



UNIVERSITAT
POLITÀCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aplicación gestión de tareas para un grupo de personas

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Joan Juan Belda

Tutor: José Angel Carsí Cubel

2020-2021

Resumen

El proyecto aborda el desarrollo de una aplicación para la creación, la gestión y el reparto de tareas entre los miembros de un grupo, así como el control de la realización de cada tarea por parte de los usuarios. Para el desarrollo de la aplicación se usará el framework o el subconjunto de sistemas llamado MEAN (acrónimo para: MongoDB, Express, Angular, Node).

Palabras clave: angular, gestión, tareas, mean, mongo.

Tabla de contenido

1.	Introducción	7
1.1.	Motivación.....	7
1.2.	Objetivos	8
1.3.	Impacto esperado.....	8
1.4.	Estructura.....	8
2.	Estado del arte	11
2.1.	Situación actual de la tecnología.....	11
2.1.1.	Remember The Milk	11
2.1.2.	Todoist	11
2.1.3.	Trello	12
2.1.4.	Jira	12
2.2.	Crítica al estado del arte.....	13
2.3.	Propuesta	13
2.4.	Conclusiones	14
3.	Metodología.....	15
3.1.	Metodología.....	15
3.2.	Enfoque	17
3.3.	Framework	18
3.4.	Control de versiones.....	19
3.5.	Integración/distribución continua.....	19
4.	Análisis y especificaciones de requisitos	21
4.1.	Actores.....	21
4.2.	Casos de uso	21
4.3.	Requisitos no funcionales	30
5.	Diseño	31
5.1.	Análisis de requisitos	31
5.2.	Modelado de la base de datos	33
5.3.	Arquitectura	36
5.4.	Interfaz (prototipos).....	38
6.	Implementación	43
6.1.	Contexto tecnológico.....	43
6.2.	Gestión de versiones	48
6.3.	Entornos.....	50
6.4.	Integración continua	51

6.5.	Base de datos.....	55
6.6.	Backend.....	57
6.6.1.	Organización del proyecto	58
6.6.2.	Dependencias.....	59
6.6.3.	Implementación.....	61
6.7.	Frontend.....	66
6.7.1.	Organización del proyecto	66
6.7.2.	Dependencias.....	68
6.7.3.	Implementación.....	69
7.	Pruebas	89
7.1.	Pruebas unitarias	89
7.2.	Pruebas de integración.....	92
7.2.1.	Cliente-Servidor-Base de datos.....	93
7.2.2.	Cliente-Servidor	94
7.3.	Pruebas E2E.....	95
8.	Conclusión	97
8.1.	Relación proyecto-estudios cursados.....	97
8.2.	Trabajos futuros y líneas de mejora	98
9.	Referencias	99

1. Introducción

El proyecto actual tiene como objetivo el desarrollo de una aplicación para la gestión de las tareas en un determinado grupo de usuarios para permitir una gestión de las tareas rápida y eficaz, así como poder ver en cualquier momento una imagen del estado actual de las tareas que tienen asignadas y permitir al gestor del grupo ver el estado actual de lo que se está haciendo por parte de los demás usuarios. También se ofrece la posibilidad al gestor de obtener informes del tiempo que se ha dedicado a cada tarea durante un período determinado de tiempo.

El proceso de desarrollo de la aplicación está enfocado en la integración continua, por lo que se detallará a lo largo del proyecto como se ha implementado cada fase de la integración como la gestión de versiones y entornos, los tipos de test implementados para cada parte de la aplicación y los métodos de subida a los diferentes entornos. Todo esto para intentar simular y entender cómo debería estar implementado todo el proceso de desarrollo y mantenimiento de un software en un entorno real.

1.1. Motivación

La realización de este proyecto surge a partir de la necesidad de organizar la cantidad de tiempo dedicado a las tareas que se estaban gestionando en el trabajo.

Actualmente las aplicaciones de gestión de tareas que hay en el mercado, al menos las gratuitas, solamente permiten ver las tareas pendientes, las que se están haciendo y las que se han hecho ya. Con esto puedes llegar a hacer una imagen de la carga de trabajo que se tiene. El problema surge cuando las tareas tienen una estimación de horas y una fecha de entrega determinada, además de tener que mandar informes diarios del tiempo dedicado a cada tarea. Esta última parte es en la que no se enfocan la mayoría de las aplicaciones y que se necesita automatizar, ya que consume una cantidad ingente de tiempo gestionándolo “a mano”.

Por todo esto se decidió desarrollar una aplicación con la idea de facilitar la gestión del tiempo de cada una de las tareas y poder ver cuánto tiempo lleva un usuario en una tarea determinada y si queda mucho para que se pase del tiempo que se estimó en el momento de su creación.

A parte de lo anterior, hay que tener en cuenta la gran importancia del apartado de test en un software y de la facilidad y ahorro de tiempo que supone implementar todo el proceso de desarrollo en la integración continua. Así que se decidió darle otro punto de enfoque y aprender sobre el test de software y la creación de un entorno de desarrollo real, con una buena gestión de versiones y entornos reales.

En resumen la motivación surge al ver la cantidad de tiempo que se tenía que invertir en dos de los apartados que tiene cualquier empresa de desarrollo de software, que son: la gestión de las tareas y el tiempo que se le dedica a cada una, así como el reporte de horas al mánager del proyecto y el reporte de horas para facturar por parte del mánager del proyecto al cliente, el tiempo y la poca seguridad que implica probar el software manualmente y subir los cambios manualmente a los respectivos entornos.

1.2. Objetivos

El objetivo principal es la realización de una aplicación que permita gestionar el tiempo dedicado a cada tarea. Con una interfaz sencilla y usable.

Otro de los objetivos fundamentales es el buen uso y la buena combinación de todos los elementos que componen el framework MEAN, que es el entorno en el que se desarrolla la aplicación. Este framework se compone de la base de datos no relacional MongoDB, la parte de servidor Express y la parte visual Angular.

Otro de los objetivos importantes en los que se enfoca el proyecto es en la parte de pruebas del software. Esta parte se centra en la implementación de las pruebas de cada una de las partes que componen la aplicación. Se crearán pruebas unitarias para cada una de las partes, pruebas de integración para cada par de componentes que se usen conjuntamente y pruebas e2e para simular y probar cómo reacciona el sistema delante de un usuario normal en un entorno real.

Por último, también se quiere desarrollar todo el proyecto en un proceso de integración continua. Para esto hay que definir los entornos en los que se implementará la aplicación (desarrollo, preproducción y producción), así como estructurar los entornos en la gestión de versiones y ejecutar una serie de procesos (pruebas y subida) cuando se suba una versión en un entorno determinado.

1.3. Impacto esperado

Mediante esta aplicación se espera reducir el tiempo que se invierte por parte de los usuarios de un grupo o proyecto en la gestión de sus tareas y el tiempo que invierten en cada una para que cada uno tenga una imagen clara de la carga de trabajo y de lo que se tiene que entregar en cada momento.

Además, se espera que el manager de un grupo pueda tener una foto del estado actual de las tareas del grupo, así como poder obtener reportes del tiempo que se ha invertido en un intervalo de tiempo en el grupo o en una determinada tarea. Así como ver la carga de trabajo que lleva cada uno de los usuarios.

1.4. Estructura

El presente proyecto se ha estructurado en bloques que representan el proceso habitual en el desarrollo de software:

- En este primer punto se ha explicado las motivaciones que han provocado la realización de la aplicación, los objetivos marcados y el impacto que tiene que tener en los usuarios.
- El segundo punto cita las diferentes aplicaciones que hay disponibles en el ámbito de la presente aplicación. Y finalmente se nombran que propone este proyecto frente a las anteriores.
- El tercer punto se explica la metodología de desarrollo de software que se ha seguido, así como el framework o conjunto de subsistemas en los que se ha desarrollado la aplicación.

- El cuarto punto se enfoca en el análisis del problema previo y se recopilan los requisitos que resuelven la problemática.
- El quinto punto explica la arquitectura que se ha seguido, así como el modelado de la base de datos (el conjunto de objetos que participan en la resolución del problema) y los prototipos de la interfaz.
- El sexto punto pasa a los detalles técnicos del desarrollo de la aplicación, como las tecnologías que se han usado para el desarrollo, en que se compone cada capa de la aplicación y como se ha implementado la integración continua.
- El séptimo punto detalla todas las pruebas implementadas para asegurar la robustez y la calidad del software desarrollado.

2. Estado del arte

Existen múltiples aplicaciones en el mercado para realizar la gestión de tareas, tanto para un solo individuo como para un grupo de usuarios. El problema que se observa en la mayoría de las aplicaciones es la falta de una gestión de tiempo orientada a lo que sería un proyecto real, es decir, en la necesidad de tener un seguimiento de los recursos (en este caso temporales, medidos en horas) que se espera que se le vaya a dedicar a una tarea, en la necesidad de tener una imagen de los recursos que se han aportado o se están aportando en cada una de las tareas y el reporte de las horas que le ha dedicado un usuario o un grupo de usuarios a una tarea concreta o las horas que ha invertido un usuario en sus distintas tareas.

Cabe señalar que actualmente las herramientas de gestión de tareas se pueden dividir en dos tipos según el sistema de trabajo:

- Método GTD "Get the things done": la organización de las tareas sigue un orden de cinco pasos, recopilar y organizar las tareas en contenedores, procesar las tareas según su prioridad, organizarlas según sus necesidades, revisarlas y ejecutarlas.
- Sistema Podomoro: establece tiempos para la ejecución de las tareas (estimaciones) y fechas de entrega según los objetivos que el gestor crea convenientes.ⁱ

2.1. Situación actual de la tecnología

En este apartado se describen varias aplicaciones dedicadas a la gestión de tareas que están en el mercado.

2.1.1. Remember The Milk

La primera aplicación es Remember The Milk. Se trata de un gestor de tareas enfocado al método GTD para organizar tarea.

Su principal característica es la integración con diversos entornos como: Gmail, Twitter, Hangouts, Skype. Todos estos se podrían vincular con Remember The Milk y poder enviar recordatorios de las tareas en marcha. También se puede disfrutar de esta aplicación en varios dispositivos, ya que cuenta con aplicación web, y aplicación para dispositivos móviles, tanto Smartphone como relojes.ⁱⁱ

2.1.2. Todoist

Probablemente sea la primera que se puede encontrar al buscar un gestor de tareas. Según los requerimientos del usuario se puede usar según el método GTD o el sistema Podomoro.

Las características más importantes son la capacidad de programar tareas repetitivas, por ejemplo, una tarea rápida que hacer cada día. Y el tablero Kanban que te permite tener una imagen clara de tu carga de trabajo.

Una funcionalidad que se echa de menos es la capacidad de estimar cuanto tiempo podría tardar un usuario a realizar una determinada tarea y la capacidad de poder ver el historial de tareas. Esta última funcionalidad solamente se encuentra en el plan de pago. Se puede decir que esta herramienta está más enfocada a la gestión personal de las tareas que a la gestión de un proyecto real.ⁱⁱⁱ

2.1.3. Trello

Trello es un gestor de tareas que usa el sistema Podomoro para organizar las tareas de un grupo. Su funcionamiento está basado en los paneles.

La principal ventaja es la capacidad de adjuntar o crear tareas en base a cualquier cosa, como una imagen, un correo electrónico que haya llegado pidiendo una determinada cosa etc. También cuenta con la herramienta de la creación de informes en los que se resume, usando gráficos de tarta, barras y lineales, las tareas que se han hecho en un determinado periodo de tiempo.

Aunque, a priori, parezca muy completa, se sigue echando en falta la capacidad de estimar el tiempo que se va a dedicar a cada tarea, no solo indicar la fecha de entrega y la elaboración de informes sobre las horas que han invertido los usuarios en una o varias tareas.^{iv}

2.1.4. Jira

Jira es un gestor de tareas que sigue el sistema Podomoro y es de los más completos.

Jira es una herramienta en línea para la gestión de tareas de un proyecto, el seguimiento de sus errores e incidencias y para la gestión operativa de proyectos.

Su principal característica es su organización de flujos de trabajo. Esta característica permite que una organización pueda definir el flujo que tiene que seguir una tarea, y actuar en esa tarea según su criterio, definiendo unas actuaciones en cada una de sus fases.

Tiene las principales características que las otras aplicaciones: creación de paneles, notificaciones a correo, gestión de las fechas de entrega, gestión de prioridades y gestión de usuarios del proyecto.

Además de poder usar la herramienta como un gestor de tareas en un proyecto, se puede enfocar el uso de la herramienta a un ámbito de soporte. Ya que permite la creación de incidencias por parte de los usuarios. Estas incidencias serán tratadas como tareas, pero tendrán un solicitante y en todo momento se puede contactar directamente con éste desde la propia herramienta.

A diferencia de los anteriores gestores, este sí que cuenta con la capacidad de poder estimar los recursos que se le va a dar a una tarea, asignar los recursos que se les va dando y hacer un informe al finalizar la tarea o en cualquier momento del proyecto.

La única pega es la complejidad, pero queda ensombrecida por la gran cantidad de funcionalidades y soluciones que aporta la herramienta.

2.2. Crítica al estado del arte

Las aplicaciones que se han descrito anteriormente, y en general, la mayoría que existen en el mercado tienden a estar o bien muy enfocadas a la gestión de las tareas que lleva un único usuario o bien muy enfocadas a la gestión de las tareas de un proyecto o grupo de usuarios.

En el caso de las que están más enfocadas a la gestión de tareas, suelen seguir el método GDT, es decir, solamente hacen hincapié en el listado de tareas que quedan por hacer (ordenándolas por prioridad o por fechas de entrega). Con este enfoque se pierde por ejemplo las horas que lleva un usuario dedicadas a cada tarea o un indicador de cuánto tiempo le queda hasta que se llegan a las horas que se han acordado para dar por finalizada la tarea. Por otra parte, también se echa en falta la funcionalidad para obtener un informe de las horas que se han dedicado a cada una de las tareas en una semana o en un intervalo de tiempo determinado.

En el caso de las que están dirigidas a la gestión de las tareas de un proyecto se echa en falta la simplicidad para un usuario de ver únicamente sus tareas y tener una imagen clara de la situación de sus tareas en el proyecto. Las aplicaciones que mejoran esta parte con tableros Kanban o la utilidad de sacar informes de horas no las ofrecen en su plan gratuito.

En conclusión, se echa en falta la gestión de tareas que pueda necesitar una empresa, ya que en estas se tiene mucho en cuenta el poder dar una estimación al cliente del tiempo que va a necesitar una tarea. A todo esto, hay que sumarle la necesidad de poder obtener informes de horas usadas por parte de los trabajadores en cada tarea o en cada proyecto. Ya que esas horas son al final de lo que cobraría la empresa.

2.3. Propuesta

Como se ha especificado en la introducción, en este trabajo se pretende desarrollar una herramienta para la gestión de tareas tanto para las tareas individuales como para las tareas que llevan a cabo varios trabajadores en un proyecto. A diferencia de las aplicaciones que hay en el mercado, se va a enfocar la aplicación para una gestión de las tareas de un proyecto, pero sin perder las características que necesitaría un usuario individual a la hora de gestionarse él mismo sus tareas. Todo esto sin llevar al usuario a perder interés en la aplicación por la alta complejidad que pueda suponer, es decir, simplificarle la gestión. Para conseguir esto, primero se ha de listar lo que necesitaría un usuario gestor de proyecto y lo que necesitaría un usuario que trabajase en un proyecto.

Al ponerse en la piel del gestor, a parte de la capacidad para crear, editar, estimar en tiempo, eliminar y repartir las tareas a los usuarios, necesitaría tener la capacidad de ver en todo momento una imagen del proyecto, es decir, las tareas que hay en curso en cada estado, las tareas que están cerca de sobrepasar el tiempo que se ha estimado en ellas y las que ya se ha pasado. También podría filtrar todas las tareas por un usuario, para comprobar la carga de trabajo que está soportando. Por último, también necesitaría tener la capacidad de elaborar informes con las horas dedicadas al proyecto o a una tarea específica en un intervalo de tiempo determinado.

Al ponerse en la piel de un usuario normal, lo más importante sería poder ver el estado o la carga de trabajo que está soportando, así como saber la relación entre la cantidad de tiempo invertido actualmente en las tareas y sus estimaciones (al igual que lo necesitaba saber el gestor de proyecto). También habría que mejorar la forma en la que los usuarios añaden horas a las tareas, ya que añadirlas a mano podría ser un poco tedioso o una pérdida de tiempo, se había pensado en que el tiempo se añadiese automáticamente a las tareas. Es decir, cuando el usuario mueva una tarea al estado de “en desarrollo”, se iniciará un temporizador con el tiempo que le va dedicando. Para flexibilizar este tema, el usuario podría editar siempre que quisiese el tiempo dedicado a las tareas. Además, podría obtener informes de las horas que ha dedicado a las tareas en un intervalo de tiempo determinado.

También se va a hacer que toda la aplicación se mueva en un entorno lo más ágil posible. Para ello se ha elegido crear una SPA ya que, aunque la carga inicial de la aplicación es más costosa, el cambio entre interfaces es mucho más rápido que las páginas webs antiguas, dando una experiencia más agradable al usuario.

2.4. Conclusiones

El mercado ya cuenta con una gran variedad de aplicaciones dedicadas a la gestión de tareas. Se han descrito las dos tipologías en las que se podrían clasificar y se han enumerado unos ejemplos de las aplicaciones más populares. Se han descrito sus ventajas y sus inconvenientes para ver que se puede aportar a este nicho.

Lo que se va a aportar es una aplicación más centrada a la gestión de los tiempos, tanto del tiempo dedicado a las diferentes tareas como el tiempo que se cree que va a costar desarrollar una tarea. Todo esto sin perder al usuario en complejidades y dándole una experiencia ágil y simple.

3. Metodología

Este apartado se centrará en describir el proceso de desarrollo de software que se ha seguido en este proyecto. Se explicará que metodología y enfoque se han usado, en qué consisten y el porqué de su elección. Al mismo tiempo se explicarán apartados más técnicos como son la elección del framework de desarrollo, el sistema de control de versiones y la parte de la integración continua y su utilidad respecto a la metodología y el enfoque que se han seguido.

3.1. Metodología

La metodología de desarrollo de software se refiere al entorno de trabajo que se usa para estructurar, planear y controlar el desarrollo del producto de software y que consisten en un enfoque sobre el proceso en sí del desarrollo y las herramientas, modelos y métodos que asisten al proceso de desarrollo de software.^v

De entre todos los modelos de desarrollo, este proyecto se centra en el desarrollo ágil del software, más concretamente en la metodología Scrum.

El desarrollo ágil del software se basa en el desarrollo iterativo e incremental, que en gran medida consiste en repartir el proceso de desarrollo en varias etapas repetitivas (iteraciones). En cada etapa se añaden una porción de las funcionalidades respecto de la totalidad de los requerimientos del proyecto. Se explicará en más profundidad el desarrollo iterativo e incremental en el siguiente apartado.^{vi}

Una de las principales características del desarrollo ágil del software es el ciclo de vida de cada iteración, que incluye la planificación, análisis de requisitos, diseño, codificación, pruebas y documentación. Cabe mencionar la importancia de finalizar las tareas en cada fase iterativa, ya que el objetivo de esta metodología no es lanzar el producto con toda su funcionalidad, sino ir lanzando versiones del producto finalizadas en cada iteración. Entendiendo finalizadas como versiones funcionales y sin errores, de ahí la importancia de la fase de pruebas en cada iteración.

Con todo esto en mente, la metodología ágil que se ha aplicado en este proyecto es la metodología Scrum.

Se trata de un marco de trabajo sobre la metodología de desarrollo ágil en la que se aplican un conjunto de buenas prácticas para trabajar colaborativamente y obtener el mejor resultado posible. Se basa y se caracteriza por:

- Estrategia de desarrollo incremental. Muy diferente a la planificación y ejecución completa del producto. En cada fase se muestra el resultado al cliente, para que este tome las decisiones necesarias.
- Se solapan las distintas fases del desarrollo, en lugar de realizarlas secuencialmente o en cascada. No se espera que finalice una fase para empezar otra.

- Desarrollo incremental de los requisitos totales del proyecto en bloques temporales cortos y fijos. Dando prioridad al que tiene más valor para el cliente.
- El equipo de trabajo (normalmente pequeños, de 3 a 9 personas cada uno) se sincroniza diariamente y se realizan las adaptaciones o modificaciones pertinentes (Scrum diario).

El proceso de un proyecto con una metodología Scrum constaría de varias fases.

En la primera fase de obtendrían los requisitos de toda la aplicación. Es decir, se definirían las funcionalidades que tendría, a priori, sin realizar aún ninguna iteración del proceso, la aplicación final. De todos estos requisitos se crearían tareas a realizar que se dividirían en diferentes Sprint según las funcionalidades que se va a ir entregando en las diferentes fases o Sprint. Cabe destacar que las longitudes temporales de los Sprint no deberían superar el mes.

Una vez definidos y planificados los diferentes Sprint con sus diferentes tareas, obtenidas de los requisitos iniciales, el equipo se reunirá para empezar el primer sprint. Cada día se acuerda una reunión llamada Scrum diario, que tiene como objetivo que los diferentes miembros del equipo se mantengan actualizados unos a otros sobre el trabajo de cada uno desde la última reunión, qué problemas han encontrado o prevén encontrar, que hacer y hablar sobre las distintas soluciones que pueden ir aportando los demás miembros del equipo, o discutir sobre las integraciones que puede haber entre las diferentes partes del proyecto.

Al finalizar un sprint, el equipo realiza dos eventos, la revisión del sprint, donde se presenta el trabajo realizado, y la retrospectiva del sprint, donde los miembros dejan sus impresiones del sprint superado y se proponen mejoras sobre el proceso.

Los beneficios observados a la hora de trabajar con esta metodología son los siguientes:

- Capacidad de reacción a los cambios. Al ser un proceso iterativo de un mes, en el que cada mes se entrega una versión al cliente, se tiene el beneficio de obtener rápidamente su opinión sobre el producto presentado y si el producto se ajusta a los requerimientos obtenidos inicialmente. Si hay algunas variaciones con lo que realmente necesita el cliente final no supone un gran problema, ya que aún no se ha entregado la versión definitiva del producto y adaptarlo sería más fácil. Además, al tener una reunión diaria con el equipo de trabajo, se pueden obtener diferentes puntos de vista que pueden ayudar a ver si se está desviando del requerimiento inicial o poder detectar si un requerimiento está mal tomado y pedir una reunión con el cliente final. Pudiendo así detectar de una manera rápida cualquier duda entre el requerimiento que se ha obtenido inicialmente y lo que realmente quería el cliente.
- El cliente puede obtener una versión inicial de la aplicación antes de que esté totalmente terminado. Todo esto en el entorno final en el que

se quedará la aplicación final, por lo que se evita el costoso y la arriesgada subida a producción de un producto final con todas sus funcionalidades. Así mismo, en cada iteración se realizan pruebas de cada nueva funcionalidad.

- Se logra una mayor productividad del equipo, ya que, entre otras razones, se elimina la burocracia y el equipo puede estructurarse de manera autónoma.
- Se puede llegar a conocer con cierto margen la velocidad del equipo en cada sprint, con lo que se consigue poder decir al cliente una predicción de los tiempos en el que estará cada iteración y cada nueva funcionalidad.
- Divide y vencerás. Con esta metodología se puede estructurar y dividir el proyecto final en diferentes fases claramente separadas por la funcionalidad que representan. Esto consigue no saturar a los miembros del equipo con toda la información de un proyecto completo e ir centrándose en la porción de funcionalidades que representa cada sprint.^{vii}

En este proyecto, aun no siendo un equipo completo de personas, sí que se ha podido experimentar los beneficios de esta metodología referentes a la capacidad de ir lanzando versiones del proyecto de una forma incremental y obtener una mejora en la productividad al dividir todo el total de tareas en diferentes iteraciones. En el caso de las reuniones como los Scrum diarios, se ha sustituido por una revisión de las tareas que había pendientes en ese sprint y ver si las soluciones que se iban implantando satisfacían lo esperado en un primer momento.

3.2. Enfoque

En este apartado se va a explicar en qué consiste el enfoque que se ha dado a la metodología de desarrollo de software seguida, porque se ha seguido y los beneficios obtenidos.

El enfoque que se ha dado es el desarrollo iterativo e incremental. Se basa en dividir las tareas agrupándolas en pequeñas etapas repetitivas (iteraciones). El objetivo de cada etapa es sumar una serie de funcionalidades ya planificadas anteriormente al producto final (incremental), así, el punto de vista del cliente es de una primera entrega con lo que más necesita, y una sucesión de entregas que van ampliando la funcionalidad de la aplicación a lo que se acordó inicialmente. Todo esto con las diferentes modificaciones que proponga el cliente en cada etapa.

Este proceso se divide en dos etapas principales:

- Etapa de inicialización: Se presenta una versión inicial de la aplicación. La meta de esta etapa es crear un producto con el que el cliente pueda interactuar y retroalimentar todo el proceso que seguirá.
- Etapa de iteración: El análisis de una iteración se basa en la retroalimentación por parte del cliente y en el análisis de las funcionalidades de las que dispone la aplicación.^{viii}

3.3. Framework

Este apartado tiene como objetivo explicar el entorno de trabajo elegido para el desarrollo de la aplicación. En el contexto del desarrollo de software, el framework es el entorno de trabajo que sirve como base para la organización y desarrollo de software. Suele incluir una serie de programas y librerías, que, actuando conjuntamente facilitan llegar al objetivo. Más tarde, se explicará también el control de versiones y la integración continua.

En este proyecto se ha elegido como entorno de trabajo el MEAN Stack (acrónimo para: MongoDB, ExpressJS, Angular 2, NodeJS). Se trata de un framework que tiene como objetivo la creación de páginas webs dinámicas basándose en el lenguaje de programación JavaScript. Cada subsistema del MEAN Stack es de código abierto y gratuito. Está compuesto por los siguientes elementos:

- **MongoDB:** Se trata de un sistema de base de datos NoSQL. La principal característica es que este sistema no usa SQL como lenguaje principal de consultas ni requiere una estructura fija de tablas. Tiene una estructura en documentos JSON y una baja cantidad de restricciones y relaciones en comparación con un sistema SQL, todo esto le aporta una mayor velocidad y escalabilidad en las operaciones de lectura, pero con el coste de aumentar la gestión y coste de las operaciones de escritura, ya que, al poder tener datos duplicados en otros documentos, al modificarlos habría que modificarlos en todos los otros objetos.
- **NodeJS:** Es un entorno en tiempo de ejecución asíncrona multiplataforma basado en el lenguaje de programación JavaScript. Fue creado con el objetivo de ser útil en la creación de servidores altamente escalables.
- **ExpressJS:** Se trata de un módulo básico que se puede incorporar a un servidor NodeJS que proporciona la capacidad de facilitar y manejar las diferentes peticiones HTTP.
- **Angular 2:** Angular es un framework para el desarrollo de aplicaciones web desarrollada en TypeScript (hereda de JavaScript añadiendo la gestión de objetos). Su objetivo es facilitar el desarrollo de aplicaciones dinámicas de una página con capacidad de Modelo Vista Controlador, así como facilitar las pruebas. Angular está basado en componentes, este elemento es el principal con el que se trabajará, estos componentes incluyen una parte de la vista total de la aplicación y está ligada a una parte de lógica, en la que se tratarán todos los datos. Están pensadas para ser reutilizadas a lo largo de toda la aplicación. También cuenta con unos elementos llamados servicios, que contienen solamente partes lógicas que se encargan de la obtención, administración y gestión de datos. En base, son los que proveen a los componentes de datos, mientras que los componentes son los encargados de cómo mostrarlos y cómo manejarlos interactuando con otros componentes.

