métodos disponemos de un sistema que imprime todas las queries ejecutadas y el resultado de estas en cuanto a datos devueltos o tiempo de respuesta entre otros. También disponemos de la librería de node.js Morgan la cual imprime en el log todas las request que recibe la api así como información sobre estas como método http, código de respuesta o tiempo de respuesta entre otros. Si detectamos algún error procedemos a replicarlo en local, solucionarlo e iniciar de nuevo el proceso de despliegue.

Capítulo 3

Modelo de datos e implementación de la Base de datos

Este capítulo tiene como objetivo describir en detalle el modelo de datos utilizado en nuestra aplicación de gestión de tareas (*GTD*), proporcionando una visión exhaustiva de cómo se organizan y relacionan los datos esenciales para su funcionamiento.

En primer lugar, presentaremos las principales entidades que componen nuestro modelo, incluyendo tareas, usuarios, proyectos, áreas, etiquetas y las relaciones asociadas a la autorización *OAuth*. Cada entidad es examinada en profundidad, detallando sus atributos y el propósito que cumplen dentro del contexto de la aplicación *GTD*.

Posteriormente, analizaremos las relaciones entre estas entidades, destacando cómo se conectan y cómo estas conexiones facilitan el flujo de información y la interacción dentro de la aplicación.

Además, discutiremos la implementación física del modelo de datos en la base de datos. Nuestra base de datos está alojada en un entorno *Docker* en *AWS* (_Amazon Web Services_), utilizando *PostgreSQL* como sistema de gestión de bases de datos. Describiremos la estructura de tablas, índices y restricciones de integridad referencial, resaltando cómo estas decisiones de diseño se traducen en la configuración final de la base de datos.

Por último, analizaremos aspectos críticos como la seguridad de la base de datos, las consideraciones de rendimiento y escalabilidad. Estos temas son esenciales para garantizar la integridad, confidencialidad y disponibilidad de los datos.

3.1 Descripción de entidades

En la figura 4.1 se muestra la relación entre las distintas entidades. A continuación, desarrollaremos las principales entidades de nuestra aplicación, junto con sus atributos

y funciones dentro del sistema:

- **Tarea:** Representa las actividades a realizar dentro de la aplicación. Cada tarea es creada por un usuario y puede contener atributos como título, descripción, fecha límite y prioridad. Además, las tareas pueden ser modificadas en cualquier momento por el usuario propietario, completadas cuando se finalicen, ser asignadas a proyectos específicos o etiquetadas con *tags* relevantes para una mejor organización.
- **Usuario:** Los usuarios tienen un rol central en la organización y gestión de la aplicación, ya que tienen la capacidad de crear, modificar y eliminar tanto tareas como proyectos. Pueden también crear y gestionar áreas para una organización más eficiente de su espacio de trabajo. Además, tienen el control sobre aspectos de personalización de la aplicación, junto con la capacidad de modificar su propio perfil.
- **Proyecto:** Permite a los usuarios organizar sus tareas en conjuntos más amplios relacionados con un objetivo común. Los proyectos pueden ser creados, editados y eliminados por los usuarios, y las tareas pueden ser asignadas a proyectos específicos para una gestión más eficiente.
- **Área:** También conocida como "Contextos", ofrece una forma adicional de categorizar las tareas. Los usuarios pueden crear, modificar y eliminar áreas según sus necesidades, y asignar tareas a áreas específicas para una mejor organización y seguimiento.
- **Etiqueta:** La entidad etiquetas permite etiquetar las tareas con palabras clave relevantes para una clasificación más detallada. Los usuarios pueden crear, editar y eliminar etiquetas, y asignarlas a tareas individuales para una organización más flexible y personalizada.
- **Relaciones de *OAuth*:** Las entidades *oauth_authcode*, *oauth_clients* y *oauth_tokens* están relacionadas con el proceso de autorización *OAuth* para la autenticación de usuarios en la aplicación, facilitando la seguridad y la gestión de accesos.

3.2 Relaciones entre entidades

En esta sección, exploraremos las relaciones entre las diferentes entidades dentro del Modelo de Datos de nuestra aplicación. Estas relaciones son fundamentales para comprender cómo interactúan los distintos componentes del sistema y cómo se organiza la información.

