Facultad de Ciencias

Desarrollo de una **plataforma para la gestión de tareas y hábitos dentro del estudiantado**

(Development of a platform for task management and habits for students)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Raúl Ruiz Puente

Director\es: Pablo Sánchez Barreiro

Junio – 2023

Contenido

Resumen

El objetivo de este Trabajo de Fin de Máster es desarrollar una plataforma web que permita a los estudiantes organizar y gestionar una serie de tareas de manera que éstos puedan satisfacer unos determinados objetivos y adquirir unos determinados hábitos. Se pondrá especial interés en la gestión de tareas, eventos y hábitos necesarios para superar diversas asignaturas, objetivo principal y común de la mayoría de los estudiantes.

Dentro de la plataforma, cada estudiante dispondrá de un espacio propio donde definir los objetivos que quiere alcanzar. Estos objetivos pueden requerir de la adquisición de diversos hábitos, de la consecución de ciertos hitos y de la realización de una determinada tarea. Para facilitar la gestión de estos elementos, cada usuario dispondrá de: (1) un espacio para organizar notas personales; (2) otro espacio para almacenar recursos que puedan ser necesarios para desarrollar determinadas tareas; y, (3) un calendario donde poder visualizar los eventos, tareas e hitos que debe ir completando.

El desarrollo de la aplicación se realizará mediante una metodología de desarrollo ágil centrada en *Test-Driven Development*. La aplicación se implementará sobre React, Spring y se analizará la conveniencia de utilizar bases de datos NoSQL, como MongoDB.

Por último, me gustaría seguir a lo largo del proyecto una serie de buenas prácticas a la hora de desarrollar y programar que creo que son fundamentales. Así como en el propio diseño de la arquitectura del sistema.

A lo largo del trabajo iremos abordando cada parte del desarrollo y de los componentes que componen la aplicación, repasando los retos, las decisiones, nuestras necesidades y como se ha decidido desarrollar.

Palabras claves: Desarrollo de una plataforma web, Gestión de tareas del estudiantado, React, Javascript, Despliegue automático

Abstract

The objective of this Master's is to develop a web platform that allows students to organize and manage a series of tasks so that they can meet certain objectives and acquire certain habits. Special interest will be placed on the management of tasks, events and habits necessary to pass various subjects, which is the main and common objective of most students.

Within the platform, each student will have his or her own space where he or she can define the objectives he or she wants to achieve. These objectives may require the acquisition of various habits, the achievement of certain milestones and the completion of a certain task. To facilitate the management of these elements, each user will have: (1) a space to organize personal notes; (2) another space to store resources that may be necessary to develop certain tasks; and (3) a calendar where he/she can visualize the events, tasks and milestones to be completed.

The development of the application will be carried out using an agile development methodology focused on Test-Driven Development. The application will be implemented on React, Spring and the convenience of using NoSQL databases, such as MongoDB, will be analyzed.

Finally, I would like to follow throughout the project a series of good practices when developing and programming that I think are fundamental. As well as in the design of the system architecture itself.

Throughout the work we will be addressing each part of the development and the components that make up the application, reviewing the challenges, decisions, our needs and how it has been decided to develop.

Keywords: Web platform development, Student assignment management, React, Javascript, Automated Deployment

1. Introducción, Objetivos y Planificación

1.1 Introducción

A lo largo del trabajo de fin de máster descrito, se va a desarrollar una aplicación Web utilizando tecnologías actuales, a pesar de ser un área tecnológica donde hay una constante aparición de nuevas herramientas y aplicaciones. Se van a exponer las distintas fases del desarrollo, además del porqué de las decisiones técnicas y visuales de las partes.

La aplicación web va a consistir en una plataforma que apoye el logro de los objetivos que el usuario se marque. Actuará como una agenda personal, la cual, gracias a sus herramientas, que se complementarán entre ellas, ayudará a los usuarios a mantenerse al día y conseguir sus objetivos personales que se propongan. La aplicación cuenta con una sección para dar de alta los objetivos propuestos, a través de este objetivo contamos con un calendario de fácil uso para organizar sus tareas o eventos personalizados, un apartado de notas para personales para apuntar consejos, tareas o frases motivacionales, un repositorio de links para tener al alcance recursos externos y por ultimo un contador de tiempo diario personalizado, donde podemos hacer un seguimientos del tiempo que dedicaremos a diferentes ejercicios para determinados días de la semana con el fin de lograr nuestro objetivo propuesto.