Se ha elegido este entorno de trabajo porque es uno de los más populares actualmente, es propiedad de Google, por lo que tiene un gran soporte. A parte de esto, cubre todo el ciclo completo de desarrollo utilizando únicamente JavaScript, admite la

arquitectura Modelo Vista Controlador para que el proceso fluya sin problemas, la estructura del proyecto en componentes y servicios mantiene el desarrollo y el mantenimiento muy organizado, tiene una extensa cantidad de herramientas, vistas y librerías ya hechas y está integrado con varias herramientas de pruebas, tanto para pruebas unitarias como para pruebas E2E.

3.4. Control de versiones

Otra de las partes imprescindibles en todo proyecto software, es el sistema de control de versiones que se emplee. El sistema de versiones ofrece la capacidad de gestionar los diversos cambios que se realizan, quien las ha realizado y que se ha modificado exactamente. Así como la capacidad de poder volver hacia atrás en cualquier momento. También crear, gestionar y mezclar ramas del desarrollo, con lo que se facilita bastante el proceso de añadir una mejora o solucionar un error a una versión concreta ya desplegada y funcional. En este caso el desarrollador solamente tendría que crear una rama en base a otra, trabajar en esa rama que ha creado, testearla para asegurarse de que no exista ningún error y mezclar su rama con la de destino. Al hecho de mezclar se le conoce comúnmente como *merge* cuando se mandan cambios a la rama principal, o *pull request* si se obtienen los cambios de la rama original a la que se ha desarrollado. Más adelante se explicará la parte de la integración continua y su implicación en esta parte concreta del proceso de desarrollo.

En este proyecto se ha elegido Git como gestor de control de versiones por la compatibilidad con la plataforma online y gratuita GitHub. GitHub es una plataforma que permite el alojamiento de proyectos con sistema de gestión de versiones Git. Una de las principales características es su integración con varios sistemas que se encargan de la integración continua que se explicará más adelante.

Cabe mencionar que se ha usado una herramienta para la gestión de las tareas y los Sprint llamada ZenHub. Se trata de una aplicación web que permite gestionar las tareas o agrupaciones de tareas (Sprint) de un repositorio en GitHub. Las tareas creadas son interpretadas como incidencias de GitHub, y se pueden añadir comentarios a cada tarea y comentar un enlace del *commit* para que al pulsar se mueva directamente a los cambios que se han hecho.

3.5. Integración/distribución continua

La integración/distribución continua es un método para distribuir aplicaciones a los clientes con una frecuencia más alta gracias al uso de la automatización de tareas en las etapas del desarrollo de aplicaciones. Se trata de una solución para los problemas que pueda generar la integración y distribución de las nuevas funcionalidades que se van desarrollando y su puesta en marcha en los equipos de operaciones. Concretamente, estas herramientas incorporan la automatización y el control permanente en todo el ciclo de vida de las aplicaciones, desde las etapas de integración y pruebas hasta las de distribución e implementación.^{ix}

Ha sido interesante añadir estas herramientas o esta metodología al proceso de desarrollo porque en proyectos grandes donde participen un número elevado de personas (incluso de empresas diferentes), el proceso de hacer unos cambios al código,

o añadir nuevas funcionalidades y después hacer las pruebas unitarias, de integración y de E2E a mano, junto con la fusión con la rama principal del producto y luego su distribución a los equipos reales puede llegar a ser muy tediosa. Con estas herramientas se permite que una vez se realiza una fusión con la rama principal del código, automáticamente se ejecuten todas las pruebas del código pertinentes (que deberían estar bien desarrolladas) y si han sido satisfactorias, se distribuye el código automáticamente.

Para implementar la parte de la integración/distribución continua se ha usado la herramienta GitHub Actions. Esta herramienta permite automatizar, personalizar y ejecutar los flujos de trabajo que se crean convenientes. Todo esto directamente en el repositorio de GitHub. Se analizará más detenidamente en el apartado de implementación, pero por ahora basta con saber que cada trabajo que se defina en el archivo de la acción se ejecuta cuando se le indique (un *push*, un *pull request*, un *merge* a la rama que se le indique) en una máquina remota que inicia GitHub en el momento de ejecución y ejecuta los comandos que se definan. Comandos que podrían ser por ejemplo hacer un *build* a la aplicación y ejecutar una serie de pruebas y luego subir al entorno definido.

Como se ha mencionado, las GitHub Actions permiten subir la aplicación donde se quiera, pero en este proyecto se ha optado por la plataforma Heroku. Esta permite alojar la aplicación en la nube sin ningún coste, y tiene la característica de que puede estar ligada a un repositorio de GitHub, por lo que cuando las GitHub Actions ejecuten las pruebas con resultado satisfactorio, se haría la fusión de código, por lo que Heroku lo detectaría y lo subiría a la plataforma.

4. Análisis y especificaciones de requisitos

Después de hacer una propuesta de una aplicación que mejore un aspecto concreto de lo que flaquean las demás en el mercado, se procede a realizar el análisis del problema. En este capítulo se van a recopilar las necesidades de la aplicación, así como definir las acciones que pueden hacer los diferentes tipos de usuarios que tendrá.

Se trata de una de las fases más importantes en el desarrollo de la aplicación, ya que si se falla en la interpretación de lo que querrá el usuario final o en la relevancia de una funcionalidad específica después será mucho más costoso de arreglar cuando se tenga ya desarrollado el código. También es de gran ayuda dejar todos los requisitos y todos los posibles flecos claro en esta etapa, ya que, con todo esto claro, la complejidad del desarrollo se aligera bastante.

Primero se definen los diferentes actores o tipos de usuarios que van a interactuar con la aplicación. Se explican que roles tienen y luego se definen los casos de uso.

4.1. Actores

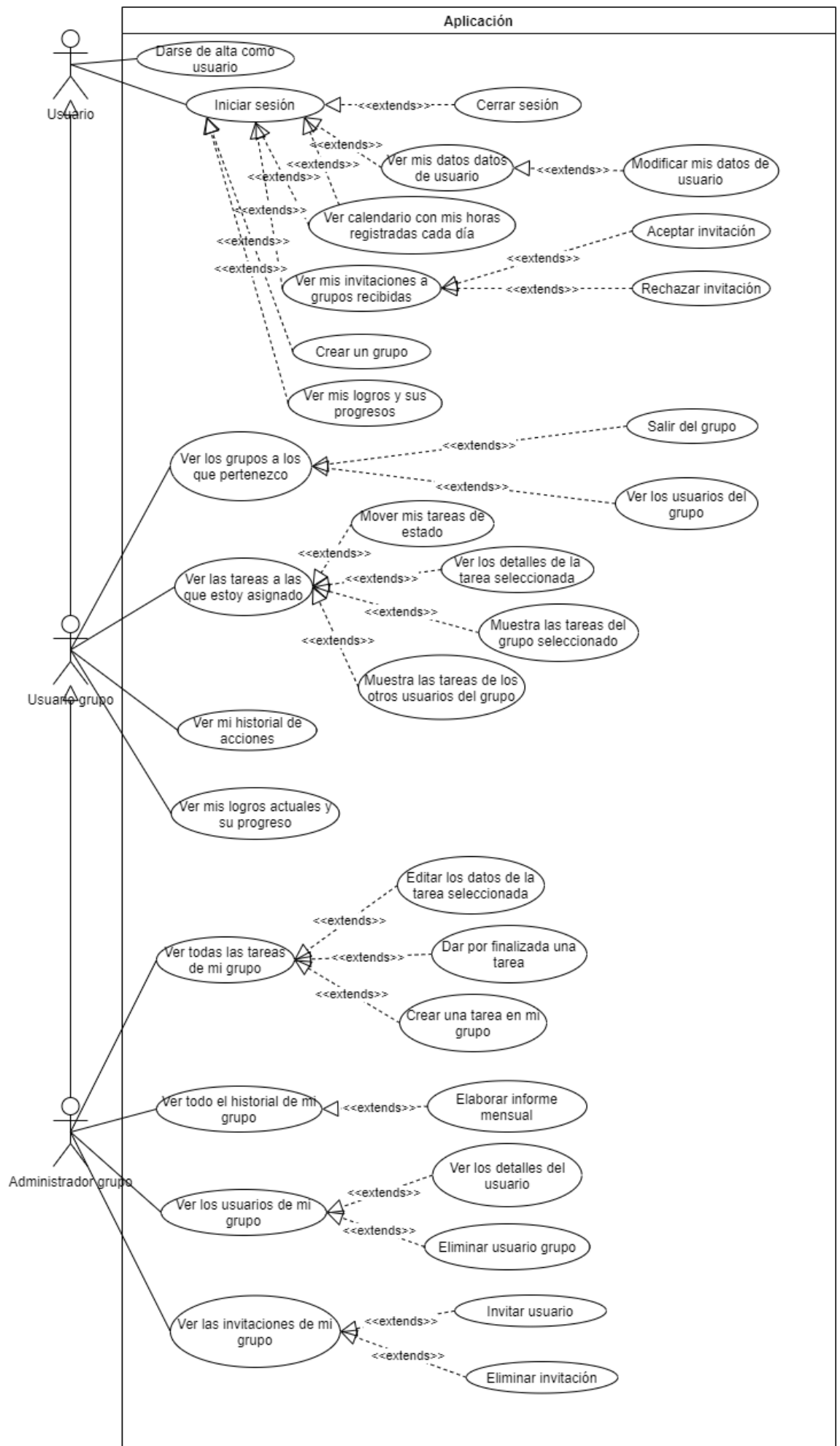
Los actores son los diferentes tipos de usuario que van a interactuar con la aplicación, y según sean de un tipo u otro, podrán hacer unas determinadas acciones. Los actores que se han definido son los siguientes:

- **Usuario:** Se trata de un usuario que entra a la aplicación por primera vez. En este caso únicamente podría registrarse, crear un grupo o aceptar una invitación de un grupo. Se entiende como grupo un equipo de trabajo en un proyecto determinado.
- **Usuario grupo:** Es el usuario que ya ha aceptado una invitación a un grupo y por lo tanto pertenece a un grupo. La idea es que pueda gestionar sus propias tareas, mover de estado sus tareas y ver las tareas de los demás usuarios del grupo. Este usuario hereda todas las acciones que puede realizar un usuario normal.
- **Administrador grupo:** Es un mánager de proyecto. Se trata de un usuario registrado que ha creado un grupo. Puede crear, editar, eliminar y asignar tareas a usuarios. Puede ver el estado de todas las tareas y puede generar informes de horas en cualquier momento. Este usuario hereda todas las acciones que puede hacer un usuario de grupo.

4.2. Casos de uso

Un caso de uso es una descripción de una acción que puede realizar un usuario sobre el sistema. El diagrama de casos de uso representa un sistema con un conjunto de interacciones que se desarrollarán entre el usuario y el sistema.

El diagrama obtenido es el siguiente, donde se reflejan los actores a la izquierda, fuera del rectángulo que representa el sistema. Las acciones están dentro de los óvalos y las líneas y flechas representan la interacción entre el usuario y el caso de uso. Las líneas continuas representan una asociación o comunicación y las discontinuas representan la opcionalidad, es decir, el usuario podría hacer o no la acción.



A continuación, después de ver una imagen general de cómo se relacionarían los usuarios con el sistema, se define cada caso de uso:

Caso de uso	Darse de alta como usuario
Actores	Usuario
Propósito	Registrarse como usuario de la aplicación
Resumen	El usuario rellena un formulario con los siguientes datos: Nombre, apellido, correo y contraseña. Si todo es válido, al pulsar en aceptar se registra como usuario
Precondiciones	Pulsar en registrar en la pestaña inicial de la aplicación
Postcondiciones	El usuario queda registrado en la aplicación

Caso de uso	Iniciar sesión
Actores	Usuario
Propósito	Autenticarse para poder acceder a la aplicación
Resumen	El usuario puede rellenar su correo y su contraseña para acceder a la aplicación
Precondiciones	-
Postcondiciones	Si el usuario está registrado con esas credenciales, accede a la pantalla principal, si no, se le muestra un mensaje con que no existe

Caso de uso	Cerrar sesión
Actores	Usuario
Propósito	Cerrar la sesión que había iniciado
Resumen	Un usuario que ya se haya autenticado, puede cerrar sesión en todo momento
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	Se finaliza la sesión del usuario

Caso de uso	Ver mis datos de usuario
Actores	Usuario
Propósito	Ver los datos con los que el usuario se ha registrado
Resumen	Un usuario puede ver los datos con los que se registró (nombre, apellido, correo)
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	-

Caso de uso	Modificar mis datos de usuario
Actores	Usuario
Propósito	Editar los datos con los que el usuario se ha registrado
Resumen	Un usuario puede editar los datos con los que se registró

Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	Los datos del usuario quedan modificados

Caso de uso	Ver calendario con mis horas registradas cada día
Actores	Usuario
Propósito	Ver las horas propias dedicadas a las tareas en una vista de calendario
Resumen	Un usuario puede ver las horas que ha ido dedicando a cada tarea en una vista de calendario. En un principio se muestra el mes actual, pero puede navegar por todos los años y meses
Precondiciones	El usuario debe haber hecho horas en alguna tarea
Postcondiciones	-

Caso de uso	Ver mis invitaciones a grupos recibidas
Actores	Usuario
Propósito	Ver las invitaciones recibidas de otros grupos
Resumen	Un usuario puede ver las invitaciones que otros grupos le han hecho para acceder a ellos.
Precondiciones	Un grupo debe invitar al usuario
Postcondiciones	-

Caso de uso	Aceptar invitación
Actores	Usuario
Propósito	Aceptar una invitación de un grupo
Resumen	Un usuario puede aceptar la invitación de un grupo para inscribirse en él
Precondiciones	Un grupo debe invitar al usuario
Postcondiciones	El usuario pertenece al grupo aceptado y la invitación se elimina

Caso de uso	Rechazar invitación
Actores	Usuario
Propósito	Rechazar una invitación de un grupo
Resumen	Un usuario puede rechazar la invitación de un grupo
Precondiciones	Un grupo debe invitar al usuario
Postcondiciones	La invitación se elimina

Caso de uso	Crear un grupo
Actores	Usuario
Propósito	Crear un grupo
Resumen	Un usuario puede crear un grupo y

	automáticamente se convierte en el administrador de ese grupo
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	El usuario pertenece a su grupo como administrador de grupo

Caso de uso	Ver mis logros y sus progresos
Actores	Usuario
Propósito	Ver los logros obtenidos y los progresos acumulados
Resumen	Un usuario puede ver los logros y los progresos de cada logro. Los progresos se aumentan al hacer acciones y los logros se obtienen al llegar al 100% de progreso
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	-

Caso de uso	Ver los grupos a los que pertenezco
Actores	Usuario grupo
Propósito	Ver los grupos a los que pertenece el usuario
Resumen	Un usuario puede ver los grupos a los que pertenece
Precondiciones	El usuario debe haber iniciado sesión
Postcondiciones	-

Caso de uso	Salir del grupo
Actores	Usuario grupo
Propósito	Salir del grupo seleccionado
Resumen	Un usuario de un grupo puede salir del grupo en todo momento
Precondiciones	El usuario debe estar en el grupo
Postcondiciones	El usuario ya no se encuentra en el grupo

Caso de uso	Ver los usuarios del grupo
Actores	Usuario grupo
Propósito	Ver los usuarios de un grupo
Resumen	Un usuario puede ver los usuarios que hay en el grupo al que pertenece
Precondiciones	El usuario debe estar en el grupo
Postcondiciones	-

Caso de uso	Ver las tareas a las que estoy asignado
Actores	Usuario grupo
Propósito	Ver las tareas que el usuario tiene asignadas
Resumen	Un usuario puede ver las tareas a las que está asignado y el estado en el que están

	en un tablero Kanban
Precondiciones	El usuario debe estar en el grupo
Postcondiciones	-

Caso de uso	Mover mis tareas de estado
Actores	Usuario grupo
Propósito	Mover la tarea de estado
Resumen	El usuario puede mover la tarea de estado
Precondiciones	El usuario debe estar en el grupo y tener tareas asignadas
Postcondiciones	La tarea cambia de estado y de columna en el tablero Kanban

Caso de uso	Ver los detalles de la tarea seleccionada
Actores	Usuario grupo
Propósito	Ver los detalles de la tarea (descripción, horas estimadas, fecha de entrega, horas hechas)
Resumen	El usuario puede pulsar en una tarea que tenga asignada y ver los datos de esa tarea
Precondiciones	El usuario debe estar en el grupo y tener tareas asignadas
Postcondiciones	Se muestran los detalles de la tarea

Caso de uso	Muestra las tareas del grupo seleccionado
Actores	Usuario grupo
Propósito	Ver las tareas totales que tiene el grupo
Resumen	El usuario puede ver todas las tareas que hay en el grupo
Precondiciones	El usuario debe estar en el grupo
Postcondiciones	-

Caso de uso	Ver mi historial de acciones
Actores	Usuario grupo
Propósito	Ver el historial de horas en tareas
Resumen	El usuario puede ver el historial de sus horas dedicadas a cada tarea y filtrarlo por intervalo de tiempo, grupo y tarea
Precondiciones	El usuario debe estar en el grupo
Postcondiciones	-

Caso de uso	Editar los datos de la tarea seleccionada
Actores	Administrador grupo
Propósito	Editar los datos de una tarea seleccionada
Resumen	El administrador del grupo puede seleccionar una tarea y editar sus datos

Precondiciones	Ser administrador de grupo y seleccionar una tarea
Postcondiciones	Se registran los cambios en la tarea

Caso de uso	Dar por finalizada una tarea
Actores	Administrador grupo
Propósito	Finalizar el ciclo de una tarea
Resumen	El administrador puede dar por finalizada la tarea que desee
Precondiciones	Ser administrador de grupo y seleccionar una tarea
Postcondiciones	La tarea se finaliza

Caso de uso	Crear una tarea en mi grupo
Actores	Administrador grupo
Propósito	Crear una tarea en el grupo
Resumen	El administrador puede rellenar el formulario de creación de la tarea y crear la tarea
Precondiciones	Ser administrador de grupo
Postcondiciones	Se crea la tarea

Caso de uso	Ver todo el historial de mi grupo
Actores	Administrador grupo
Propósito	Ver el historial de todas las horas en tareas
Resumen	El administrador de grupo puede ver el historial de todas las horas dedicadas a cada tarea y filtrarlo por intervalo de tiempo, grupo y tarea
Precondiciones	Ser administrador de grupo
Postcondiciones	-

Caso de uso	Elaborar informe mensual
Actores	Administrador grupo
Propósito	Elaborar informe en Excel de las horas mensuales
Resumen	El administrador de grupo puede generar un informe de las horas del grupo, o de una tarea determinada en un intervalo de tiempo determinado en formato Excel
Precondiciones	Ser administrador de grupo
Postcondiciones	Genera un informe que se descarga

Caso de uso	Ver los usuarios de mi grupo
Actores	Administrador grupo
Propósito	Ver los usuarios que hay en el grupo
Resumen	El administrador del grupo puede ver los usuarios que hay en el grupo

Precondiciones	Ser administrador del grupo
Postcondiciones	-

Caso de uso	Ver los detalles del usuario
Actores	Administrador grupo
Propósito	Ver los detalles del usuario seleccionado del grupo
Resumen	El administrador del grupo puede ver los detalles de los usuarios del grupo
Precondiciones	Ser administrador de grupo y seleccionar un usuario
Postcondiciones	-

Caso de uso	Eliminar usuario grupo
Actores	Administrador grupo
Propósito	Eliminar a un usuario de un grupo
Resumen	El administrador del grupo puede eliminar a cualquier usuario del grupo
Precondiciones	Ser administrador de grupo y seleccionar un usuario
Postcondiciones	El usuario se elimina del grupo

Caso de uso	Ver las invitaciones de mi grupo
Actores	Administrador grupo
Propósito	Ver las invitaciones que hay pendientes del grupo
Resumen	El administrador puede ver las invitaciones que se han lanzado y están pendientes de respuesta por parte del usuario invitado
Precondiciones	Ser administrador de grupo
Postcondiciones	-

Caso de uso	Invitar usuario
Actores	Administrador grupo
Propósito	Invitar a un usuario al grupo
Resumen	El administrador puede invitar a un usuario al grupo introduciendo un correo
Precondiciones	Ser administrador de grupo
Postcondiciones	Se crea una invitación al usuario con ese correo

Caso de uso	Eliminar invitación
Actores	Administrador grupo
Propósito	Eliminar una invitación
Resumen	El administrador puede eliminar una invitación hecha
Precondiciones	Ser administrador de grupo y tener invitaciones hechas

Postcondiciones	La invitación se elimina
------------------------	--------------------------

4.3. Requisitos no funcionales

Los requisitos no funcionales se refieren a todos los requisitos que no describen las funciones descritas en los requisitos funcionales, entendiendo como función un conjunto de entradas, comportamientos y salidas. Se centran más en las características que complementan el funcionamiento que en el funcionamiento específico de la aplicación. A continuación, se detallan los requisitos funcionales:

- **Interfaz de usuario:** La aplicación debe presentar una interfaz intuitiva, dinámica y sencilla. Debe ofrecer un tablero Kanban de las tareas a los usuarios. Y la aplicación debe estar implementada como una SPA para que sea lo más dinámica posible.
- **Adaptabilidad:** La interfaz de la aplicación debe adaptarse a los posibles formatos de pantalla de los diferentes dispositivos. Como móviles, tabletas y ordenadores.
- **Feedback:** La aplicación debe retornar una señal, en forma de mensaje o en forma de actualización en la interfaz, cada vez que el usuario realice una acción.
- **Disponibilidad:** El sistema debe estar disponible las 24 horas de todos los días del año.
- **Compatibilidad de navegadores:** La aplicación debe funcionar exactamente en todos los navegadores.
- **Tiempo de peticiones HTTP:** El tiempo de cada petición que se haga a la API REST no debe superar 1 segundo.

5. Diseño

Una de las fases del ciclo por el que tiene que pasar el desarrollo de software es la etapa de diseño. En esta etapa se tiene que llegar a una planificación o diseño de las soluciones que tienen que resolver los requisitos definidos en el apartado anterior. Cabe destacar el concepto de diseño, ya que no se trata de soluciones ya codificadas sino de modelar una solución sin llegar a implantarla, esta última fase se verá en el apartado de implementación.

Normalmente, la fase de diseño suele contar con estas etapas:

- **Análisis de requisitos:** En esta parte se extraen los objetos y sus relaciones que debe tener la aplicación para satisfacer los requisitos.
- **Modelado de datos:** El modelado de datos es el proceso de documentar la forma en que los datos necesitan fluir en un sistema de software usando un diagrama de fácil comprensión.
- **Diseño de la arquitectura:** La arquitectura de software es el diseño de más alto nivel de la estructura en un proyecto software. Consiste en un conjunto de elementos y abstracciones que proporcionan un marco claro para interactuar con el código fuente del software.
- **Diseño de la interfaz:** El diseño de la interfaz de usuario tiene como objetivo definir la forma, función, utilidad y ergonomía para solucionar los requisitos definidos anteriormente ofreciéndole la máxima usabilidad al usuario.

5.1. Análisis de requisitos

De los casos de uso se pueden obtener los objetos que van a ser necesarios y que va a gestionar la aplicación. Se puede ver que hay un caso de uso en el que un usuario debe registrarse para poder acceder a la aplicación, por lo tanto, el objeto del tipo usuario es necesario.

Por otro lado, están los casos de uso que tienen que ver con las invitaciones, pues un usuario debe poder aceptar o rechazar una invitación y un usuario administrador del grupo debe poder enviar una invitación. Por lo tanto, de estos casos de uso se obtienen dos tipos de objetos, el de las invitaciones y el de los grupos. Estos dos objetos se deben relacionar con el objeto usuario, pues un usuario puede estar inscrito a varios grupos y en una invitación se invita a un usuario a pertenecer a un grupo. Por lo que un usuario puede tener ninguno o muchos grupos, un grupo puede tener ninguno o muchos usuarios y una invitación tiene que tener un usuario y un grupo, y un usuario puede tener varias invitaciones.

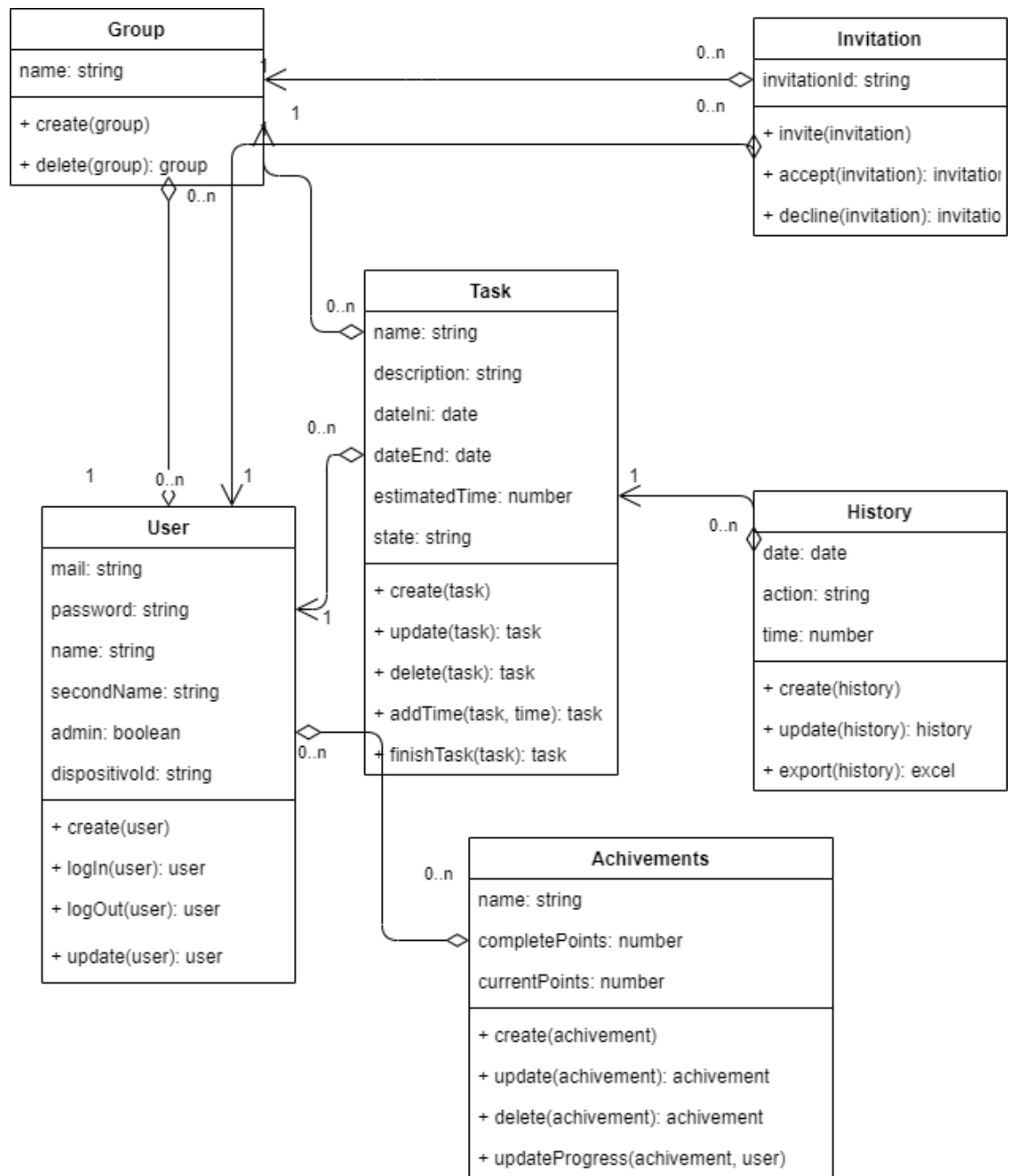
También hay que gestionar las tareas, ya que la mayoría de los casos de uso involucran el leer, la creación, actualización de estado y la eliminación o finalización de tareas. Como se puede obtener de los casos de uso, una tarea debe estar asignada a un usuario y pertenecer a un grupo, por lo que la tarea se relacionará con estos objetos,

con el grupo y con el usuario, de forma en la que un usuario pueda tener ninguna o muchas tareas, al igual que un grupo.

