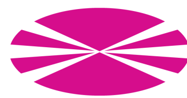

Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DEL SOFTWARE

Desarrollo de una aplicación de gestión documental: frontal React y backend GraphQL con almacenamiento git

Estudiante: Sergio Fernández Fraga
Dirección: David Barral Precedo
Tutor: David Cabrero Souto

A Coruña, junio de 2021

A los que he conocido, conozco y conoceré

Agradecimientos

Han sido cinco largos años de una etapa de mi vida llena de alegrías y nuevas emociones. También ha habido momentos tensos y complicados que se han llevado de la mejor manera posible y de los que siempre se aprende. He conocido a gente extraordinaria y muy capaz con visiones diferentes que siempre han aportado algo a mi forma de ser. Gracias a David Barral por la ayuda durante estos meses. Y en especial a mi familia por darme la oportunidad y el apoyo incondicional.

Resumen

La finalidad de este proyecto es el desarrollo de una nueva funcionalidad para una herramienta de intranet de la empresa Trabe Soluciones S.L.

La funcionalidad implementa un prototipo para la gestión de documentos de la empresa en un formato *markdown* utilizando almacenamiento `Git`. *Markdown* es un tipo de formato de texto enfocado en facilitar la escritura en internet. La herramienta permite:

- Visualizar los documentos existentes y sus autores.
- Modificar la documentación.
- Crear versiones.
- Previsualizar.
- Almacenar en repositorios `Git`.

Al usar `Git` y *markdown*, se consigue trabajar de dos maneras con los documentos: el acceso mediante un cliente `Git` y mediante el uso de la *User Interface* (UI) de la aplicación. En consecuencia, la creación y edición de la documentación se puede realizar con las mismas herramientas que se utilizan para el desarrollo de software: IDE, terminal, editor de texto, etc.

El proyecto se compone de un *backend* implementado en NodeJS que se comunica mediante GraphQL con un *frontend* desarrollado en React.

Abstract

Palabras clave: React, Git, Javascript, Documento, Formato markdown, Versión, Contenido

Keywords: React, Git, Javascript, Document, Markdown format, Version, Content

Índice general

1 Introdución **1**

1.1 Situación actual 1

1.2 Objetivos y alcance 2

2 Tecnología **3**

2.1 Lenguajes de programación 3

2.2 Conjunto de herramientas 3

2.3 Librerías 5

2.4 Bases de datos 7

3 Metodología **9**

3.1 Introducción 9

3.2 Code Review 10

3.3 Pair Programming 10

3.4 Pruebas 10

4 Análisis de la viabilidad del proyecto **11**

5 Requisitos del sistema **13**

5.1 Introducción 13

5.2 Actores 14

5.3 Casos de uso 14

6 Arquitectura **17**

6.1 Introducción 17

6.2 Tecnología 17

6.2.1 *Backend* 18

6.2.2 *Frontend* 20

7 Iteraciones y estimación **23**
7.1 Iteraciones 23

8 Desarrollo **25**
8.1 Versiones del software 25
8.2 Iteraciones 26
8.2.1 **Iteración 0:** formación en las tecnologías y definición de requisitos funcionales. 26
8.2.2 **Iteración 1:** desarrollo de la capa de acceso a datos (repositorios `Git`). 28
8.2.3 **Iteración 2:** desarrollo de los casos de uso en el *backend*. 32
8.2.4 **Iteración 3:** desarrollo de acceso a base de datos PostgreSQL. 34
8.2.5 **Iteración 4:** desarrollo de resolvers de GraphQL. 36
8.2.6 **Iteración 5:** desarrollo en el *frontend*: lista de documentos. 39
8.2.7 **Iteración 6:** desarrollo en el *frontend*: vista de documento. 42
8.2.8 **Iteración 7:** desarrollo en el *frontend*: funcionalides del documento (edición, versionado, etc). 44

9 Análisis de los costes **51**

10 Conclusiones **53**
10.1 Conclusiones 53
10.2 Futuras líneas de trabajo 54

Lista de acrónimos **55**

Glosario **57**

Bibliografía **59**

Índice de figuras

1.1 Página de inicio de *trabenet*. 2

2.1 Herramienta Trello 4

3.1 Metodología incremental. 10

6.1 Arquitectura general de la aplicación. 18
6.2 Estructura en detalle del *backend*. 19
6.3 Estructura en detalle del *frontend*. 21
6.4 Diagrama de secuencia del caso de uso Modificación del contenido de un documento. 22

8.1 Estructura del código de la aplicación *trabenet*. 26
8.2 Configuración en `docker-compose.yml` de la imagen de Gitolite. 27
8.3 Estructura del código del *backend*. 28
8.4 Ejemplo de configuración de los repositorios en gitolite. 29
8.5 Entidades *users* y *documents*. 35
8.6 Estructura del código del *frontend*. 40
8.7 Componentes empleados para la página de la lista de documentos. 41
8.8 Ubicación del enlace para ver la lista de documentos en la aplicación *trabenet*. 41
8.9 Vista de la página documentos en la aplicación *trabenet*. 42
8.10 Componentes empleados para la página del detalle de un documento. 43
8.11 Vista de la página del detalle de un documento en la aplicación *trabenet*. . . . 44
8.12 Modificación de la descripción de un documento dentro de la aplicación. . . . 45
8.13 Modificación del contenido de un documento dentro de la aplicación. 46
8.14 Creación de una nueva versión del documento en la aplicación. 47
8.15 Lista de versiones del documento con la nueva versión. 47
8.16 Previsualización del contenido del documento en la aplicación. 48

8.17 Guardado del documento con opción de previsualización. 49

Índice de tablas

7.1 Iteraciones del proyecto con estimación temporal. 24
9.1 Iteraciones del proyecto con los tiempos estimados y reales. 51

Capítulo 1

Introdución

Trabe Soluciones es una pequeña empresa de Coruña dedicada al desarrollo de proyectos software. Utilizan un conjunto de aplicaciones propias para el funcionamiento diario de la empresa. Una parte del trabajo diario consiste en la elaboración de ofertas para clientes y otros tipos de documentación, que requieren de un formato corporativo.

1.1 Situación actual

Actualmente, Trabe no dispone de un sistema de documentación estandarizado para la creación de documentos corporativos. Para su creación, disponen de un CLI escrito en Ruby que empleando Pandoc y Latex, convierte un documento markdown en un PDF con formato corporativo. Posteriormente, los documentos se almacenan en un directorio compartido en Dropbox.

Una solución posible para la gestión y creación de documentación sería el uso de servicios externos como Legito [1]. El uso de estos aporta ventajas como:

- Facilidad de uso: no habría que utilizar herramientas como el terminal para la generación de un documento.
- Ubicuidad y rapidez de acceso: los empleados pueden acceder a los documentos desde cualquier punto en el que trabajen y de una manera más rápida y eficiente. En este momento, los empleados de Trabe necesitan una cuenta activa en los servicios de Dropbox para acceder a la documentación de la empresa.
- Colaboración: los usuarios podrán colaborar y trabajar juntos en un mismo documento evitando retrasos de tiempo.

La empresa dispone de algunas herramientas internas, una de ellas, *trabenet*, ofrece a los empleados información sobre sus horas trabajadas, dispone de boletín informativo, etc. Trabe

quiere agrupar toda la documentación relevante en un mismo lugar y permitir el acceso a los empleados de manera sencilla y eficaz.

La empresa ha optado por una solución propia que se integre en *trabenet*. Se utilizará como backend de almacenamiento de documentos Git y se integrará el sistema de documentación con la interfaz de usuario para poder editar la documentación.

1.2 Objetivos y alcance

El objetivo es el desarrollo de una versión prototipo de un sistema de gestión documental y su integración dentro de la aplicación interna de Trabe, *trabenet* (figura 1.1 (página 2)).

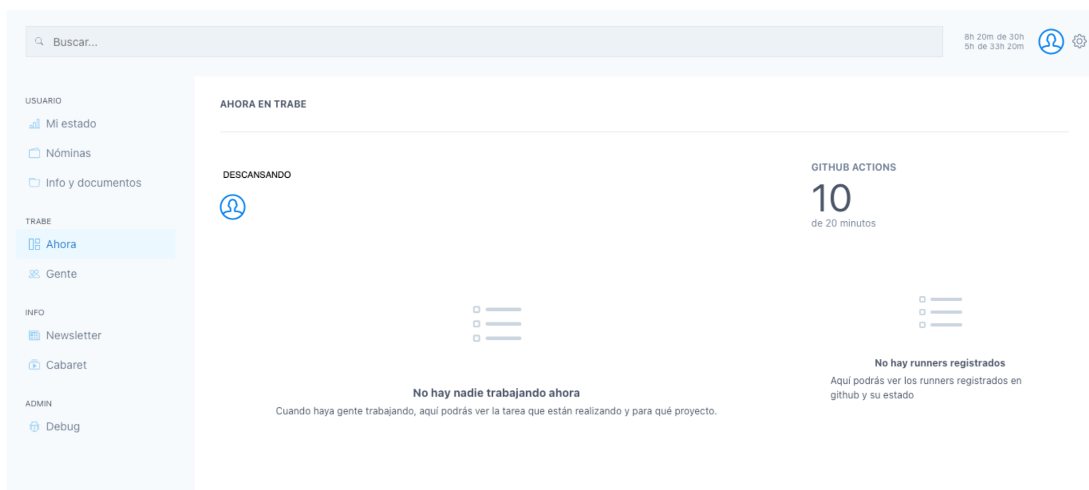

Figura 1.1: Página de inicio de *trabenet*.

La aplicación permitirá mostrar un listado de documentos, acceder al detalle de cada uno y realizar operaciones de actualización y versionado. En los sucesivos capítulos de esta memoria se profundizará en un mayor nivel de detalle en cada una de las funcionalidades que ofrece. Previamente se ha comentado que se utilizarán repositorios git para el almacenamiento de la documentación porque ofrece la posibilidad de la gestión de documentos desde la interfaz gráfica de la aplicación o desde línea de comandos utilizando un terminal. Respecto a la UI, la solución debe integrarse y seguir las guías de estilo de la actual *trabenet*.

Se desarrollarán unas funcionalidades básicas que permitan comenzar a utilizar este módulo de gestión de documentación para, en futuras versiones, ir realizando iteraciones con un *feedback* más realista.

Capítulo 2

Tecnología

En las siguientes secciones explicaremos toda la tecnología que se va a utilizar en este proyecto: lenguajes de programación, herramientas y librerías utilizadas. Una gran parte de las tecnologías citadas a continuación se usan diariamente en Trabe Soluciones y formaban parte del *stack* de *trabenet* antes del comienzo de este proyecto.

2.1 Lenguajes de programación

A continuación se detallan los lenguajes de programación:

- Javascript (JS) [2]: se trata de un lenguaje de alto nivel, orientado a objetos y de tipado débil y dinámico. Soporta programación dirigida por eventos, funcional e imperativa. Una característica a destacar es que es un lenguaje interpretado, por lo tanto no requiere de tiempo de compilación antes de ser ejecutado.

- GraphQL[3]: se trata de un lenguaje de consultas que se ejecuta en el lado del servidor en tiempo de ejecución para su uso en APIs. Está diseñado para ser muy flexible y rápido de cara al desarrollador. Su mayor ventaja es el menor uso de llamadas al servidor para la obtención o manipulación de información respecto a otros sistemas como Representational state transfer (REST).

2.2 Conjunto de herramientas

- Intellij[4]: se trata de un Integrated Development Environment (IDE), escrito en el lenguaje de programación Java. Integra multitud de funcionalidades de todo tipo: integración con un sistema de control de versiones, sistema de base de datos, ejecución de pruebas, terminal, etc. También dispone de herramientas de completado automático, inyección de lenguaje (añadir fragmentos JS en un string de Java, por ejemplo). A pesar

de no ser un programa gratuito dispone de una versión de prueba. Se ha utilizado para el desarrollo del código del proyecto.

- Google Chrome: se trata de un navegador web y es uno de los más populares y utilizados durante abril de 2021[5]. En el proyecto se ha utilizado para realizar pruebas en la aplicación y detectar posibles *bug's*. Dispone de una serie de funcionalidades para desarrolladores y extensiones para la librería React que han sido de gran utilidad.

- Slack[6]: se trata de una aplicación de mensajería para el entorno empresarial. Ofrece lo que Slack denomina canales y consiste en la forma de organizar las conversaciones entre empleados de una organización. Pueden ser abiertos o cerrados dependiendo de su uso en el proyecto. Esta herramienta se ha utilizado para la comunicación con el director del proyecto.

- Git[7]: se trata de una herramienta para la gestión de control de versiones. Está ampliamente considerada como una de las mejores para este propósito. Requiere una curva de aprendizaje pero es muy escalable y potente. Su código es abierto y la documentación es extensa. Se ha utilizado en este proyecto para llevar el control de las versiones del código.

- Trello[8]: es un servicio que se enfoca en la gestión de proyectos de una manera fácil, flexible y escalable empleando tableros tipo kanban. La figura 2.1 (página 4) muestra una captura de la herramienta siendo empleada en el proyecto. En estos se organizan las diferentes partes del proyecto. Cada una de estas partes contiene una o varias tarjetas, que contienen información relevante respecto a una tarea: nombre, miembros que colaboran, descripción, actividad reciente, etc. Trello permite integración con otros servicios como GitHub o Slack creando así un potente ecosistema de trabajo.