1.2 Objetivos

- Facilitar la organización personal: La aplicación debe permitir a los usuarios organizar sus objetivos personales de manera eficiente y mantener un registro claro de sus tareas y eventos.
- Ayudar a los usuarios a mantenerse al día: La aplicación debe proporcionar un lugar o vista para mostrar los recordatorios y alertas.
- Apoyar la consecución de objetivos: La aplicación debe proporcionar herramientas y funciones que ayuden a los usuarios a alcanzar sus objetivos personales, como la gestión del tiempo y la planificación de actividades.
- Proporcionar un calendario fácil de usar: La aplicación debe incluir un calendario intuitivo y de fácil manejo que permita a los usuarios programar sus tareas y eventos de manera sencilla.
- Ofrecer un apartado de notas personales: La aplicación debe contar con un espacio donde los usuarios puedan tomar notas personales, guardar consejos, tareas pendientes o frases motivacionales relacionadas con sus objetivos.
- Repositorio de enlaces útiles: La aplicación debe permitir a los usuarios almacenar enlaces a recursos externos relevantes que puedan ayudarles a alcanzar sus objetivos, proporcionando un acceso rápido y fácil a dichos recursos.

- Contador de tiempo personalizado: La aplicación debe incluir un contador de tiempo que permita a los usuarios realizar un seguimiento del tiempo dedicado a diferentes ejercicios o actividades específicas en función de sus objetivos establecidos.
- Privacidad y seguridad de los datos: La aplicación debe garantizar la protección de la privacidad de los usuarios y la seguridad de sus datos personales, implementando medidas de seguridad adecuadas.
- Interfaz intuitiva y atractiva: La aplicación debe contar con una interfaz de usuario intuitiva y atractiva que facilite la navegación y el uso de todas las funcionalidades de manera amigable y agradable.

1.3 Planificación

El desarrollo de la aplicación fue iniciado con cero conocimientos de las tecnologías empleadas, es por eso por lo que en un primer lugar empecé desarrollando muchas funcionalidades con el único propósito principal de aprender y teniendo la aplicación en un segundo plano. Cuando el proyecto ya estaba bastante avanzado solicite la ayuda y dirección de este proyecto y consecuentemente ya era tarde para implementar una metodología.

Pero si puedo comentar un poco las fases aproximadas por las que ha atravesado el proyecto, si tuviéramos que declarar los tiempos de desarrollo, podríamos comentar que:

- Octubre 2022 a enero 2023: Fase de aprendizaje y desarrollo pruebas de concepto donde se prueba que las tecnologías y herramientas utilizadas son válidas para el conjunto del proyecto que se quiere abordar.
- Enero 2023: Se definen los requisitos de la aplicación.
- Enero 2023 a febrero 2023: Se desarrollan las funcionalidades y pruebas fijándose en los requisitos de la aplicación definidos.
- Marzo 2023: Se enseña y revisa la aplicación con el tutor del proyecto, fijándonos en los requisitos propuestos. se proponen nuevas mejoras y cambios funcionales en la aplicación.
- Abril a junio 2023: Se realizan las diferentes mejoras propuestas y se concluye el proyecto.

No ha habido una metodología definida expresamente, ni un sistema de gestión de tareas donde estas sean estimadas o descritas. Principalmente debido a, como he comentado, empezar por mi cuenta investigando y aprendiendo y luego reconvertir el trabajo en este proyecto. Si empezase el proyecto de nuevo, sin duda utilizaría un sistema de gestión de las diferentes tareas fijándome en las metodologías ágiles, que son las que utilizo día a día en mi faceta profesional, definiendo cada tarea correctamente, especificando qué hay que hacer, estimando las duraciones y enlazándola a cada requisito.

Por otro lado, quiero destacar que para las pruebas y desarrollo de los distintos requisitos se ha explorado la opción *Test-Driven Development* [1] (Desarrollo guiado por pruebas de software), sobre todo en la parte del back-end. Esta técnica de desarrollo consiste en primero crear una prueba, generar el código necesario para que la prueba sea correcta y una vez pasada, refactorizar y limpiar el código. Después de estos tres pasos se volvería a crear una nueva prueba y repetiríamos el ciclo hasta completar el desarrollo de un requisito. Esta forma de desarrollo es bastante útil ya que nos permite desarrollar con la confianza de que se están superando una serie de pruebas y una vez acabados el desarrollo, ya disponemos de estas pruebas automáticos que nos servirán para probar la aplicación siempre que se realice algún cambio en el código o refactorización.

Por último, en cuanto a la gestión de la configuración, a pesar de ser el único desarrollador de la aplicación, desde el primer momento se ha trabajado con un controlador de versiones, en este caso Git, utilizando distintas ramas dependiendo de la fase del desarrollo en la que nos encontramos. En los repositorios existen una rama principal donde se introducen las funcionalidades ya implementadas y probadas y las ramas se van creando en demanda por las funciones o requisitos que se van desarrollando para poder manejarlos de forma independiente. También ha sido creado un repositorio común de archivos en la nube, para guardar los distintos requisitos/documentos necesarios para la aplicación web.