Así mismo, como ya se ha explicado en los casos de uso, todo usuario debe poder ver si historial de acciones sobre las tareas, por lo que cada entrada en el historial deberá ser gestionada por el objeto historial. Este historial deberá estar relacionada con el objeto tarea. De la forma en la que una tarea puede tener varios objetos del tipo historial y un objeto historial debe estar relacionado con únicamente una tarea.

Para finalizar, cada usuario puede tener una serie de logros, por lo que aquí se declara otro objeto, el objeto de logros. En el que un usuario puede tener varios logros, y un logro puede tenerlo varios usuarios.

En el siguiente diagrama de clases, se definen los atributos que tienen que tener los diferentes objetos, así como sus métodos y las relaciones con los otros objetos para satisfacer los requisitos obtenidos en los casos de uso:



5.2. Modelado de la base de datos

El sistema de información representa un gestor de tareas. También hay que tener en cuenta que los usuarios deben estar en un grupo, aunque el grupo sea solamente de una persona, para poder gestionar las tareas. Por lo que los objetos principales serán los usuarios, los grupos y las tareas. Por otro lado, la aplicación tiene que gestionar las invitaciones que se crean así que también hay que definir este objeto. Las tareas también pueden tener el estado como un objeto independiente, ya que en un futuro es posible que los usuarios creen diferentes estados y estos, en el momento de creación de la tarea puedan asignarse, por lo que también se creará el objeto estado de la tarea. También habría que definir el historial de acciones que un usuario lleva a cabo en una tarea determinada y los objetos de los logros que están disponibles para que

adquieran los usuarios. Por último, habría que crear un registro con el identificador del dispositivo o dispositivos que usa el usuario, esto será de gran utilidad a la hora de enviar notificaciones a su dispositivo.

Los objetos que se han definido anteriormente deberían tener los siguientes atributos para satisfacer las necesidades de los usuarios:

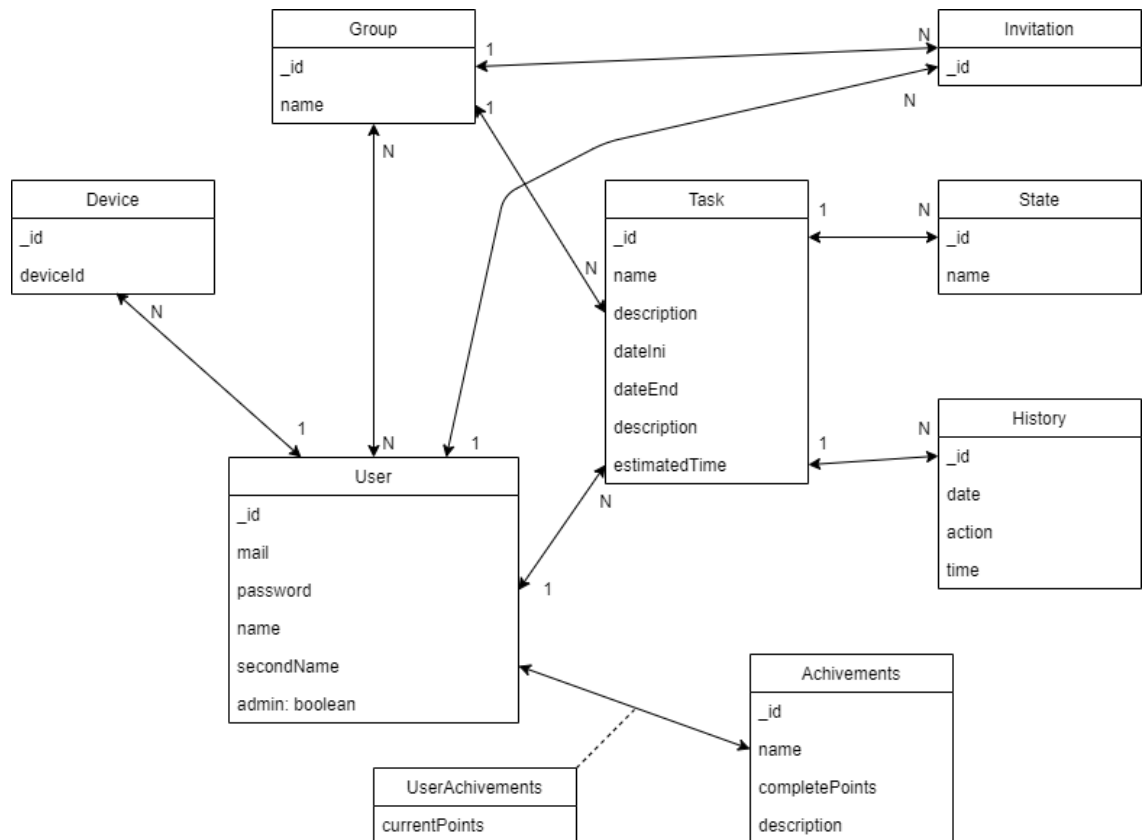
- Grupo
 - Identificador único
 - Nombre
 - Usuarios
 - Tareas
- Usuario
 - Identificador único
 - Correo
 - Nombre
 - Apellidos
 - Contraseña
 - Administrador
 - Grupos
 - Tareas
 - Invitaciones
- Tarea
 - Identificador único
 - Nombre
 - Descripción
 - Grupo
 - Usuario
 - Fecha de inicio
 - Fecha fin
 - Estado
 - Tiempo estimado
 - Fecha fin estimada
 - Tiempo imputado
 - Historiales
- Invitación
 - Identificador único
 - Grupo invitado
 - Usuario invitado
- Historial acciones
 - Identificador único
 - Tarea
 - Usuario
 - Grupo
 - Fecha
 - Acción
 - Tiempo de la acción
- Dispositivo
 - Identificador único

- Usuario
- Logros
 - Identificador único
 - Nombre
 - Descripción
 - Puntos para completar

Los objetos se relacionan con los otros de la siguiente forma:

- Grupo
 - Un grupo contiene diferentes usuarios y al menos debe contener uno, que será el usuario administrador de grupo.
 - Contendrá también las tareas que tienen los usuarios en el grupo
 - Un grupo tiene un número ilimitado de invitaciones que ha hecho a otros usuarios.
- Usuario
 - Un usuario puede tener diferentes tareas asignadas.
 - Un usuario puede tener uno o varios grupos a los que está inscrito.
 - Un usuario puede ver las invitaciones que le han hecho.
 - Además, tendrá uno o varios logros que ha completado o que aún están en progreso.
 - El usuario podrá tener relacionados varios dispositivos.
- Tarea
 - Cada tarea, necesariamente, debe estar relacionada con un usuario y con un grupo.
- Invitación
 - Una invitación debe involucrar necesariamente a un grupo y a un usuario.
- Historial acciones
 - Una entrada en el historial debe estar relacionada con una tarea y un usuario.
- Dispositivos
 - Un dispositivo debe estar relacionado con un usuario.
- Logros
 - Un logro debe tener varios usuarios, y cada relación con uno deberá incluir si el logro está completo o el progreso actual.

El diagrama del modelado de los objetos que se usará en el proyecto quedaría de la siguiente forma:



5.3. Arquitectura

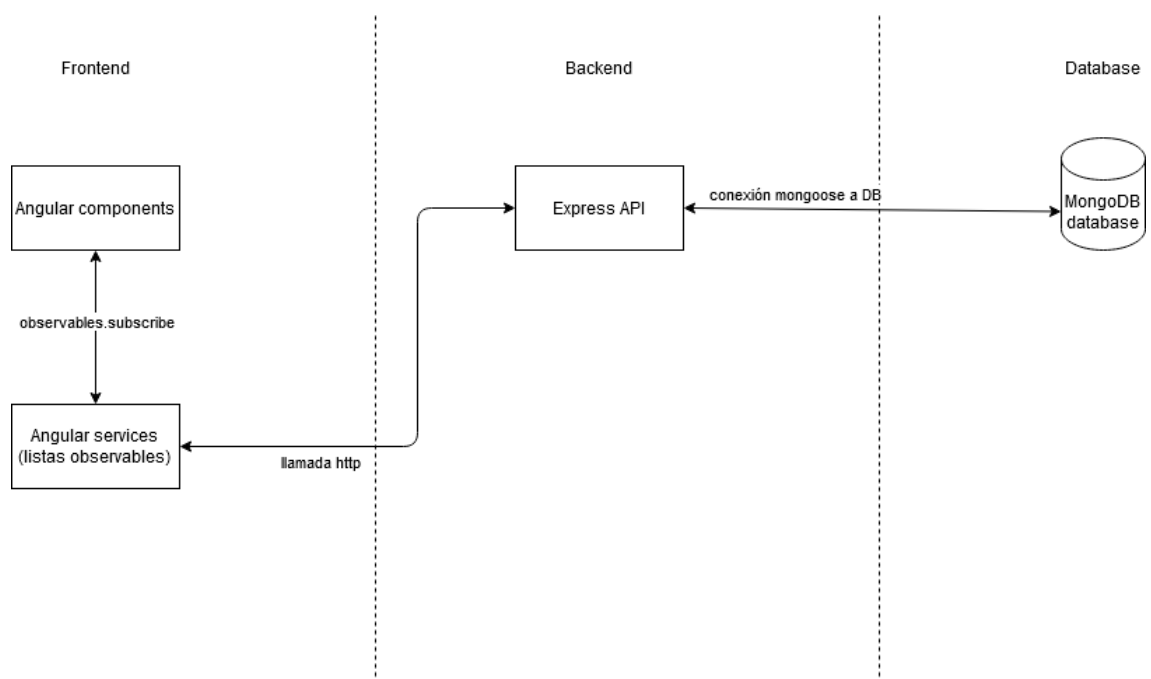
Se define la arquitectura de software como la estructura que debe tener un proyecto software en tanto a las piezas individuales que lo forman y el modo en el que estas piezas se deben juntar y cómo trabajan entre ellas.^x

La arquitectura del proyecto se puede dividir en tres partes claramente diferenciadas:

- **Frontend:** Se trata de la parte que se encarga de mostrar las vistas a los usuarios, así de cómo mostrar los datos que se recuperan desde la base de datos y como introduce datos el usuario para que sean almacenados. Esta parte estará desarrollada en Angular. Cabe mencionar que Angular no maneja un patrón clásico de MVC (modelo, vista, controlador), sino que el modelo tiene mucha relación con la vista. Esto es así por el concepto base de Angular de two-way data binding, ya que la forma de sincronizar los datos entre la vista y el modelo-vista es totalmente dependiente, es decir, en la vista se puede modificar el modelo y en el modelo se puede modificar la vista, se hará más hincapié en esto en el apartado de implementación. Ahora mismo habría que dejar claro que Angular es el encargado de proporcionar las vistas a los usuarios.

- **Backend:** Es la parte que se encarga de todos los procesos necesarios para que la aplicación funcione de forma correcta. Se trata de un servidor que espera peticiones que hará el frontend. Es decir, esperará por ejemplo una petición de obtener las tareas de un usuario y el servidor se conectará con la base de datos y efectuará la consulta necesaria, modelará los datos obtenidos al formato correcto y los enviará al frontend, que se encargará de mostrarlos. También se pueden programar acciones cada cierto tiempo, como repartir tareas programadas o enviar notificaciones a los dispositivos de los usuarios. En este proyecto se ha optado por crear un servidor Express.js por simplicidad a la hora de gestionar peticiones HTTP y por poder instalar un modelador de objetos frente a la base de datos llamado Mongoose, que permite obtener objetos JSON de los esquemas que se definan. Es decir, al crear un esquema, se pueden efectuar peticiones a la base de datos usando ese esquema y la petición devolverá ya un objeto JSON con el mismo formato definido en el esquema. Esto ayudará a mapear los datos de la base de datos con los objetos definidos en el frontend.
- **Base de datos:** Se trata de la parte donde se almacenan todos los datos que va a gestionar la aplicación. Se ha optado por la base de datos NoSQL MongoDB por su gran adaptación en todo el framework de Angular + Express.js.

El diagrama de la arquitectura software quedaría de la siguiente forma:



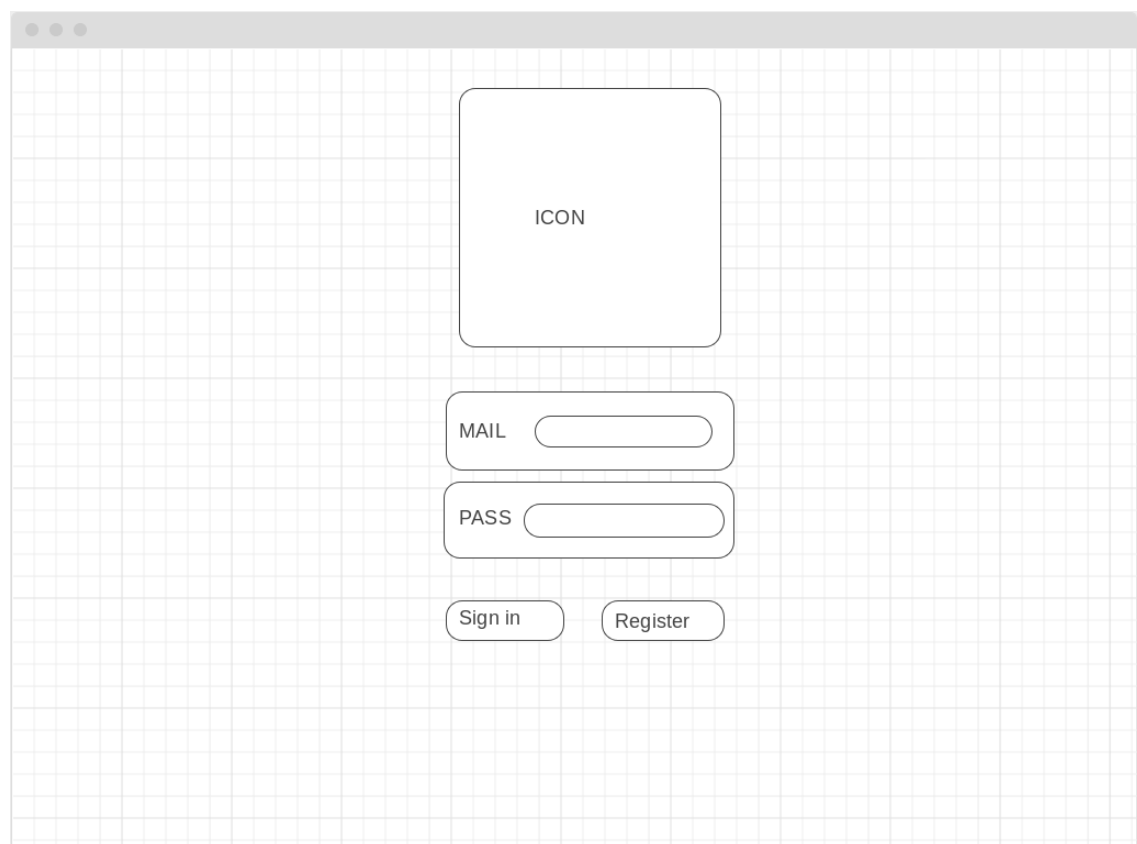
En el diagrama se pueden ver las tres partes principales definidas anteriormente y como se comunican entre ellas. Mencionar que en la parte de frontend, el componente de Angular es el modelo-vista, este modelo se suscribe a un tipo de dato llamado Observable, estos observables se encuentran en los archivos de servicios de Angular, que son las partes de Angular encargadas de comunicarse con el backend por

peticiones http y pasan los datos obtenidos a las listas observables, que al final son las listas donde lee y muestra los datos el componente.

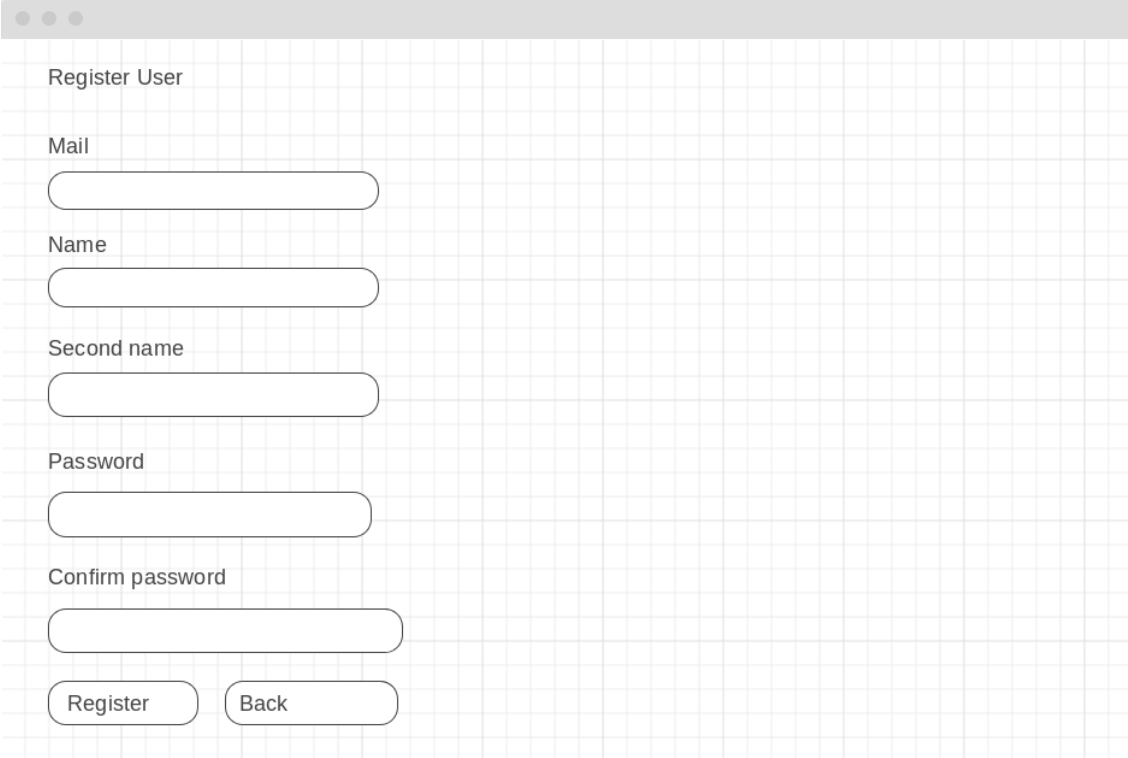
5.4. Interfaz (prototipos)

La interfaz es la encargada de permitir a los usuarios interactuar con la aplicación y permitirles cumplir los requisitos y casos de uso definidos. Para tener una idea clara de cómo se va a hacer, se han creado una serie de prototipos y definido como el usuario va a interactuar entre las diferentes pantallas.

- Pantalla inicial: Esta es la primera pantalla que ve el usuario al entrar a la aplicación, le permite acceder usando sus credenciales o acceder a la pantalla de registro si aún no está registrado, en grande aparecerá el icono de la aplicación:



- Pantalla de registro: Esta pantalla se accede desde la pantalla de inicio, en esta interfaz el usuario puede introducir sus datos personales y registrarse como usuario de la aplicación, donde después podrá acceder desde la pantalla de inicio.



Register User

Mail

Name

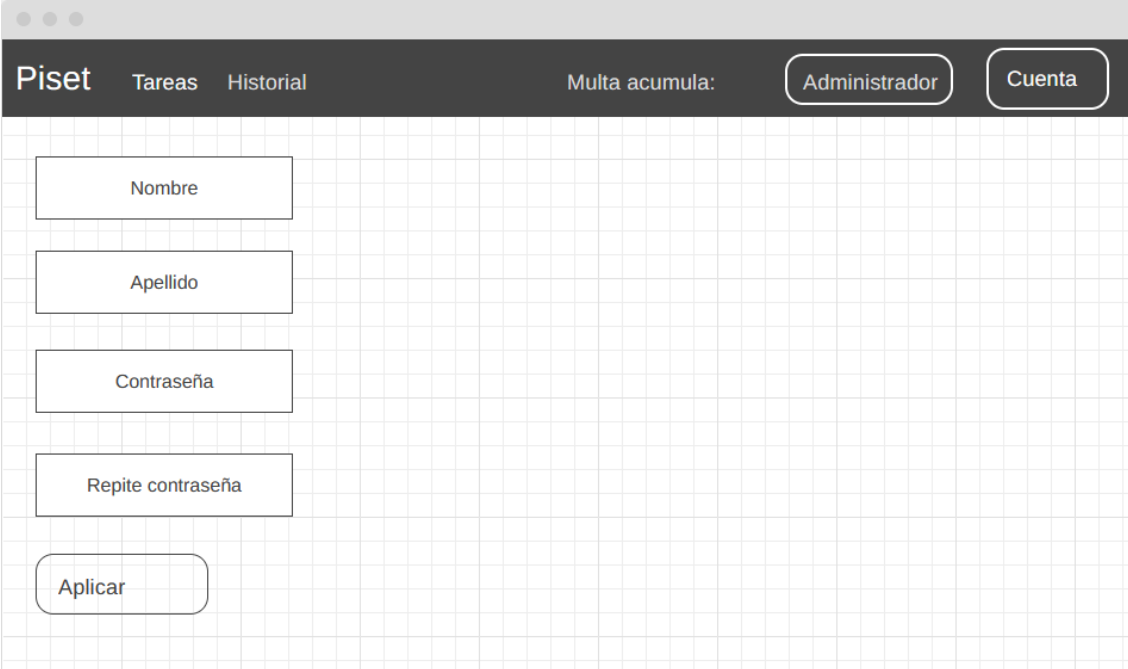
Second name

Password

Confirm password

Register Back

- Pantalla de ajustes de usuario: En esta vista el usuario ya registrado puede actualizar sus datos de usuario. Puede actualizar su nombre, apellidos y contraseña.



Piset Tareas Historial Multa acumula: Administrador Cuenta

Nombre

Apellido

Contraseña

Repite contraseña

Aplicar

- Pantalla de invitaciones: En la siguiente pantalla un usuario puede ver las invitaciones que le han llegado, y puede aceptar y denegar las invitaciones que quiera.



- Pantalla principal de las tareas: En esta vista los usuarios pueden ver en un tablero Kanban el estado de todas las tareas en las que están asignados. Al pulsar en una tarea pueden ver sus detalles, y en el caso de los administradores del grupo pueden editar los detalles de las tareas. En la parte de arriba del panel pueden aplicar los filtros que quieran, para mostrar las tareas según el grupo y los usuarios. Los usuarios pueden mover las tareas de estados.

- Pantalla de gestión de usuarios: En esta vista los gestores del grupo pueden eliminar o añadir los usuarios que quieran al grupo que tienen seleccionado, así como ver una lista de los usuarios de cada grupo.

Piset Tareas Historial Multa acumula: Administrador Cuenta

Grupo ▼

Nombre	Apellidos	Email	Eliminar

correo usuario Invitar

6. Implementación

En este apartado se explica todo el proceso de desarrollo del proyecto. Una vez ya se tienen todas las partes diseñadas, se procede a implementar todas esas soluciones diseñadas en el apartado anterior.

Para explicarlo de la mejor forma posible primero se comentan todas las herramientas de desarrollo utilizadas, el porqué de su uso y el propósito. Seguidamente se explica el modelo de gestión de versiones usado, los repositorios sobre los que se van a trabajar y las ramas que tendrá cada uno de ellos. Después de esto se definen los entornos donde se ejecutará cada parte del proyecto y como se relacionarán con los diferentes repositorios y ramas que se han definido anteriormente. Una vez todo esto definido se pasa a explicar cómo se ha implementado la integración continua, es decir, como se ha automatizado la relación entre las ramas de los repositorios y los diferentes entornos para que cuando se produzca un cambio en una rama, se ejecuten las pruebas pertinentes y se suba al entorno que toque.

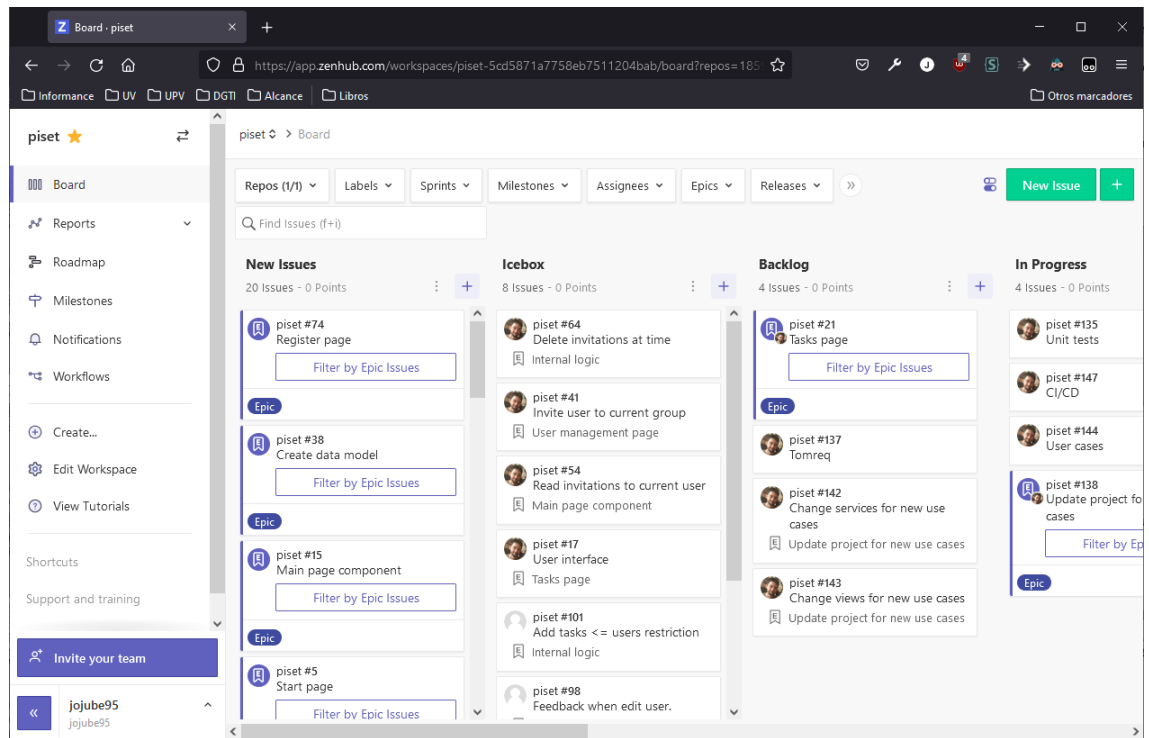
Con toda la metodología ya definida con la que se va a trabajar, ya se puede pasar a explicar el desarrollo del proyecto como tal, para hacerlo de la forma más simple posible, se ha optado por ir explicando el desarrollo empezando por la implementación de la base de datos, pasando por el backend y finalizando por el frontend.