- **Tarea y Usuario:** Cada tarea de la aplicación está asociada a un usuario que la crea y gestiona. Esta relación permite a los usuarios tener control total sobre sus propias tareas, incluyendo la creación, modificación y eliminación.

- **Tarea y Proyecto:** Las tareas pueden estar vinculadas a proyectos específicos, lo que facilita la organización y seguimiento de las actividades dentro de entornos más amplios. Esta relación permite a los usuarios agrupar las tareas relacionadas bajo un objetivo común y gestionarlas de manera eficiente.
- **Tarea y Área:** Las tareas también pueden estar asociadas a áreas o contextos específicos, lo que proporciona una categorización adicional para una mejor organización. Los usuarios pueden asignar tareas a áreas relevantes según entorno en el que deben realizarse, lo que facilita la priorización y gestión de estas.
- **Tarea y Etiqueta:** Las etiquetas se utilizan para clasificar y categorizar las tareas según temas o características comunes. Las tareas pueden estar etiquetadas con una o más etiquetas, lo que permite una organización más detallada y organizada. Esta relación permite a los usuario filtrar y buscar tareas según etiquetas específicas para una gestión más eficiente.
- **Usuario y Proyecto/Área:** Los usuarios tienen la capacidad de crear, modificar y eliminar tanto proyectos como áreas dentro de la aplicación. Esta relación permite a los usuario organizar y personalizar su espacio de trabajo de acuerdo con sus necesidades y preferencias.

3.3 Modelo físico de la BD

En la figura 3.1 se mostrará un diagrama de la estructuración de las tablas en la base de datos. A continuación, vamos a detallar la implementación concreta del modelo de datos en la base de datos real que respalda nuestra aplicación. Describimos la estructura de tablas, los tipos de datos utilizados, así como las relaciones y restricciones de integridad referencial.

Empezamos con las descripciones detalladas de las tablas que componen la base de datos:

- **tasks:**
  - "task_id": Identificador único de la tarea
  - "user_id": Identificador del usuario al que pertenece la tarea (obligatorio).
  - "context_id": Identificador del área/contexto en el que puede estar una tarea (opcional).
  - "project_id": Identificador del proyecto al que pertenece la tarea (opcional).
  - "title": Título de la tarea (obligatorio).
  - "description": Descripción de la tarea (opcional).
  - "state": Sección en la que se encuentra la tarea (Inbox, Cuanto Antes, Programadas, etc.).
  - "completed": Booleano que indica si la tarea está completa o no.

Figura 3.1: Diagrama Entidad-Relación en la base de datos

  - "important_fixed": Booleano que indica si la tarea es importante (prioridad a la hora de hacer tareas).
  - "date_added": Fecha en la que se añade la tarea.
  - "date_completed": Fecha en la que se completa la tarea.
  - "date_limit": Fecha límite para realizar la tarea (opcional).
  - "date_changed": Fecha en la que se realiza algún cambio en la tarea.
  - "num_version": Número de versión por la que se va en la tarea (por cada cambio).

- **users:**
  - "user_id": Identificador del usuario
  - "name": Nombre del usuario
  - "email": Correo del usuario
  - "password": Contraseña del usuario

- **projects:**
  - "project_id": Identificador del proyecto
  - "title": Título del proyecto
  - "description": Descripción del proyecto
  - "user_id": Identificador del usuario al que pertenece el proyecto
  - "completed": Boolean que indica si está completo el proyecto o no
  - "date_added": Fecha en la que se añade el proyecto
  - "date_changed": Fecha en la que se modifica el proyecto
  - "date_completed": Fecha en la que el proyecto ha sido completado
  - "num_version": Número de version en la que se encuentra el proyecto
  - "color": Color que corresponde al proyecto

- **areas_contexts:**
  - "context_id": Identificador del área/contexto
  - "name": Nombre del área
  - "user_id": Identificador del usuario al que corresponde el área