Figura 2.1: Herramienta Trello

- Docker[9]: es una herramienta que automatiza el despliegue de aplicaciones en contenedores. Está diseñada para simplificar el uso de creación y ejecución de aplicaciones. Un contenedor es un entorno aislado en el que se ejecuta una aplicación, es similar a máquina virtual pero evitando la creación del sistema operativo. Sus ventajas son: rapidez de despliegue, escalabilidad y eficiencia de los recursos. En el caso de este proyecto se ha utilizado para la ejecución de la base de datos, otros servicios y el despliegue de la aplicación en el entorno de producción.

- Github[10]: es un servicio de almacenamiento de código de aplicaciones. Implementa git para el control de versiones además de otras funcionalidades propias. A día de hoy, se ha convertido en uno de los mayores host de código en 2021 con más de 50 millones de usuarios [11]. En el proyecto se ha empleado para el versionado y revisado mediante *pull requests* del código de la aplicación.

- Zoom[12]: es una herramienta orientada a la comunicación. Ofrece funcionalidades como llamadas de audio y vídeo, integración con el calendario, gestión de tareas, salas de espera. Para el proyecto, se ha usado para videollamadas y compartición de pantalla para solución de problemas de configuración o código.

2.3 Librerías

En esta sección se ha incluido NodeJS como librería. Técnicamente no se trata de una librería sino de un entorno en tiempo de ejecución. A pesar de ello es la sección más idónea en la que incluirlo.

Se detallan a continuación las librerías utilizadas en el proyecto:

- NodeJS[13]: se trata de un entorno de ejecución, permite ejecutar un programa escrito en JS y diseñar aplicaciones muy escalables. Fue creado por los desarrolladores originales de JS para utilizar algo que ya existía en los navegadores y llevarlo al lado del servidor.

- Nodegit[14]: es una librería asíncrona que implementa funcionalidades expuestas en `libgit2` (implementación de la herramienta Git desarrollada en C. Su uso está destinado a la manipulación de repositorios Git, desde la autenticación y gestión de permisos de usuarios hasta gestión de archivos, objetos commit y acciones de git como *push*, *clone* o *stash*. Esta librería ha sido de esencial para la gestión de repositorios de documentación.

- Jest[15]: jest es una librería de *testing* basada en Javascript que busca resultar lo más sencilla posible de utilizar de cara al desarrollador. Busca poder instalarse con la mínima configuración, ofrece una API muy extensa y bien documentada y los tests se ejecutan en paralelo para obtener el mejor rendimiento posible. En el transcurso del proyecto, ha sido muy útil de cara a los tests de unidad e integración del código fuente.

- Gitolite[16]: provee de un servidor de repositorios Git con acceso ssh y discriminación de usuarios mediante claves públicas. Sus ventajas son: el uso de un solo usuario en el servidor, control en múltiples repositorios y un sistema robusto de reglas de acceso. Para este proyecto ha sido de utilidad para definir reglas de acceso para los usuarios mediante el uso de sus claves públicas Secure Shell Protocol (SSH).

- React[17]: es una librería basada en javascript para la construcción de interfaces de usuario. Se basa en componentes, es decir, cualquier elemento es un componente aislado con unas propiedades y un estado. Hoy en día, posee mucha popularidad en el desarrollo de páginas web. El *frontend* de la aplicación *trabenet* y las nuevas funcionalidades están desarrolladas con esta librería.

- Prisma[18]: se trata de un Object-relational Mapping (ORM), es el encargado de transformar los datos de los objetos en un formato correcto para su uso en base de datos, de manera que se delega en este la construcción explícita de *queries*. Su núcleo principal es el esquema, en este se definen las entidades y atributos con sus tipos. Utiliza un modelo de tipado seguro evitando posibles errores. En el proyecto se ha utilizado para definir una nueva tabla de en base de datos y métodos de búsqueda, creación, actualización y eliminación.

- Prettier[19]: es un formateador de código en el que se definen unas reglas que se aplicarán sobre el código fuente. Reglas como el uso de comillas dobles o simples, indentación, etc. En los equipos de desarrollo de software es muy útil pues cada programador tiende a escribir código de una manera y utilizando esta librería se crea un estándar para todo el equipo. Se ha empleado en el proyecto para estandarizar el código fuente de la aplicación con unas reglas predefinidas.

- Yarn[20]: es un gestor de paquetes de código. Se encarga de instalar paquetes utilizados por los desarrolladores y la gestión de dependencias y versiones. En este proyecto se ha utilizado para actualizar algunas dependencias y ejecutar scripts predefinidos.

- Apollo Client[21]: es una librería de gestión de estado para JS para manejar tanto información local como remota con GraphQL. Permite su integración con React moderno facilitando su uso y aprovechando las ventajas que este ofrece. Además se puede utilizar

con cualquier API de GraphQL. Se ha usado para la comunicación con el *backend* en las consultas de datos.

- Apollo Server[22]: es un servidor que implementa GraphQL de código abierto compatible con cualquier cliente GraphQL. Su uso requiere de la definición de un esquema graphql, con definiciones de tipos y un conjunto de *resolvers*, encargados de resolver las queries y devolver los resultados correspondientes.

- Koa[23]: esta librería se utiliza para crear servicios HTTP. Se trata de un framework que intenta tener un tamaño reducido, ser robusto para aplicaciones y APIs y emplea funciones asíncronas para evitar *callbacks*. Basicamente, una aplicación Koa es un *pipeline* con funciones *middleware* que se ejecutan en base a una petición. La aplicación emplea esta librería para la ejecución del *backend*.

2.4 Bases de datos

- PostgreSQL[24]: es un sistema de gestión de bases de datos relacional de código abierto. Ofrece una óptima fiabilidad, integridad de información y un robusto conjunto de funcionalidades. Estas características hacen que disponga de una buena reputación entre los programadores.

Capítulo 3

Metodología

3.1 Introducción

El tipo de metodología seguido ha sido el de un enfoque iterativo incremental[25]. Consta de sucesivas etapas incrementales en las que cada una tiene unos requisitos, diseño, desarrollo y pruebas. Al final de cada etapa se obtiene un producto o versión entregable al cliente.

Las ventajas de esta metodología son las siguientes:

- Los riesgos se minimizan y se distribuyen durante todo el progreso de ejecución de las sucesivas versiones del producto.

- Permite tener un producto entregable con un mínimo de funcionalidades al inicio del proyecto.

- Las pruebas e integración de los diferentes módulos de código son constantes.

- La planificación puede realizarse al principio priorizando funcionalidades más críticas para el cliente.

La figura 3.1 (página 10) expone la estructura de esta metodología.

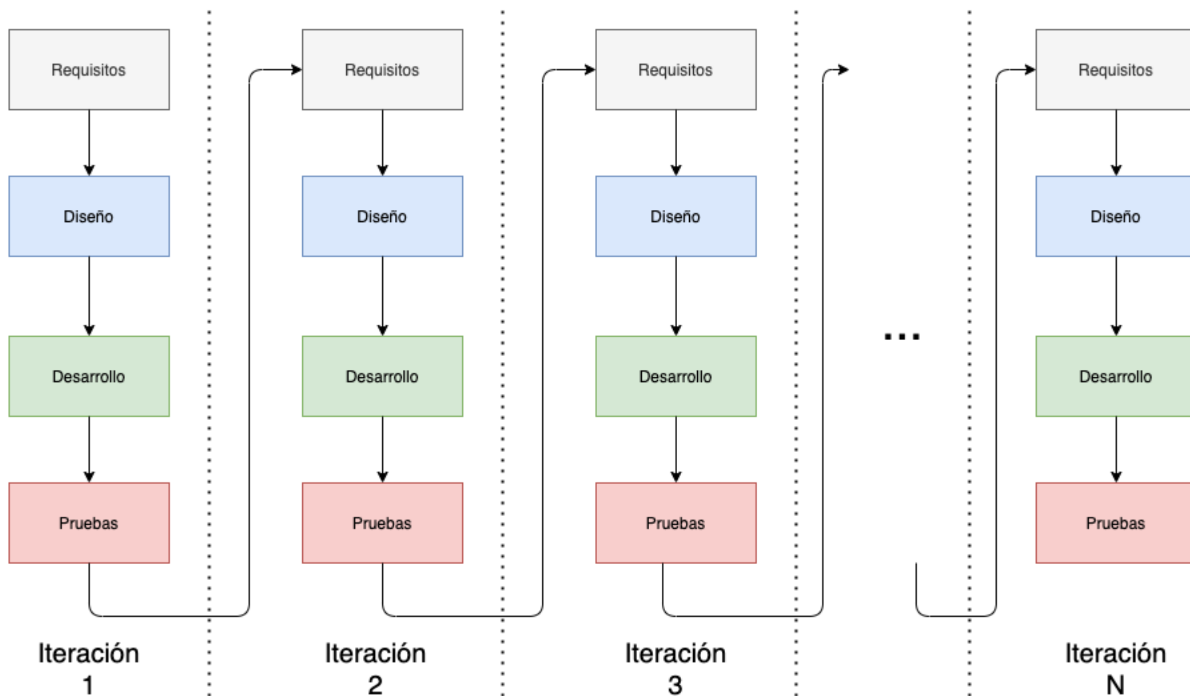

Figura 3.1: Metodología incremental.

3.2 Code Review

En este proyecto se han realizado revisiones de código mediante el uso de *pull requests* en la herramienta GitHub. Por cada nuevo conjunto de funcionalidades desarrolladas se crea una *PR* con el objetivo de la detección temprana de errores en el código o alternativas más eficientes a las implementaciones.

3.3 Pair Programming

Esta técnica ágil de desarrollo *software* se ha utilizado en algunas situaciones concretas:

- Solución de problemas con la configuración e instalación de la aplicación al inicio del proyecto.
- Resolución de errores de ejecución del código.
- Explicación de funcionalidades de la aplicación *trabenet*.

3.4 Pruebas

Se han realizado pruebas de unidad e integración en el código del proyecto.

Capítulo 4

Análisis de la viabilidad del proyecto

En primer lugar, este proyecto conforma la ejecución de un trabajo de fin de grado (TFG) de universidad, por lo que tiene asignado doce créditos (ECTS). Para el desarrollo de esta versión prototipo de gestión documental se ha tomado como referencia su finalización en 320 horas.

En segundo lugar, conviene mencionar las tecnologías para la ejecución del proyecto, algunas de ellas ya heredadas de la propia aplicación de Trabe. *Trabenet* emplea como lenguaje de programación Javascript y el *framework* Node.js, asíncrono y dirigido por eventos. La parte frontal de la aplicación está desarrollada en React, esta librería es muy escalable, con amplia comunidad de usuarios y documentación. Respecto a la comunicación con los repositorios git se empleará NodeGit.

En tercer lugar conviene analizar las restricciones económicas a las que el proyecto ha de restringirse. Como se ha comentado anteriormente, es un proyecto que conforma un trabajo de fin de grado de ingeniería informática. Los recursos físicos necesarios serían un ordenador portátil y una conexión a internet. El primero ya está disponible debido a la naturaleza de la ingeniería. Respecto al coste de internet, su coste sobre el total del proyecto es reducido. El *software* utilizado es código abierto por lo que su coste es nulo.

El principal riesgo del proyecto es a nivel técnico: varias tecnologías no han sido usadas por el alumno del proyecto y requieren una curva de aprendizaje. Otras librerías como NodeGit o Gitolite nunca se habían empleado con anterioridad en Trabe, por lo que no había *know how* dentro de la empresa.

Finalmente, analizadas las limitaciones temporales, tecnológicas, económicas y asumiendo los riesgos, se concluye que la ejecución del proyecto es viable con los recursos disponibles.

Capítulo 5

Requisitos del sistema

5.1 Introducción

Esta versión prototipo pretende implementar un sistema de gestión de documental e integrarlo dentro de una aplicación interna de la empresa Trabe Soluciones. Mediante el uso de la tecnología *Git* para el almacenamiento, se pretende guardar la información de manera que sea accesible y mantenible por dos vías. La primera, mediante la interfaz gráfica de usuario de la aplicación. La segunda vía de acceso mediante el uso de un terminal de comandos. Los documentos que se almacenen emplearán un formato *markdown*, esto permite la modificación del archivo utilizando diferentes herramientas: un IDE o editor de texto, o la UI de la aplicación.

La aplicación permitirá las siguientes funcionalidades:

1. Mostrar la documentación en una lista ordenada por título del documento.

2. Filtrar la búsqueda de documentos por título.

3. Mostar la información del documento: título, descripción, contenido, autores y versiones.

4. Editar el contenido y la descripción del documento.

5. Deshacer los últimos cambios realizados en el contenido del documento.

6. Crear una versión de un documento almacenado en base de datos.

7. Previsualizar el contenido del documento.

5.2 Actores

En este proyecto consideraremos como los actores a los usuarios con una cuenta corporativa de *trabenet*. Cada usuario posee una serie de roles pero en este caso, solo estamos interesados en el rol de ***rrhh:manager***.

5.3 Casos de uso

Los casos de uso desarrollados en esta versión prototipo son los siguientes:

CU-01	**Mostar lista de documentos**
Precondiciones	El usuario ha iniciado sesión y posee el rol ***rrhh:manager***.
Secuencia	El usuario selecciona la opción “Documentación” en el menú de la aplicación. Se muestra la lista de documentos.
Poscondiciones	N/A
Excepciones	N/A

CU-02	**Mostar detalle de documento**
Precondiciones	El usuario ha iniciado sesión y posee el rol ***rrhh:manager***.
Secuencia	El usuario selecciona un documento de la lista de documentos. Se muestra el detalle del documento.
Poscondiciones	N/A
Excepciones	N/A

CU-03	**Crear versión de un documento**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, ha accedido al `detalle de un documento (CU-02)` y ha seleccionado la opción “Crear versión”.
Secuencia	El usuario detalla el nombre y mensaje de la versión y pulsa “Aceptar”.
Poscondiciones	La nueva versión del documento es creada en la aplicación.
Excepciones	El usuario no rellena todos los campos requeridos: nombre y descripción.