6.1. Contexto tecnológico

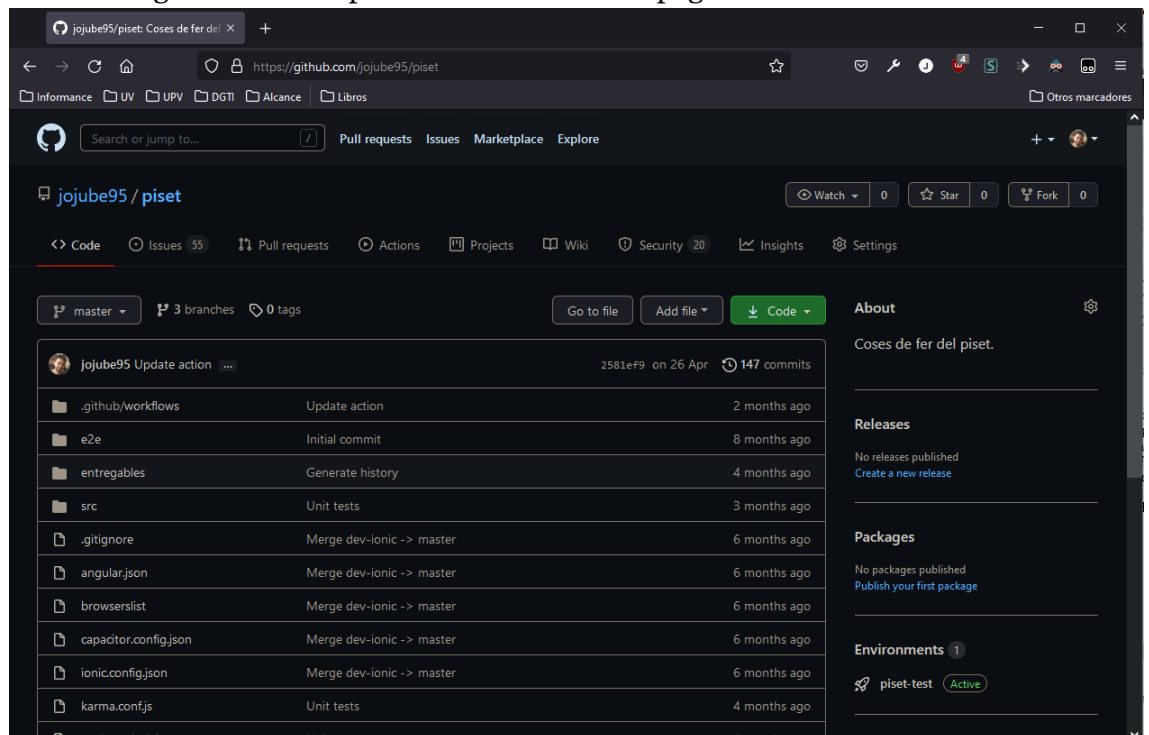
Se dividen las aplicaciones según la parte de la arquitectura del proyecto en las que han participado.

- Gestión de carga de trabajo:
 - ZenHub: Se trata de una aplicación que permite crear tareas sobre repositorios Git. Esto permite tener bien gestionadas cada tarea que se va a hacer para el desarrollo del proyecto con cada *commit* del repositorio donde se aloja el código del proyecto. Permite la creación de tareas más grandes, formadas por un conjunto de tareas más simples:

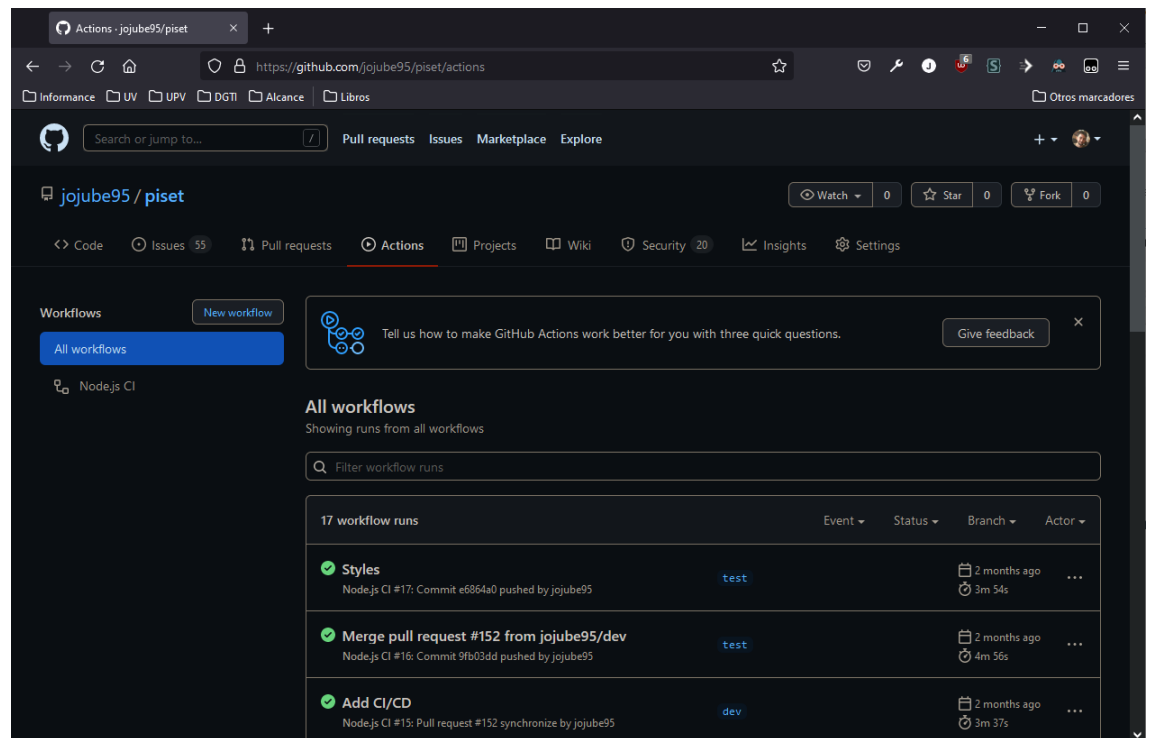
Aplicación gestión de tareas para un grupo de personas



- Gestión de versiones:
 - Git: Se ha usado Git para la gestión de versiones del código que se ha desarrollado. Se trata de una aplicación por línea de comandos, aunque hay una versión con interfaz de usuario. Esta herramienta permite realizar *commits* de cambios en el código y subirlos a GitHub usando el comando *push*.
 - GitHub: Se trata de una plataforma donde alojar y poder gestionar un repositorio Git desde una página web:

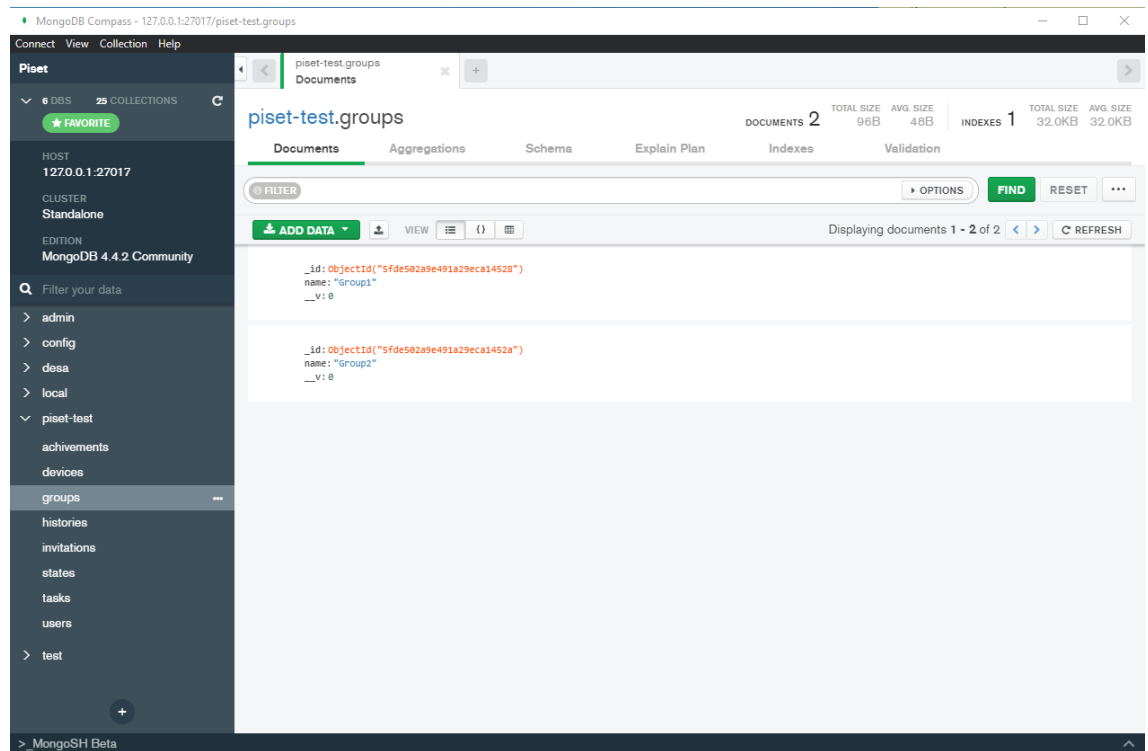


- Integración continua:
 - GitHub Actions: Para implementar la integración continua se ha optado por usar la herramienta GitHub Actions que proporciona la propia plataforma de GitHub. Se ha elegido por su integración directa con el repositorio de GitHub. En esta herramienta se puede definir un archivo con una serie de acciones que se ejecutarán en una máquina virtual que crea GitHub al vuelo cuando se detecte una determinada acción en el repositorio, como un *push*, *merge* o *pull request*, así como ver el resultado de cada ejecución que se haya lanzado de esas acciones:

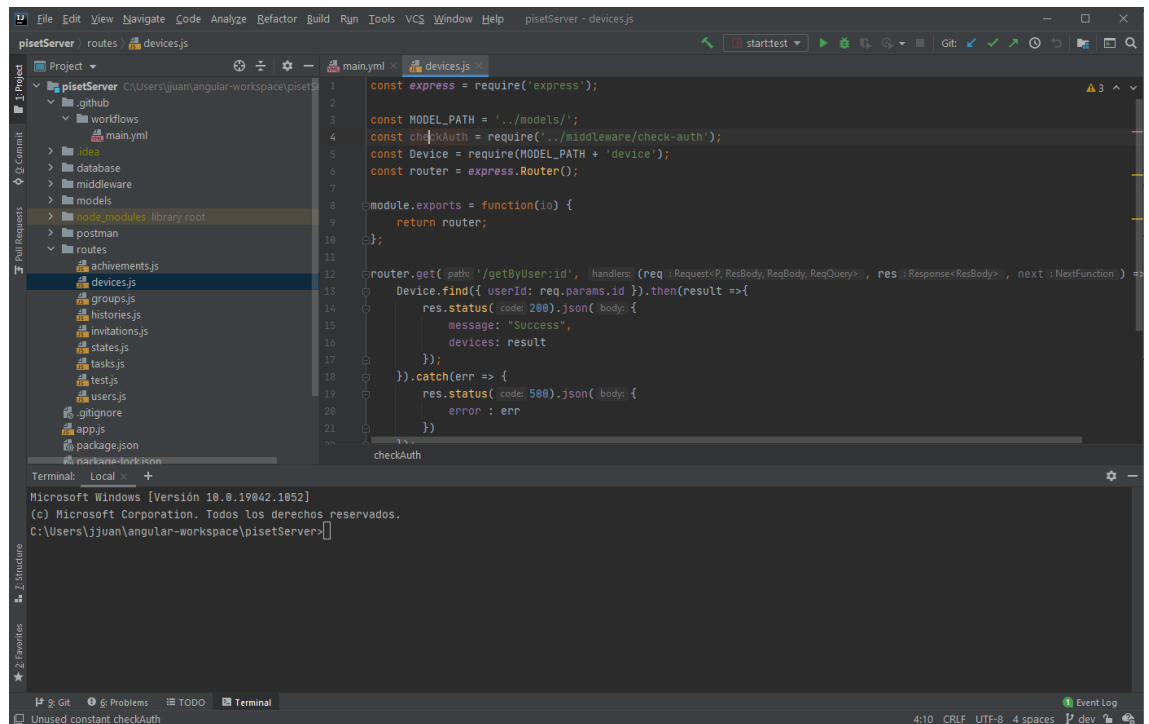


- Base de datos:
 - MongoDB Compass: Esta aplicación permite gestionar las diferentes bases de datos añadiéndolas al programa simplemente con un enlace de MongoDB, que incluye la dirección donde se aloja la base de datos y el usuario y contraseña del administrador. Con este programa se puede ver en una interfaz intuitiva el contenido de cada una de las bases de datos, así como editar sus datos y cargar documentos desde un archivo JSON:

Aplicación gestión de tareas para un grupo de personas



- Backend y frontend:
 - IntelliJ IDEA: Se trata de un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Proporciona todas las herramientas y facilidades a la hora de desarrollar código, sobretodo código en JavaScript, ya que viene con un revisor de calidad de código y la función de autocompletado, así como una extensión para generar componentes y servicios de Angular. También permite ver en todo momento en que rama de la versión se está trabajando y que archivos han sido editados respecto al último cambio de la rama. Se ha usado tanto para el desarrollo de la parte cliente como del servidor Express:



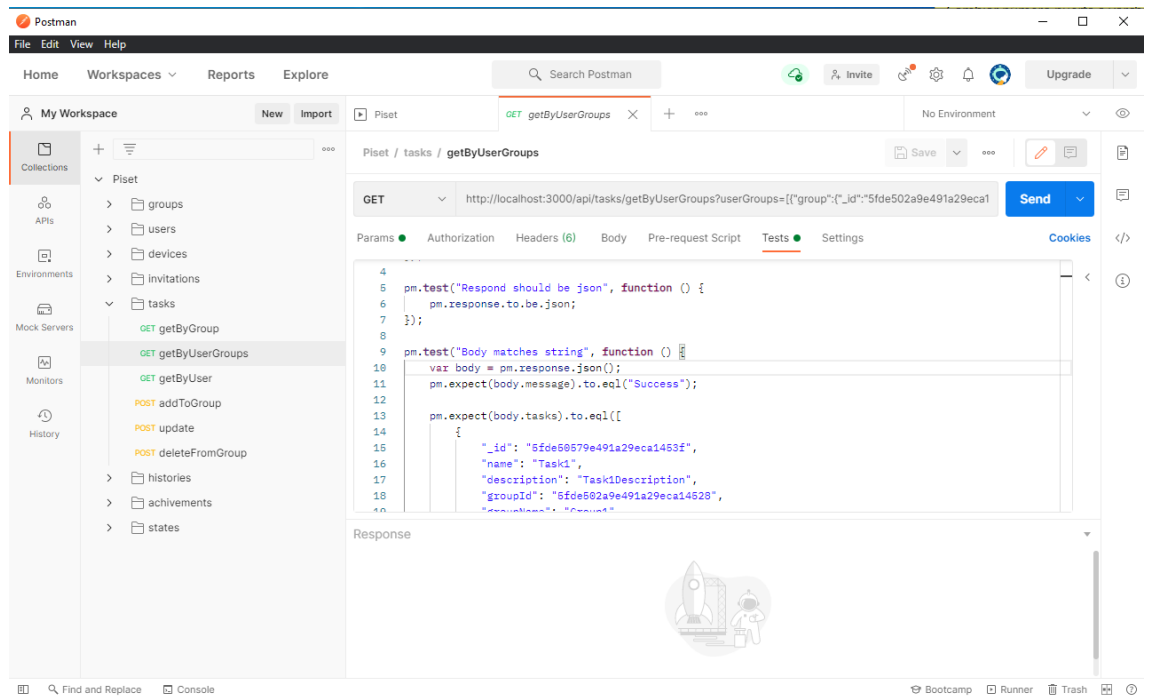
```
const express = require('express');

const MODEL_PATH = '../models/';
const checkAuth = require('../middleware/check-auth');
const Device = require(MODEL_PATH + 'device');
const router = express.Router();

module.exports = function(io) {
  return router;
};

router.get('/getByUser/:id', checkAuth, (req, res) => {
  Device.find({ userId: req.params.id }).then(result => {
    res.status(200).json({
      message: "Success",
      devices: result
    });
  }).catch(err => {
    res.status(500).json({
      error: err
    });
  });
});
```

- Postman: Postman es una herramienta de gestión de consultas HTTP. Con esta herramienta se pueden crear consultas al servidor y ver cómo este responde. Básicamente permite testear el servidor creando las consultas pertinentes. Al ejecutar las consultas definidas, la herramienta muestra los resultados de cada una. Se pueden crear pruebas para cada consulta, como por ejemplo el tipo de resultado HTTP, el contenido del resultado, o el tiempo que tarda en hacer la consulta. Luego este listado de consultas se puede exportar a un solo archivo para ser ejecutado desde la línea de comandos. Esto último es muy importante, ya que es lo que se usará para probar el servidor en la integración continua:



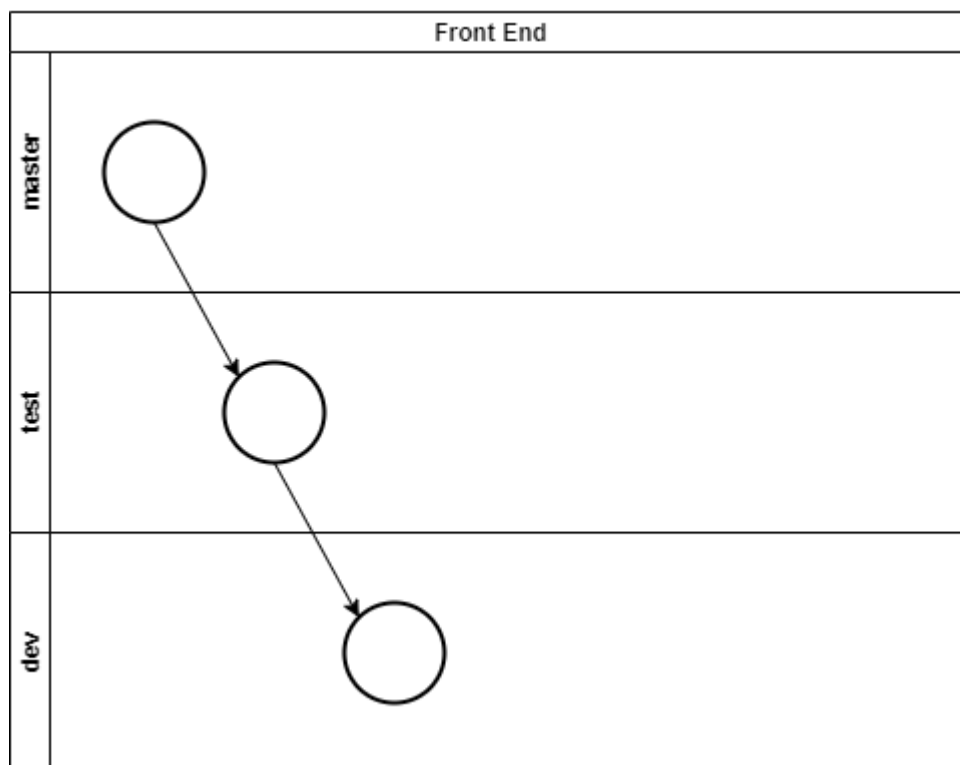
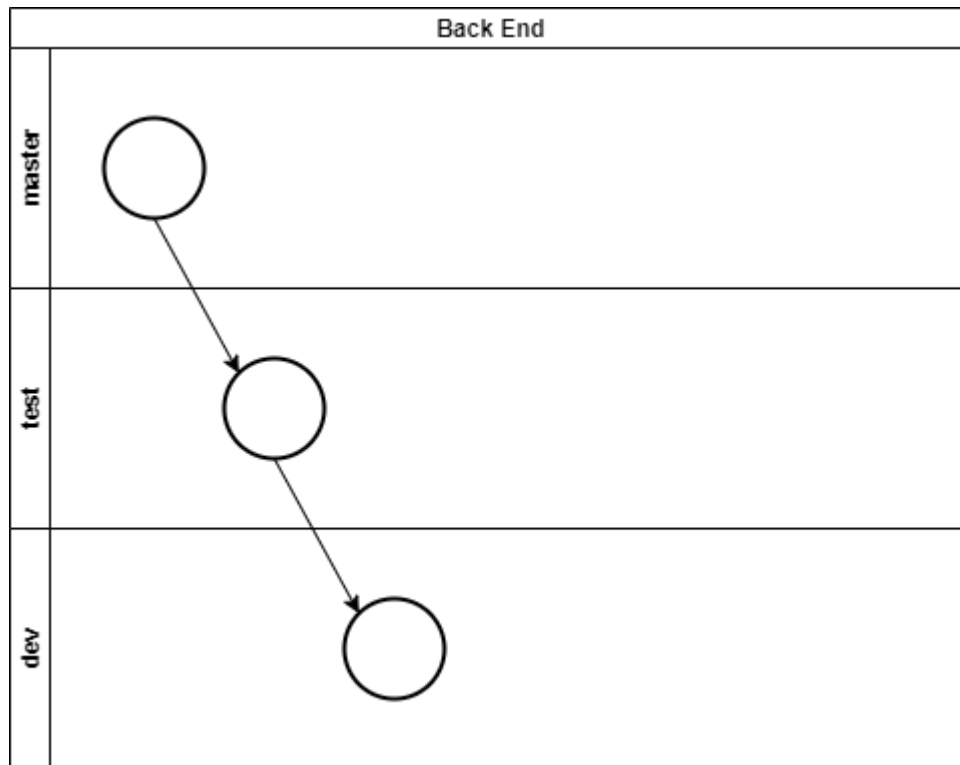
6.2. Gestión de versiones

Como se ha comentado en apartados anteriores, la gestión de versiones es esencial a la hora de mantener el código fuente, así como para almacenarlo y tener un histórico de los cambios que se van aplicando con la posibilidad de volver a una versión anterior en todo momento. También sería muy útil a la hora de trabajar en equipo, ya que otro trabajador solamente tendría que crear una rama en base a otra principal y trabajar en su rama, hasta que ya lo tuviera todo listo y fusionaría sus cambios con la rama principal.

En este proyecto se usa el controlador de versiones Git. Se van a definir qué repositorios se van a crear y qué ramas van a tener.

La idea principal es tener en repositorios diferentes la parte de frontend y la de backend. Esto se debe a que las dos partes estarán alojadas en entornos diferentes, por lo que a la hora de desplegarlas es conveniente tenerlas también en repositorios separados ya que del despliegue se encargará el GitHub Action que se defina en cada repositorio. Se podrían haber alojado ambas partes en un mismo repositorio, pero entonces se debería definir en el GitHub Action que archivos van a ser desplegados y en que entorno irían a parar, con la separación en dos repositorios se simplifica este trabajo.

En el siguiente diagrama se ha representado el esquema Git del control de versiones de este proyecto:



Cada piscina representa un repositorio, el repositorio del backend y del frontend. Cada línea de la piscina representa una rama. En ambos repositorios se han creado las tres ramas principales, la rama master, que representa el código que estará alojado en el entorno de producción, la rama test, que debe ser lo más parecida a la rama master y es donde se alojarán los cambios más recientes para que lo prueben los usuarios finales, y la rama de dev que es donde se trabaja a la hora de desarrollar los

cambios pertinentes. Como se observa, gracias a las flechas del diagrama, cada rama está creada en base a la anterior. Por lo tanto, la metodología de trabajo sería la siguiente:

1. Crear una rama en base a la rama principal dev y hacer los cambios pertinentes.
2. Una vez ya hechos con sus pruebas, se fusionan la rama creada anteriormente con la rama principal dev. Gracias a la integración continua, al fusionar la rama del desarrollo con la rama dev, se ejecutarán todas las pruebas del proyecto, más las que se han hecho para la parte nueva. Si han ido correctamente, se tendrá la rama principal dev con los nuevos cambios.
3. Una vez la rama principal dev esté estable con todos los nuevos cambios, hay que fusionar la rama dev con la rama test. Una vez más, la integración continua realizará todas las pruebas y si son satisfactorias dejará el código fusionado en la rama test y desplegará el proyecto en el entorno de test.
4. Una vez lo hayan validado los usuarios y den su confirmación para subir los cambios a la rama de master o producción, se fusiona la rama de test con la rama master, la integración continua hará su trabajo y ya quedarían los nuevos cambios implementados y desplegados en producción.

6.3. Entornos

En este apartado se va a definir en qué entornos estarán alojadas las diferentes partes del proyecto y que relación guardan con los repositorios y ramas definidas en el apartado anterior. El proyecto consta de tres partes principales que hay que alojar en diferentes servidores: frontend, backend y base de datos. En el apartado anterior se han definido tres ramas o entornos: entorno de dev o desarrollo, entorno de test o preproducción y entorno master o producción. Por lo tanto, cada parte del proyecto tendrá que alojarse en tres entornos diferentes, se va a definir en qué servidores se alojan cada parte:

- Desarrollo
 - Frontend: Se crea una instancia local de desarrollo de Angular usando el comando de Angular “ng serve”. Este comando crea una instancia de desarrollo de la aplicación Angular que se reinicia rápidamente cuando se aplique un cambio en el código fuente, lo que permite mayor agilidad a la hora del desarrollo. Aplicando este comando, Angular crea la instancia usando el archivo de entorno de desarrollo de Angular. Los archivos de entorno son archivos donde se guardan parámetros que hay que tener en cuenta al ejecutar en diferentes entornos, como puede ser la dirección del servidor backend.
 - Backend: Se crea una instancia local de desarrollo usando el comando “nodemon server.js desa”. Este comando ejecuta el servidor usando el paquete nodemon, que permite aplicar cambios en el código del servidor sin tener que reiniciarlo

manualmente, ya se encarga de reiniciarlo la propia máquina. Se le pasa el parámetro “desa” para indicarle que la base de datos se encuentra en la máquina local.

- Base de datos: Se crea una instancia de base de datos MongoDB en la máquina local. El servidor sabe que tiene que apuntar a esta base de datos por el parámetro “desa” que se le pasa al iniciarlo anteriormente.
- Test
 - Frontend: La parte de frontend estará alojada en la plataforma llamada Heroku. Se trata de una plataforma como servicio que permite alojar aplicaciones en la nube de forma gratuita. Además, tiene integración con el código alojado en GitHub, lo que resultará de mucha ayuda para la integración continua. También cuenta con variables de entorno, que se usarán también para indicarle a las aplicaciones en que entorno se van a desplegar en el momento de despliegue. Es decir, se pueden almacenar aquí los enlaces y credenciales para que la parte de frontend se conecte con el servidor, y la parte de servidor se conecte con la base de datos.
 - Backend: La parte de backend estará también alojada en la plataforma Heroku.
 - Base de datos: Se aloja la base de datos en la plataforma MongoDB Atlas. Esta plataforma permite mantener una base de datos de MongoDB en un servidor externo de forma gratuita, y proporciona un enlace de conexión para que se conecte el servidor.
- Producción
 - Frontend: Alojada en otra máquina Heroku.
 - Backend: También se encuentra en Heroku.
 - Base de datos: Alojada en otra máquina de MongoDB Atlas

6.4. Integración continua

En este apartado se va a explicar cómo se ha implementado la integración continua en este proyecto. Lo que se quiere conseguir es la automatización de los test y el despliegue en el entorno. Esto se ejecutará cada vez que se suban cambios a las ramas principales del repositorio que se definieron anteriormente (master, test, dev). Más tarde se explicará cómo se ha desarrollado las pruebas de software que se ejecutará en la integración continua.

Como se ha mencionado anteriormente, se va a usar la herramienta de GitHub Actions ya que se va a alojar el código en la misma plataforma de GitHub, por lo que la integración con esta herramienta es instantánea.

Para implementar la integración continua hay que generar un archivo yml y añadirlo en la ruta `.github/workflows`. Ahora se va a explicar el archivo de configuración de la integración continua que se ha definido para cada uno de los repositorios.