En próximas fases el trabajo explicaré como se ha llevado a cabo el despliegue automático de la aplicación. De esta forma es muy fácil controlar que las nuevas funcionalidades que son subidas a la rama son correctas, no provocan errores y mantienen la calidad definida.

[1] Test-Driven development

2. Requisitos

2.1 Requisitos Funcionales

La aplicación creada es ficticia, por lo que se carece de fuentes de las cuales extraer los requisitos. Para poder tener unos requisitos de los cuales partir, entre mi director de este trabajo y yo consensuamos una serie de requisitos mínimos que la aplicación debe soportar.

Para la especificación de requisitos usamos la técnica de historias de usuarios[2]. Esta técnica consiste en especificar los requisitos a lo largo de la vida del desarrollo del producto, cambiando y adaptando los ya existentes, ya que plantear todo al inicio es imposible e improductivo. Estos requisitos van ligados fuertemente con pruebas automatizadas que sirve para corroborar el buen estado de la aplicación a lo largo del desarrollo.

Las historias de usuario definen funcionalidades atómicas de alguna parte del sistema, Las historias de usuario identificadas son:

01 - Notas

01.01 - Crear Nota
01.02 - Modificar descripción e importancia de una nota
01.03 - Eliminar nota
01.04 - Buscar/Filtrar nota

02 - Eventos de calendario

02.01 - Crear eventos en el calendario
02.02 - Modificar evento
02.03 - Eliminar evento
02.04 - Buscar/Filtrar evento

03 - Habito

03.01 - Crear habito
03.02 - Eliminar habito
03.03 - Control del habito
03.04 - Buscar/Filtrar evento

04 - Objetivo

04.01 - Crear objetivo
04.02 - Modificar objetivo
04.03 - Crear recurso
04.04 - Eliminar recursos de objetivo

[2] Historia de usuarios

La anterior lista de requisitos son los descritos a través de historias de usuarios para las distintas funcionalidades de nuestra aplicación. En la tabla 1 vemos a modo de ejemplo la historia de usuario que crear un evento en el calendario.

ID	02.01
Titulo	Crear eventos en el calendario
Descripción	Yo, como usuario, puedo crear un evento en el calendario, de manera que pueda organizar todos los eventos del mes, semana y día en el calendario personal
Pruebas de aceptación	02.01.01- Crear evento • El usuario accede al calendario a través del menú lateral • El usuario selecciona el periodo del evento deseando, puede ser de horas, un día o varios días • Al seleccionar el periodo aparecerá un formulario • El usuario debe rellenar el nombre del evento y si lo desea asignarlo a un objetivo • Al crear un evento se valida que aparece correctamente en el calendario 02.01.02- Rellenar el formulario del evento forma errónea • El usuario no rellena correctamente el formulario de creación • Se valida que no se ha creado un evento de forma errónea 02.01.03 - Fallo sin conexión y BBDD • Si se produce un fallo en la conexión o en BBDD se notifica al usuario y no se ejecuta ninguna acción

Tabla 1. Historia de usuario, creación de eventos

Las descripciones siguen una estructura común: "Yo, <como rol>, quiero <funcionalidad>, de manera que <objetivo>". La descripción es una forma muy natural de contar lo que queremos conseguir con el requisito, nos aporta información de cómo, quien y por qué. En este caso, describimos que un usuario puede crear eventos en el calendario y expresa el motivo de la creación de este requisito, que es que el usuario pueda organizar los eventos próximos en el calendario de la aplicación.

Por último, describimos las pruebas de aceptación necesarias para que el usuario acepte la completitud del requisito. Utilizamos un lenguaje coloquial entendible tanto para el usuario como para el desarrollador. En este caso contemplamos la propia creación del evento y describimos el proceso que tiene que tomar el usuario. La segunda prueba, contempla el caso en el que se rellenan el formulario de creación de forma errónea y la aplicación captura correctamente el problema y lo notifica al usuario. Y, por último, también contempla el caso en donde se produzca un error debido a la conexión, conexión del servidor o problemas con la conexión de la BBDD, también deberá ser correctamente notificado al usuario.

2.2 Requisitos no funcionales

En cuanto a la especificación de requisitos no funcionales principalmente nos podemos fijar en la ISO 25010, modelo de calidad que forma la piedra angular en torno a la cual se establece el sistema para la evaluación de la calidad de los productos.

Algunos de los requisitos son obvios, como el de rendimiento de la aplicación o utilización mínima de recursos, por lo que nos los saltaremos. No obstante, al ser una aplicación donde existe el usuario como entidad podemos definir los siguientes recursos no funcionales.