CU-05	**Modificar contenido de un documento**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, y ha accedido al `detalle de un documento (CU-02)`.
Secuencia	El usuario pulsa el botón de "Editar". El usuario modifica el contenido del documento y pulsa el botón de "Actualizar".
Poscondiciones	El nuevo contenido se guarda en base de datos.
Excepciones	N/A

CU-04	**Modificar descripción de un documento**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, y ha accedido al `detalle de un documento (CU-02)`.
Secuencia	El usuario modifica la descripción y pulsa el botón de "Aceptar".
Poscondiciones	La nueva descripción se asocia con el documento y se guarda en base de datos.
Excepciones	N/A

CU-06	**Previsualizar el contenido de un documento**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, ha accedido al `detalle de un documento (CU-02)` y selecciona la opción de "*Preview*".
Secuencia	Se muestra el contenido del documento en una nueva ventana.
Poscondiciones	N/A
Excepciones	N/A

CU-07	**Guardar un documento**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, ha accedido al `detalle de un documento (CU-02)` y selecciona la opción de "Guardar".
Secuencia	Se muestra por pantalla una nueva ventana con la opción de `previsualizar el documento (CU-05)`. El usuario pulsa "Aceptar".
Poscondiciones	Se guarda el documento en base de datos.
Excepciones	N/A

CU-08	**Deshacer últimos cambios**
Precondiciones	El usuario ha iniciado sesión, posee el rol ***rrhh:manager***, ha accedido al `detalle de un documento (CU-02)` y selecciona la opción de "Deshacer cambios".
Secuencia	La ventana de la aplicación se recarga con la anterior información del contenido del documento.
Poscondiciones	Se actualiza el contenido del documento.
Excepciones	N/A

CU-09	**Acceder a un repositorio a través de git**
Precondiciones	El repositorio existe y el usuario posee permiso para acceder al mismo.
Secuencia	El usuario accede mediante un comando en un terminal o un cliente git al repositorio. El acceso es correcto y se muestra el contenido del repositorio.
Poscondiciones	N/A
Excepciones	El usuario no posee permiso para acceder.

Capítulo 6

Arquitectura

6.1 Introducción

En el capítulo 2 de esta memoria de trabajo de fin de grado se han expuesto las tecnologías para el desarrollo de este proyecto. Para implantar las funcionalidades descritas en el capítulo 5 se utiliza la siguiente arquitectura.

6.2 Tecnología

El proyecto se puede dividir en dos partes, por un lado, el *backend* y por otro, el *frontend*. Sigue un modelo de arquitectura cliente-servidor[26], muy empleado en el desarrollo de aplicaciones web. Un modelo cliente-servidor consiste en lo siguiente:

- El lado del servidor posee la lógica de la aplicación y contiene los recursos que serán consumidos por uno o varios clientes.

- El lado del cliente se ocupa de realizar peticiones concretas que ha hecho un usuario al servidor.

- El flujo de información es un ciclo y en una sola dirección, es decir, se inicia en el cliente, el servidor procesa la petición y este devuelve algún tipo de respuesta.

- Los clientes no establecen comunicación entre ellos.

La imagen 6.1 (página 18) muestra la arquitectura de la aplicación.

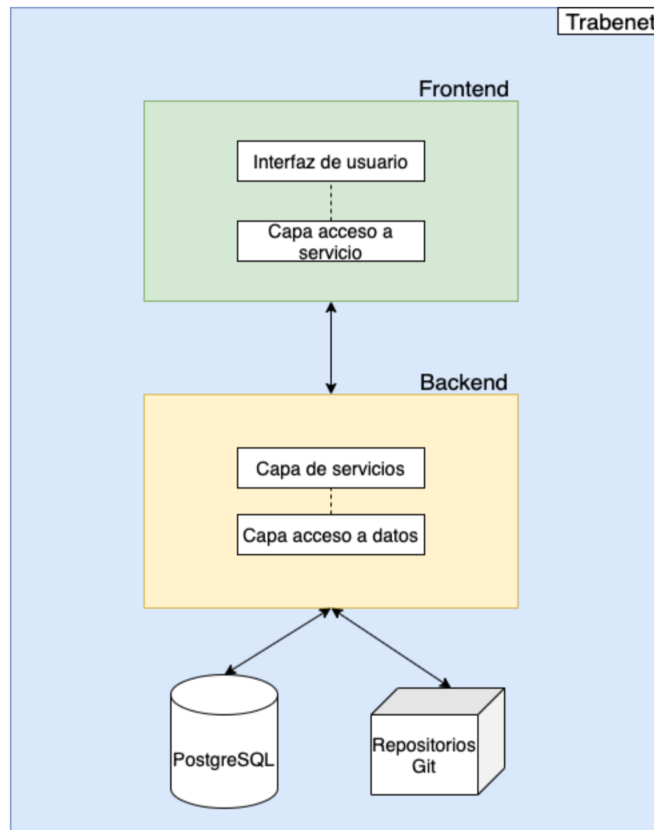

Figura 6.1: Arquitectura general de la aplicación.

6.2.1 *Backend*

Entre las responsabilidades del *backend* estarían la lógica de negocio, gestión de la información y el procesado de peticiones. El código fuente de este proyecto parte del código de trabenet, la aplicación de Trabe Soluciones, por lo tanto no se construye un proyecto desde cero.

A continuación se describen características importantes:

- El lenguaje de programación es Javascript[2], lenguaje moderno de alto nivel y tipado dinámico.
- La librería que se ocupa de la lógica está desarrollada con NodeJS[13], posee rapidez de ejecución y sigue un modelo *event-driven*[27].
- Yarn[20] es el gestor de paquetes utilizado para instalar nuevas dependencias o ejecutar *scripts*.
- Koa[23] es el servidor que se utiliza para la ejecución de la aplicación.

- Mediante una API implementada en GraphQL se accede las funcionalidades del *backend*.

El *backend* está estructurado como se muestra en la siguiente figura (ver 6.2 (página 19)):

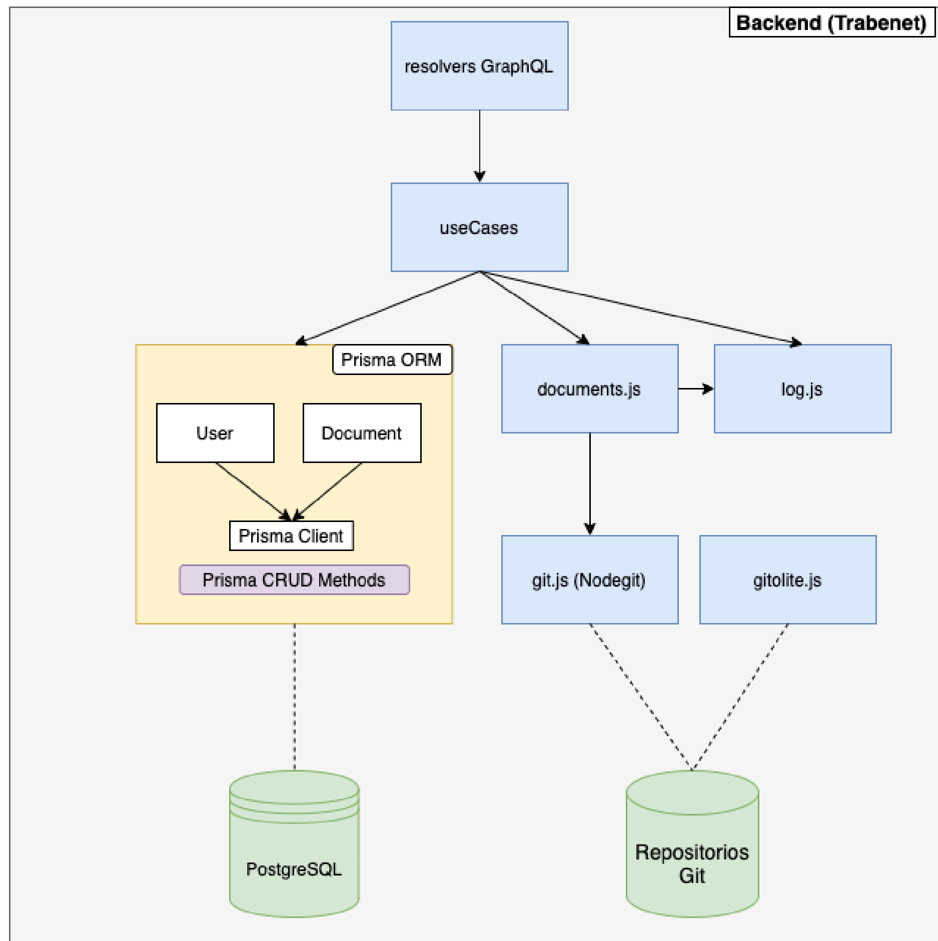

Figura 6.2: Estructura en detalle del *backend*.

El desarrollo tiene una serie de capas que se explicarán a continuación:

1. La primera capa corresponde a métodos (*resolvers*) que reciben las peticiones de la parte frontal de la aplicación y ejecutan llamadas a los casos de uso de la siguiente capa. La comunicación con el *frontend* se realiza a través de Apollo Server que es el encargado de recibir las peticiones. Estas peticiones incluyen un *token* JWT que contiene información acerca del usuario conectado como el login y sus roles. Los *resolvers* emplean esta información para limitar el acceso a las operaciones en función de los roles del usuario. Este sistema de autorización y autenticación ya estaba presente en *trabenet* antes del comienzo de este proyecto.

2. La segunda capa consiste en el desarrollo de casos de uso realizando llamadas a servicios de la última capa, un ejemplo sería un método que almacene nuevos archivos en un repositorio.

3. En la tercera capa se define utilizando la librería Prisma un nuevo modelo que corresponde a un documento con una serie de atributos. También se implementan métodos Create, Read, Update and Delete (CRUD)[28], es decir, búsqueda, creación, actualización y eliminación de entidades “Documento” que realizan llamadas al gestor de base de datos.

4. En la última capa se ha utilizado NodeGit para codificar métodos para inicializar repositorios, realizar un *commit*, ejecutar un *push*, obtener los *tags*, etc. Se emplea Git para el control de versiones de los repositorios y Gitolite para la gestión de los mismos: creación y permisos de los usuarios.

6.2.2 *Frontend*

El *frontend* es la parte visual de la aplicación y con la que el cliente final interactúa. Esta se encarga de captar las acciones del usuario y actuar acorde a ello, mostrando nueva información o enviando peticiones al *backend*.

Las características del *frontend* son las siguientes:

- El lenguaje de programación empleado sigue siendo Javascript.
- La librería utilizada para la interfaz de usuario es React[17].
- Para la renderización del contenido de los documentos, se ha empleado el componente react-markdown[29]. Este renderiza una cadena de texto en formato markdown.
- Muchos de los componentes empleados son heredados de la librería de componentes de *trabenet*. Solamente en algunas funcionalidades es necesario la modificación del componente.
- Para la navegación entre páginas de la aplicación se emplea react-router-dom[30].
- En las consultas con el *backend* se ha empleado el lenguaje GraphQL[3]. Las peticiones utilizan *tokens* JWT temporales. El sistema de login y generación de estos *tokens* ya estaba implementado en *trabenet*.
- La comunicación se realiza a través de Apollo Client[21]. Este permite realizar *queries*, consultas de información, y *mutations*, empleadas para enviar consultas de actualización de datos.

El *frontend* está estructurado como se muestra en la siguiente figura (ver 6.3 (página 21)):

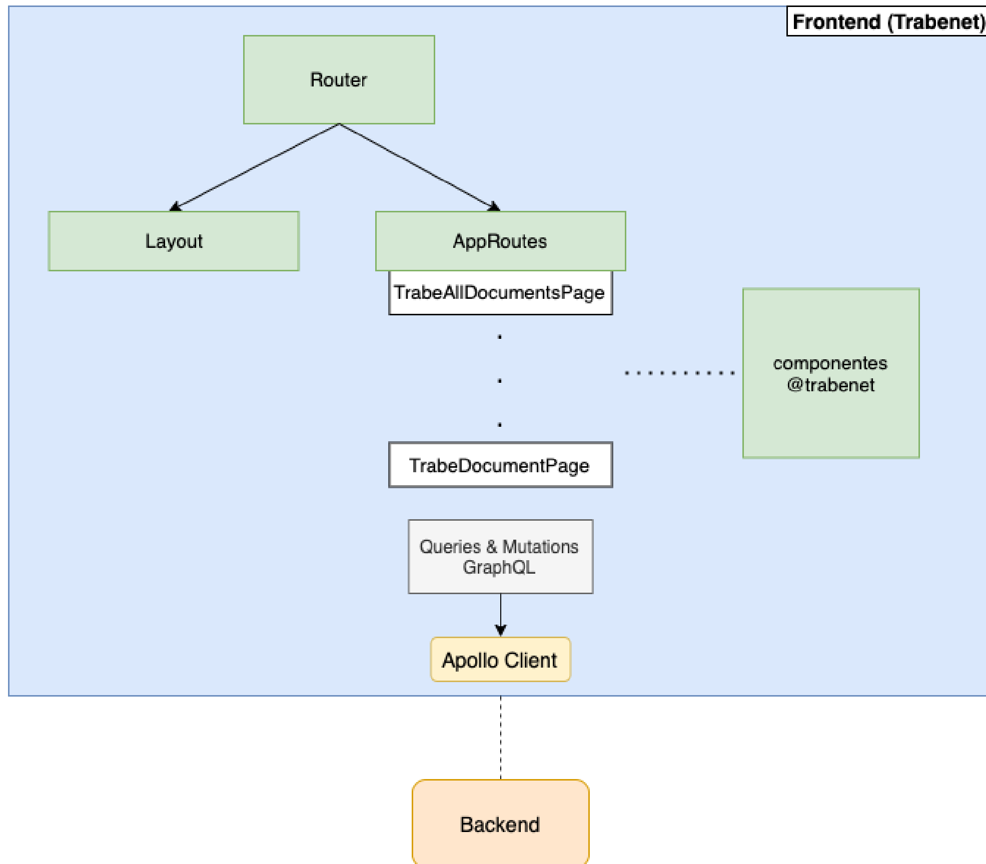

Figura 6.3: Estructura en detalle del *frontend*.