- Backend:

```

3   name: CI
4
5   # Controls when the action will run.
6   on:
7     # Triggers the workflow on push or pull request events but only for the master branch
8     push:
9       branches: [ test ]
10    pull_request:
11      branches: [ test ]
12
13    # Allows you to run this workflow manually from the Actions tab
14    workflow_dispatch:
15
16    # A workflow run is made up of one or more jobs that can run sequentially or in parallel
17    jobs:
18      build:
19        runs-on: ubuntu-latest
20        strategy:
21          matrix:
22            node-version: [14.x]
23            mongodb-version: [4.4]
24
25        steps:
26          - name: Git checkout
27            uses: actions/checkout@v2
28
29          - name: Use Node.js ${ matrix.node-version }
30            uses: actions/setup-node@v1
31            with:
32              node-version: ${ matrix.node-version }
33
34          - name: Start MongoDB
35            uses: supercharge/mongodb-github-action@1.3.0
36            with:
37              mongodb-version: ${ matrix.mongodb-version }
38          - run: mongorestore "mongodb+srv://cluster0.53xnf.mongodb.net/piset-test" -u root -p root --archive=database/test --drop
39          - run: npm install
40          - run: npm run ci
41          env:
42            CI: true

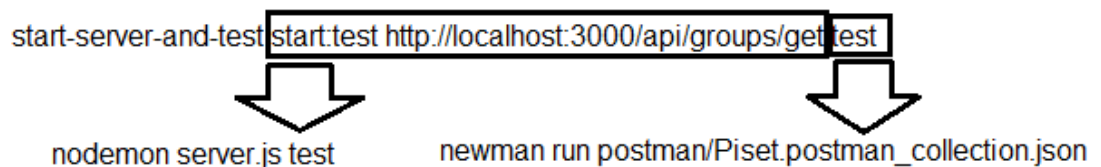
```

El archivo de ejecución de la integración continua se divide en varias partes:

- name: Indica el nombre de la acción de GitHub Actions.
- on: Esta parte indica en que eventos del control de versiones git se ejecutará la acción, en este caso se ha definido que se ejecute cuando se haga un push o un pull request a la rama de test.
- jobs: Esta parte agrupa los trabajos que se van a ejecutar en distintas maquinas, para este caso solo se ha definido una secuencia de actividades en una maquina Ubuntu, indicando la versión de node y la versión de mongodb.
- steps: Agrupa las acciones que se van a ejecutar. En esta parte se puede usar código que ya está hecho y es reutilizable, es decir, se pueden incluir Actions. En este caso hay un Action que instala node usando la versión indicada anteriormente y otro que instala mongodb.
- run: Se usa para ejecutar los comandos pertinentes en la máquina virtual, estos comandos se ejecutarán en la ruta absoluta del repositorio alojado en GitHub. Se va a explicar cada comando:
 - mongorestore
"mongodb+srv://cluster0.53xnf.mongodb.net/piset-test" -u root -p root --archive=database/test --drop:

Este comando ejecuta un restore de la base de datos de MongoDB (alojada en el entorno de test de MongoDB Atlas, comentado en apartados anteriores) usando unos datos de test almacenados en el archivo del repositorio Git database/test. Con este comando se deja la base de datos lista para ejecutar los tests pertinentes sobre el servidor. Se explicará en el apartado de pruebas porque se ha usado esta metodología.

- npm install: Instala las dependencias de node definidas en el archivo package.json que necesita el servidor para ejecutarse correctamente.
- npm run ci: Este comando sirve para ejecutar los scripts definidos en el archivo package.json del proyecto. En este caso ejecuta el siguiente script, explicado en esta imagen para que quede más claro:



- start-server-and-test: Este comando o herramienta admite dos entradas, un comando que inicie un servidor y un comando que ejecute pruebas. El primero se ejecuta con nodemon server.js test y el segundo ejecuta unas pruebas de Postman sobre el servidor usando su ejecutable por línea de comandos llamado Newman, si todas las pruebas son satisfactorias el resultado es positivo y si falla una es negativo, indicándole a la integración continua que no debe desplegar nada en el entorno de Heroku.

- Frontend:

```

4     name: Node.js CI
5
6     on:
7       push:
8         branches: [ test ]
9       pull_request:
10        branches: [ test ]
11
12    jobs:
13      build:
14
15        runs-on: ubuntu-latest
16
17        strategy:
18          matrix:
19            node-version: [10]
20
21        steps:
22          - uses: actions/checkout@v2
23          - name: Use Node.js ${ matrix.node-version }
24            uses: actions/setup-node@v2
25            with:
26              node-version: ${ matrix.node-version }
27          - run: npm ci
28          - run: npm run lint
29          - run: npm run test:gitHubActions
30          - run: npm run e2e

```

En este archivo se declaran los pasos que sigue la integración continua para el repositorio de frontend. Los únicos cambios están en los steps que se usan. Se explica cada comando:

- npm ci: Se trata de un comando similar al npm install, pero se usa en un entorno automático como es el caso de plataformas de pruebas o entornos de integración continua, como este caso.
- npm run lint: Se trata de un comando que comprueba la buena escritura de código, es decir, si está todo bien formateado para favorecer la futura lectura.
- npm run test:gitHubActions: Ejecuta el script definido en el archivo package.json: `ng test --browsers=ChromeHeadless --watch=false --code-coverage`. Este comando ejecuta las pruebas unitarias de la aplicación Angular, definidas en su lenguaje integrado llamado Jasmin, se explicarán en más detalle en el apartado de pruebas.

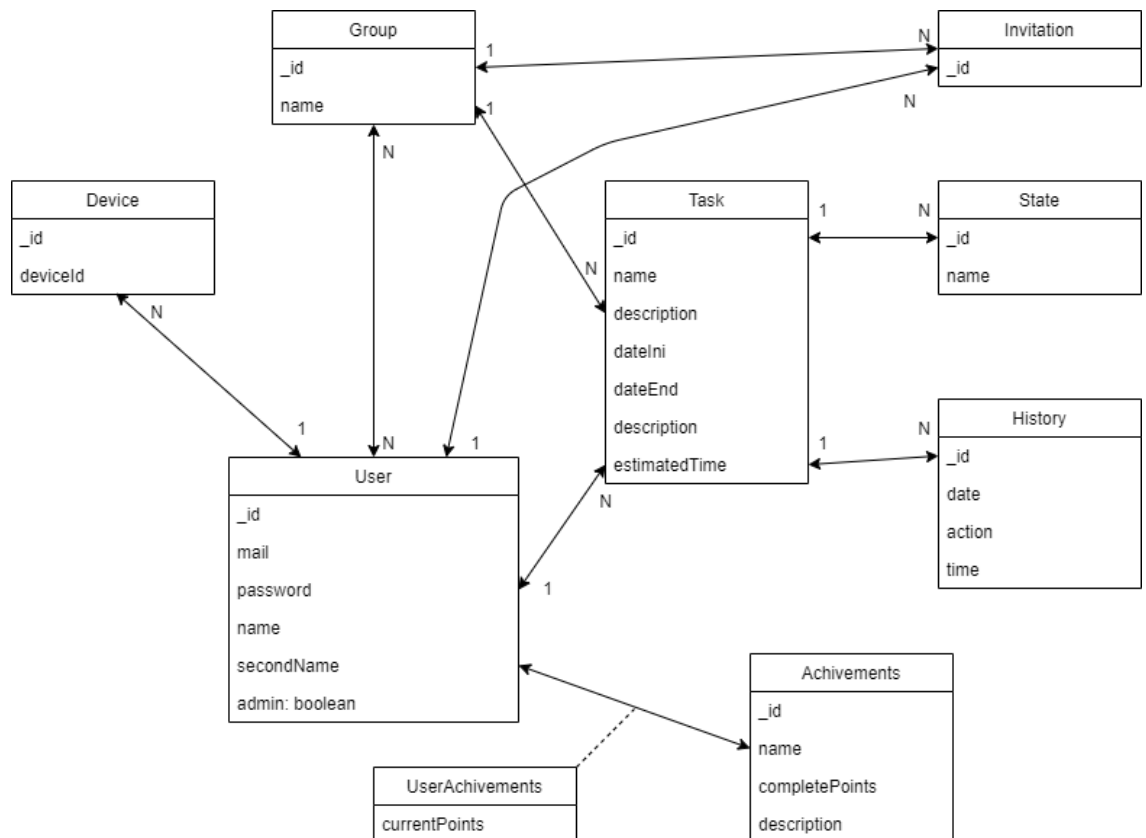
- npm run e2e: Ejecuta las pruebas end to end, es decir las pruebas que consisten en simular lo que haría un usuario final de la aplicación. Se ejecutan usando la herramienta de pruebas Cypress.io. Estas pruebas se explicarán también en el apartado de pruebas.

Con esto ya se tendrían los procesos de integración continua para ambos repositorios implementada. Cabe mencionar que, si algún comando de pruebas definido termina su ejecución con algún test fallido, automáticamente se pausa el proceso de integración, por lo que esa versión no llegaría a desplegarse en el entorno indicado del servidor Heroku.

6.5. Base de datos

Para la implementación de la base de datos hay que tener en cuenta que se trata de una base de datos no relacional, por lo que la implementación se vuelve más flexible y abierta a posibles cambios futuros. Hay que tener en cuenta también la posibilidad de la repetición de algunos datos en pos de mejorar las consultas de obtención de datos y evitar la concatenación entre los diferentes documentos. Estas últimas cualidades son propias de las bases de datos no relacionales y es importante comentarlas antes de presentar la implementación.

Se va a partir del diseño presentado en el apartado anterior y se va a ir añadiendo los campos necesarios para implementar las relaciones indicadas. Así como definir la tipología de cada campo. El diseño inicial es el siguiente:

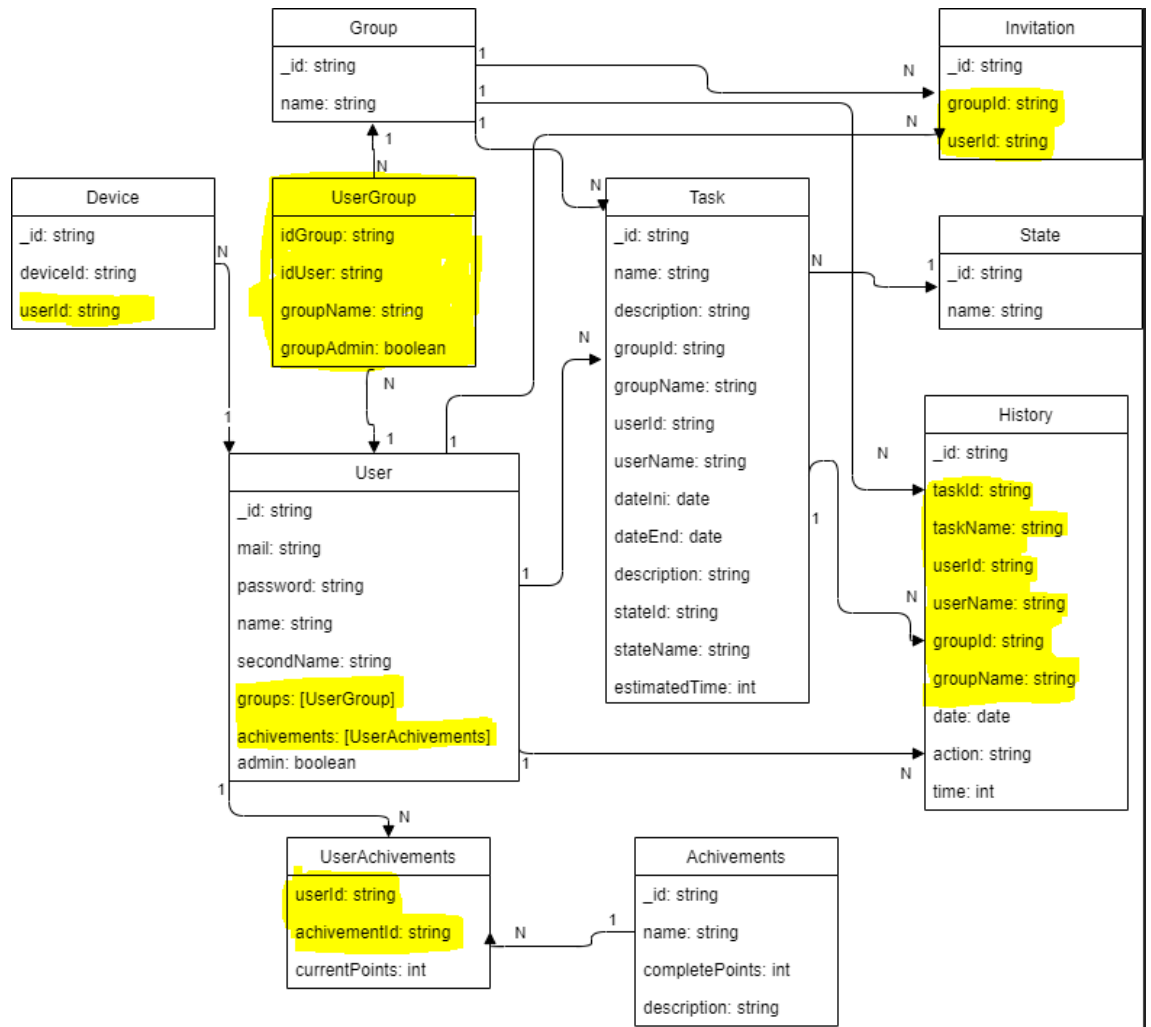


En el diseño se ha dejado definido cada objeto que hay que guardar en la base de datos, cada campo que tiene que tener y las relaciones que mantiene sobre los otros objetos. Para satisfacer esas relaciones hay que definir una serie de campos que actúen como referencia a otros objetos, se implementa cada relación de la siguiente forma:

- **Grupo-Usuario:** Se trata de una relación de muchos a muchos. Esta relación se resuelve añadiendo un objeto adicional que tendrá la referencia a los identificadores de cada objeto de la relación, en este caso, este objeto auxiliar tendrá un campo de identificación de grupo y otro campo de identificación de usuario para implementar la relación. Además, hay que añadir un campo de tipo binario para indicar si el usuario es administrador de ese grupo y se repetirá en este objeto el campo del nombre del grupo. Así, pensando en optimizar futuras consultas, se mejora la consulta de listar los grupos en los que está inscrito un usuario. Ya que con este último campo se evita concatenar el objeto usuario, con el objeto auxiliar usuario-grupo y con el objeto grupo.
- **Usuario-Dispositivo:** Se trata de una relación de uno a muchos, es decir, un usuario puede tener muchos dispositivos. Para implementar esta relación bastaría con incluir el campo de identificación de usuario como referencia en el objeto de dispositivos.
- **Grupo-Tarea:** Esta relación es de uno a muchos. Un grupo puede tener varias tareas, pero una tarea solo puede estar relacionada con un único grupo. Esta relación se implementa al igual que el punto anterior, añadiendo un campo de referencia al grupo en el objeto de la tarea. Además, para mejorar futuras operaciones de consulta a la base de datos, se añade un campo con el nombre del grupo.
- **Usuario-Tarea:** Se implementa de la misma manera que la relación Grupo-Usuario. También en esta parte se añade un campo para facilitar una consulta, en este caso se trata del nombre del usuario.
- **Tarea-Historial, Usuario-Historial, Grupo-Historial:** Un grupo, una tarea y un usuario están relacionados con una entrada en el historial de la forma de uno a muchos. Para implementarla se han añadido los campos de referencia a los identificadores de grupo, tarea y usuario en el objeto del historial. Para facilitar futuras operaciones de consulta se han añadido los campos de nombre de grupo, nombre de usuario y nombre de tarea también en el objeto del historial. Cabe destacar que con una relación de Tarea-Historial habría bastado para satisfacer los requerimientos de la aplicación, pero se han creado las tres relaciones con sus respectivos campos duplicados de nombres para facilitar y optimizar las operaciones de consulta y búsqueda sobre los historiales de acciones.
- **Usuario-Logros:** Se trata de una relación de muchos a muchos. Un usuario puede tener varios logros y un logro puede tener varios usuarios. Se ha resuelto implementando un objeto auxiliar y además a este objeto auxiliar se le ha añadido un campo para indicar el progreso que tiene cada usuario en un logro determinado.

- Invitación-Grupo e Invitación-Usuario: La aplicación necesita guardar las invitaciones que hace un grupo a un usuario. Para esto se ha creado el objeto invitación, que referencia a los campos de identificación del grupo y al del usuario que se ha invitado.
- Tarea-Estado: Se ha creado esta relación para que en un futuro se puedan añadir varios estados a las tareas. Por lo que la relación es de uno a muchos, una tarea puede tener un estado. Para implementarla se ha añadido la referencia al estado en el objeto tarea, así como el nombre del estado para facilitar la operación de consulta.

Después de todos estos cambios para implementar las relaciones, se han definido los tipos de datos de cada campo. Hay que recordar que se trata de una base no relacional, por lo que se entiende como tabla en una base relacional ahora se resume en un archivo JSON o colecciones y las filas de cada tabla relacional serían ahora entradas JSON en el archivo de colección o documento. El diagrama de la implementación quedaría de la siguiente forma, se han resaltado los campos añadidos respecto al diagrama de diseño:



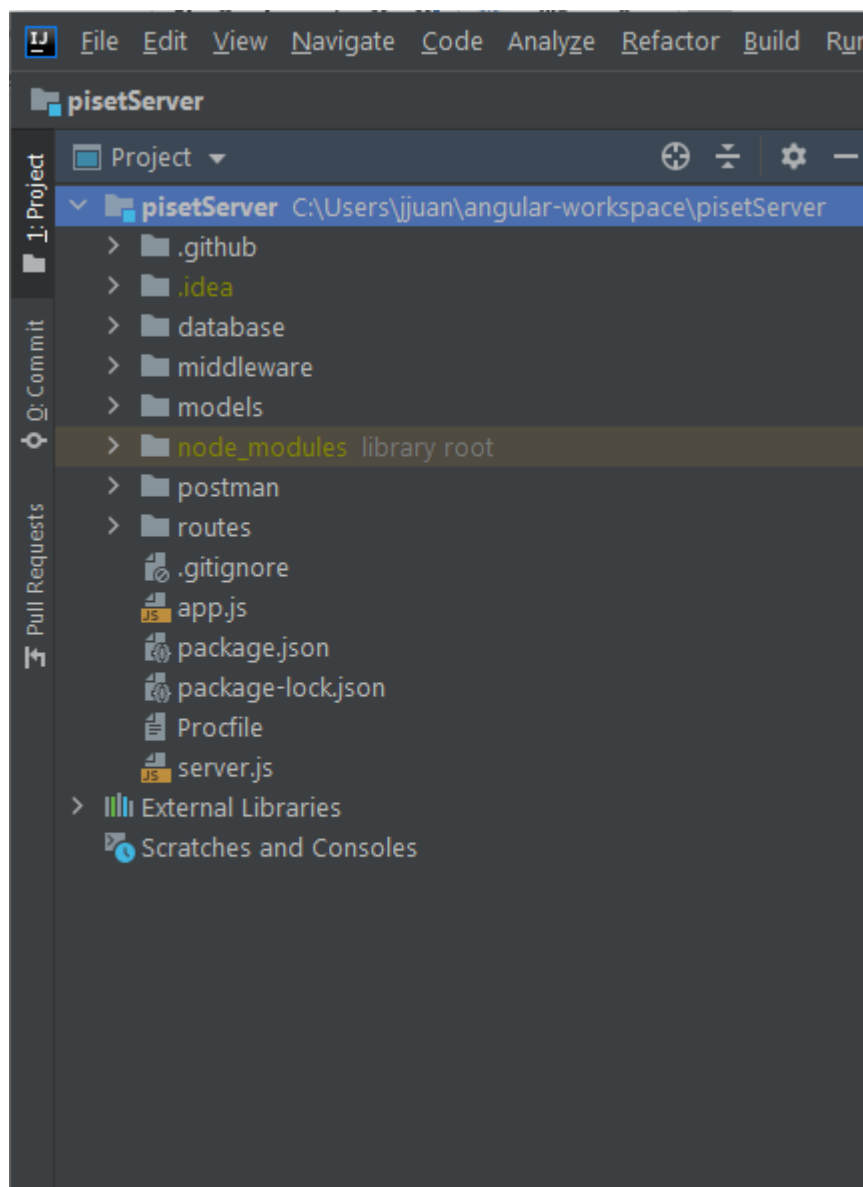
6.6. Backend

En este apartado se detalla todo el código y el proceso que se ha realizado para la implementación de la parte de servidor de la aplicación. Esta parte se encargará de responder a las peticiones HTTP que realice el frontend.

6.6.1. Organización del proyecto

Para detallar todo el proceso primero se expondrá la estructura del proyecto NodeJS del servidor y luego se explicará cada parte lo que hace y como lo hace, todo esto siguiendo un orden lógico de desarrollo para que quede lo más claro posible.

La estructura del proyecto NodeJS es la siguiente:



- .github: Contiene los archivos necesarios para la gestión del repositorio Git. Todos los repositorios Git contienen esta carpeta.
- .idea: Contiene los archivos necesarios para el uso del entorno de desarrollo IntelliJ. Se genera automáticamente al abrir un proyecto con este programa.

- **database:** Contiene los archivos de backup de la base de datos de test. Se usarán para futuras pruebas del servidor contra la base de datos. Se ha explicado su uso en la implementación de la integración continua.
- **middleware:** Contiene la validación de la autenticación del usuario que gestiona el paquete jsonwebtoken. Este código se ejecuta antes de cada petición HTTP para evitar que otros usuarios no autorizados ejecuten consultas que puedan dañar la aplicación.
- **models:** Contiene los modelos de los objetos a los que se mapearán los datos que se obtengan de base de datos. Para esta tarea se ha usado el paquete Mongoose.
- **node_modules:** Aquí se encuentran todas las dependencias que se han declarado en el archivo package.json.
- **postman:** Contiene el archivo con las consultas de pruebas que se ejecutarán sobre el servidor. Se ha explicado su uso en la parte de la implementación de la integración continua.
- **routes:** Contiene los archivos de las rutas a las que atenderá las peticiones HTTP el servidor. Estas rutas se inicializarán en la variable router del servidor Express y habrá un archivo de rutas por cada objeto de base de datos. Así se consigue un mejor mantenimiento del código al tener las diferentes funcionalidades lo más separadas posibles.
- **server.js:** Se trata del archivo de ejecución del servidor, al ejecutar este archivo se define el puerto por donde va a leer las peticiones, elige la url de conexión con la base de datos según el entorno en el que se inicie el servidor (se pasa por parámetro en el comando de ejecución), se conecta con la base de datos usando el paquete Mongoose e inicia el servidor creando una instancia de un servidor Express.
- **app.js:** Es el archivo donde se ha definido el servidor Express. En este archivo se indican las rutas a las que atenderá el servidor.

6.6.2. Dependencias

La primera parte que se ha hecho es la inicialización del proyecto NodeJS, para esto se ha creado una carpeta y ejecutado el comando “npm init”. Este comando deja un proyecto inicial al que hay que instalar una serie de dependencias, que se indican en el archivo package.json que ha generado el anterior comando.

El contenido del archivo package.json es el siguiente:

```

1  {
2    "name": "piresetserver",
3    "version": "1.0.0",
4    "description": "node api hosted on heroku",
5    "main": "index.js",
6    "scripts": {
7      "start:desa": "nodemon server.js desa",
8      "start:test": "nodemon server.js test",
9      "start:prod": "node server.js prod",
10     "start": "node server.js ${environment}",
11     "test": "newman run postman/Piset.postman_collection.json",
12     "ci": "start-server-and-test start:test http://localhost:3000/api/groups/get test"
13   },
14   "dependencies": {
15     "bcrypt": "^5.0.0",
16     "express": "^4.17.1",
17     "immutable": "^3.7.6",
18     "jsonwebtoken": "^8.5.1",
19     "mongoose": "^5.10.9",
20     "node-cron": "^2.0.3",
21     "request": "^2.88.2"
22   },
23   "devDependencies": {
24     "newman": "^5.2.2",
25     "nodemon": "^2.0.4",
26     "start-server-and-test": "^1.12.1"
27   },
28   "engines": {
29     "node": "10.13.0"
30   },
31   "repository": {
32     "type": "git",
33     "url": "git+https://github.com/jojube95/pisetServer.git"
34   },
35   "author": "",
36   "license": "ISC",
37   "bugs": {
38     "url": "https://github.com/jojube95/pisetServer/issues"
39   },
40   "homepage": "https://github.com/jojube95/pisetServer#readme"
41 }

```

La parte que interesa en estos momentos es la de “dependencies”, en la que se define el nombre de la dependencia y la versión que se instalará:

- **bcrypt:** Se trata de una utilidad para encriptar datos, se usa a la hora de guardar las contraseñas de los usuarios en la base de datos.
- **express:** Es el marco de aplicación web para NodeJS y en el que está basado todo el desarrollo del servidor.
- **jsonwebtoken:** Se trata de una librería para implementar una validación de seguridad a la hora de transmitir información en JSON entre cliente y servidor.
- **mongoose:** Es una librería para escribir consultas para la base de datos MongoDB usando unos modelos que hay que definir previamente para mapear la información de la base de datos a objetos JSON.
- **node-cron:** Permite crear una rutina que se ejecuta cada cierto tiempo.

- request: Su objetivo es simplificar la forma de hacer y recibir peticiones HTTP.

Una vez estén declaradas las dependencias en el archivo package.json, solamente hay que ejecutar el comando “npm install” para que las instale en el proyecto.

6.6.3. Implementación

6.6.3.1. server.js

Como ya se ha comentado anteriormente, este es el archivo de ejecución del servidor. Primero se importa el módulo de Express app.js. Se definirá su implementación más tarde.

```
1 const app = require("./app");
```

Seguidamente se define cual es la URL a la base de datos según el entorno que se le pasa al servidor en el comando, una vez conectado, se le pasa la variable de conexión al módulo de Express definida anteriormente en la variable app.

```
50 const uris = {
51   "desa": "mongodb://127.0.0.1:27017/desa",
52   "test": "mongodb+srv://root:root@cluster0-53xnf.mongodb.net/piset-test",
53   "prod": "mongodb+srv://root:root@cluster0-53xnf.mongodb.net/piset-master"
54 };
55
56 let database;
57
58 switch (process.argv[2]){
59   case 'desa':
60     database = uris.desa;
61     app.set('database', uris.desa);
62     break;
63
64   case 'test':
65     database = uris.test;
66     break;
67
68   case 'prod':
69     database = uris.prod;
70     break;
71
72   default:
73     database = uris.desa;
74     break;
75 }
76
77
78 mongoose.connect(database, {options: {useNewUrlParser: true, useUnifiedTopology: true}}).then( () => {
79   console.log('Connected to database!');
80   app.set('mongoose', mongoose);
81 }).catch((err) => {
82   console.log(err);
83   console.log('Connection to database failed!');
84 });
```



Una vez hecho todo esto, se crea el servidor pasándole el módulo de Express y se le pide que escuche sobre el puerto.

```
86  ▶ const server = http.createServer(app);  
87  server.listen(port);
```

6.6.3.2. app.js

En este archivo se define el módulo Express. El primer paso es importar la librería Express e importar todos los archivos de las rutas, que es donde se añaden las peticiones que tiene que atender el servidor.

```
1  const express = require('express');  
2  const bodyParser = require("body-parser");  
3  const usersRoutes = require('./routes/users');  
4  const groupsRoutes = require('./routes/groups');  
5  const tasksRoutes = require('./routes/tasks');  
6  const historiesRoutes = require('./routes/histories');  
7  const invitationsRoutes = require('./routes/invitations');  
8  const devicesRoutes = require('./routes/devices');  
9  const achievementsRoutes = require('./routes/achivements');  
10 const statesRoutes = require('./routes/states');  
11 const testRoutes = require('./routes/test');  
12 const Group = require('./models/group');  
13 const cron = require('node-cron');  
14 const request = require('request');  
15 const app = express();
```

Una vez importado todo lo necesario e inicializado el módulo Express, hay que indicarle al servidor que transforme el contenido que envía el cliente a formato JSON, para ello se usa la librería llamada “body-parser” y se le indica al servidor que la utilice a modo de middleware:

```
21  app.use(bodyParser.json());
```

Ahora tocaría indicar al servidor que use los archivos de las rutas que contienen las peticiones que hay que atender del cliente. Esto se implementa al igual que el middleware anterior:

```

84     app.use('/api/users', usersRoutes);
85
86     app.use('/api/groups', groupsRoutes);
87
88     app.use('/api/tasks', tasksRoutes);
89
90     app.use('/api/histories', historiesRoutes);
91
92     app.use('/api/test', testRoutes);
93
94     app.use('/api/history', historiesRoutes);
95
96     app.use('/api/invitations', invitationsRoutes);
97
98     app.use('/api/devices', devicesRoutes);
99
100    app.use('/api/achivements', achivementsRoutes);
101
102    app.use('/api/states', statesRoutes);

```

Y finalmente habría que exportar el módulo Express para que pueda ser usado en el servidor.

```

104    module.exports = app;

```

Con esto ya estaría implementado el servidor, a falta de la implementación de las peticiones que tiene que atender del cliente. En gran medida, estas peticiones son operaciones sobre la base de datos, por lo que antes de explicar las peticiones, hay que explicar cómo se ha implementado el mapeo entre los datos de la base de datos y objetos en formato JSON que es lo que le llegará al cliente.