- **tags:**
  - "name": Nombre de la tag
  - "color": Color de la tag

- **tagstotask:**
  - "task_id": Identificador de la tarea a la que pertenece la tag
  - "nametag": Nombre de la tag

- **oauth_authcode:**
  - "authorization_code": Código de autorización
  - "expires_at": Fecha de expiración del código

  - "redirect_uri": Dirección a la que te redirige si la obtención del código de autenticación es correcto.
  - "client_id": Identificador del cliente
  - "user_id": Identificador del usuario que recibe el código

- **oauth_clients:**
  - "client_id": Identificador del cliente
  - "client_secret": Número secreto del cliente
  - "redirect_uri": Dirección a la que te redirige si la obtención del código de autenticación es correcto.
  - "grants": Indica los flujos de Oauth configurados para el cliente.

- **oauth_tokens:**
  - "access_token": Token de acceso
  - "access_token_expires_at": Fecha de expiración del token
  - "refresh_token": Token refrescado
  - "refresh_token_expires_at": Fecha de expiración del token refrescado
  - "client_id": Identificador del cliente
  - "user_id": Identificador del usuario

A continuación, comentamos cómo se establecen las relaciones entre las tablas para mantener la coherencia de los datos y garantizar su integridad:

- **Usuarios con Tareas/Proyectos/Áreas:** La tabla de "users" con las tablas "tasks", "projects", "areas_contexts" tiene una relación uno a muchos. (Creo que se podría quitar) Un usuario puede tener muchas tareas, proyectos y áreas, pero cada una de ellas pertenece a un solo usuario. Esta relación uno a muchos se establece mediante la clave foránea "user_id" en las tablas "tasks", "projects" y "areas_contexts", que referencia al identificador único del usuario en la tabla "users".

- **Proyectos y Tareas:** La tabla "projects" tiene una relación uno a muchos con la tabla "tasks". Un proyecto puede tener muchas tareas, pero cada tarea pertenece solo a un proyecto. Esta relación uno a muchos se establece mediante la clave foránea "project_id" en la tabla "tasks", que referencia al identificador único del proyeto en la tabla "projects".

- **Áreas y Tareas:** La tabla "areas_contexts" tiene una relación uno a muchos con la tabla "tasks". Un área puede tener muchas tareas, pero cada tarea pertenece solo a un área. Esta relación uno a muchos se establece mediante la clave foránea "context_id" en la tabla "tasks", que referencia al identificador único del área en la tabla "areas_contexts".

- **Tareas y Tags:** Una tarea puede tener muchas etiquetas, y una etiqueta puede estar asociada a muchas tareas. Esta relación muchos a muchos se implementa mediante una tabla intermedia "tagstotask", que contiene las claves foráneas "task_id" y

"nametag" que relacionan las "tasks" con las "tags".

3.4 Rendimiento y Escalabilidad de la Base de Datos

En esta sección, detallamos la estructura y el desempeño de la base de datos implementada. Describimos las acciones concretas llevadas a cabo para mejorar la eficiencia y capacidad de respuesta del sistema ante un crecimiento progresivo de carga de trabajo.

Para mejorar el rendimiento de la base de datos hemos realizado una optimización de consultas, definiendo claves primarias en las tablas pertinentes para garantizar la unicidad de las filas y mejorar el rendimiento de las consultas. Además de diseñar consultas eficientes para minimizar la carga en el servidor de la base de datos.

Con respecto a la escalabilidad, hemos diseñado la estructura de la base de datos con flexibilidad y adaptabilidad, permitiendo una fácil expansión y ajuste para satisfacer futuras necesidades de crecimiento. Cada aspecto del diseño ha sido cuidadosamente planificado para garantizar la escalabilidad del sistema y facilitar la incorporación de nuevas funcionalidades según sea necesario, sin comprometer la integridad de los datos ni la eficiencia del sistema.