En la siguiente imagen (ver 6.4) se muestra un diagrama de secuencia del caso de uso de Modificación del contenido de un documento (ver 5.3) en el que se muestran las distintas capas del *backend* y *frontend* y cómo las peticiones fluyen entre las capas.

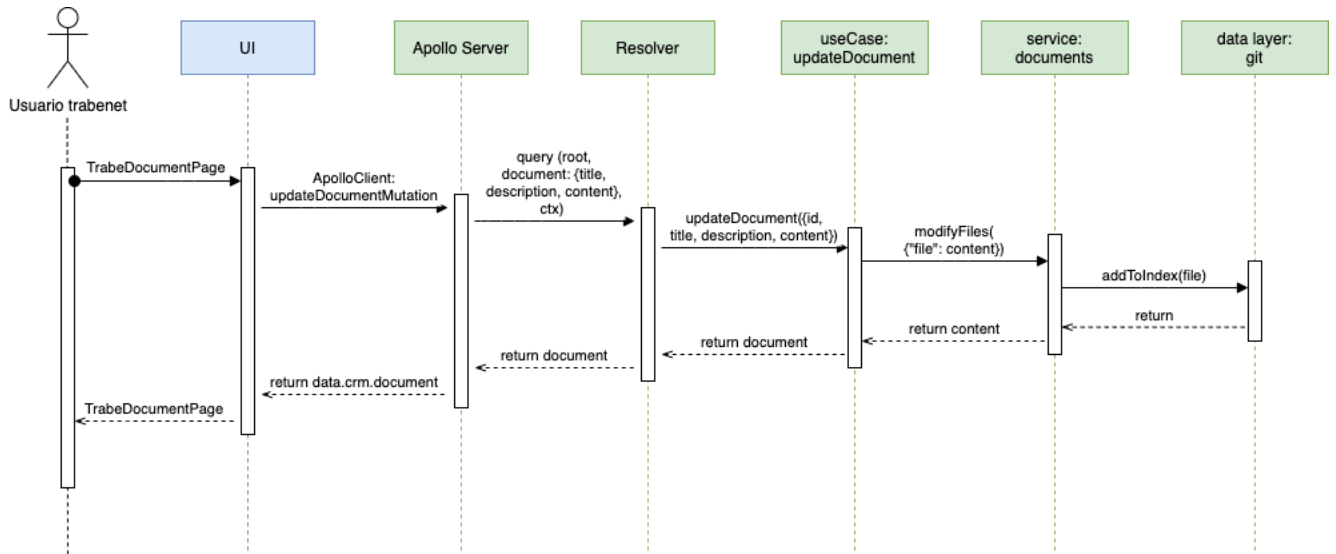

Figura 6.4: Diagrama de secuencia del caso de uso Modificación del contenido de un documento.

Capítulo 7

Iteraciones y estimación

7.1 Iteraciones

Las iteraciones del desarrollo del proyecto se detallan a continuación con sus estimaciones:

Nota: todas las fases descritas (excepto la iteración 0) poseen subfases de requisitos, diseño, desarrollo de código y de pruebas.

Tabla 7.1: Iteraciones del proyecto con estimación temporal.

Iteración	Objetivos	Estimación
0	Toma de contacto con las tecnologías. Instalación y configuración de herramientas requeridas. Descarga del código fuente de *trabenet* e instalación. Familiarización con la aplicación.	30 h
1	Desarrollo de los servicios de *git* para comunicación con repositorios git. Desarrollo y ejecución de pruebas de unidad de los servicios.	60 h
2	Desarrollo de los casos de uso en el *backend*. Desarrollo y ejecución de pruebas de unidad de los casos de uso.	50 h
3	Definición de nueva tabla *documents* en base de datos PostgreSQL. Desarrollo de métodos CRUD para acceso a tabla *documents*.	5 h
4	Definición de nuevos tipos de datos en el esquema de GraphQL. Desarrollo de los *resolvers* de GraphQL. Ejecución de pruebas.	8 h
5	Desarrollo de la consulta GraphQL para la obtención de todos los documentos. Desarrollo de la funcionalidad lista de documentos en el *frontend*. Ejecución de pruebas.	10 h
6	Desarrollo de la consulta GraphQL para la obtención de un documento concreto. Desarrollo de la funcionalidad detalle de un documento en el *frontend*. Ejecución de pruebas.	40 h
7	Desarrollo de la funcionalidad de deshacer los cambios en el contenido de un documento. Desarrollo de la funcionalidad de previsualización del contenido de un documento. Desarrollo de la funcionalidad de crear una versión de un documento. Desarrollo de la funcionalidad de guardado de un documento. Ejecución de pruebas.	35 h
Total:		**238 horas**

Capítulo 8

Desarrollo

8.1 Versiones del software

Las versiones del *software* con las que se ha trabajado en este proyecto son las siguientes:

- Node: 14.15.0
- Koa: 2.13.1
- NodeGit: 0.27.0
- Gitolite: 3.6.11
- Prisma: 2.22.1
- PostgreSQL: 13.2
- React: 16.14.0
- React-markdown: 6.0.2
- Apollo Server: 2.21.0
- Apollo Client: 3.3.11
- Yarn: 1.22.10
- Prettier: 2.2.1
- GraphQL: 15.5.0
- Docker: 20.10.5

8.2 Iteraciones

8.2.1 Iteración 0: formación en las tecnologías y definición de requisitos funcionales.

Iteración	Objetivos	Estimación	Desarrollo
0	Toma de contacto con las tecnologías. Instalación y configuración de herramientas requeridas. Descarga del código fuente de *trabenet* e instalación. Familiarización con la aplicación.	30 h	45 h

En esta primera iteración se estudia la documentación de las herramientas y librerías de código que se van a emplear. Algunas de ellas eran desconocidas como Docker, NodeGit, Gitolite, GraphQL, por lo tanto se han aprendido nuevos conceptos y tecnologías.

El código fuente de la aplicación *trabenet* se instala en la máquina local y se instalan todas las dependencias necesarias. Parte de la estructura del código se muestra en la figura 8.1 (página 26)

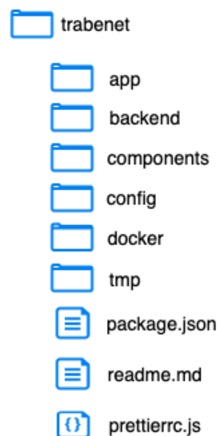

Figura 8.1: Estructura del código de la aplicación *trabenet*.

- El directorio `app` contiene las páginas del *frontend* de la aplicación y los componentes de React.
- El directorio `backend` se estructura en varios subdirectorios: `prisma`, `graphql`, `services` y `repos`.

- El directorio `config` contiene en su mayoría archivos de configuración con variables de entorno y otros para librerías, por ejemplo, prettier.

- El directorio `components` contiene componentes propios desarrollados por Trabe para su uso en el *frontend*: Avatar, Icon, File, CheckBox, SearchInput, etc.

- El directorio `tmp` contiene las carpetas locales de las imágenes del cliente de Docker. Estas se encargan de almacenar la información fuera de los contenedores de Docker para poder preservarla en la máquina local [31]. Las más importantes son: la imagen de gitolite, utilizada en el entorno de pruebas para almacenamiento de repositorios Git y la imagen de la base de datos PostgreSQL.

- El directorio `docker` contiene archivos de utilidad para la herramienta Docker. Uno de ellos, `docker-compose.yml`, expone los diferentes contenedores que se desplegarán para la aplicación. Utiliza un formato *YAML Ain't Markup Language* (YAML), formato de serialización [1] de datos legible por personas. En cada contenedor se configura la imagen que utiliza, el puerto que utilizará en el sistema local, variables de entorno como pueden ser usuario y contraseña y los *volumes*, que son rutas en las que se guardan los datos de los contenedores. Todas se almacenan en el directorio `tmp`. La figura 8.2 (página 27) expone un ejemplo de la imagen de gitolite y su configuración:

```
gitolite:
  image: jgiannuzzi/gitolite:latest
  volumes:
    - "../tmp/gitolite/keys:/etc/ssh/keys"
    - "../tmp/gitolite/repos:/var/lib/git"
  ports:
    - "2222:22"
  env_file:
    - config/gitolite.env
```

Figura 8.2: Configuración en `docker-compose.yml` de la imagen de Gitolite.

El código del *backend* se estructura de la siguiente manera (ver 8.3 (página 28)):

- `graphql`: este directorio contiene los *resolvers* que realizan las llamadas a los casos de uso.

- `prisma`: contiene el esquema de Prisma, el ORM encargado de transformar objetos en datos manipulables por el sistema de gestión de base de datos.

[1] La serialización consiste en el proceso de codificar un objeto en un formato de manera que pueda ser almacenado o distribuido.

- `repos`: dispone de servicios para realizar las llamadas a la base de datos y obtener la información relativa a los documentos.
- `services`: este directorio contiene todos los servicios empleados en la aplicación. Entre ellos estarían los encargados de implementar funcionalidades de git y los casos de uso.
- `package.json` es el archivo de gestión de paquetes del *backend*.
- `server.js` es el archivo para la creación e inicilización del servidor Koa.

Figura 8.3: Estructura del código del *backend*.

8.2.2 Iteración 1: desarrollo de la capa de acceso a datos (repositorios `Git`).

Iteración	Objetivos	Estimación	Desarrollo
1	Desarrollo de los servicios de *git* para comunicación con repositorios git. Desarrollo y ejecución de pruebas de unidad de los servicios.	60 h	75 h

Tras la primera iteración, la segunda está centrada en la capa de acceso a datos: los repositorios git. En estos se almacena la documentación de la aplicación *trabenet*. Esta capa se desarrolla utilizando la librería NodeGit que consiste en un módulo asíncrono que permite realizar llamadas a libgit2, otra librería desarrollada en C que implementa métodos git. De esta manera, permite ejecutar comandos git sin asumir nada sobre la máquina en la que se está ejecutando.

Dentro de `/backend/services` se encuentra el archivo `git.js` que define esta capa. Gitolite es la librería que se encarga de la gestión de acceso a los repositorios git. Su funcionamiento es el siguiente:

- Posee un repositorio administrador que contiene: directorio con las claves públicas SSH del usuario administrador y resto de usuarios y otro directorio con un archivo de configuración `gitolite.conf`.
- Mediante una serie de reglas se establecen los permisos de acceso y acciones permitidas de los usuarios sobre el repositorio git.

En la figura 8.4 (página 29) se expone un ejemplo del contenido de `gitolite.conf`:

```
repo gitolite-admin
    RW+     =   admin

repo testing
    RW+     =   @all

repo 000034
    RW+ =   john.smith alice23
```

Figura 8.4: Ejemplo de configuración de los repositorios en gitolite.

La imagen anterior muestra la siguiente información:

- Existen tres repositorios: `gitolite-admin`, `testing` y `000034`.
- Al primero solo tiene acceso el usuario "admin".
- Al repositorio `testing` tiene acceso "@all". Esta directiva "@all" de gitolite engloba a todos los usuarios.
- Al repositorio `000034` tienen acceso dos usuarios: "john.smith" y "alice23".
- En todos los repositorios, los usuarios que tienen acceso tienen permisos completos sobre el repositorio.

En `gitolite.js` se definen una serie de servicios para interactuar con el repositorio administrador. Los métodos son los siguientes:

- **manageGroups**: este método recibe como parámetro un objeto con nombres de grupos y la lista de usuarios de cada uno. Su funcionamiento consiste en:

- Devolver una cadena de texto con cada nombre de grupo y usuarios

- **manageRepos**: este método recibe un objeto en el que las claves son los nombres de los repositorios. El valor de esas claves a su vez es otro objeto que contiene cada uno de los permisos con la lista de usuarios que poseen esos permisos. El funcionamiento consiste en:

    - Devolver una cadena de texto con los nuevos repositorios con usuarios y permisos de estos últimos.

- **writeConfig**: este método recibe como parámetro una cadena de texto y su función consiste en escribir esa cadena de texto en el archivo `gitolite.conf`.

- **manageConf**: en este método se engloba el uso de los tres métodos anteriores.

- **manageGit**: este método recibe un objeto con los usuarios y sus claves públicas y un *path* del directorio administrador en caso de que no se use el valor por defecto. Su funcionamiento consiste en:

    - Escribir las claves públicas como ficheros en el directorio `gitolite-admin/keys`.
    - Crear un objeto *commit* con estos mismos archivos.
    - Realizar un *push* al directorio gitolite.

- **updateConfig**: este método se encarga de conectar mediante SSH a la imagen `docker` de gitolite. Después, realiza llamadas a los métodos **manageConf** y **manageGit**.

- **deleteOldRepos**: este método se encarga de eliminar del sistema de ficheros los repositorios que no estén presentes en la configuración de gitolite.

- **deleteRepoFromConf**: este método recibe como parámetro un nombre de repositorio y se encarga de borrarlo de la configuración de gitolite.