6.6.3.3. modelos

Para explicar la implementación de los modelos se va a usar como ejemplo el modelo del objeto usuario, ya que es el más representativo al tener varios tipos de campos y dos campos de relaciones a otros modelos.

```

1  const mongoose = require('mongoose');
2  const userGroupSchema = require ('../models/userGroup').schema;
3  const userAchievementSchema = require ('../models/userAchievement').schema;
4
5  const userSchema = mongoose.Schema({
6    mail: { type: String, required: true},
7    password: { type: String, required: true},
8    name: { type: String, required: true},
9    secondName: { type: String, required: true},
10   admin: {type: Boolean, required: true},
11   groups: {type: [userGroupSchema], required: false},
12   achievements: {type: [userAchievementSchema], required: false},
13 });
14
15 module.exports = mongoose.model( name: 'User', userSchema);

```

En la primera línea se obtiene la variable mongoose de la librería usada para la creación del modelo. También hay que obtener los dos modelos con los que se va a relacionar el modelo del usuario, que son los objetos de usuario-grupo y usuario-logro. Son los dos objetos auxiliares definidos en la implementación de la base de datos. Después de requerir todo lo necesario, se declara el modelo usando la función mongoose.Schema(modelo), donde modelo es un objeto que contiene cada campo que se quiera declarar. Por lo que se declara un campo, el tipo y si es requerido o no. Por ejemplo, el caso del campo mail:

```

6    mail: { type: String, required: true},

```

Este campo se declara en formato JSON, donde la clave es el nombre del campo (mail) y el valor es otro objeto JSON con los atributos del tipo y si es requerido.

Para implementar las relaciones de base de datos como modelo de objeto, simplemente hay que declarar como tipo de campo el modelo de Mongoose del objeto al que se hace referencia, se ve más claro en el ejemplo:

```

11   groups: {type: [userGroupSchema], required: false},

```

Con toda esta implementación de los modelos de objetos de base de datos, lo que se consigue es indicarles a las consultas de base de datos de Mongoose como recuperar los objetos y cómo crearlos para luego enviárselo al cliente que los ha solicitado, en vez de crear los objetos a mano al finalizar cada consulta. Así se consigue simplificar las consultas futuras a bases de datos.

6.6.3.4. rutas

En estos archivos están definidas las consultas que estará atendiendo el servidor, como obtiene los datos que le pasa el cliente, como hace lo que le pide el cliente y como le devuelve la respuesta al cliente. Únicamente se explican cómo se han implementado algunos casos representativos. Por ejemplo, la petición de dar de alta a un usuario:


```

152 router.post( path: '/signup', handlers: (req : Request<P, ResBody, ReqBody, ReqQuery> , res : Response<ResBody> , next : NextFunction ) => {
153   bcrypt.hash(req.body.password, salt: 10).then(hash => {
154     const user = new User({
155       'mail': req.body.mail,
156       'password': hash,
157       'name': req.body.name,
158       'secondName': req.body.secondName,
159       'admin': false,
160       'groups': [],
161       'achievements': []
162     });
163     user.save().then(result => {
164       res.status( code: 201).json( body: {
165         message: 'Success',
166         user: result
167       });
168     }).catch(err => {
169       res.status( code: 500).json( body: {
170         error: err
171       });
172     });
173   });
174 });

```

Cuando llega una petición de registro de usuario por parte del cliente, lo primero que se hace es encriptar la contraseña del usuario. Una vez encriptada, se crea el objeto de usuario usando el modelo de Mongoose definido anteriormente. Gracias a este modelo, se puede hacer la petición de guardado a la base de datos únicamente usando un método. Llamando a `user.save()` se guarda el usuario en base de datos, y cuando se guarde se responde la petición al cliente con `res.status(201)` o en caso de error `res.status(500)`.

Ahora, el caso de la petición de acceso de un usuario:

```

176 router.post( path: '/signin', handlers: (req : Request<P, ResBody, ReqBody, ReqQuery> , res : Response<ResBody> , next : NextFunction ) => {
177   let fetchedUser;
178
179   User.findOne({mail: req.body.mail}).then(user => {
180     if (!user) {
181       return res.status( code: 401).json( body: {
182         message: 'Auth failed'
183       });
184     }
185     else {
186       fetchedUser = user;
187       return bcrypt.compare(req.body.password, user.password);
188     }
189   }).then(result => {
190     if (!result) {
191       return res.status( code: 401).json( body: {
192         message: 'Auth failed'
193       });
194     }
195     else {
196       const token = jwt.sign( payload: {
197         mail: fetchedUser.mail,
198         userId: fetchedUser._id
199       }, secretOrPrivateKey: 'secret_this_should_be_longer', options: {expiresIn: '1h'});
200       res.status( code: 201).json( body: {
201         message: 'Success',
202         token: token,
203         user: fetchedUser
204       });
205     }
206   }).catch(err => {
207     console.log(err);
208   });
209 });
210 });

```

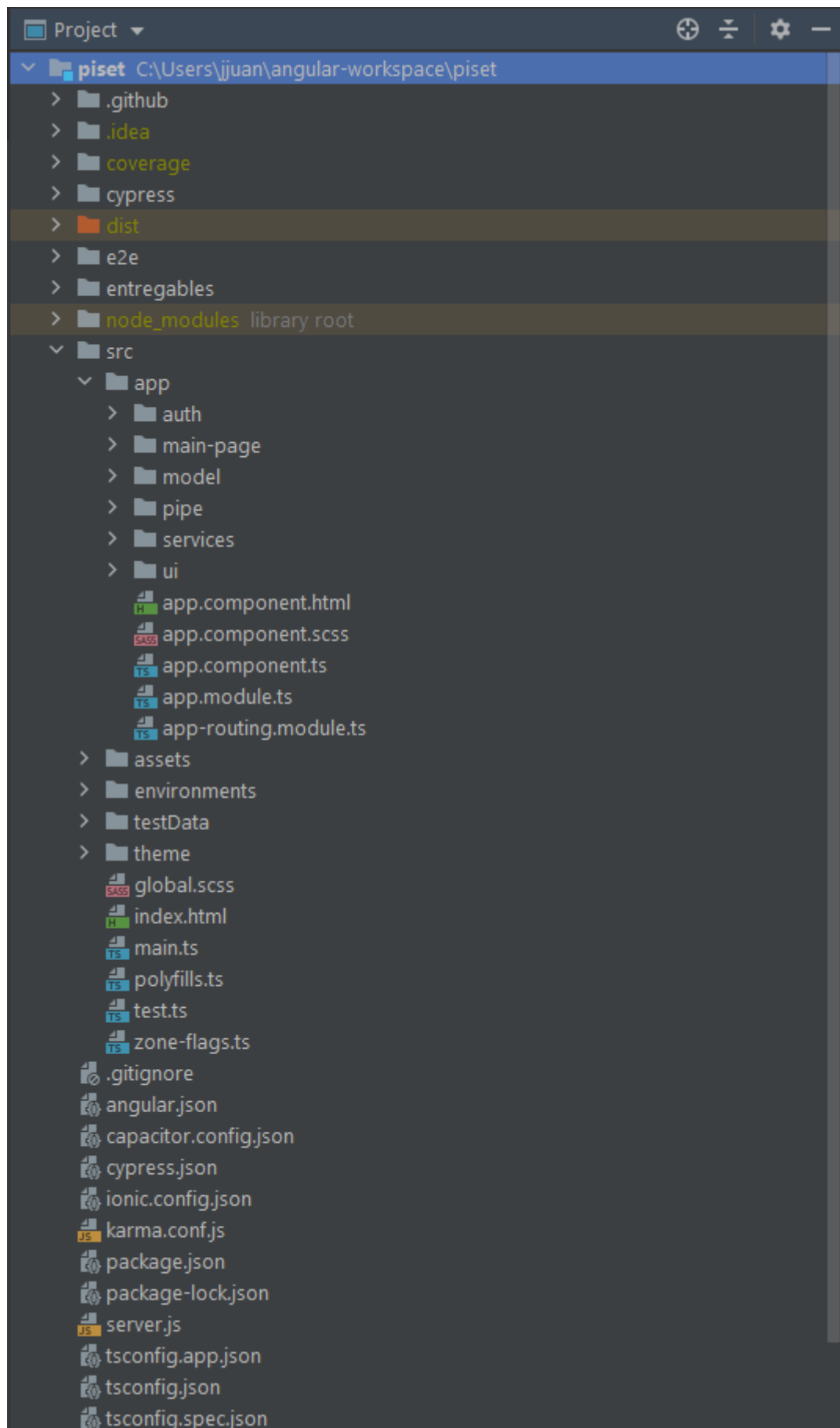
El comportamiento de esta petición es el siguiente, primero se busca el usuario según su correo usando el método Mongoose sobre el modelo definido `User.findOne()` y filtrando por el correo, que se obtiene de la petición del cliente en `req.body.mail`. Si no encuentra el usuario se devuelve al cliente un error 401 con un mensaje de que ha fallado la autenticación. Si encuentra el usuario, se almacena en la variable `fetchedReader` y se devuelve la comparación entre las dos contraseñas al siguiente callback en la variable `result`. Si las contraseñas son iguales, se crea un token de sesión y se da de alta en el servidor, este token se usa como validador de seguridad antes de cada consulta al servidor, se ha mencionado en apartados anteriores, y se le da una duración de una hora. Después solo queda devolver el token, el usuario y el status de completado al cliente.

6.7. Frontend

En esta parte se detalla todo el proceso de implementación de la parte con la que interactuará directamente el cliente. Por lo que se enfatizará en el desarrollo del interfaz de usuario, así como la navegación entre vistas y de cómo se hacen las peticiones al servidor.

6.7.1. Organización del proyecto

En este apartado se detalla la organización de ficheros del proyecto, explicando en que consiste cada componente para así, partir con una idea general de las partes que tiene y cómo interactúan entre ellas. La organización del proyecto es la siguiente:



Las carpetas de `.github`, `.idea` y `node_modules` no se explicarán porque el proyecto de la parte de servidor también los tienen y ya se han explicado anteriormente.

- `dist`: Se trata de los archivos compilados, es la página web que se compila al hacer el build del proyecto para su despliegue.
- `e2e` y `cypress`: Son las carpetas encargadas de las pruebas end-to-end, en la carpeta `e2e` se almacenan las pruebas con la herramienta Protractor, que en este proyecto no se usará. Y la carpeta `cypress` contiene las pruebas con la herramienta Cypress que se ha usado en este proyecto y que se detallan en el apartado de pruebas.
- `entregables`: Contiene ilustraciones sobre el modelo de datos y la memoria del proyecto.
- `src`: Se trata de la carpeta donde están alojados los archivos de código que se desarrolla. Dentro de esta carpeta se encuentra lo siguiente:
 - `app`: Dentro se encuentran los componentes o vistas de Angular, así como los servicios, las pipes, los modelos y el componente inicial, desde donde parte toda la aplicación llamado `app.component`.
 - `assets`: Aquí se almacenan los archivos estáticos de imágenes usadas en la interfaz.
 - `environments`: Se trata de los archivos de entornos, en estos archivos se definen las variables de entorno, como por ejemplo la URL del servidor en los diferentes entornos o si se trata de un entorno de prueba o de producción. Al ejecutar la aplicación, se puede especificar el entorno para que se elija un archivo u otro.
 - `testData`: Aquí se encuentran los archivos de datos para testear la integración de las peticiones al servidor. Su uso se explicará en el apartado de pruebas de integración.
 - `theme`: Contiene variables de hojas de estilos.

6.7.2. Dependencias

En este apartado se enumeran y explican las dependencias o librerías que tiene instalado este proyecto. Las dependencias del archivo `package.json` son las siguientes:

```

16  "dependencies": {
17    "@angular/animations": "~10.0.0",
18    "@angular/cdk": "~10.0.0",
19    "@angular/common": "~10.0.0",
20    "@angular/core": "~10.0.0",
21    "@angular/forms": "~10.0.0",
22    "@angular/material": "~10.0.0",
23    "@angular/platform-browser": "~10.0.0",
24    "@angular/platform-browser-dynamic": "~10.0.0",
25    "@angular/router": "~10.0.0",
26    "@capacitor/core": "2.4.2",
27    "@ionic-native/core": "^5.0.0",
28    "@ionic-native/splash-screen": "^5.0.0",
29    "@ionic-native/status-bar": "^5.0.0",
30    "@ionic/angular": "^5.0.0",
31    "express": "^4.17.1",
32    "immutable": "^3.7.6",
33    "moment": "^2.29.1",
34    "rxjs": "~6.5.5",
35    "tslib": "^2.0.0",
36    "zone.js": "~0.10.3"
37  },

```

- @angular: Se trata de las librerías que necesita todo proyecto Angular, son necesarias para la aplicación.
- @ionic: Se trata de una librería que permite adaptar luego un proyecto web a una aplicación móvil. Se ha usado por si en un futuro se quiere pasar la aplicación a móviles. La única pega es que hay que usar sus interfaces predefinidas, que se verán más adelante en la implementación.
- express: La librería Express para poder iniciar un pequeño servidor con la aplicación a la hora de desplegar.
- moment: Añade funcionalidades para tratar con variables de tipo fecha.
- rxjs: Permite usar los objetos de tipo Observable, se detallará su uso más adelante en la implementación.

6.7.3. Implementación

En esta parte se detallan las partes más importantes del proyecto y como se ha desarrollado el código para llegar a cumplir los requisitos que se definieron anteriormente.

6.7.3.1. Entornos

En este apartado se explica cómo se implementa la diferenciación de entornos que pueda tener la aplicación.

Los entornos en los que se moverá la aplicación son los siguientes: desarrollo, test y producción. Esto plantea un problema a la hora de hacer las peticiones al servidor, y es que el servidor también tiene diferentes entornos, en este caso los mismos, por lo que la parte de cliente y la parte de servidor tienen que tener una concordancia de entornos, es decir, sería desastroso que el entorno de desarrollo del cliente hiciera peticiones al entorno de producción del servidor.

Angular permite implementar una solución a este problema usando dos cosas: sus archivos de entorno y el archivo `angular.json` donde se configuran los diferentes parámetros de ejecución según entorno.

Los archivos de entorno, como se ha mencionado en la organización del proyecto, son una serie de archivos que contienen las variables de entorno. Aquí dos ejemplos:

```
5 export const environment = {
6   environment: 'dev',
7   production: false,
8   API_URL: 'http://localhost:3000'
9 };
```

El primero contiene las variables para el entorno de desarrollo, se define el nombre del entorno, si se trata de un entorno de producción y la URL de conexión al servidor, en este caso, al tratarse del entorno de desarrollo, la parte del cliente se conecta al servidor inicializado en local.

El siguiente ejemplo es el del entorno de test:

```
1 export const environment = {
2   environment: 'test',
3   production: true,
4   API_URL: 'https://pisetserver-test.herokuapp.com'
5 };
```

En este caso se define el entorno como de producción, ya que el entorno de test es una copia del entorno de producción. Y la URL de conexión al servidor de test.

El archivo del entorno de producción es igual que el anterior, pero con la URL dirigida al servidor de producción.

La otra parte necesaria para solucionar el problema de los entornos se encuentra en el archivo `angular.json`. Dentro de este archivo se encuentra declarado un objeto llamado `configurations`, que contiene una lista con las configuraciones en las que se inicializará la aplicación, y los diferentes parámetros que usa cada una. Hay un parámetro que indica que archivo de entorno se usa, que es el siguiente:

```
"production": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.prod.ts"
    }
  ]
},
```

Con esto se le indica que compile la aplicación usando el archivo de entorno de producción.

El parámetro de configuración para el entorno de test es el siguiente:

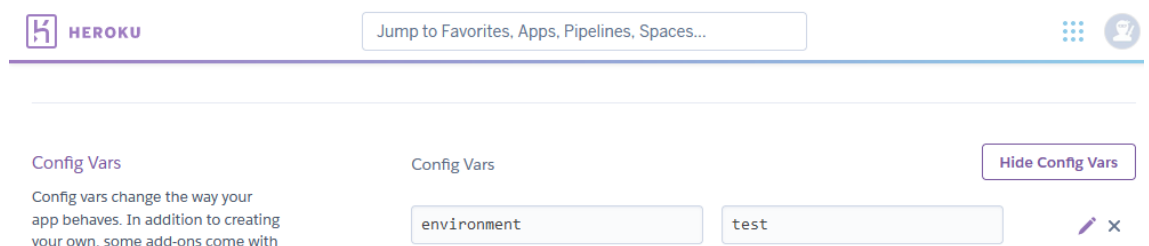
```
"test": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.test.ts"
    }
  ]
},
```

En este caso se usa el entorno de test.

Para finalizar, hay que indicar en el archivo package.json la forma en la que se va a indicar el entorno en el comando de compilación.

```
9 ▶ "build": "ng build --configuration=${environment}",
```

En el apartado de scripts, se define el anterior comando, donde “\${environment}”, es una variable declarada en el servidor remoto Heroku, y es la que se usará en el comando. Aquí se ve cómo está declarada esta variable:



Con todo esto ya está solucionada la selección del entorno correcto del cliente y su correlación con el entorno del servidor. Todo esto a la hora del despliegue, ya que la variable que decide el entorno se encuentra en el propio servidor, por lo que una vez se ejecute la integración continua, al desplegar en el servidor ya se asigna automáticamente el entorno en el que operará la aplicación.

6.7.3.2. Modelos

En esta parte de la aplicación se definen los modelos de los objetos con los que trabaja la aplicación. Estos modelos recuerdan a los modelos que se definieron en el apartado del servidor. Son iguales en gran medida porque lo que devuelve el servidor tiene que mapearse a los modelos definidos en el cliente. Se trata de archivos que permiten definir los atributos de un objeto y su constructor, para así crear los objetos, muy parecido a las clases de Java. Se expondrán los dos modelos más representativos de la aplicación, el usuario y la tarea.

El modelo de usuario es el siguiente:

```

1  import {UserGroup} from './userGroup';
2  import {UserAchiement} from './userAchiement';
3
4  export class User {
5      _id: string;
6      mail: string;
7      password: string;
8      name: string;
9      secondName: string;
10     admin: boolean;
11     groups: UserGroup[];
12     achivements: UserAchiement[];
13
14     constructor(mail: string, password: string, name: string, secondName: string,
15                 admin: boolean, groups: UserGroup[],
16                 achivements: UserAchiement[], _id?: string) {
17         this.mail = mail;
18         this.password = password;
19         this.name = name;
20         this.secondName = secondName;
21         this.admin = admin;
22         this.groups = groups;
23         this.achivements = achivements;
24         this._id = _id || '';
25     }
26 }

```

En las primeras dos líneas se importan los otros dos modelos en los que se relaciona el actual.

Después se crea la clase de TypeScript, la forma de describirla es listar sus atributos indicando el tipo de variable, se pueden ver algunas variables del tipo literal y una del tipo binario. Las dos últimas, son las relacionales y son de la tipología de lista de objetos del tipo UserGroup y UserAchiements, ya que un usuario puede tener varios objetos de estos tipos.

Más abajo se define el constructor, usando este método para crear un objeto de este tipo. Se pasan los atributos de la clase al método del constructor, el indicador de “?” al lado del atributo del id, indica que es un atributo opcional.

El modelo de la tarea es el siguiente:


```

3  export class Task{
4      _id: string;
5      name: string;
6      description: string;
7      groupId: string;
8      groupName: string;
9      userId: string;
10     userName: string;
11     dateIni: Date;
12     dateEnd: Date;
13     estimatedTime: number;
14     state: State;
15
16     constructor(name: string, description: string, groupId: string, groupName: string,
17                 |userId?: string, userName?: string, dateIni?: Date, dateEnd?: Date,
18                 estimatedTime?: number, state?: State, _id?: string){
19         this.name = name;
20         this.description = description;
21         this.groupId = groupId;
22         this.groupName = groupName;
23         this.userId = userId || null;
24         this.userName = userName || null;
25         this.dateIni = dateIni || null;
26         this.dateEnd = dateEnd || null;
27         this.estimatedTime = estimatedTime || null;
28         this.state = state || null;
29         this._id = _id || null;
30     }

```

La definición de este modelo sigue los patrones del anterior, a diferencia que hay varios atributos que pueden ser opcionales, y si lo son, se inicializan a nulos en el objeto del tipo tarea. Esto se puede ver en la asignación de atributos siguiente:

```

23     this.userId = userId || null;

```

Los demás modelos de la aplicación se definen usando el mismo patrón de clases de TypeScript, estas clases son las características de la programación orientada a objetos que añade TypeScript a JavaScript.

6.7.3.3. Servicios

En este apartado se va a explicar en qué consisten los servicios en Angular, para que son útiles y como se han implementado en la aplicación.

Los servicios se encargan de toda la gestión de los datos que use la aplicación. Es decir, aquí estarán las variables observables mencionadas en apartados anteriores y todas las llamadas al servidor. Cada una de estas llamadas, en su callback, rellenará los datos de las variables Observables, estas variables son las encargadas de alimentar de datos a las vistas o componentes, en el apartado de componentes se verá como los usan.

Primero se define que es un Observable, o el subtipo que usa esta aplicación llamado BehaviorSubject. Los Observables tienen la función de actualizar los datos automáticamente cuando cambian en la vista en los que son usados. Es decir, una vista se suscribe a una variable Observable y el servicio alimenta de datos ese Observable,

con lo que los datos que muestra la vista se modifican automáticamente. La forma de definirlo es la siguiente:

```
16 public _tasksGroups: BehaviorSubject<List<Task>> = new BehaviorSubject(List<array> ([]));
```

Esto define una variable del tipo BehaviorSubject, que es un subtipo de Observable. Se define el tipo de datos que va a manejar, que en este caso es una lista de tareas “List<Task>” y se inicializa con “new”. A esta variable es a la que se le pasarán los datos al finalizar una consulta al servidor y de la que leerá en las vistas.

Una vez explicado el concepto de Observable, se va a ver cómo se define un servicio en Angular y como se ha implementado el servicio más representativo de toda la aplicación, el servicio de tareas:

```
1 import { Injectable } from '@angular/core';
2 import { Group } from '../model/group';
3 import { Task } from '../model/task';
4 import { BehaviorSubject, Observable, Subscription } from 'rxjs';
5 import { HttpClient, HttpParams } from '@angular/common/http';
6 import { List } from 'immutable';
7 import { User } from '../model/user';
8 import { environment } from '../../environments/environment';
9 import { UserGroup } from '../model/userGroup';
10
11 @Injectable({
12   providedIn: 'root'
13 })
14 export class TaskStorageService {
15   private API_URL = environment.API_URL;
16   public _tasksGroups: BehaviorSubject<List<Task>> = new BehaviorSubject(List<array> ([]));
17   public _tasksGroup: BehaviorSubject<List<Task>> = new BehaviorSubject(List<array> ([]));
18
19   constructor(private http: HttpClient) {}
20 }
```

Al inicio del fichero se declaran las dependencias que hacen falta importar, aquí está todo lo relacionado con los Observables: BehavoirSubjet, Observable, Subscription. También está el tipo de dato de Lista y los modelos que hacen falta, así como el fichero del entorno para obtener la URL del servidor, para hacer la consulta.

Más adelante se define la clase del servicio y las variables, al final en el constructor se le pasa la clase HttpClient, que es la encargada de realizar las peticiones.

A partir de aquí se definen las peticiones que hace el cliente al servidor, que son los métodos que se ejecutan desde las vistas o componentes Angular, se van a explicar las más representativas.

```
35 getGroupTasks(group: Group) {
36   return this.http.get<{message: string, tasks: any}>({ url: this.API_URL + '/api/tasks/getByGroup' + group._id}).subscribe(
37     next res => {
38       const tasks = (res.tasks as Task[]).map((task: any) =>
39         new Task(task.name, task.description, task.groupId, task.groupName, task.userId, task.userName, task.dateIni, task.dateEnd,
40           task.estimatedTime, task.state, task._id));
41       this._tasksGroup.next(List(tasks));
42     },
43     error err => console.log('Error retrieving Todos')
44   );
45 }
```

El código anterior es la función de obtención de las tareas que tiene un grupo, que se pasa como argumento. La función utiliza la librería HttpClient, instanciada en el constructor en la variable http, para hacer una consulta HTTP del tipo GET. En esta consulta se puede definir el formato de la respuesta que dará el servidor, en este caso espera un mensaje del servidor y las tareas. El método get de la librería httpClient se le pasa como argumento la URL de la consulta, en este caso le pasa la ruta del servidor, que obtiene del archivo de entorno y le concatena la ruta de la consulta concreta y el identificador del grupo, que usará el servidor para hacer la consulta a la base de datos de las tareas que tiene el grupo. A esa consulta se puede suscribir usando el método subscribe, lo que quiere decir que se ejecuta la consulta, y el en callback del método subscribe se obtiene la respuesta del servidor y se ejecuta el código que sigue a continuación.

El código que se ejecuta como respuesta del servidor no es más que la obtención de los datos de respuesta y la transferencia de estos al objeto Observable BehaviorSubject que se definió anteriormente. Como se puede ver, se obtiene el objeto de las tareas que responde el servidor. Este objeto no es más que un objeto en JSON, pero se quiere mapear al modelo de objeto de tarea que se definió anteriormente. Para ello se usa la función map de JavaScript. La función map ejecuta cambios en cada elemento de la lista en el que es ejecutado. En este caso se obtiene el elemento y se crea un objeto del tipo tarea. Esto resulta en una lista de elementos del tipo tarea, que es lo que se transfiere a la lista observable usando el método next.

Con esto se ha explicado el mecanismo básico que tiene el cliente para hacer peticiones al servidor, obtener su respuesta, transformarla a un tipo de dato más manejable por la aplicación y transferirla a las listas Observables, que serán consumidas por la parte de la vista o componentes de Angular.

Las demás consultas de ejemplo, siguen el mismo mecanismo explicado anteriormente, pero con algunas particularidades que se explicarán a continuación.

```
47 addTaskToGroup(task: Task){
48   this.http.post( url: this.API_URL + '/api/tasks/addToGroup', body: {task}).subscribe( next: response => {
49     // Add task to _tasksGroup and push
50     this._tasksGroups.next(this._tasksGroups.getValue().push(task));
51   });
52 }
```

La consulta anterior se trata de una del tipo POST, es decir, se esta mandando información al servidor para que la almacene. Para hacerlo, no hay más que añadir la información como atributo del método post, en este caso se pasa la tarea que se va a guardar. La respuesta no es más que transmitir al observable, el mismo observable más la nueva tarea usando el método next para pasar el observable más la tarea insertada usando el método push.

```
63 updateTask(updatedTask: Task){
64   this.http.post( url: this.API_URL + '/api/tasks/update', body: {task: updatedTask}).subscribe( next: response => {
65     const tasks: List<Task> = this._tasksGroups.getValue();
66     const index = tasks.findIndex( predicate: (task :Task| undefined ) => task._id === updatedTask._id);
67     this._tasksGroups.next(tasks.set(index, updatedTask));
68   });
69 }
```

La consulta anterior tiene la particularidad de que se actualiza un valor, por lo tanto, es una consulta del tipo POST y hay que modificar el valor en la lista observable.