ID	01
Titulo	Control del acceso
Descripción	Existe un control de acceso para la información personal de cada usuario, así como a sus herramientas e información.

Tabla 2. Requisito no funcional, control del acceso

ID	02
Titulo	Autenticación de usuario
Descripción	La autenticación de los usuarios se realiza a través de una contraseña elegida en el registro del usuario y que permite acceder a la aplicación.

Tabla 3. Requisito no funcional, autenticación de usuario

ID	03
Titulo	Uso de JWT para la sesión de los usuarios
Descripción	Los usuarios, al iniciar sesión, recibirán un Web Token JSON, que será utilizado para los consecuentes peticiones a los servicios de la web.

Tabla 4 Requisito no funcional, JWT

En este requisito detallamos el control de acceso a los datos, la autenticación y la seguridad que utilizará la plataforma para proteger las peticiones al back-end de la aplicación. Utilizar los JWT es una forma sencilla y segura de realizar esta tarea, además nos permite guardando el token en el explorador local de cada usuario, mantener la sesión del usuario iniciado durante un tiempo, esto amenizará el uso diario de la aplicación.

3. Arquitectura y diseño

3.1 Arquitectura de la aplicación

La arquitectura de la aplicación va a utilizar una aproximación tradicional en tres capas. En la ilustración 1 podemos ver la arquitectura completa.

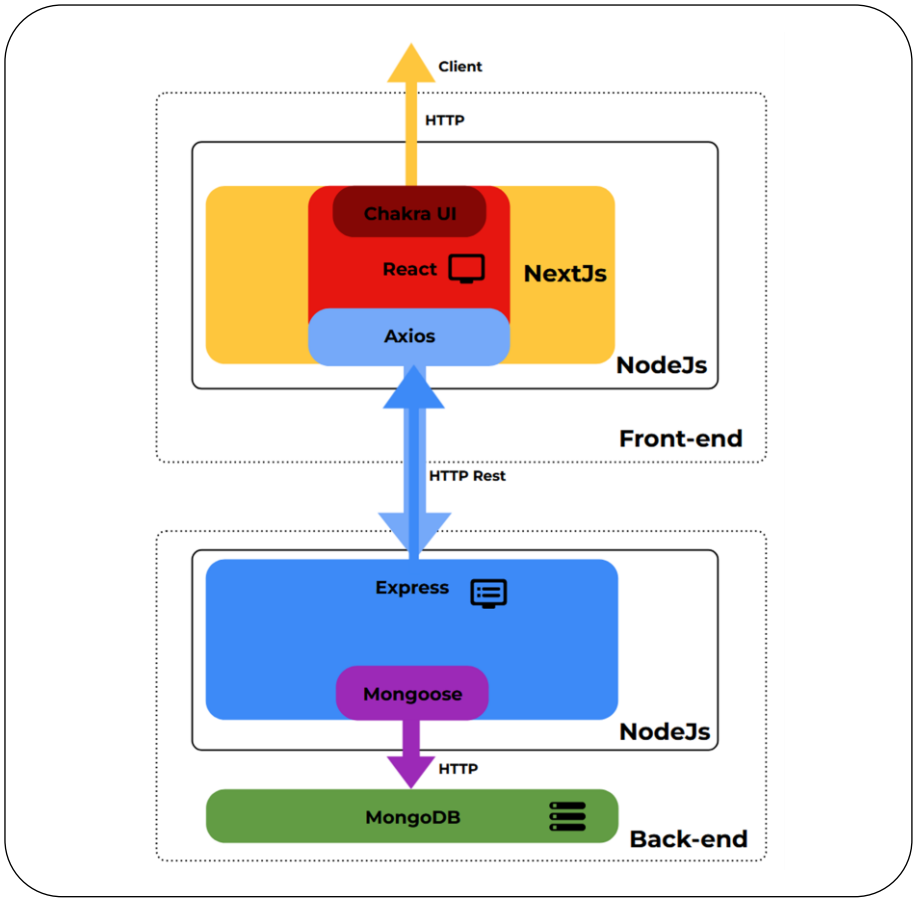


Ilustración 1. Arquitectura completa de la aplicación

El front-end de la aplicación se ejecutará sobre el framework de Next.js[3]. Este framework nos proporcionará una serie de herramientas y utilidades que enriquecen tanto la experiencia de desarrollo como la del usuario final. Nos va a solucionar una serie de problemas como el enrutamiento de páginas, las optimizaciones de imágenes y texto, y el soporte de CSS. También nos ayudará a la hora de desplegar nuestra aplicación de forma automática en Vercel[4].