El funcionamiento para emplear los métodos anteriores es el siguiente:

- Se crea una instancia que se llama *gitClient*. Para la creación de esta, se precisan de unos parámetros que son:

    - **repoUrl**: la *Uniform Resource Locator* (URL) del repositorio donde esté definido.
    - **repoPath**: la ruta local donde se realizará el clonado.

  - **user**: un objeto formado por el nombre del usuario, un *path* que apunte a la clave pública, otro *path* que apunte a la clave privada y una *passphrase*. Esta *passphrase* consiste en una cadena de texto para aumentar el nivel de seguridad de manera local de la clave SSH en caso de que el sistema local se vea comprometido.
  - **committer**: el usuario que realizará los *commits*.

- *gitClient*, usando las claves proporcionadas, crea una credencial en memoria utilizando un método propio de NodeGit.

Los métodos definidos en `git.js` son los siguientes:

- **pull**: realiza un *pull* dentro un repositorio. En `git`, el comando *pull* es utilizado para actualizar el contenido local con el contenido del repositorio remoto. No recibe parámetros.

- **createRepo**: permite crear e inicializar un repositorio en una ruta local del sistema.

- **cloneRepo**: realiza un clonado de otro repositorio en una ruta local del sistema y hace que apunte sobre el que se clona.

- **createCommit**: realiza un objeto commit dentro del repositorio. Como parámetros recibe tres: **files** (objeto con nombres de archivo y contenido), **author** y **message**.

- **getTags**: este método obtiene una lista de los nombres de los *tags* existentes en el repositorio.

- **push**: realiza un *push* dentro un repositorio. En git, el comando *push* es utilizado para actualizar el contenido remoto con el contenido del repositorio local.

- **commitHistory**: este método obtiene una lista de los objetos *commits* existentes en el repositorio. Cada objeto contiene: el *Secure Hash Algorithm* (SHA), mensaje, autor (nombre y correo electrónico) y el *committer*, aquella persona o servicio que ha realizado el *commit*.

- **addToIndex**: este método escribe y añade los ficheros que recibe como parámetro al índice del repositorio git.

- **createTag**: este método se encarga de crear un objeto *tag* en el repositorio git. Como parámetros recibe el identificador SHA, el autor y el nombre y mensaje del *tag*.

- **getAuthors**: este método se encarga de obtener los autores que han realizado algún *commit* en el repositorio.

- **getTagByName**: obtiene el objeto *tag* haciendo una búsqueda por el nombre que se recibe como argumento.

- **getVersions**: haciendo uso del método **getTags** y **getTagByName** obtiene los objetos *tag* definidos dentro del repositorio. Un objeto *version* está compuesto de: nombre, mensaje, *tagger* e identificador y fecha del *commit*.

- **getLastestCommit**: obtiene el objeto *commit* al que el *HEAD* apunta. El *HEAD* en git es un puntero a la rama actual en la que se está trabajando.

- **getCommitFilesContent**: recibe como parámetro el SHA de un *commit* y devuelve una lista con los archivos de ese *commit*. Un elemento de la lista está compuesto del nombre y el contenido del archivo.

- **discardLocalChanges**: este método actúa sobre el contenido de un documento descartando sus cambios locales que todavía no hayan sido "commiteados".

8.2.3 Iteración 2: desarrollo de los casos de uso en el *backend*.

Iteración	Objetivos	Estimación	Desarrollo
2	Desarrollo de los casos de uso en el *backend*. Desarrollo y ejecución de pruebas de unidad de los casos de uso.	50 h	90 h

Las iteraciones uno y dos, son las de mayor duración en el proyecto. Esta parte está enfocada en el desarrollo de los casos de uso[2] de la aplicación. El código está localizado en `documents.js` y se hace uso de los métodos definidos en la iteración uno.

El funcionamiento de `documents.js` es el siguiente:

- Se crea una instancia *documentsClient* para poder realizar llamadas a los métodos, esta posee los siguientes parámetros:
  - **repoName**: el nombre del directorio donde se creará el repositorio git.
  - **keys**: un objeto que contiene nombres de usuarios y sus claves públicas.

[2] Un caso de uso es una secuencia de acciones llevadas a cabo por los usuarios de un sistema que ofrecen un resultado observable.

  - **perms**: un objeto que define por cada conjunto de permisos, una lista con los usuarios que poseen los mismos.
  - **adminPath**: el *path* del repositorio gitolite de administrador, en caso de que no se use el valor por defecto.

- Se realiza una llamada a *gitClient*, la instancia para utilizar métodos de la capa de acceso a datos.

- Se actualiza la configuración con los nuevos usuarios, permisos y claves en el repositorio administrador de gitolite empleando el método **updateConfig** definido en los servicios de `gitolite.js`.

- Se clona el nuevo repositorio.

Los casos de uso presentes en `documents.js` son los siguientes:

- **modifyFiles**: recibe como parámetro un objeto con nombres de archivos y su contenido como valor del objeto. Se encarga de añadir estos archivos al *index* del repositorio git.

- **saveFiles**: recibe como parámetros un objeto autor y un objeto archivos. Se encarga de crear un objeto *commit* y realizar un *push*.

- **getFiles**: se encarga de leer el directorio del repositorio git y devuelve los archivos que haya leído.

- **fileContent**: recibe como parámetro un nombre de archivo y se encarga de devolver el contenido de ese archivo en caso de que exista.

- **getAuthors**: realiza una llamada al método **getAuthors** de `git.js`.

- **getVersions**: realiza una llamada al método **getVersions** de `git.js`.

- **createVersion**: recibe como parámetros un nombre, mensaje y autor. Se encarga de crear una nueva versión en el repositorio utilizando métodos presentes en `git.js`.

- **getCommitFilesContent**: recibe como parámetro el identificador SHA de un objeto *commit* y devuelve los archivos implicados en el mismo.

- **undoLatestChanges**: realiza una llamada al método **discardLocalChanges** de `git.js`.

8.2.4 Iteración 3: desarrollo de acceso a base de datos PostgreSQL.

Iteración	Objetivos	Estimación	Desarrollo
3	Definición de nueva tabla *documents* en base de datos PostgreSQL. Desarrollo de métodos CRUD para acceso a tabla *documents*.	5 h	10 h

Esta iteración tiene como objetivos desarrollar el acceso a base de datos con PostgreSQL y los métodos CRUD.

En el esquema de Prisma se modelan los siguientes tipos:

```
model User {
  id Int @id @default(autoincrement())
  login String @unique
  // and other data fields ...

  @@map(name: "users")
}

model Document {
  @@map(name: "documents")

  id          Int        @id @default(autoincrement())
  createdAt   DateTime   @db.Timestamp(4) @default(now())
    @map("created_at")
  updatedAt   DateTime   @db.Timestamp(4) @updatedAt
    @map("updated_at")
  @@index([updatedAt])

  title        String   @db.VarChar(128)
  description  String   @db.VarChar(512)
}
```

Podemos observar que:

- Se emplean dos modelos en Prisma, User y Document con sus respectivos atributos, algunos emitidos por simplificación.
- Los modelos se mapean a nombre de tablas en base de datos PostgreSQL mediante la sentencia “@@map(name: <name>)”.

- Prisma también permite mapear atributos mediante “@map(<atributo>)” y elegir el tipo nativo de PostgreSQL que se utilizará, además de otras funcionalidades concretas como claves autoincrementales, actualización automática de acceso a registros, etc.

Una vez definido el esquema en Prisma, mediante la sentencia “prisma generate” se genera un cliente Prisma que se empleará para invocar los métodos CRUD. Prisma también dispone de la herramienta Migrate que permite llevar un historial sobre el esquema de base de datos en el directorio `backend/prisma/migrations`. A través del comando “prisma migrate dev –name <name>” se crea un nuevo archivo SQL con los cambios realizados que puede subirse a una herramienta de control de versiones como git. También es útil para realizar un despliegue de la aplicación por si se produce algún fallo durante el mismo y ver si los cambios en base de datos han provocado el error.

El modelo *User* se corresponde con la tabla *users* en el gestor de base de datos. Este modelo no se modifica pero se expone porque se utiliza en el proyecto. Con las modificaciones realizadas en el esquema de Prisma se genera la siguiente tabla (ver 8.5 (página 35)):

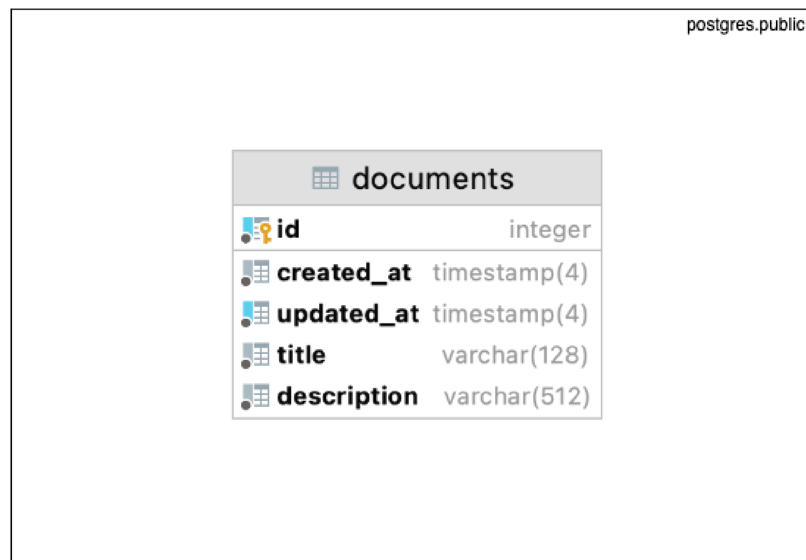

Figura 8.5: Entidades *users* y *documents*.

La tabla *documents* que corresponde con el modelo *Document* contiene los siguientes atributos:

- **id**: identificador de tipo *Int* de cada documento de la tabla y autoincrementado por el gestor de base de datos.

- **createdAt**: fecha de creación de tipo *DateTime*.

· **updatedAt**: fecha de actualización de tipo *DateTime*. Sobre este, se crea un índice.

· **title**: título del documento de tipo *String*.

· **description**: descripción del documento de tipo *String*.

Por último, en `backend/repos/documents.js` se definen los métodos de creación, búsqueda, eliminación y actualización (CRUD) empleando las funcionalidades que ofrece la librería Prisma.

- **all**: obtención de todos los documentos
- **find**: búsqueda de un documento
- **create**: creación de un documento
- **update**: actualización de un documento
- **remove**: eliminación de un documento

8.2.5 Iteración 4: desarrollo de resolvers de GraphQL.

Iteración	Objetivos	Estimación	Desarrollo
4	Definición de nuevos tipos de datos en el esquema de GraphQL. Desarrollo de los *resolvers* de GraphQL. Ejecución de pruebas.	8 h	15 h

En esta iteración el objetivo es desarrollar los *resolvers* de GraphQL, funciones que reciben las peticiones del *frontend* de la aplicación y devuelven un resultado al mismo y definir los nuevos tipos de datos en el esquema de GraphQL.

Los nuevos tipos de datos han sido los siguientes:

- El tipo **Doc**:

```
type Doc {
   id: ID!
   title: String
   description: String
   updatedAt: Date
```

```
6        content: String
7        authors: [Person]
8        versions: [Version]
9     }
10
```

El tipo *Person* que se emplea para los autores de los documentos ya estaba definido en *trabenet*. Este tipo contiene toda la información de un usuario de la aplicación: nombre, correo electrónico, roles, login, etc.

- El tipo **Version**:

```
1     type Version {
2        name: String
3        message: String
4        tagger: Person
5        commit: String
6        date: Date
7     }
8
```

- El tipo **Crm**: en este se agrupan las consultas de búsqueda sobre documentos.

```
1     type Crm {
2        document(id: Int!, commit: String): Doc
3        documents: [Doc]
4     }
5
```

- El tipo **DocumentMutation**: en este se agrupan las mutaciones que se pueden realizar sobre el tipo Doc.

```
1     type DocumentMutation {
2        create(document: DocumentInput): Doc
3        update(id: Int!, document: DocumentInput): Doc
4        save(id: Int!): Doc
5        remove(id: Int!): Doc
6        createVersion(id: Int!, version: VersionInput): Doc
7        undoChanges(id: Int!): Doc
```

```
8      }
9
```

- El tipo **DocumentInput**: el tipo GraphQL *input* se utiliza en consultas de actualización frecuentes que requieran los mismos parámetros.

```
1      input DocumentInput {
2         title: String!
3         description: String
4         content: String
5      }
6
```

- El tipo **VersionInput**:

```
1      input VersionInput {
2         name: String!
3         message: String!
4      }
5
```

Respecto a las funciones *resolvers*, estas poseen tres parámetros: un contexto con información como el usuario conectado, los argumentos de la *query* y el objeto padre. Se encargan de realizar la llamada a cada uno de los servicios definidos en `backend/graphql/useCases/documents`. En todos estos métodos se realizan llamadas a los servicios de la base de datos (Prisma, PostgreSQL) y a los casos de uso de `documents.js`. Los *resolvers* son los siguientes:

- **/entry/create**: realiza una llamada al caso de uso **createDocument** que se encarga de crear una nueva instancia en base de datos y un nuevo repositorio git asociado a ese documento.

- **/entry/remove**: realiza una llamada al caso de uso **removeDocument** que elimina el documento de base de datos y el repositorio.

- **/entry/save**: realiza una llamada al caso de uso **saveDocument** que guarda el contenido de un documento.