Para ello, se obtiene la posición del elemento usando la función `findIndex` con el atributo identificador de la tarea y luego se pasa la lista con el atributo modificado usando la función `set`.

```

54 deleteTaskFromGroup(deletedTask: Task){
55   this.http.post<Task>(url: this.API_URL + '/api/tasks/deleteFromGroup', body: {taskId: deletedTask._id}).subscribe( next: response => {
56
57     const tasks: List<Task> = this._tasksGroups.getValue();
58     const index = tasks.findIndex( predicate: (task :Task|undefined ) => task._id === deletedTask._id);
59     this._tasksGroups.next(tasks.delete(index));
60   });
61 }

```

La consulta anterior es muy parecida a lo que se ha explicado, pero al eliminar un elemento, tiene la particularidad de que se tiene que eliminar de la lista usando la función `delete` para pasársela al observable.

6.7.3.4. Pipes

Los pipes son una herramienta de Angular que permite transformar visualmente la información, por ejemplo, cambiar un texto a mayúsculas o darle formato de fecha y hora o filtrar información de una lista de valores, que es lo que se va a explicar a continuación.

Para usar un pipe no hay más que añadirlo a la derecha de la lista de la que se está iterando usando el operador “|”, seguido de todos sus argumentos después del operador “:”. A continuación, se puede ver su uso en el componente del listado de historiales de acciones:

```

60 <ion-row *ngFor="let history of (historyStorage._historiesGroups | async| historyFilter : selectedGroup:selectedUser:selectedTask:selectedDate); index as i">
61   <ion-col>{{ history.taskName }}</ion-col>
62   <ion-col>{{ history.userName }}</ion-col>
63   <ion-col>{{ history.date.toString() | date: format: 'dd/MM/yyyy' }}</ion-col>
64   <ion-col>{{ history.action }}</ion-col>
65   <ion-col>{{ history.time }}</ion-col>

```

Aquí se puede ver que se está iterando la lista observable obtenida del servicio `historyStorage._historiesGroups`. Se están filtrando sus datos usando el pipe `historyFilter` y los argumentos del grupo, usuario, tarea y fecha seleccionada.

Ahora se explica cómo se ha implementado el pipe.

```

1 import { Pipe, PipeTransform } from '@angular/core';
2
3 @Pipe({
4   name: 'historyFilter'
5 })
6 export class HistoryFilterPipe implements PipeTransform {
7
8   transform(value: any, ...args: any[]): any {
9     const filtered = [];
10
11     const selectedGroup = args[0];
12     const selectedUser = args[1];
13     const selectedTask = args[2];
14     const selectedDate = args[3];
15
16     value.forEach(element => {
17       if (((selectedGroup === undefined) || (element.groupId === selectedGroup._id))
18         && ((selectedUser === undefined) || (element.userId === selectedUser._id))
19         && ((selectedTask === undefined) || (element.taskId === selectedTask._id))
20         && ((selectedDate === undefined) ||
21           (new Date(element.date).getMonth() === selectedDate.getMonth() && new Date(element.date).getFullYear() === selectedDate.getFullYear()))){
22         filtered.push(element);
23       }
24     });
25     return filtered;
26   }
27 }

```

El pipe no es más que una función que transforma la lista a filtrar, en este caso el argumento llamado value, usando una serie de valores que se le pasan en la vista, llamado args. Lo que se hace en este caso es obtener los valores por los que se va a filtrar e iterar la lista usando el método forEach y obtener los valores de la lista que concuerdan con las condiciones definidas y añadirlas a la lista de filtrado. Al terminar la iteración de retorna la lista filtrada, que es la que se verá en la vista o componente. Este método de filtrado de ejecuta cada vez que algún argumento es cambiado en la vista, por lo que en la vista se verá que automáticamente se filtra la lista de valores al cambiar un argumento.

6.7.3.5. Navegación entre vistas o componentes

Para definir la navegación entre los componentes que forman la aplicación Angular, hay que crear un archivo llamado app-routing.module.ts y definir en el componente principal de la aplicación, el app.component. Para ello se incluye dentro la etiqueta “<ion-router-outlet>”, dentro de esta etiqueta se irán instanciando los componentes que toquen según en la ruta en la que esté el usuario:

```

1 <ion-app>
2   <ion-router-outlet></ion-router-outlet>
3 </ion-app>

```

En el archivo app-routing.module.ts se define la constante de las rutas:

```

17  const routes: Routes = [
18    { path: '', redirectTo: '/signIn', pathMatch: 'full'},
19    { path: 'signUp', component: SignUpComponent},
20    { path: 'signIn', component: SignInComponent},
21    { path: 'main', component: MainPageComponent, canActivate: [AuthGuard], children: [
22      { path: 'admin', canActivate: [AuthGuard], children: [
23        {path: 'taskManagement', component: TaskManagementComponent},
24        {path: 'userManagement', component: UserManagementComponent},
25        {path: 'test', component: TestComponent},
26      ]
27    },
28    { path: 'user', canActivate: [AuthGuard], children: [
29      {path: 'info', component: UserInfoComponent},
30      {path: 'settings', component: UserSettingsComponent},
31      {path: 'invitations', component: UserInvitationsComponent}
32    ]
33  },
34    { path: '', redirectTo: 'tasks', pathMatch: 'full'},
35    { path: 'tasks', canActivate: [AuthGuard], component: TasksComponent},
36    { path: 'history', canActivate: [AuthGuard], component: HistoryComponent},
37  ]
38  ];

```

Como se ve, el objeto rutas contiene las relaciones entre las rutas y el componente que tienen que abrir. Así como la relación entre las rutas padre e hijas, como muestra el ejemplo de la ruta “user” y sus rutas hijas: “info”, “settings” y “invitations”. También se pueden definir las redirecciones a las rutas que sean necesarias, por ejemplo, para un usuario que acaba de acceder a la aplicación, su ruta inicial sería la absoluta o la ruta vacía. Si entra a esta ruta lo redirige a la vista de acceso, como muestra la siguiente entrada en el objeto rutas:

```

18  { path: '', redirectTo: '/signIn', pathMatch: 'full'},

```

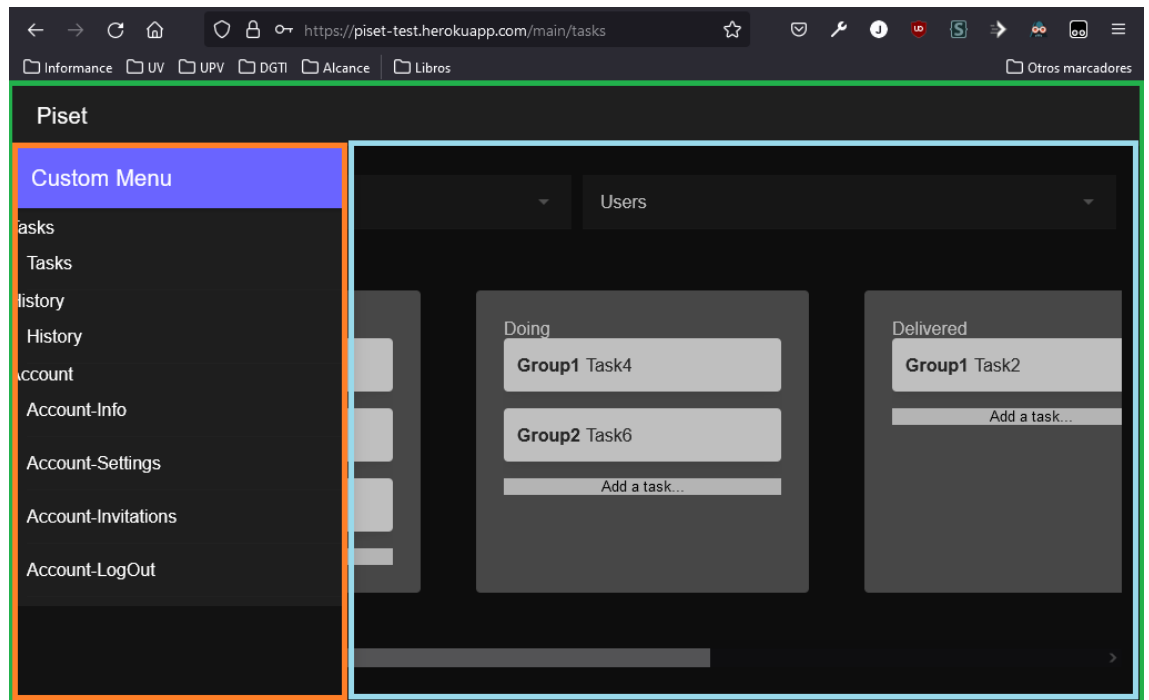
También se puede implementar esta misma funcionalidad para rutas hijas, es decir, si el usuario accede a una ruta determinada, se le redirigirle automáticamente a alguna ruta hija. En el caso de la aplicación, se observa que hay una redirección desde la ruta vacía a la vista de las tareas, como se puede apreciar en esta línea:

```

34  { path: '', redirectTo: 'tasks', pathMatch: 'full'},

```

Para esta aplicación se ha diseñado el sistema de navegación para que el usuario, una vez autenticado, se le muestre una pestaña inicial, en la que, si pulsa a la izquierda de su pantalla, se le muestra un desplegable con el menú de navegación, en el que, si pulsa, se le muestra la ventana concreta insertada como hija de esta pestaña principal. Inicialmente cuando entra a la pestaña principal, se le redirige automáticamente a la vista Kanban de las tareas. Aquí se ve la imagen:



Se puede ver en la imagen que la zona en verde es el componente principal. Este componente esta durante toda la navegación del usuario en la aplicación. La zona naranja es el menú, del componente principal, que se despliega al pulsar en la parte izquierda de la pantalla, y se oculta al pulsar en otra parte de la pantalla. La zona azul es el componente que se instanciara como componente hijo del componente principal, según lo que se pulse en el menú de navegación. Al ocultarse el menú de navegación se muestra la pantalla completa del componente hijo.

Como se verá más adelante, los componentes se forman por la vista en un archivo HTML y la parte del modelo en el archivo TypeScript. La implementación de la vista principal es la siguiente:

```

1  <ion-header>
2    <ion-toolbar>
3      <ion-row>
4        <ion-col>
5          <ion-title id="pisetTitle" (click)="openMenu( menuId: 'custom')">Piset</ion-title>
6        </ion-col>
7      </ion-row>
8    </ion-toolbar>
9  </ion-header>
10 <ion-content class="ion-padding">
11   <ion-router-outlet id="main"></ion-router-outlet>
12   <ion-menu id="ionMenu" side="start" menuId="custom" contentId="main" class="my-custom-menu">
13     <ion-header>
14       <ion-toolbar color="tertiary">
15         <ion-title>Custom Menu</ion-title>
16       </ion-toolbar>
17     </ion-header>
18     <ion-content>
19       <ion-list>
20         <ion-label>Tasks</ion-label>
21         <ion-item id="clickTasks" (click)="onClickTasks()">Tasks</ion-item>
22
23         <ion-label>History</ion-label>
24         <ion-item id="clickHistory" (click)="onClickHistory()">History</ion-item>
25
26         <div *ngIf="userLogged.admin">
27           <ion-label>Admin</ion-label>
28           <ion-item id="clickTaskManagement" (click)="onClickTaskManagement()">Admin-Tasks</ion-item>
29           <ion-item id="clickUserManagement" (click)="onClickUserManagement()">Admin-Users</ion-item>
30           <ion-item *ngIf="!getEnvironment()" id="clickTest" (click)="onClickTest()">Admin-Test</ion-item>
31         </div>
32
33         <ion-label>Account</ion-label>
34         <ion-item id="clickUserInfo" (click)="onClickUserInfo()">Account-Info</ion-item>
35         <ion-item id="clickUserSettings" (click)="onClickUserSettings()">Account-Settings</ion-item>
36         <ion-item id="clickUserInvitations" (click)="onClickUserInvitations()">Account-Invitations</ion-item>
37         <ion-item id="logout" (click)="logout()">Account-Logout</ion-item>
38       </ion-list>
39     </ion-content>
40   </ion-menu>
41 </ion-content>

```

Se puede ver que la vista tiene dos partes claramente diferenciadas, el contenido de la etiqueta “<ion-header>”, que representa la barra de arriba de la vista, y el contenido de la etiqueta “<ion-content>”, que representa el contenido del componente hijo al que navegará el usuario con la etiqueta “<ion-router-outlet>” y el menú de navegación que se despliega a la izquierda de la pantalla “<ion-menu>”. Dentro del contenido de la etiqueta del menú, está definida cada una de las entradas con dos etiquetas, el título “<ion-label>”, y el propio enlace para que al pulsar navegue al sitio que toca, “ion-item”, cada uno tiene un evento de clic al método que toca.

Los métodos anteriores que se ejecutan al pulsar una entrada del menú se declaran en el modelo del componente, es decir, en el archivo TypeScript:


```

7 import {Router} from '@angular/router';
8
9 @Component({
10   selector: 'app-main-page',
11   templateUrl: './main-page.component.html',
12   styleUrls: ['./main-page.component.scss'],
13 })
14 export class MainPageComponent implements OnInit {
15   loading = true;
16   userLogged: User;
17   private production = environment.production;
18
19   selectOptions = {
20     title: 'Pizza Toppings',
21     subTitle: 'Select your toppings',
22     mode: 'md'
23   };
24
25   constructor(private authService: AuthService, private menu: MenuController, private testService: TestService, private router: Router) { }
26
27   ngOnInit() {
28     this.userLogged = this.authService.getCurrentUser();
29     this.loading = false;
30   }
31
32   logout(){
33     this.authService.signOut();
34   }
35
36   openMenu(menuId){
37     this.menu.open(menuId);
38   }
39
40   onClickTasks(){
41     this.router.navigate( commands: ['/main/tasks']);
42     this.menu.close( menuId: 'custom');
43   }

```

Lo primero que se escribe a la hora de crear un componente en Angular es la parte de `@Component`, donde se declara el selector del componente, y la etiqueta que se usará en las otras vistas para instanciar este componente, la ruta de la vista HTML y la ruta de los estilos en SCSS.

Más adelante se crea la clase del componente, en el constructor se le pasa las dependencias que necesita, que se instanciarán automáticamente cuando se cree el componente. Aquí se definen varias funciones:

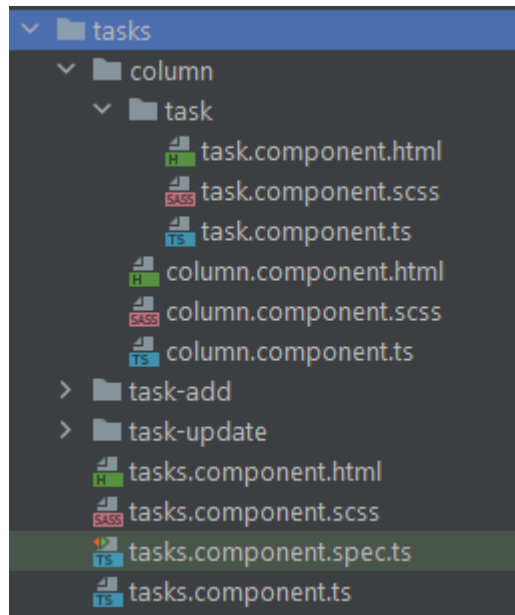
- `ngOnInit`: Es una función que se puede definir en todos los componentes, se ejecuta cuando se inicia el componente. En este caso se obtiene el usuario autenticado usando el servicio de autenticación.
- `logout`: Se elimina la sesión del usuario y se le redirige a la página de acceso. Se usa también el servicio de autenticación.
- `openMenu`: Abre el menú de navegación de la izquierda de la vista.
- `onClickTasks`: Este método usa la librería de rutas de Angular para navegar a una ruta concreta. Lo que, gracias al archivo de rutas definido, instanciará el componente correcto dentro de la etiqueta `<ion-router-outlet>` de la vista principal. Acto seguido cierra el menú de navegación.

Con todo esto queda implementado el entramado de la navegación entre vistas a las que puede acceder el usuario.

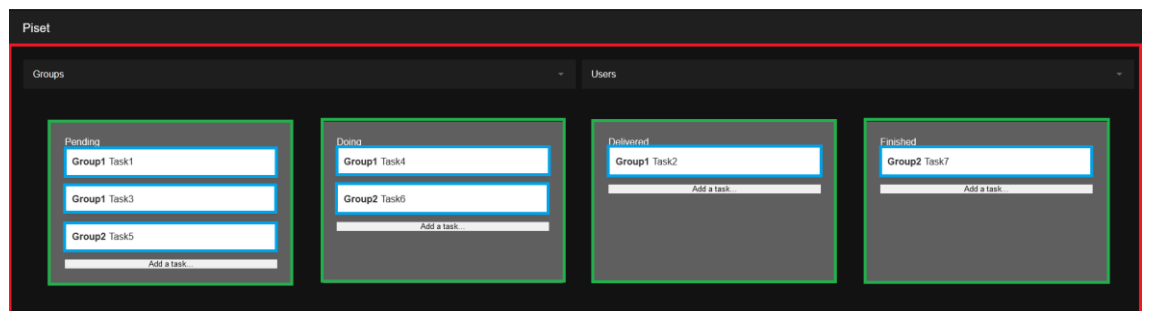
6.7.3.6. Componentes

En este apartado se detalla cómo se ha implementado una de las vistas más importantes de la aplicación, la vista Kanban de las tareas. Esta vista consta de varios

componentes Angular anidados y comunicados entre ellos. La estructura de componentes sería la siguiente:



Antes de explicar nada, la imagen de la vista quedaría de la siguiente forma:



Como se aprecia en la imagen, hay varios recuadros de diferentes colores para marcar que es cada componente. La zona marcada en rojo es el componente de las tareas, llamada tasks, en la parte de arriba tiene varios desplegables, que se usarán para filtrar las tareas que muestre el panel Kanban. Dentro de este componente hay un componente de columna por cada estado que puedan tener las tareas en este grupo, en la imagen están representados con el color verde. Dentro de cada columna, están todas las tareas que tengan el estado de la columna. Estas tareas están representadas con el color azul. Cada tarea se puede arrastrar con el ratón a la columna que desee el usuario, y se puede añadir una tarea a una columna pulsando en el botón de añadir tarea.

La vista del componente principal de las tareas quedaría de la siguiente forma:

```

1 <ion-content *ngIf="!loading" class="ion-padding">
2   <div *ngIf="!isUserInGroup">
3     <ion-label id="withoutGroupLabel">User don't have a group</ion-label>
4   </div>
5   <div *ngIf="isUserInGroup">
6     <ion-grid>
7       <ion-row>
8         <ion-col>
9           <ion-item>
10            <ion-label>Groups</ion-label>
11            <ion-select id="groupSelect" value="unselected" okText="Okay" cancelText="Dismiss"
12              (ionChange)="onGroupSelect($event)">
13              <ion-select-option *ngFor="let userGroup of (userService.getCurrentUser().groups)" [value]="userGroup" >
14                {{userGroup.group.name}}
15              </ion-select-option>
16            </ion-select>
17          </ion-item>
18        </ion-col>
19        <ion-col>
20          <ion-item>
21            <ion-label>Users</ion-label>
22            <ion-select id="userSelect" [value]="selectedUser" okText="Okay" cancelText="Dismiss"
23              (ionChange)="onUserSelect($event)">
24              <ion-select-option *ngFor="let user of (userService._usersGroup | async)" [value]="user" >{{user.name}}
25            </ion-select-option>
26          </ion-select>
27        </ion-item>
28      </ion-col>
29    </ion-row>
30    <ion-row>
31      <div class="board">
32        <div class="board-wrapper">
33          <div class="board-columns" cdkDropListGroup<!-- cdkDropListGroup-->
34            <app-column class="board-column"
35              *ngFor="let state of (stateService._states | async)" [state]="state" [group]="selectedGroup"
36              [user]="selectedUser">
37            </app-column>
38          </div>
39        </div>
40      </div>
41    </ion-row>
42  </ion-grid>
43 </div>
44 </ion-content>
45 <ion-content *ngIf="loading" class="ion-padding">
46   <app-loading-spinner></app-loading-spinner>
47 </ion-content>

```

Las dos etiquetas más externas del fichero definen si se muestra la vista normal o se muestra una rueda de cargando usando el atributo “*ngIf”, esto está porque la obtención de datos se hace de una forma asíncrona, por lo que puede pasar que cuando se cargue el componente, aún falten datos. Dentro de la etiqueta de la vista normal, también se puede mostrar la vista normal del panel Kanban o una vista con la etiqueta de que el usuario no tiene grupo, ya que, si no tiene ningún grupo, no hay posibilidad de tener tareas, por lo que hay que avisar al usuario.

Dentro de la vista Kanban (etiqueta de la sexta línea), se definen dos filas usando la etiqueta “<ion-row>”. La primera fila representa los desplegables de filtro de tareas. El primer desplegable lista los grupos en los que está el usuario y el segundo lista los usuarios que tiene el grupo. Los datos se obtienen usando las variables Observables de los servicios explicados en apartados anteriores, para el caso de los grupos se hace de la siguiente manera:

```

<ion-select id="groupSelect" value="unselected" okText="Okay" cancelText="Dismiss" [(ionChange)="onGroupSelect($event)">
  <ion-select-option *ngFor="let userGroup of (userService.getCurrentUser().groups)" [value]="userGroup" >
    {{userGroup.group.name}}
  </ion-select-option>
</ion-select>

```

Donde se puede ver en la etiqueta “<ion-select-option>” que se itera usando el atributo “*ngFor” sobre el método de suscripción al Observable que se explicó anteriormente:

```
*ngFor="let userGroup of (userStorage.getCurrentUser().groups)"
```

Con esto se crea una etiqueta “<ion-select-option>” por cada grupo en el que esté el usuario actual. El filtro por usuarios funciona de una forma similar.

Volviendo otra vez a la vista principal de tareas, la segunda fila contiene las columnas del panel Kanban, que, de una forma similar a como se crean dinámicamente las opciones en los filtros, de crean componentes de columnas en base a los estados de tareas que existan:

```
32 <div class="board-columns" cdkDropListGroup><!-- cdkDropListGroup-->
33 <app-column class="board-column"
34   *ngFor="let state of (stateStorage._states | async)" [state]="state" [group]="selectedGroup"
35   [user]="selectedUser">
36 </app-column>
37 </div>
```

Aquí se crean componentes de columnas por los estados que tenga. Además, a cada componente de columna, se le pasa el grupo y usuario seleccionados en el componente principal de tareas, esta es una de las formas de pasar datos de un componente padre a un componente hijo:

```
[group]="selectedGroup" [user]="selectedUser"
```

Ahora se detalla la implementación del componente de las columnas. Este componente, como se ha visto, se crea dinámicamente según los estados que hayan registrados. Además, se le pasa el grupo y usuario seleccionados en el componente padre de tareas. La vista quedaría de la siguiente manera:

```
<div class="board-column" id="{{state._id}}" cdkDropList [cdkDropListData]="taskStorageService._tasksGroups | async" (cdkDropListDropped)="drop($event)">
  <div class="column-title">
    {{ state.name }}
  </div>

  <div class="tasks-container"><!-- cdkDropList -->
    <app-task *ngFor="let task of (taskStorageService._tasksGroups | async | taskFilter : state.group:user)"
      [task]="task" (click)="openModalTaskUpdate(task)" cdkDrag [cdkDragData]="task">
    </app-task>
  </div>
</div>
<button class="btn" (click)="openModalTaskAdd()">Add a task...</button>
```

Como se puede apreciar, se ha definido un primer elemento con un atributo “cdkDropList”, esto indica que es una lista en la que se pueden arrastrar otros elementos. Se define también los elementos que tiene esta lista, que son las tareas que se obtienen del grupo:

```
[cdkDropListData]="taskStorageService._tasksGroups | async"
```

Y se define la función que se ejecuta cuando se arrastra un ítem a esta lista:

```
(cdkDropListDropped)="drop($event)"
```

Más adelante se define la lista de las tareas que contiene el componente de columna, se hace de una forma similar a como relaciona el componente padre de tareas con los componentes hijos de columnas. Aquí es lo mismo, pero relacionando el componente padre de columna con los componentes hijos de tareas. La lista de tareas hijas es una lista dinámica, ya que se van creando conforme llegan desde la respuesta del servidor. Este componente de tarea tiene el atributo “cdkDrag”, para indicar que es un elemento que se puede arrastrar. Y se definen sus datos con el atributo:

```
[cdkDragData]="task"
```

En la parte de modelo del componente de columna hay que mencionar la función:

```
50 drop(event: CdkDragDrop<string[]>) {  
51   const task = event.item.data;  
52   task.state = this.state;  
53   this.taskStorageService.updateTask(task);  
54 }
```

Esta función es la encargada de llamar al servicio de tareas para que actualice la tarea de estado, ya que se ha movido de columna de estado.

Ahora se detalla la vista del componente de tarea:


```
1 <div class="task" id="{{task._id}}">  
2   <span class="projectName">{{task.groupName}}</span>  
3   <span class="taskName">{{task.name}}</span>  
4 </div>
```

Aquí únicamente se muestran los datos de la tarea. Se añade el id del elemento al id de la tarea y se muestra el nombre del grupo y el nombre de la tarea.

El resto de las vistas, que no se van a explicar con detalle su implementación, ya que sigue el mismo procedimiento que la que se ha explicado, son las siguientes:

- Acceso:

Environment: test



Mail user1@user.com

Password ●●●●●●●●

[SIGN IN](#) [SIGN UP](#)

- Registro en la aplicación:

Register User

Mail

Password

Confirm Password

Name

Second name

[REGISTER](#) [BACK](#)

- Historial de acciones:

Piset

Group	Selec... ▾	Task	Selec... ▾	User	Selec... ▾	Date
Task	User	Date	Action	Time		
Task1	User1	30/11/2020	Finish	10		
Task2	User2	30/11/2020	Work	15		
Task3	User3	30/11/2020	Move state	0		
Task4	User4	30/11/2020	Move state	0		
Task5	User5	30/11/2020	Finish	20		
Task6	User6	30/11/2020	Work	40		
Task7	User7	30/11/2020	Work	30		

- Información de la cuenta:

Piset

Name

User1

Second Name

User1

Email

user1@user.com

- Ajustes de cuenta:

Piset

Name User1

Second name User1

UPDATE PROFILE

- Invitaciones a grupos:

Aplicación gestión de tareas para un grupo de personas

Piset

Group invited

Accept

Decline

Email

INVITE

7. Pruebas

Las pruebas son las investigaciones técnicas por las que se obtiene una información objetiva e independiente sobre la calidad del producto software. Se trata de una actividad más en el proceso de desarrollo de software. Los tipos de pruebas se pueden dividir en pruebas unitarias, pruebas de integración y pruebas E2E.

7.1. Pruebas unitarias

Las pruebas unitarias son aquellas que comprueban el correcto funcionamiento de una unidad de código, como podría ser una función. Cada una de las pruebas unitarias tienen que ejecutarse independientemente de las otras, es decir, una prueba no tiene que depender de la ejecución de otra.

En el caso de una aplicación escrita en Angular, el módulo de pruebas que se usa es Jasmine. Un archivo de prueba en Jasmine consta de los siguientes elementos:

- **describe:** Se trata de una agrupación de pruebas unitarias, es una forma de agruparlas por temas, por ejemplo, si se quiere probar todo un componente, todas sus pruebas unitarias irán dentro del describe correspondiente. En el describe se indica el nombre de las pruebas, así como las variables globales que van a usar las pruebas unitarias.
- **beforeEach:** Todo el código introducido aquí se ejecutará antes de cada prueba unitaria del describe. Se usa básicamente para instanciar los elementos necesarios para las pruebas, para estas instancias se usa un módulo llamado TestBed, que se encarga de aportar una instancia del componente, con todas sus dependencias, que se crea nuevo cada vez antes de cada prueba unitaria, así se garantiza la independencia entre pruebas.
- **it:** Aquí se declara la prueba. En esta parte se pueden usar diferentes mecanismos para hacer llamadas de prueba a métodos para ver cuántas veces se llama a una función, con que parámetros se llaman, todo esto sin que la función se llegue a ejecutar. También permite testear cada uno de los valores de las variables del componente, que valores retornan las funciones... Estos mecanismos son los siguientes:
 - **spyOn:** Esta función declara un espía para un método, este espía permite ver cuántas veces se llama o con que parámetros se le llama a la función. Se puede definir si la función llega a hacer algo, o si no hace nada y es solo para pruebas para ver si se tiene que llamar o no.
 - **expect:** Se trata de una función con la siguiente forma, `expect(variable).toBe(lo que tiene que ser)`. Esta función se puede usar para variables o para llamadas a funciones.
 - **el.query(By.css(identificador)):** Se trata de una función para obtener los elementos del DOM de la vista del componente.