3.5 Seguridad de la Base de Datos

La seguridad de la base de datos es un componente fundamental para proteger la integridad, confidencialidad y disponibilidad de los datos almacenados. En esta implementación, hemos adoptado diversas medidas para garantizar un entorno seguro:

- **Autenticación y Autorización:** Hemos implementado un sistema de autenticación robusto que requiere credenciales válidas para acceder a la base de datos. Se hablará de este sistema en los siguientes apartados.
- **Cifrado de datos:** Implementamos técnicas de cifrado utilizando la biblioteca *bcrypt* para proteger la información sensible almacenada en la base de datos que pueda ser vulnerable a accesos no autorizados (contraseñas). *Bycript* es un algoritmo de hashing adaptativo diseñado específicamente para almacenar contraseñas de manera segura. Este enfoque garantiza que las contraseñas estén protegidas contra ataques de fuerza bruta y de diccionario, proporcionando una capa adicional de seguridad para mantener la informacion confidencial protegida en todo momento.
- **Registro de actividades:** Empleamos las capacidades integradas de registro y auditoría proporcionadas por *AWS* para supervisar todas las actividades realizadas en nuestra base de datos.Estas funciones nos permiten rastrear quién accede a la base de datos, cuándo lo hace y qué operaciones realiza, garantizando la integridad y seguridad de los datos almacenados en la nube.

Para finalizar este capítulo, es crucial destacar la importancia del diseño y la implementación eficientes del modelo de datos en nuestra aplicación de gestión de tareas. A través de un análisis exhaustivo de las entidades, relaciones y consideraciones técnicas, hemos establecido una base sólida para el funcionamiento de nuestra base de datos. Al comprender la estructura subyacente y las decisiones de diseño, estamos mejor preparados para abordar los desafíos futuros y garantizar la integridad, seguridad y escalabilidad continuas de nuestra aplicación *GTD*.

Capítulo 4

Diseño e implementación del Backend

La arquitectura cliente-servidor es una de las más usadas a día de hoy y por lo tanto tener un *backend* robusto es de vital importancia en cualquier aplicación. Por ello hemos invertido gran cantidad de esfuerzo en el diseño y la implementación del *backend* para asegurar un correcto funcionamiento de la aplicación a nivel de lógica de negocio además de garantizar seguridad en la comunicación con los clientes a la hora de acceder a datos y operaciones.

4.1 Uso de REST

Una de las características a destacar es el uso de la arquitectura *REST* para nuestra *API*. La clave de esta arquitectura es el uso del protocolo HTTP para el acceso a una interfaz de operaciones bien definidas. Además la arquitectura *REST* busca definir una sintaxis para identificar los recursos y sus diferentes operaciones consiguiendo así una organización clara y mantenible de los diferentes elementos de nuestra aplicación para facilitar el acceso desde los clientes. El uso de esta arquitectura en nuestro proyecto viene dada por la necesidad de organizar los distintos recursos a los que puede acceder la aplicación cliente con el objetivo de estandarizar el servicio y facilitar la comunicación y evolución de este.

Por otro lado aunque REST permite la comunicación entre el cliente y el servidor mediante cualquier formato usamos el formato JSON ya que es un formato muy sencillo de comprender y manipular en comparación a otros como por ejemplo XML donde además JSON es mucho más ligero en cuanto a demanda de ancho de banda respecto a este.

4.2 Diseño de la API

4.2.1 Descripción tecnologías: Node JS + Express

Para el desarrollo del *backend* nos hemos decantado por el uso del entorno Node.js en conjunto con el *framework* de este mismo llamado Express.js el cual proporciona una manera muy sencilla de implementar gran cantidad de servicios de aplicaciones web.

Node.js es una entorno de ejecución multiplataforma basado en el lenguaje de programación JavaScript para la capa del servidor. Este entorno utiliza el motor de JavaScript V8 utilizado en el navegador Google Chrome pero en este caso se utiliza fuera de un navegador lo cual incrementa el rendimiento considerablemente. Una aplicación con Node.js se ejecuta en un mismo proceso, es decir no crea un proceso para cada *request* y es por ello que este entorno provee de un conjunto de estándares asíncronos para prevenir que el código JavaScript tenga bloqueos. Cuando Node.js realiza operaciones de entrada/salida como acceder a una base de datos o leer del sistema de ficheros en vez de bloquear el hilo y perder ciclos de CPU lo que hará será reanudar las operaciones cuando vuelva la respuesta a la lectura/escritura. En resumen Node.js permite realizar gran cantidad de conexiones en el servidor sin emplear concurrencia entre hilos. Otra de las principales características de Node.js es el lenguaje JavaScript el cual es ampliamente utilizado por desarrolladores de *front-end* en la web y que con Node.js tienen la posibilidad de escribir código de parte del servidor sin necesidad de aprender otros lenguajes.