- **/entry/update**: realiza una llamada al caso de uso **updateDocument** que actualiza la descripción o el contenido de un documento.
- **createVersion**: realiza una llamada al caso de uso **createVersion** que crea una nueva versión de un documento.
- **document**: realiza una llamada al caso de uso **findDocument** que recupera toda la información asociada a un documento: título, descripción, versiones, autores y contenido.
- **documents**: realiza una llamada al caso de uso **getDocuments** que recupera la lista de documentos existentes.
- **undoChanges**: realiza una llamada al caso de uso **undoLatestChanges** que elimina los últimos cambios en el contenido del documento.

8.2.6 Iteración 5: desarrollo en el *frontend*: lista de documentos.

Iteración	Objetivos	Estimación	Desarrollo
5	Desarrollo de la consulta GraphQL para la obtención de todos los documentos. Desarrollo de la funcionalidad lista de documentos en el *frontend*. Ejecución de pruebas.	10 h	10 h

A partir de la quinta iteración, el desarrollo está enfocado en la parte frontal de la aplicación. En esta iteración se ha desarrollado la funcionalidad de la lista de documentos: consulta en formato GraphQL y codificación de los componentes. Solamente determinados roles pueden acceder a la documentación, en este caso, el rol de ***rrhh:manager***.

El código del *frontend* (ver 8.6 (página 40)) se estructura de la siguiente manera :

- `components`: contiene el código de los componentes React empleados en la aplicación.
- `pages`: contiene el código de cada una de las páginas de la aplicación.
- `package.json` es el archivo de gestión de paquetes del *frontend*.
- `main.jsx` contiene la estructura de los componentes de la aplicación: enrutado, componente de notificaciones, de idioma, gestión de errores, etc.

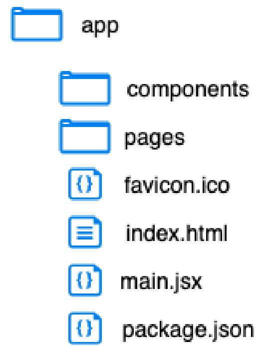

Figura 8.6: Estructura del código del *frontend*.

La consulta GraphQL para obtener los documentos es la siguiente:

```
query documents {
    crm {
      documents {
        id
        title
        description
        content
        updatedAt
        authors {
          firstName
          lastName
          gravatar
          login
          email
        }
      }
    }
  }
```

La figura 8.7 (página 41)) muestra los componentes que forman la página de la lista de documentos:

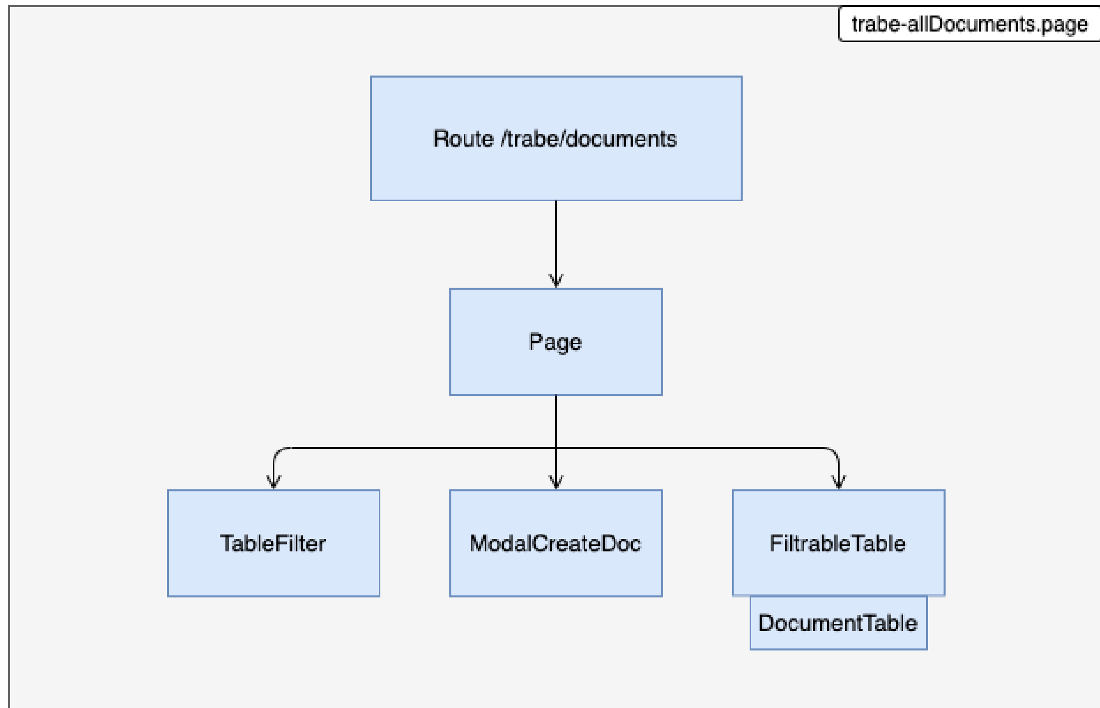

Figura 8.7: Componentes empleados para la página de la lista de documentos.

En la aplicación se ha ubicado la opción para acceder a la documentación en el menú principal de la aplicación (figura 8.8 (página 41)).

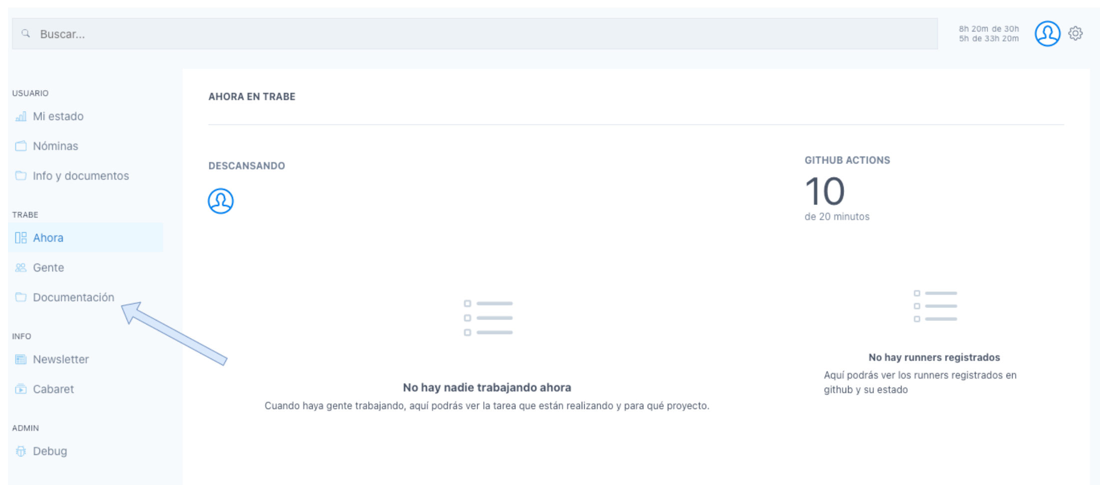

Figura 8.8: Ubicación del enlace para ver la lista de documentos en la aplicación *trabenet*.

La figura 8.9 (página 42) muestra la página de la lista de documentos:

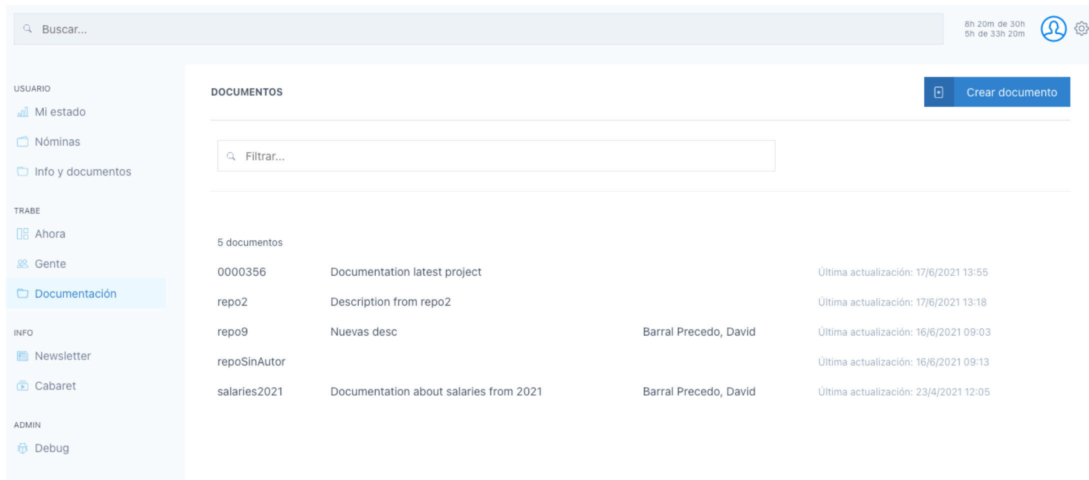

Figura 8.9: Vista de la página documentos en la aplicación *trabenet*.

8.2.7 Iteración 6: desarrollo en el *frontend*: vista de documento.

Iteración	Objetivos	Estimación	Desarrollo
6	Desarrollo de la consulta GraphQL para la obtención de un documento concreto. Desarrollo de la funcionalidad detalle de un documento en el *frontend*. Ejecución de pruebas.	40 h	50 h

La sexta iteración consiste en el desarrollo de la página del detalle de un documento después de haberlo seleccionado en la lista de documentos. En esta página recaen muchos casos de uso desarrollados en anteriores iteraciones. El detalle del documento incluye: título, descripción, lista de autores, lista de versiones y contenido del documento.

La consulta GraphQL para obtener la información de un documento es la siguiente:

```
query document($id: Int!, $commit: String) {
    crm {
        document(id: $id, commit: $commit) {
            id
            title
            description
            content
            updatedAt
```

```
9	                authors {
10	                    firstName
11	                    lastName
12	                    email
13	                    login
14	                    gravatar
15	                    active
16	                }
17	                versions {
18	                    name
19	                    message
20	                    commit
21	                    date
22	                    tagger {
23	                        gravatar
24	                        login
25	                        firstName
26	                        lastName
27	                    }
28	                }
29	            }
30	        }
31	}
```

La figura 8.10 (página 43)) muestra los componentes que forman la página del detalle de un documento:

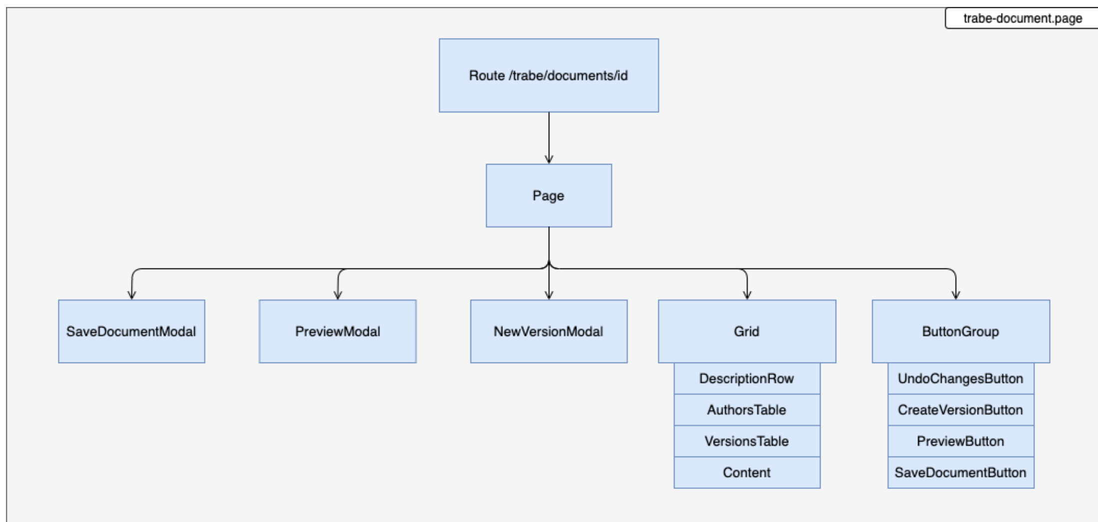

Figura 8.10: Componentes empleados para la página del detalle de un documento.

La figura 8.11 (página 44) muestra el detalle de un documento dentro de la aplicación:

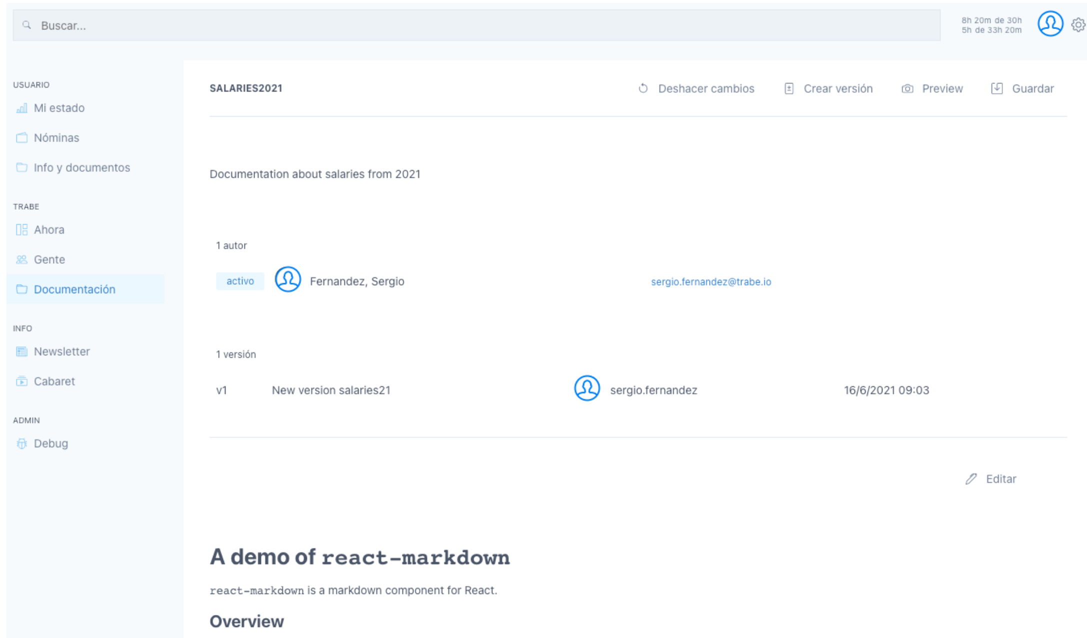

Figura 8.11: Vista de la página del detalle de un documento en la aplicación *trabenet*.