Sobre Next.js utilizaremos React[5], una biblioteca JavaScript que nos permite crear nuestra aplicación web de una forma sencilla describiendo vistas y componentes simples para

[3] Next.js
[4] Vercel
[5] React

cada estado de la aplicación. React se encargará de actualizar y renderizar los componentes automáticamente cuando los datos cambien. Haremos uso de la librería Chakra UI[6], la cual nos aportará una serie componentes y plantillas los cuales utilizaremos para dar la forma necesaria a la aplicación y cumplir nuestros requisitos. Lo que nos aporta esta librería en contra posición a crear nuestros propios componentes desde cero, es una base de componentes funcionales con características y propiedades personalizables que nos facilitaran el desarrollo, dando una apariencia y experiencia de usuario más profesional. Además, los componentes tienen la flexibilidad para adaptarse a nuestro diseño y necesidades.

Para la comunicación entre la parte front-end y back-end me he decantado por una API Rest, aunque destacar que en las pruebas de concepto se han probado alternativas como GraphQL[7], el cual es un lenguaje de consulta de datos, que aporta algunas ventajas como un mayor control de la información que se recupera del servidor. Esto dota de rapidez y menos carga de trabajo para la aplicación que se nota sobre todo en dispositivos móviles.

Finalmente se decide usar una API Rest por el mayor soporte, facilidades, librerías y opciones que existen, además que es ampliamente usada, probada y más conocida a la hora de desarrollar. La API Rest será implementada utilizando Express[8] en JavaScript sobre Node.js[9].

Referente a la capa de persistencia de datos, es decir la base de datos, se opta por MongoDB[10], una base de datos no relacional. Se decide utilizar una base de datos no relacional frente a una relacional por: la facilidad de escalar la infraestructura necesaria, siendo un escalado horizontal, la flexibilidad a la hora de modelar nuestros esquemas a lo largo del tiempo, la optimización para tener un mayor rendimiento con una mayor velocidad de lectura/escritura y por la alta disponibilidad de los datos para un gran volumen de información. Las ventajas que nos aportan usar este tipo de base de datos no relacionales probablemente no lleguen a ser aprovechadas en este proyecto, principalmente debido a la pequeña cantidad de información que se va a manejar, pero creo que sería una buena decisión para el futuro con vistas al posible crecimiento y escalabilidad que se podría sufrir, además de la visión de las funcionalidades futuras que pueden ser implementadas.

Entre nuestra capa de servicios Rest y la base de datos MongoDB, se hará uso de una biblioteca llamada Mongoose[11]. Esta biblioteca es un puente objeto documental que nos ayudará a la hora interactuar en las dos direcciones, lectura y escritura, con la base de datos, facilitando las peticiones y el poblamiento de los datos entre entidades relacionadas

[6] Chakra UI
[7] GraphQL
[8] Express
[9] Node.Js
[10] MongoDB
[11] Mongoose

cuando sea necesario. Además, Mongoose nos permite modelar esquemas para nuestras entidades en la base de datos, lo que nos aporta la facilidad de asegurarnos la consistencia y correctitud de los datos.

Por último, en la ilustración 2, se presenta el modelo de dominio diseñado para nuestra aplicación. Principalmente se ha buscado la simplicidad de los objetos y minimizar las relaciones entre entidades.