Desarrollar en el entorno Node.js permite además utilizar gran cantidad de paquetes que se gestionan a través de la herramienta *npm*. Con esta herramienta, que se incluye con la instalación del entorno, puedes descargar e instalar paquetes de los repositorios de *npm* con gran facilidad ya que gestiona de manera automática todas las dependencias. Una vez instalas paquetes, *npm* genera el fichero *package.json* y *package-lock.json* el cual contiene información sobre todos los paquetes instalados y sus dependencias, de esta manera el proyecto puede ser instalado y ejecutado en cualquier máquina resolviendo *npm* todo el proceso de instalación. Esta característica de Node.js nos ha facilitado bastante la colaboración en el proceso de desarrollo de la API ya que en nuestros repositorios hemos incluido los ficheros mencionados previamente y *npm* ha gestionado completamente la instalación del entorno.

4.2.2 Express.js

Para la implementación de la API hemos utilizado el *framework* Express.js el cual proporciona herramientas para implementar gran variedad de servicios de *backend* de una manera muy minimalista y sencilla. Express proporciona principalmente herramientas para gestionar el enrutamiento en tu aplicación web, facilita el manejo de *requests* y *responses* de HTTP y la implementación de *middlewares* entre otras características. La razón por la que hemos escogido este *framework* es su sencillez para definir manejadores de ruta con los cuales podíamos implementar los diferentes *endpoints* de nuestra API

además de las facilidades que te proporciona para gestionar tanto *requests* como *responses* lo que nos ha servido para gestionar la de entrada de datos y los errores.

Como hemos mencionado anteriormente gracias a la facilidad con la que puedes gestionar la instalación de paquetes en un entorno Node.js hemos hecho uso de varios que complementan a Express.js para el desarrollo de la API, entre ellos destacan el paquete *express-validator* para facilitar la validación sintáctica de los valores de entrada a los *endpoints*. También otros paquetes como *pg* que es el módulo para conectar un sistema de Node a una base de datos PostgreSQL, el paquete *bcrypt* que hemos utilizado para cifrar las contraseñas de los usuarios cuando se registran en la aplicación y el modulo de OAuth 2.0 para Node.js del cual hablaremos más en profundidad en el apartado (Referencia!!!!!).

4.3 Diseño de la API

La arquitectura REST tiene como principal característica el empleo de una sintaxis para identificar los diferentes *endpoints*. Como se ha mencionado anteriormente esta sintaxis busca relacionar de manera clara los *endpoints* con los recursos a los que accede y sus operaciones, definiendo una serie de reglas a la hora de diseñar la API.

Entre estas prácticas que hemos seguido para nombrar los *endpoints*, están utilizar el nombre del recurso al que se accede sin utilizar verbos, es decir, sin utilizar por ejemplo el nombre de la operación (crear, borrar, modificar…) ya que esta información ya la proporciona el método de HTTP utilizado. Precisamente esta última apreciación también la hemos tenido en cuenta ya que hemos sacado partido de los diferentes métodos de HTTP para definir de manera clara y organizada las operaciones. También hemos hecho uso de rutas parametrizadas, es decir el uso de parámetros directamente en la uri, por ejemplo */example/:example_id*. Otros de los aspectos de gran importancia en cualquier API y en el que nos hemos centrado para el diseño de la nuestra es la gestión de errores. Hemos especificado para cada endpoint que códigos de status HTTP devolverán en función del resultado para poder gestionar cuando una operación ha sido ejecutada con éxito o ha ocurrido algún error.