8.2.8 Iteración 7: desarrollo en el *frontend*: funcionalides del documento (edición, versionado, etc).

Iteración	Objetivos	Estimación	Desarrollo
7	Desarrollo de la funcionalidad de deshacer los cambios en el contenido de un documento. Desarrollo de la funcionalidad de previsualización del contenido de un documento. Desarrollo de la funcionalidad de crear una versión de un documento. Desarrollo de la funcionalidad de guardado de un documento. Ejecución de pruebas.	35 h	50 h

La última iteración de este proyecto consiste en el desarrollo de las funcionalidades dentro de la página del detalle de un documento.

8.2.8.1 Modificación de la descripción

En la siguiente figura (ver 8.12 (página 45)) se expone el funcionamiento:

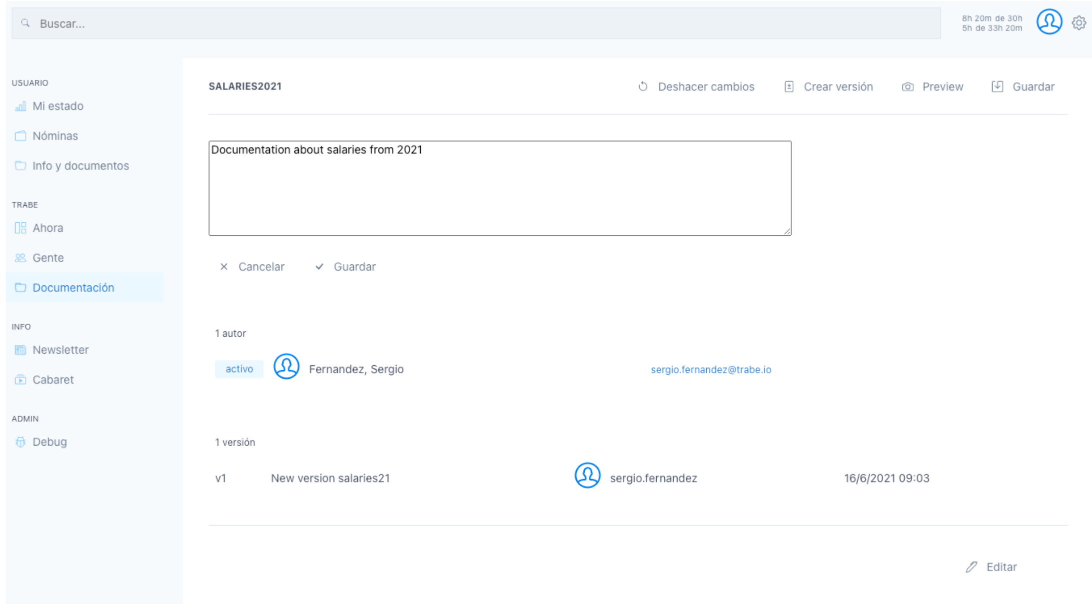

Figura 8.12: Modificación de la descripción de un documento dentro de la aplicación.

Consulta GraphQL empleada:

```
mutation update($id: Int!, $title: String!, $description: String, $content: String) {
    crm {
        document {
            update(id: $id, document: { title: $title,
                description: $description,
                content: $content }) {
                    id
                    title
                    description
                    updatedAt
        }
    }
}

```

8.2.8.2 Modificación del contenido del documento

Consulta GraphQL empleada: se corresponde con la misma empleada en la modificación de la descripción (ver 8.2.8.1).

En la siguiente figura (ver 8.13 (página 46)) se expone cómo se modifica el contenido de un documento:

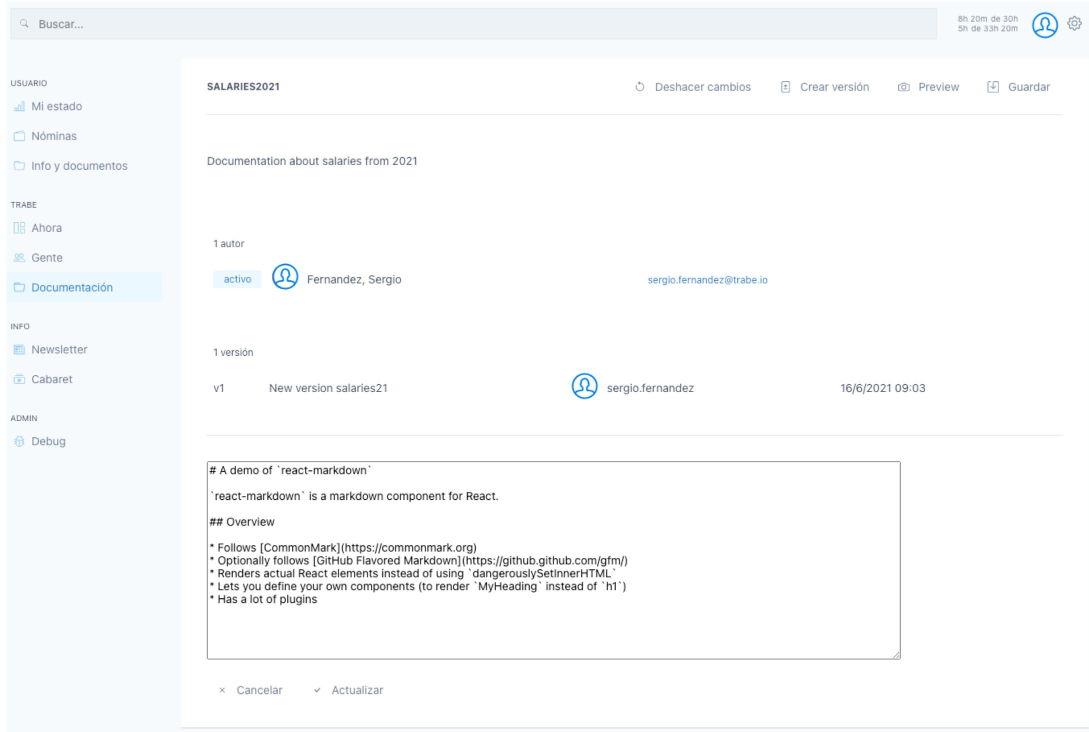

Figura 8.13: Modificación del contenido de un documento dentro de la aplicación.

8.2.8.3 Creación de versión de un documento

Consulta GraphQL empleada:

```
1   mutation createVersion($id: Int!, $name: String!, $message:
    String!) {
2       crm {
3       document {
4           createVersion(id: $id, version: {
5               name: $name,
6               message: $message }) {
7                   id
8                   title
9                   description
10                  updatedAt}
11          }
12      }}}
13
```

La figura 8.14 (página 47) muestra cómo crear una versión de un documento.

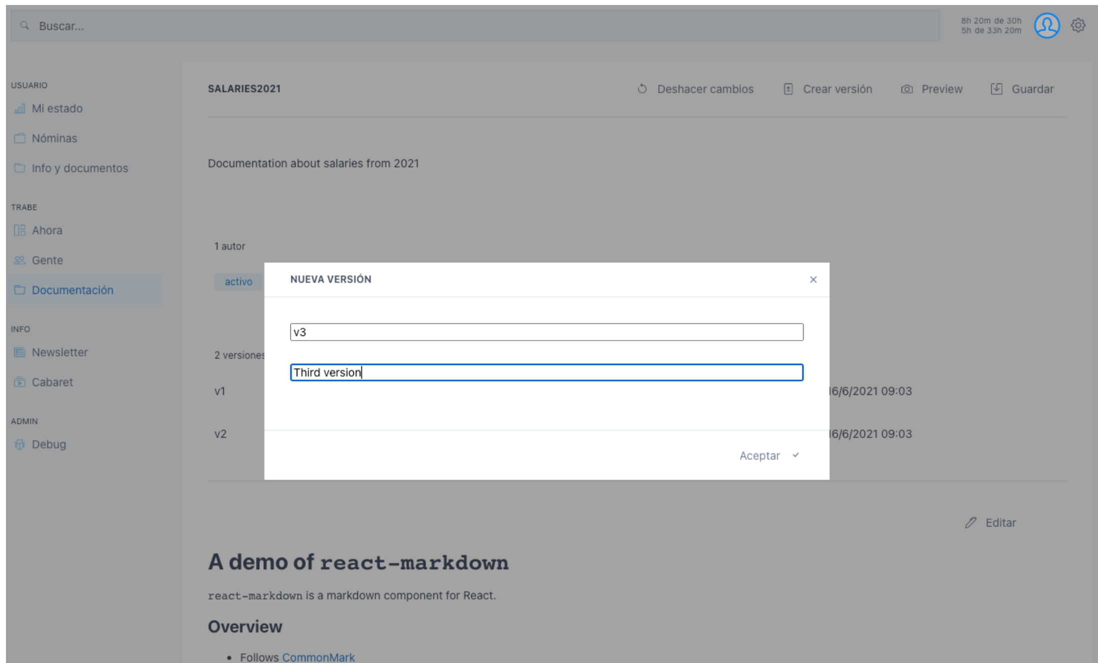

Figura 8.14: Creación de una nueva versión del documento en la aplicación.

En la figura 8.15 (página 47) se muestra la lista de versiones con la nueva versión.

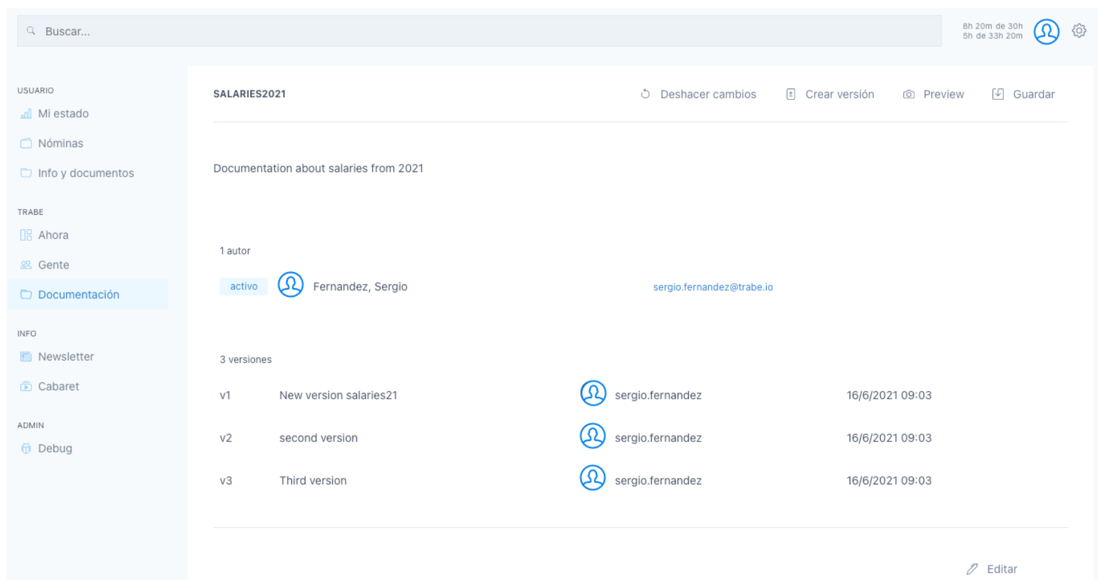

Figura 8.15: Lista de versiones del documento con la nueva versión.

8.2.8.4 Previsualización del documento

Se emplea un componente React que renderiza el contenido del documento en un formato markdown. La previsualización del documento con el formato corporativo de Trabe no se ha desarrollado porque ha quedado fuera del alcance del proyecto.

La figura 8.16 (página 48) muestra una previsualización del contenido del documento en markdown:

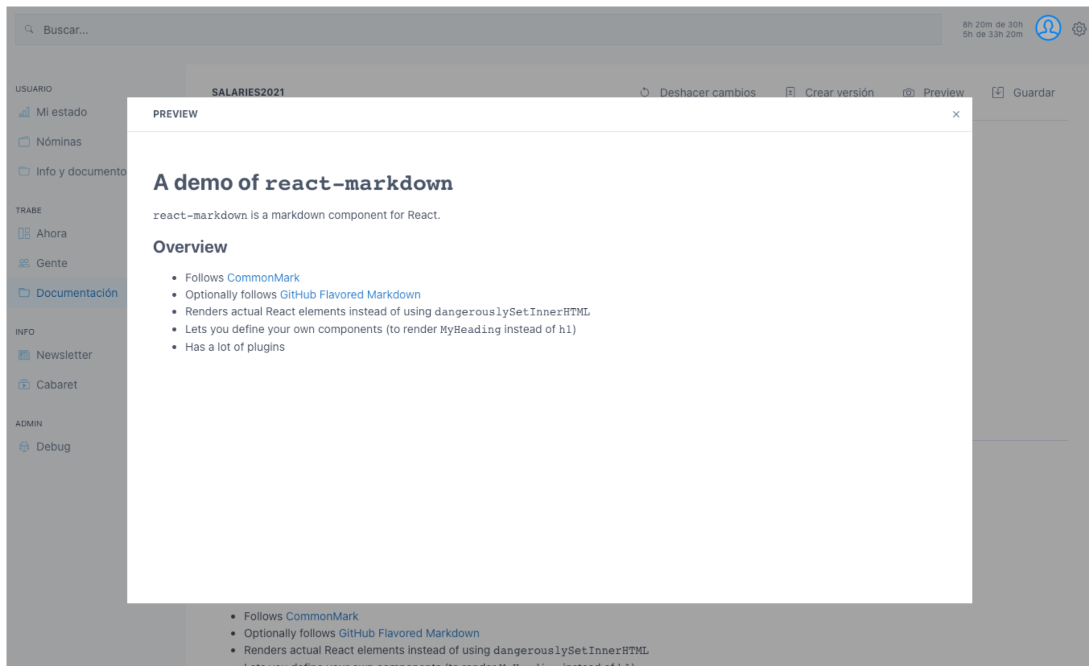

Figura 8.16: Previsualización del contenido del documento en la aplicación.