Sobre este elemento se pueden comprobar los valores que tiene, por ejemplo, si tiene que estar visible o no.

- **afterEach:** Esta función se ejecuta después de cada una de las pruebas unitarias.

Una vez explicado la sintaxis de las pruebas en Jasmine, se va a explicar la implementación de las pruebas unitarias del componente del acceso/autenticación de la aplicación. Primero se detalla el describe de la prueba:

```
12 describe('SignInComponent', specDefinitions: () => {
13   let component: SignInComponent;
14   let fixture: ComponentFixture<SignInComponent>;
15   let el: DebugElement;
16
17   let authService: any;
18
19   let router: Router;
```

En esta parte se instancian las variables que van a ser usadas en todas las pruebas, todas las variables son utilidades o herramientas para facilitar las pruebas. Está la instancia del propio componente, la instancia del fixture, que se usa para actualizar la vista del DOM cuando se hacen cambios en la parte del modelo del componente. Está la variable del DebugElement, que se usa para obtener los elementos del DOM. La instancia del servicio de autenticación no se explica en este apartado, ya que tiene que ver con una prueba de integración. Y la última instancia de router sirve para el testeo de las redirecciones de rutas.

La siguiente parte es la del beforeEach, en esta parte se usa la utilidad de Angular a la hora de crear instancias para las pruebas llamada TestBed.

```
21 beforeEach(async( fn: () => {
22
23   TestBed.configureTestingModule( moduleDef: {
24     declarations: [ SignInComponent],
25     imports: [IonicModule.forRoot(), RouterTestingModule, HttpClientTestingModule, ReactiveFormsModule],
26     providers: [AuthService]
27   }).compileComponents().then(() => {
28     fixture = TestBed.createComponent(SignInComponent);
29     component = fixture.componentInstance;
30     el = fixture.debugElement;
31     authService = TestBed.get(AuthService);
32     router = TestBed.get(Router);
33     spyOn(router, method: 'navigate');
34     spyOn(authService, method: 'signInUser').and.callFake( fn: () => {
35       router.navigate( commands: ['/main']);
36     });
37     component.ngOnInit();
38     fixture.detectChanges();
39   });
40 });
```

En esta parte se configura el TestBed, que tiene tres apartados principales:

- **declarations:** Aquí se declara el componente que se va a testear, en este caso el SignInComponent.

- **imports:** En esta parte se instancian todas las dependencias que pueda necesitar el componente. En este caso hacen falta los módulos de rutas, que se usará la extensión a posta para pruebas llamada `RouterTestingModule`, hará falta también el `HttpClientTestingModule`, que es necesario para hacer pruebas de llamadas al servidor, que se explicarán en el apartado de pruebas de integración. Y el módulo de pruebas para los formularios.
- **providers:** Aquí se insertan los servicios de los que obtiene los datos o utiliza el componente. En el caso de este componente, se declara el servicio de autenticación.

Una vez se compila el módulo que se ha creado con el TestBed, se crea el fixture usando la función de crear el componente, este fixture se usa para obtener los elementos del DOM y se instancian en las variables globales los servicios y módulos necesarios para las pruebas unitarias que se van a hacer. Finalmente espían los métodos que se quieren que no se lleguen a ejecutar y se inicia el componente y se detectan los cambios en el DOM usando la función `“fixture.detectChanges()”`

Ahora se va a explicar la implementación de las pruebas que se han considerado representativas.

```

46  it( expectation: 'form invalid when empty', assertion: () => {
47    // Form should be invalid at start
48    expect(component.form.valid).toBeFalsy();
49
50    fixture.detectChanges();
51
52    // Sign in button should be disabled
53    const signInButton = el.query(By.css( selector: '#signInButton'));
54    expect(signInButton.nativeElement.disabled).toBeTruthy();
55  });

```

Esta prueba, como su nombre indica, testea que tiene que pasar cuando el formulario de acceso sea inválido, es decir, falten campos o la contraseña sea de longitud menor a ocho caracteres. Primero se valida que cuando se inicia el componente, el formulario sea inválido, ya que el usuario aún no ha introducido datos. Luego se obtiene el botón de acceso a la aplicación y en la última línea se comprueba que su atributo `disabled` sea verdadero. Es decir, en esta prueba se valida que, si el formulario no es válido, el botón de acceso esté inhabilitado.

```

126  it( expectation: 'should open register page when click sign up', assertion: () => {
127    const signUpButton = el.query(By.css( selector: '#signUpButton'));
128
129    signUpButton.nativeElement.click();
130
131    expect(router.navigate).toHaveBeenCalledWith( params: ['/signUp']);
132  });
133

```

La prueba de la imagen valida que, al pulsar en el botón de registro, la aplicación dirija al usuario a la vista del registro. Esto se hace obteniendo el botón de registro, ejecutando su función de clic y validando que el módulo de navegación se haya llamado con el argumento correcto.

```

102 ▶ it( expectation: 'should execute logIn service when click on signIn with user', assertion: () => {
103   // set up spies, could also call a fake method in case you don't want the API call to go through
104   const signInSpy = spyOn(fixture.componentInstance, method: 'signIn').and.callThrough();
105
106   // make sure they haven't been called yet
107   expect(signInSpy).not.toHaveBeenCalled();
108   expect(authService.signInUser).not.toHaveBeenCalled();
109
110   component.form.controls['email'].setValue( value: 'user1@user.com');
111   component.form.controls['password'].setValue( value: 'useruser');
112
113   fixture.detectChanges();
114
115   expect(component.form.valid).toBeTruthy();
116   expect(el.query(By.css( selector: '#signInButton')).nativeElement.disabled).toBeFalsy();
117
118   component.signIn();
119
120   fixture.detectChanges();
121
122   expect(authService.signInUser).toHaveBeenCalledWith( params: 'user1@user.com', 'useruser');
123   expect(router.navigate).toHaveBeenCalledWith( params: ['/main'] );
124 });

```

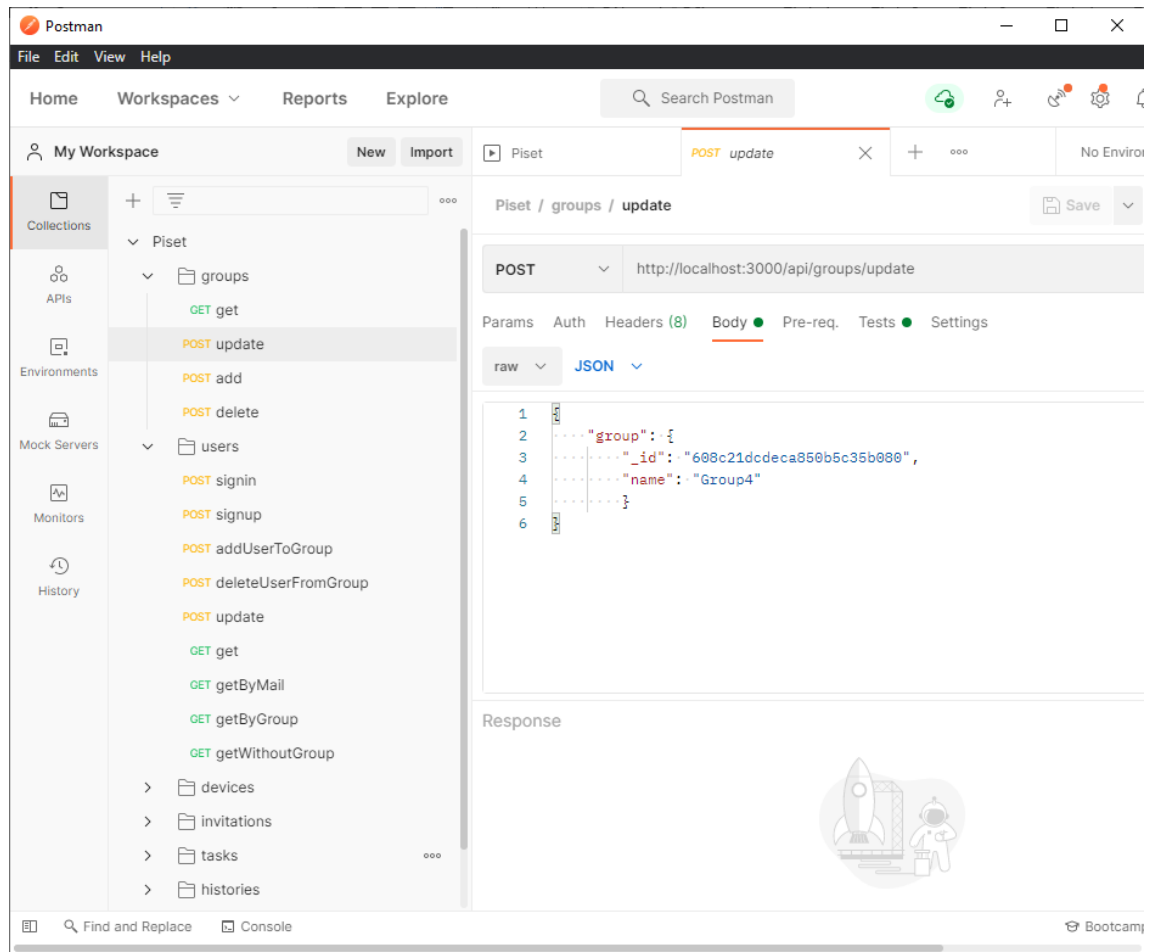
Esta prueba válida que cuando se pulsa el botón de acceso a la aplicación, se llame a la función de acceso del servicio Angular. Como se ve en la primera línea, se declara es espía a la función con nombre “signIn” del componente y se declara que se ejecute igualmente. Después se valida que aún no se hayan llamado a las funciones de “signIn” del componente ni a la función de “signInUser” del servicio de autenticación. Seguidamente se añaden los valores al formulario, el correo del usuario y la contraseña. Luego se actualiza el DOM, ya que la vista tiene que detectar los cambios en el formulario y pasar el botón de acceso a habilitado. En la siguiente línea se valida que el botón se haya habilitado y luego se llama a la función de acceso llamada “signIn”, al final se valida que se llame a la función del servicio llamada “signInUser” con los parámetros correctos obtenidos del formulario y que la aplicación de redirija a la pantalla principal, ya que en el describe, se ha modificado el método del servicio para que siempre de acceso, es decir no haga una llamada al servidor ni la simule, por lo tanto esta prueba sigue siendo una prueba unitaria y no de integración. Que son las que se verán más adelante.

7.2. Pruebas de integración

Las pruebas de integración son aquellas que se realizan una vez se han aprobado las pruebas unitarias y lo que prueban es que todos los elementos que componen el software funcionan juntos correctamente. Se centra principalmente en probar la comunicación entre sus diferentes componentes. En este caso se prueba la comunicación del cliente (frontend) con el servidor (backend) y del servidor con la base de datos.

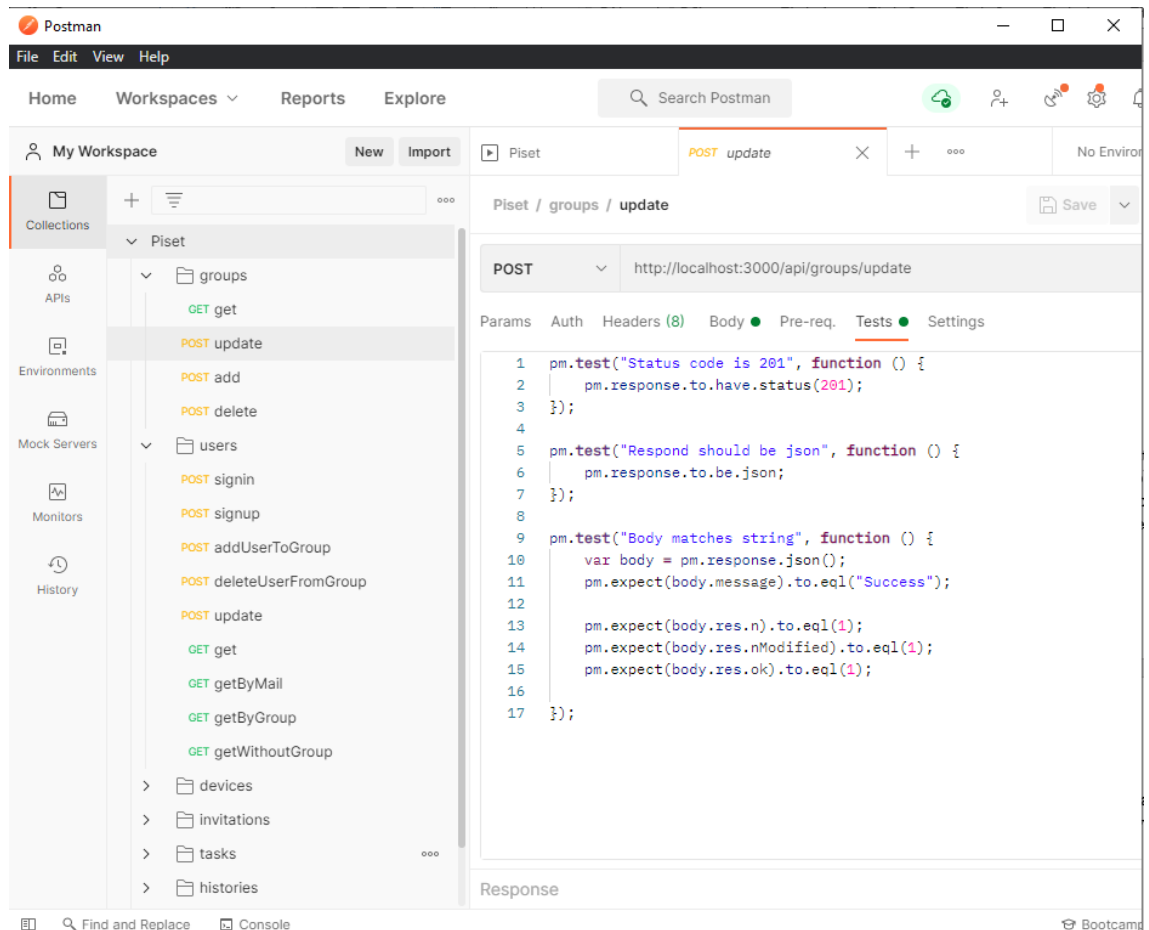
7.2.1. Cliente-Servidor-Base de datos

Para las pruebas de integración entre el cliente, el servidor y la base de datos se va a usar la herramienta Postman que se ha explicado en apartados anteriores. Postman es de gran ayuda ya que permite simular las peticiones que hará el cliente al servidor y almacenarlas en un listado para poder probar los cambios que se vayan haciendo en el servidor.



En la imagen se pueden ver el listado de peticiones que hay para el proyecto, agrupado por las carpetas que representan las rutas definidas en la parte de implementación del servidor. En la imagen se puede ver una petición del tipo Post al servidor que inicializado en el entorno local y los datos en formato JSON que envía el cliente, que en este caso se definen a modo de prueba.

A parte de esta ventana, en la herramienta Postman existe la utilidad de poder definir pruebas automáticas sobre la respuesta que devuelva el servidor.



En la parte de pruebas de la consulta de la imagen anterior se puede ver cada prueba declarada con su nombre en la función `pm.test()` y lo que se espera de la respuesta en la función `pm.expect(campo de la respuesta).to.eq(valor correcto que tiene que responder el servidor)`. Con estas funciones se pueden implementar las pruebas que se quieran en cada consulta y al final exportarlas todas en un archivo JSON, que es el que se usa para ejecutar las pruebas de la parte del servidor en la integración continua.

7.2.2. Cliente-Servidor

Estas pruebas tratan de validar la interacción entre el cliente y el servidor. Para el caso de Angular se va a volver a utilizar la herramienta de Jasmine. Este tipo de pruebas se aplican en los servicios de Angular y básicamente consisten en validar que el formato de la llamada que se hace al servidor es correcto y que la funcionalidad que se hace con los datos que devuelve el servidor es correcta, es decir, se tratan correctamente los datos para que se visualicen en las vistas del cliente. Hay que tener en cuenta que todo esto se hace haciendo una simulación de la petición al servidor y emular la respuesta del servidor con datos de pruebas, ya que el objetivo de la prueba es validar la interacción con el servidor, no el servidor en sí mismo. Este proceso se consigue gracias al módulo de pruebas de Angular llamado “`HttpClientTestingModule`”, este módulo extiende al “`HttpClient`” usado en la aplicación real y le añade la funcionalidad de poder simular la respuesta por parte del servidor y espiar la propia

consulta para ver con que parámetros se llama. Se va a explicar la implementación de un caso representativo en la aplicación:

```
50 it( expectation: 'getGroupTasks', assertion: () => {
51   // Get mock data
52   const mockGroup = testService.getGroupByName( groupName: 'Group1');
53   const mockTasks = testService.getTasksByGroupId(mockGroup._id);
54
55   // Call service method
56   service.getGroupTasks(mockGroup);
57
58   // Create the mockCall
59   const reqTasks = httpTestingController.expectOne( url: service['API_URL'] + '/api/tasks/getByGroup' + mockGroup._id);
60
61   reqTasks.flush( body: {
62     tasks: mockTasks
63   });
64
65   // Expect request method
66   expect(reqTasks.request.method).toEqual( expected: 'GET');
67   // Expect request parameters
68   expect(reqTasks.request.params.keys().length).toBe( expected: 0);
69   // Expect the returned data
70   expect(service._tasksGroup.getValue().size).toEqual( expected: 4);
71   expect(service._tasksGroup.getValue().get( key: 0).name).toEqual( expected: 'Task1');
72   expect(service._tasksGroup.getValue().get( key: 1).name).toEqual( expected: 'Task2');
73   expect(service._tasksGroup.getValue().get( key: 2).name).toEqual( expected: 'Task3');
74   expect(service._tasksGroup.getValue().get( key: 3).name).toEqual( expected: 'Task4');
75 });
```

En esta prueba se necesita el uso de datos de pruebas, ya que no se puede interactuar con el servidor. Para eso existe el servicio de test, que tiene funciones para obtener datos de archivos de objetos de pruebas en JSON. En esta prueba se obtiene primero un grupo de prueba y sus tareas. Luego se llama al servicio que hace la llamada al servidor y luego se declara la simulación de la llamada al servidor, se añade la URL y la ruta y luego usando la función flush devuelve los datos de prueba para simular la respuesta del servidor. Seguidamente se valida que la consulta sea del tipo GET, que no haya parámetros y valida que los datos que ha devuelto la consulta se han almacenado correctamente en la lista observable “_tasksGroups” del servicio.

Con este tipo de pruebas se valida que la interacción del cliente con el servidor es correcta, que se envían los datos correctos en la petición y se hace lo correcto con los datos que devolvería el servidor real.

7.3. Pruebas E2E

Las pruebas E2E se encargan de probar toda la aplicación al completo, se encargan de poner a prueba la aplicación en el entorno real intentando simular a un usuario final que interactúa con la vista de la aplicación. Para esta aplicación se han implementado estas pruebas usando la herramienta Cypress.

Cypress es una herramienta de pruebas que se puede usar para todo tipo de aplicación web, tiene la particularidad de interactuar directamente con el DOM en el propio navegador web, a diferencia de Selenium, que utiliza un driver intermedio. Esto le permite a Cypress grabar las interacciones que tiene el usuario simulado con la aplicación. En este caso, este proceso de pruebas se hace con toda la aplicación completa, es decir, en ningún momento se simula la respuesta del servidor, se usa una instancia de base de datos en el entorno de test, subida al entorno donde estará la aplicación en producción, al igual que el propio servidor. Cuando se finaliza la prueba

en Cypress, se restaura la base de datos de test al estado anterior. Esto permite simular completamente el entorno donde estará la aplicación en producción y que tengan más validez las pruebas. Esto lleva un coste temporal a la hora de ejecutar las pruebas, ya que es más lento hacer una petición real al servidor que simularla, al igual que el proceso de restauración de base de datos.

Cypress funciona con la misma sintaxis con la que funciona Jasmine. Un ejemplo ilustrativo de una prueba de este tipo sería el siguiente:

```
3 describe('Home Page', () => {
4
5   it('user login', () => {
6
7     cy.visit( url: '/');
8
9     cy.contains('Environment: test');
10
11    //cy.get('#signInButton').should('be.disabled');
12    cy.get('#signInButton').should( chainer: 'not.be.disabled');
13
14    cy.get('#mail').type( text: 'user1@user.com');
15
16    cy.get('#password').type( text: 'useruser');
17
18    cy.get('#signInButton').should( chainer: 'not.be.disabled');
19    cy.get('#signInButton').should( chainer: 'not.be.disabled');
20
21    cy.get('#signInButton').click();
22
23    cy.url().should( chainer: 'contain', value: '/main/tasks');
24  });
25 });
```

En este fragmento de código se puede apreciar lo rápido y simple que puede llegar a ser hacer una prueba con Cypress, en este caso se está validando que el usuario visite la página de la aplicación, el botón esté deshabilitado, el usuario introduzca sus datos, los botones pasen a estar habilitados y al pulsar en acceder la página acceda a la ventana principal.

Uno de los factores de esta simpleza es que como se ha explicado Cypress interactúa directamente con el DOM del navegador y además cada instrucción Cypress tiene implícito un tiempo de espera, con el cual se espera a que se carguen los elementos de las vistas necesarios o se espere a que una llamada al servidor obtenga respuesta. Si este tiempo se pasa sin cumplir los requisitos de la prueba, la prueba es errónea. En Jasmine, por ejemplo, esta gestión de la sincronía se hace de forma manual por el programador.

8. Conclusión

En este capítulo se resumen las conclusiones obtenidas tras el desarrollo de la aplicación.

Una de las cualidades en las que se enfoca este proyecto es en la metodología y el proceso de desarrollo software. Se ha hecho hincapié en la parte de las pruebas, la integración y despliegue, a parte de la implementación, que, aunque en un principio parece la más importante, ha quedado demostrado que las otras partes lo son en igual medida, ya que una buena implementación de pruebas en todas las partes del proyecto nos permite asegurar que se trata de un código de calidad y que cumple con los requisitos previamente establecidos. Todas estas pruebas, junto a la importancia de automatizar la integración y el despliegue para facilitar las tareas que tiene que hacer un desarrollador a la hora de trabajar en este proyecto demuestran el gran ahorro de tiempo que supondría para una empresa.

Lo más laborioso del proyecto ha sido aprender la metodología de desarrollo en el framework MEAN. Este framework no se explica durante los estudios, pero cuenta con una gran comunidad que facilita su aprendizaje de una forma autodidacta y una rápida solución a todos los problemas que han ido surgiendo. Cabe destacar el cambio de paradigma de las bases de datos relacionales a las no relacionales ya que estas últimas están mínimamente estructuradas y son mucho más flexibles a la hora de tratar los datos.

Cabe destacar el acierto que ha sido sustituir el framework de pruebas E2E integrado ya en Angular llamado Protractor por uno externo y aplicable a toda página web llamado Cypress. Ha sido muy satisfactorio descubrir cómo implementar pruebas E2E y la gran utilidad que aportan a la hora de mantener un software.

Por último, como en todo software creado desde cero, la finalización no es más que el inicio de una etapa de mantenimiento y ampliación de funcionalidades. En esta etapa se podrá apreciar más aún el trabajo que hay detrás de cada prueba en el proyecto y la implementación de la integración y despliegue continua. Ya que, con cada nuevo cambio, no será necesario probar a mano toda la aplicación para ver si se ha roto algo, ni desplegar a mano cada nueva versión.

8.1. Relación proyecto-estudios cursados

Los estudios que se han cursado en el Grado, han servido para poder enfocar todas las partes que debe tener un proyecto, así como para ver qué beneficios o desventajas aporta cada framework que se ha tanteado para ser usado. Además, los estudios aportan toda una base en el desarrollo de software que enseña muy bien a cómo desarrollar código de calidad, es decir, que sea simple, leíble, documentado, escalable y que tolere los cambios.

Por otro lado, la experiencia laboral da un enfoque a la hora de escribir código y seguir una metodología para simplificar y automatizar lo máximo posible todos los procesos del desarrollo de software, de ahí el enfoque en las pruebas y la integración

continua. Todo esto para facilitar no el propio desarrollo de la aplicación, sino el mantenimiento que vendrá detrás.

8.2. Trabajos futuros y líneas de mejora

En la versión final de este proyecto se han cumplido los requisitos establecidos previamente. Pero hay varias funcionalidades que se puede plantear incorporar en un futuro.

Por ejemplo, dentro de cada tarea podía haber unos hilos de comentarios que puedan insertar los usuarios de un grupo. Esto mejoraría mucho más la organización del grupo.

También se podría dividir por proyectos las tareas, ya que ahora mismo solo existe la agrupación por grupos. Esto estaría bien para una organización que quisiera tener organizadas las tareas por sus proyectos.

Por último, y ahora más enfocado en el mantenimiento, estaría bien añadir algún gestor de traducciones para los literales de la parte de cliente de Angular. Estos podrían estar insertados en archivos JSON y obtenerlos mediante un servicio de Angular. O buscar alguna librería nativa de Angular que se encargue de esto.

9. Referencias

- ⁱ Structuralia (2021): «Los 5 mejores gestores de tareas de 2021», *Structuralia*. 4 enero 2021, <<https://blog.structuralia.com/gestores-de-tareas>> [Consulta: 3 de junio 2021].
- ⁱⁱ Remember The Milk (2018). «Remember The Milk», *Capterra*. 6 enero 2018, <<https://www.capterra.es/software/162672/remember-the-milk>> [Consulta: 3 de junio 2021].
- ⁱⁱⁱ Todoist (2018). «Todoist», *Todoist*. 8 de octubre 2018, <<https://todoist.com/es/>> [Consulta: 3 de junio 2021].
- ^{iv} GesTron (2016): «Qué es Trello y cómo se usa, Guía rápida», *Ayudatpymes*. 2 agosto 2016, <<https://ayudatpymes.com/gestron/que-es-trello/>> [Consulta: 3 de junio 2021].
- ^v Santander Universidades (2020): «Metodologías de desarrollo de software: ¿qué son? », *Santander*. 21 diciembre 2020, <<https://blog.becas-santander.com/es/metodologias-desarrollo-software.html>> [Consulta: 8 de junio 2021].
- ^{vi} RedHat (2019): « ¿Qué es la metodología ágil? », *RedHat*. 3 de julio 2019, <<https://www.redhat.com/es/devops/what-is-agile-methodology>> [Consulta: 8 de junio 2021].
- ^{vii} Proyectos ágiles (2008): «Qué es SCRUM», *Proyectos Ágiles*. 4 de agosto 2008, <<https://proyectosagiles.org/que-es-scrum/>> [Consulta: 8 de junio 2021].
- ^{viii} Proyectos ágiles (2008): «Desarrollo iterativo e incremental», *Proyectos Ágiles*. 27 de septiembre 2008, <<https://proyectosagiles.org/desarrollo-iterativo-incremental/>> [Consulta: 8 de junio 2021].
- ^{ix} RedHat (2020): «¿Qué son la integración/distribución continuas (CI/CD)? », *RedHat*. 18 de junio 2020, <<https://www.redhat.com/es/topics/devops/what-is-ci-cd>> [Consulta: 14 de junio 2021].
- ^x de Zúñiga, F. G. (2020): «¿Qué es la arquitectura del software?», *Arsys*. 3 de febrero 2020, <<https://www.arsys.es/blog/arquitectura-software/>> [Consulta: 16 de junio 2021].