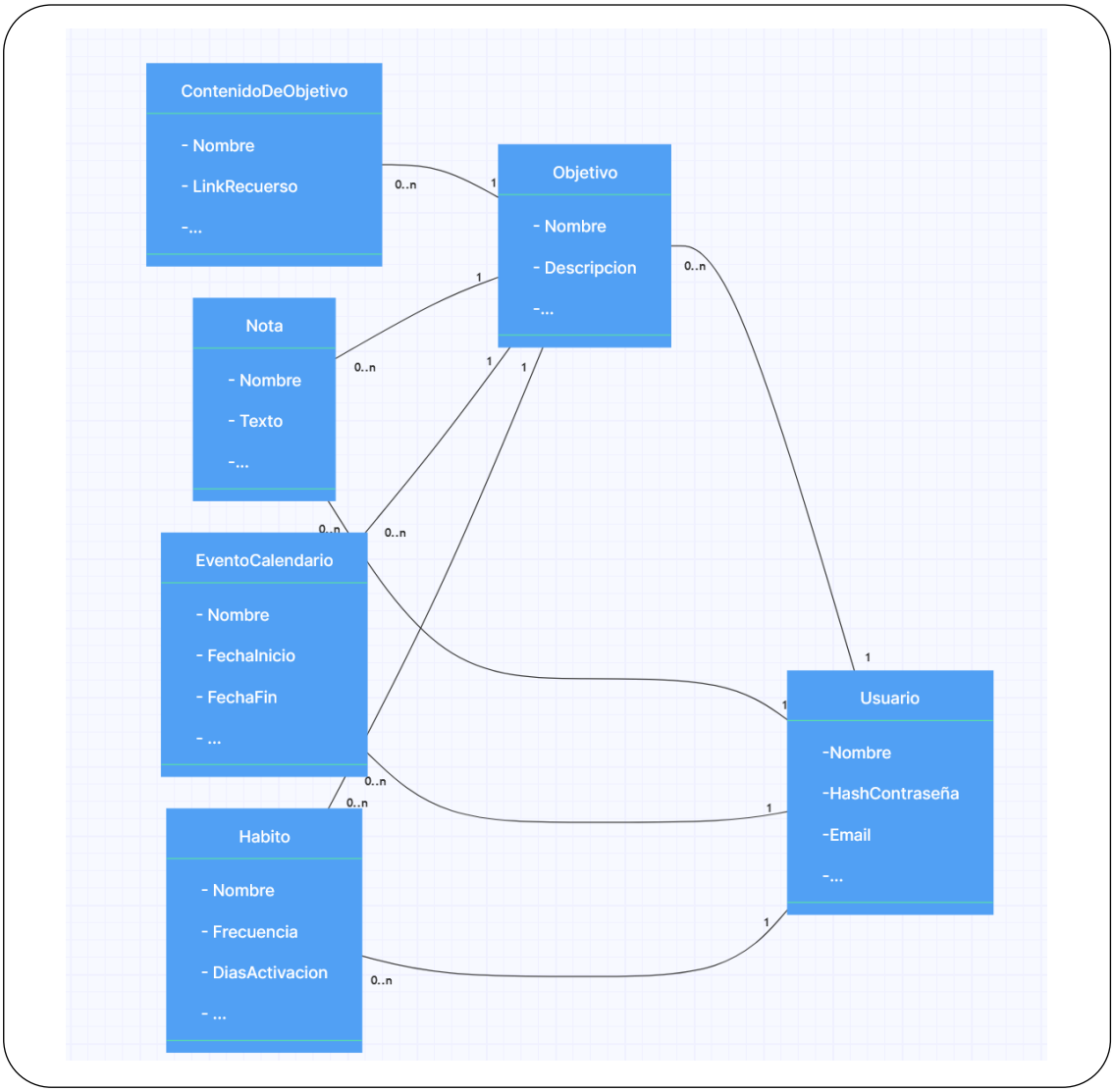


Ilustración 2. Modelo de dominio de la aplicación

3.2 Diseño de la API Rest

La API Rest está constituida por diferentes endpoints que utilizaremos desde el front-end de nuestra aplicación. El *endpoint* principal de la aplicación es el de */login*, el cual permite a los usuarios acceder a la aplicación. Al iniciar en la sesión, el servidor devuelve un token web (JWT), como mecanismo para asegurar la autenticación de los usuarios, el cual será guardado en el explorador de forma local y será necesario utilizarlo para hacer las consultas a los demás *endpoints* de la aplicación.

En segundo lugar, en importancia, tenemos una serie de *endpoints* para manejar la información de los usuarios, es decir, registro de nuevos usuarios, eliminación de usuarios y modificaciones de datos personales de cada usuario que guardaremos.

Para finalizar, tenemos los distintos *endpoints* de los objetos o entidades que utilizan las herramientas en la aplicación. Estos son: los objetivos, notas, hábitos, eventos del calendario y contenidos. Cada uno de estos *endpoints* está asociado a un usuario en concreto. En todos estos *endpoints* poseemos los métodos necesarios para crear, modificar, eliminar y obtener las entidades propias.

A continuación, a modo de ejemplo, se proporciona la descripción detallada de algunos *endpoint* que pertenecen a la entidad notas en la API Rest:

URL	/api/notes/{objectId}
Parámetros del header	JWT del usuario
Descripción	Devuelve todas las notas asignadas a un objetivo de un usuario
Operación	Get
Respuestas HTTP	TokenExpired/invalid(498), Error(404), Ok(200)
Retorno correcto	Lista JSON con las notas que pertenecen

Tabla 5. Endpoint de notas Get

URL	/api/notes
Parámetros del header	JWT del usuario
Descripción	Permite a un usuario crear una nueva nota
Operación	Post
Respuestas HTTP	TokenExpired/invalid(498), Bad Resquest(400), Error(404), Ok(200)
Retorno correcto	Objeto JSON de la nota creada

Tabla 6. Endpoint de notas Post

URL	/api/notes/{id}
Parámetros del header	JWT del usuario

Descripción	Permite a un usuario modificar una nota que corresponde con el Id
Operación	Put
Respuestas HTTP	TokenExpired/invalid(498), Bad Resquest(400), Error(404), Ok(200)
Retorno correcto	Objeto JSON de la nota modificada