4.4 Endpoints

Endpoint	/task/
Descripción	Crea una tarea para el usuario que lo solicita
Método HTTP	POST
Código HTTP (OK)	201
Código HTTP (KO)	409

Endpoint	/task/:id
Descripción	Modifica una tarea del usuario que lo solicita
Método HTTP	POST
Código HTTP (OK)	200
Código HTTP (KO)	409

Endpoint	/task/:id
Descripción	Devuelve el contenido de una tarea
Método HTTP	GET
Código HTTP (OK)	200
Código HTTP (KO)	404

Endpoint	/task/
Descripción	Devuelve todas las tareas no completadas del usuario que las solicita
Método HTTP	GET
Código HTTP (OK)	200
Código HTTP (KO)	404

4.5 Aspectos de seguridad de la API

Introducir estos aspectos

4.6 Implementación de OAuth 2.0

Uno de los aspectos más importantes que hemos podido implementar en nuestra API es el estándar *OAuth 2.0*. Este estándar introduce una nueva capa de autorización separando este rol del servidor de recursos y proporcionando así varias ventajas principalmente en términos de seguridad. A continuación se describe en profundidad cómo funciona este estándar y cuales son sus principales ventajas.

Capítulo 5

Integración con agentes conversacionales

En este capítulo, profundizaremos en el proceso de integración de nuestra aplicación GTD con la plataforma de voz de Alexa. Desde el inicio del proyecto, nos propusimos habilitar la interacción mediante comandos de voz para mejorar la experiencia del usuario.

5.1 Configuración de la skill de Alexa

El primer paso crucial en nuestra aplicación fue obtener acceso al *Amazon Developer Console* para crear una *skill* de *Alexa*. Esta *skill* fue meticulosamente configurada para permitir a los usuarios añadir tareas a nuestra aplicación *GTD* utilizando comandos de voz de manera intuitiva y eficiente. Durante la configuración, definimos cuidadosamente los tipos de interacciones que la *skill* debería admitir, como la capacidad de añadir un título obligatorio para la tarea y la opción de proporcionar información adicional, como una descripción, fecha de vencimiento o nivel de importancia.

5.2 Vinculación de la cuenta del usuario

Para habilitar la interacción entre la *skill* de *Alexa* y nuestra aplicación, implementamos un sólido flujo de autorización *Oauth* realizado también en *Amazon Developer Console*. Optamos por el método de autenticación de código de autorización (*Auth Code Grant*) para permitir que los usuarios vinculen sus cuentas de manera segura. Sin embargo, surgió un desafío significativo: para garantizar la seguridad de la transacción, *Amazon* requiere que las aplicaciones estén protegidas mediante *HTTPS*. Si bien la implementación de *HTTPS* estaba pendiente, este requisito nos incentivó a adelantar su incorporación a nuestra plataforma web, por lo tanto incorporamos *HTTPS* a nuestra plataforma web y permitir asi la vinculación de las cuentas de usuario.

5.3 Implementación del flujo de autorización Oauth

Al configurar la *skill* de *Alexa*, seguimos un diagrama de secuencia detallado que guía al usuario a través del proceso de vinculación de la cuenta (consulte la figura 5.1). Todo este flujo de autorización *Oauth* ya se prueba en la propia aplicación de *Alexa*. Para realizar pruebas y ajustes, es necesario iniciar sesión con el mismo usuario que en *Amazon Developer Console*, donde se define la *skill*.