8.2.8.5 Deshacer últimos cambios

Esta funcionalidad no posee ninguna ventana adicional. Al pulsar el botón con título "Deshacer cambios", este automáticamente ejecuta la acción de deshacer los últimos cambios en el contenido del documento.

8.2.8.6 Guardar documento

Consulta GraphQL empleada:

```
1  mutation save($id: Int!) {
2      crm {
```

```
3           document {
4               save(id: $id) {
5                   id
6                   title
7                   description
8                   updatedAt
9               }
10          }
11      }
12  }
13
```

La figura 8.17 (página 49) muestra la ventana de guardado de un documento:

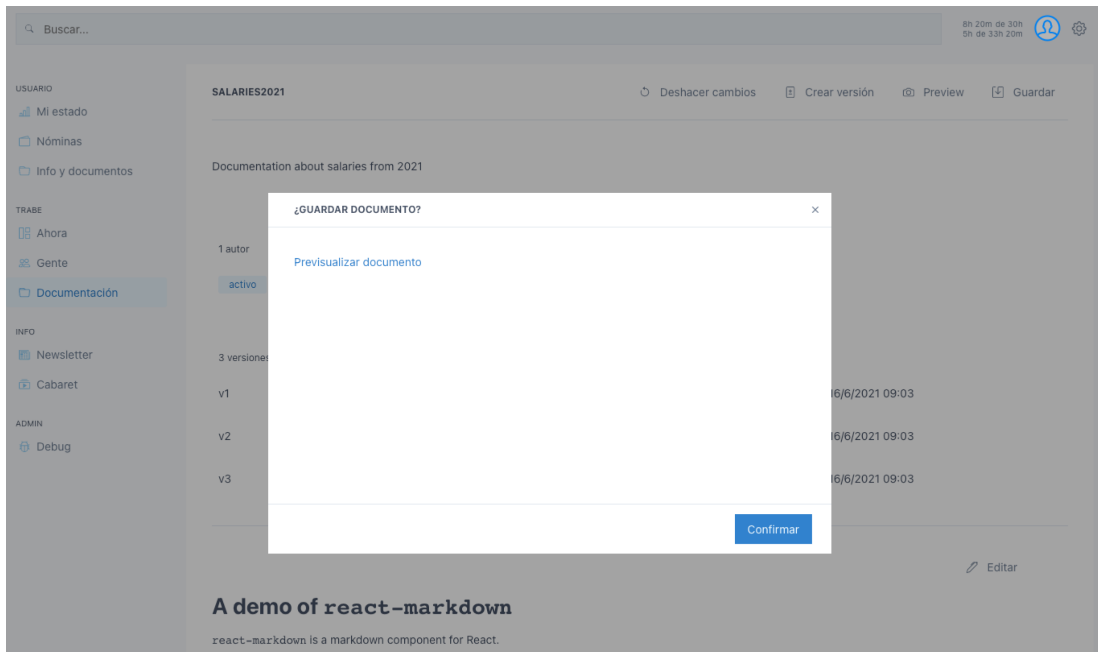

Figura 8.17: Guardado del documento con opción de previsualización.

Capítulo 9

Análisis de los costes

En la sección 3.1 del capítulo 3 se ha expuesto la metodología que este proyecto ha seguido y las diferentes iteraciones se han mostrado en el capítulo 7 de esta memoria.

Respecto a la planificación, en la tabla 9.1 se expone los tiempos de desarrollo estimados frente a los reales de cada una de las iteraciones del proyecto.

Tabla 9.1: Iteraciones del proyecto con los tiempos estimados y reales.

Iteración	Descripción	Tiempo estimado	Tiempo real
0	Formación en tecnologías	30 h	45 h
1	Capa acceso a datos	60 h	75 h
2	Casos de uso	50 h	90 h
3	Acesso a base de datos	5 h	10 h
4	Desarrollo de resolvers	8 h	15 h
5	Lista de documentos	10 h	10 h
6	Detalle de documento	40 h	50 h
7	Funcionalidades *frontend*	35 h	50 h
Total:		**238 horas**	**345 horas**

Este proyecto tuvo como asignación de beca para el alumno un total de **2.000 €**. Respecto al importe económico del proyecto, asumiento un coste por hora del desarrollo entorno a **18 €/h**, el total del proyecto sería **6.210 €**.

Capítulo 10

Conclusiones

10.1 Conclusiones

En esta sección se analizan las conclusiones del desarrollo de este proyecto. Las funcionalidades objetivo ya han sido descritas en el apartado 5.3. Todas ellas han sido desarrolladas y fueron probadas en un entorno de pruebas. Actualmente, están siendo testeadas en un entorno de producción por un conjunto limitado de usuarios para obtener un primer *feedback* inicial.

El conjunto de la aplicación se ha codificado de manera que permita reusabilidad de componentes y escalabilidad de cara a futuros desarrollos de nuevas características. Respecto a la estimación del proyecto, las iteraciones uno y dos fueron las que más problemas y conflictos generaron. Los problemas surgieron por la falta de experiencia con las tecnologías NodeGit y Gitolite. La primera no posee una gran comunidad de usuarios y la documentación de la librería es escasa. Gitolite requiere un cierto tiempo de aprendizaje sobre su funcionamiento pero es una librería muy bien documentada. En consecuencia, hubo un retraso de tiempo respecto a la estimación inicial del desarrollo de las funcionalidades.

El modelo de arquitectura cliente-servidor que emplea la aplicación ha permitido enforcarse primero en las funcionalidades del *backend* sin tener que preocuparse de cómo iban a utilizarse desde el frontal de la aplicación. El desarrollo por capas también ha permitido realizar cambios rapidamente en una capa y sin que afectase a otras.

Respecto a la tecnología que emplea *trabenet*, con la librería React el desarrollo de nueva páginas de la aplicación es un proceso relativamente rápido. En la tabla de iteraciones y sus estimaciones de tiempo (ver 9.1), se puede ver que aquellas relacionadas con el frontal de la aplicación emplearon un 31% del tiempo total del proyecto. Las consultas desarrolladas con

GraphQL también permiten que si se necesita un nuevo campo de información tan solo haya que definirlo en la propia *query* evitando desarrollar nuevos servicios o funciones.

10.2 Futuras líneas de trabajo

Parte de las líneas de trabajo futuras han de estar enfocadas en completar las funcionalidades que no se han podido desarrollar durante el transcurso de este proyecto: sistema de navegación entre directorios, comparación de archivos y la descarga y previsualización del documento con un formato corporativo.

Otras líneas de trabajo pueden ser las siguientes:

- Desarrollo de sistema de notificaciones y su integración con la herramienta Slack en caso de nuevos documentos, nuevas ediciones o versiones.
- Envío y compartición de documentación integrado en la aplicación, es decir, un nuevo servicio que permita enviar documentos a otros usuarios directamente desde la aplicación evitando el uso de servicios de correo electrónico externos.
- Sistema de comentarios: los documentos podrían tener una sección de comentarios para que los usuarios compartiesen sus opiniones o aportasen algo nuevo al documento.

Lista de acrónimos

API Application Programming Interface. 3, 6, 7, 19

CLI *Command-line Interface.* 1

CRUD Create, Read, Update and Delete. 20, 24, 34–36

ECTS European Credit Transfer and Accumulation System. 11

HTTP Hypertext Transfer Protocol. 7

IDE Integrated Development Environment. 3, 13

JS Javascript. 3, 5, 6, 11, 18, 20

JWT *JSON Web Token.* 19, 20

Node Node.js. 11

ORM Object-relational Mapping. 6, 27

PDF Portable Document Format. 1

REST Representational state transfer. 3

SHA *Secure Hash Algorithm.* 31–33

SQL Structured Query Language. 35

SSH Secure Shell Protocol. 6, 29–31

TFG Trabajo de Fin de Grado. 11

UI *User Interface*. 1, 2, 13

URL *Uniform Resource Locator*. 30

YAML *YAML Ain't Markup Language*. 27

Glosario

bug Un bug es un error en un programa que produce un resultado inesperado en la ejecución del mismo. 4

commit Un commit es una imagen concreta del estado de un repositorio git, es decir, un objeto que posee el estado de un conjunto de archivos y el autor que ha realizado modificaciones sobre esos archivos entre otros atributos. 20, 30–33

host Host se refiere al término de un anfitrión o administrador. En informática se utiliza en multitud de ámbitos: comunicaciones, videojuegos, redes, etc.. 5

markdown Lenguaje utilizado para la creación de texto formateado utilizando un editor de texto. Fue creado en 2004 para obtener un texto que fuese sencillo de entender para las personas en su formato original. 1, 13, 20

push Push se refiere a la acción de Git con la que se transfiere el contenido de un repositorio local a uno remoto. 5, 20, 30, 31, 33

scripts Un script es una series de comandos que son ejecutados para producir un resultado. 18

tags Un tag es un objeto de git que hace referencia a un objeto commit y sirve para crear una versión específica del código. 20

Bibliografía

[1] Legito, "Smart document workspace," 2021. [En línea]. Disponible en: https://www.legito.com/US/en

[2] MozillaFoundation, "Javascript," 2021. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[3] GraphQL, "Introduction to graphql," 2021. [En línea]. Disponible en: https://graphql.org/learn/

[4] Slack, "Making development an enjoyable experience," 2021. [En línea]. Disponible en: https://www.jetbrains.com/idea/features/

[5] W3Counter, "Browser platform market share," 2021. [En línea]. Disponible en: https://www.w3counter.com/globalstats.php

[6] Slack, "¿qué es slack?" 2021. [En línea]. Disponible en: https://slack.com/intl/es-es/help/articles/115004071768-%C2%BFQu%C3%A9-es-Slack-

[7] Git, "git –distributed-is-the-new-centralized," 2021. [En línea]. Disponible en: https://git-scm.com/

[8] Atlassian, "Trello," 2021. [En línea]. Disponible en: https://trello.com/en

[9] I. Docker, "Docker," 2021. [En línea]. Disponible en: https://docs.docker.com/get-started/overview/

[10] GitHub, "Github," 2021. [En línea]. Disponible en: https://github.com/about

[11] ——, "Github users," 2021. [En línea]. Disponible en: https://github.com/search?q=type:user&type=Users

[12] Z. Inc., "Zoom, we deliver happiness," 2021. [En línea]. Disponible en: https://zoom.us/about

[13] OpenJSFoundation, "About nodejs," 2021. [En línea]. Disponible en: https://nodejs.org/en/about/

[14] NodeGit, "Asynchronous native node bindings to libgit2," 2019. [En línea]. Disponible en: https://www.nodegit.org/

[15] Jest, "Jestjs," 2021. [En línea]. Disponible en: https://jestjs.io/

[16] Gitolite, "Hosting git repositories," 2021. [En línea]. Disponible en: https://gitolite.com/gitolite/

[17] React, "React, a javascript library for building user interfaces," 2021. [En línea]. Disponible en: https://reactjs.org/

[18] Prisma, "Next-generation node.js and typescript orm," 2021. [En línea]. Disponible en: https://www.prisma.io/

[19] Prettier, "Prettier, opinionated code formatter," 2021. [En línea]. Disponible en: https://www.npmjs.com/package/prettier

[20] Yarn, "Yarn, safe, stable, reproducible projects," 2021. [En línea]. Disponible en: https://yarnpkg.com/

[21] Apollo, "Introduction to apollo client," 2021. [En línea]. Disponible en: https://www.apollographql.com/docs/react//

[22] ——, "Introduction to apollo server," 2021. [En línea]. Disponible en: https://www.apollographql.com/docs/apollo-server/

[23] Koa, "Koa, next generation web framework for node.js," 2021. [En línea]. Disponible en: https://koajs.com/

[24] PostgreSQL-Global-Development-Group, "Postgresql," 2021. [En línea]. Disponible en: https://www.postgresql.org/about/

[25] UCA, "Metodologías de desarrollo de software," 2015. [En línea]. Disponible en: https://repositorio.uca.edu.ar/bitstream/123456789/522/1/metodologias-desarrollo-software.pdf

[26] U. o. W. School of Computer Science, "Client-server architecture," 2014. [En línea]. Disponible en: https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch_Design_Activity/ClientServer.pdf

[27] RedHat, "Event driven model, software architecture," 2021. [En línea]. Disponible en: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture

[28] Codecademy, “What is crud?” 2020. [En línea]. Disponible en: https://www.codecademy.com/articles/what-is-crud

[29] Múltiples, “react-markdown,” 2021. [En línea]. Disponible en: https://github.com/remarkjs/react-markdown

[30] React, “react-router-dom,” 2021. [En línea]. Disponible en: https://reactrouter.com/web/guides/quick-start

[31] A. Mouat, “Understanding volumes in docker,” 2014. [En línea]. Disponible en: https://blog.container-solutions.com/understanding-volumes-docker