Tabla 7. Endpoint de notas Put

URL	/api/notes/{id}
Parámetros del header	JWT del usuario
Descripción	Permite eliminar la nota que corresponde con el Id y el usuario
Operación	Delete
Respuestas HTTP	TokenExpired/invalid(498), Bad Resquest(400), Error(404), Ok(200)
Retorno correcto	Vacío

Tabla 8. Endpoint de notas Delete

A continuación, listo las direcciones de todos los endpoints existentes en la aplicación:

Log In

- [POST] /login
- [POST] /login/istokenvalid

Usuarios

- [GET] /users/{id}
- [POST] /users
- [PUT] /users/{id}

Objetivos

- [GET] /objective/{id}
- [POST] /objective
- [PUT] /objective/{id}
- [DELETE] /objective/{id}

Contenido de objetivos

- [GET] /objectivecontent/{id}
- [POST] /objectivecontent

- [PUT] /objectivecontent/{id}
- [DELETE] /objectivecontent/{id}

Notas

- [GET] /notes/{id}
- [POST] /notes
- [PUT] /notes/{id}
- [DELETE] /notes/{id}

Hábitos

- [GET] /habit/{id}
- [POST] /habit
- [PUT] /habit/{id}
- [DELETE] /habit/{id}

Eventos del calendario

- [GET] /calendarevent/{id}
- [POST] /calendarevent
- [PUT] /calendarevent/{id}
- [DELETE] /calendarevent/{id}

3.3 Diseño de la base de datos

La base de datos esta creada en MongoDB y encima de ella utilizamos la herramienta Mongoose. Esta herramienta nos permite definir esquemas que representaran nuestras entidades y constituyen el diseño lógico de nuestra base de datos.

Las colecciones que componen la base de datos son: usuarios, eventos del calendario, hábitos, notas, objetivos y contenido de los objetivos. Para la creación de las entidades se ha utilizado como referencia el modelo de dominio antes descrito.

El bloque de código 1 define el esquema de un objeto de usuario en la base de datos que está definido en código en la aplicación.

El proceso necesario para definir nuestro esquema es muy sencillo. Por cada propiedad existente en la clase, usuario del modelo de dominio, creamos un campo dentro del esquema y le damos un tipo de datos adecuados, conforme al tipo y que tuviese la propiedad en el modelo de dominio, además, en el caso del nombre de usuario le especificamos que ha de ser único en toda la colección, es decir no puede existir otro usuario con el mismo nombre de usuario.

En el caso de las referencias a otros objetos del modelo de dominio, representamos referencias como a raíz de referencias a otros documentos. Por ejemplo, en el caso de las notas, éstas se alojan en su propia colección y en el documento usuario, simplemente guardamos los identificadores de las notas asociadas a un determinado usuario. Cuando obtengamos un usuario de la base de datos estas listas son literalmente listas de identificadores y cuando sea necesario podremos poblar estas listas sustituyendo los identificadores por los objetos reales de la colección a la que haga referencia.

```
const userSchema = new Schema({
    username: {
      type: String,
      unique: true,
    },
    name: String,
    passwordHash: String,
    email: String,
    birthDate: Date,
    creationDate: Date,
    userPic: String,
    profilePic: String,
    language: String,
    location: String,
    notes: [
      {
        type: Schema.Types.ObjectId,
        ref: "Note",
      },
    ],
    objectives: [
      {
        type: Schema.Types.ObjectId,
        ref: "Objective",
      },
    ],
    communities: [
      {
        type: Schema.Types.ObjectId,
        ref: "Community",
      },
    ],
    calendarEvents: [
      {
        type: Schema.Types.ObjectId,
        ref: "CalendarEvent",
      },
    ],
    habits: [
      {
        type: Schema.Types.ObjectId,
        ref: "Habit",
      },
    ],
  });
```

Bloque de código 1. Esquema de usuario

3.4 Diseño y organización de la interfaz front-end

Para el desarrollo de la interfaz de usuario se ha utilizado React con Chakra UI. Chakra UI nos proporciona una seria de plantillas para poder generar nuestras interfaces, además un catálogo de componentes básicos y reutilizables.

Lo primero que ve un usuario al entrar a la página inicial del sitio web es un formulario que permite al usuario acceder a su cuenta o la posibilidad de registrarse en la plataforma.

Una vez que el usuario ha accedido a la plataforma existe un esquema general que se mantiene presente a lo largo de toda la aplicación. Este esquema consta de una barra lateral persistente la cual tiene un acceso directo a todas las opciones y herramientas disponibles. La barra lateral se implanta con la idea de que el usuario sepa en todo momento en donde está ubicado en la página y poder navegar por ella. Al tener la barra lateral siempre presente (barra roja, ilustración 3), disponemos de un espacio (fondo azul, ilustración 3) para poder ir implementando las diferentes herramientas de la aplicación.