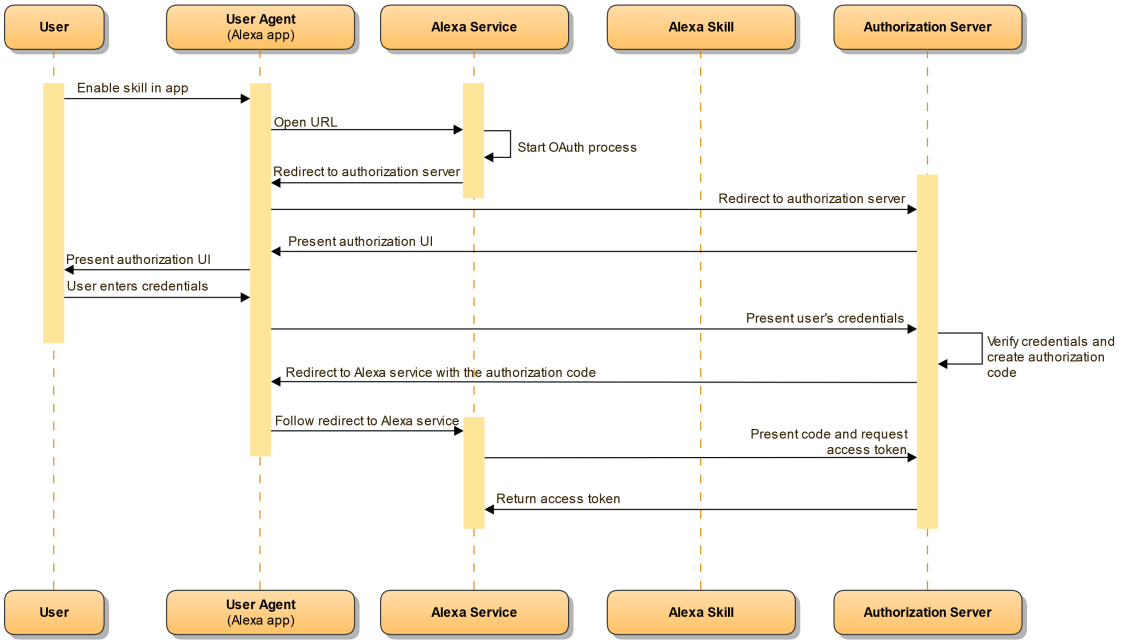

Figura 5.1: Diagrama de secuencia del proceso de vinculación de la cuenta

El proceso comienza cuando el usuario habilita *skill* en su dispositivo, la cual informa al usuario que necesita vincular su cuenta a la aplicación *GTD* para que funcione.

Posteriormente, se redirige al usuario a la página de autorización de nuestro servidor. Aquí, el usuario introduce sus credenciales de la cuenta de la aplicación *GTD*. Nuestro servidor verifica las credenciales proporcionadas por el usuario y, una vez verificadas, genera un código de autorización y un estado.

Este código de autorización y el estado son devueltos por nuestro servidor a la *skill* de *Alexa*. La *skill* intenta entonces obtener un *token* de acceso utilizando el código de autorización proporcionado. Si hay un error en el proceso, la *skill* muestra un mensaje de error al usuario,y en caso de éxito, la *skill* vincula la cuenta del usuario y muestra un mensaje de confirmación.

5.4 Problemas en la vinculación

A pesar de nuestros esfuerzos, durante la implementación nos encontramos con un problema en el proceso de vinculación. Aunque nuestro servidor devuelve el código de autorización correctamente, *Alexa* no logra completar el paso final para solicitar el *token* de acceso. Como resultado, recibimos un mensaje de error que indica: "No fue posible vincular la cuenta Alexa". Actualmente, estamos investigando a fondo la causa de este problema y trabajando diligentemente en una solución para garantizar una integración fluida entre nuestra aplicación *GTD* y la *skill* de *Alexa*.

Capítulo 6

Introducción

Esto es solo un ejemplo de capítulo. Vamos a citar algo [2], [3].

En la figura 6.1 se muestra el escudo de la Universidad Complutense de Madrid.

Figura 6.1: Esto es el título

- UNO
- DOS
- TRES
- CUATRO
- TEST DE CAPITULO

OK	A	B
AHSAJHD	ASHJDHAJSHº	SAHJDHJASH
SADHJHJ	ASHJDFHJASH	ASDASD

Tras esta breve introducción al capítulo continuamos con la sección 6.1.

6.1 Ejemplo de sección

```
#ifdef CONFIG_PMC_PERF
    int safety_control;
    unsigned char prof_enabled;
#endif
```

Epílogo

Esto es solo un ejemplo de apéndice.

Bibliografía

[1] D. Allen, *Getting Things Done. The Art of Stress-Free Productivity*. Penguin, 2003.

[2] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[3] ARM, «AMBA(R) AXI(TM) and ACE(TM) Protocol Specification». http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html, 2011.