Ilustración 3. Layout general

Las herramientas se diseñan con dos objetivos: que sean claras, simples y concisas y que sean adaptativas, es decir que se adapten automáticamente a la resolución del usuario. En el caso de la barra lateral es necesario cambiar su comportamiento cuando detecte que está en un dispositivo móvil o en una resolución muy pequeña. Si dejamos ver la barra completa en resoluciones pequeñas tapará el contenido principal de las herramientas, por lo que si se detecta una resolución pequeña la barra lateral desaparece y aparecerá un botón que desplegará la barra principal cuando se pulse.

A continuación, mostramos el aspecto visual de las diferentes interfaces de la aplicación, las ilustraciones son diseños de la aplicación utilizados en el desarrollo, pero no son el resultado visual final de la Web.

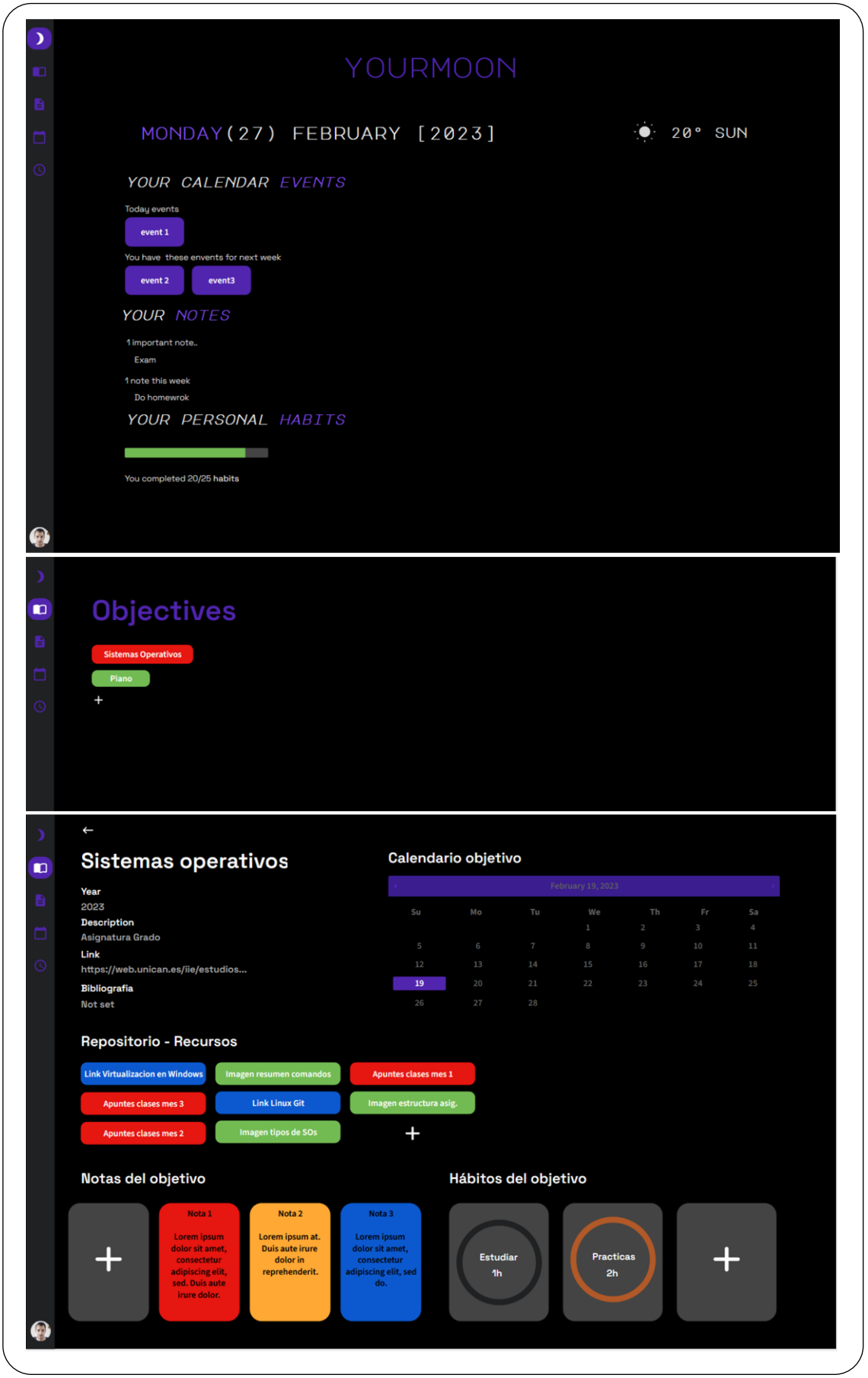


Ilustración 4. Diseño del inicio y vista de los objetivos

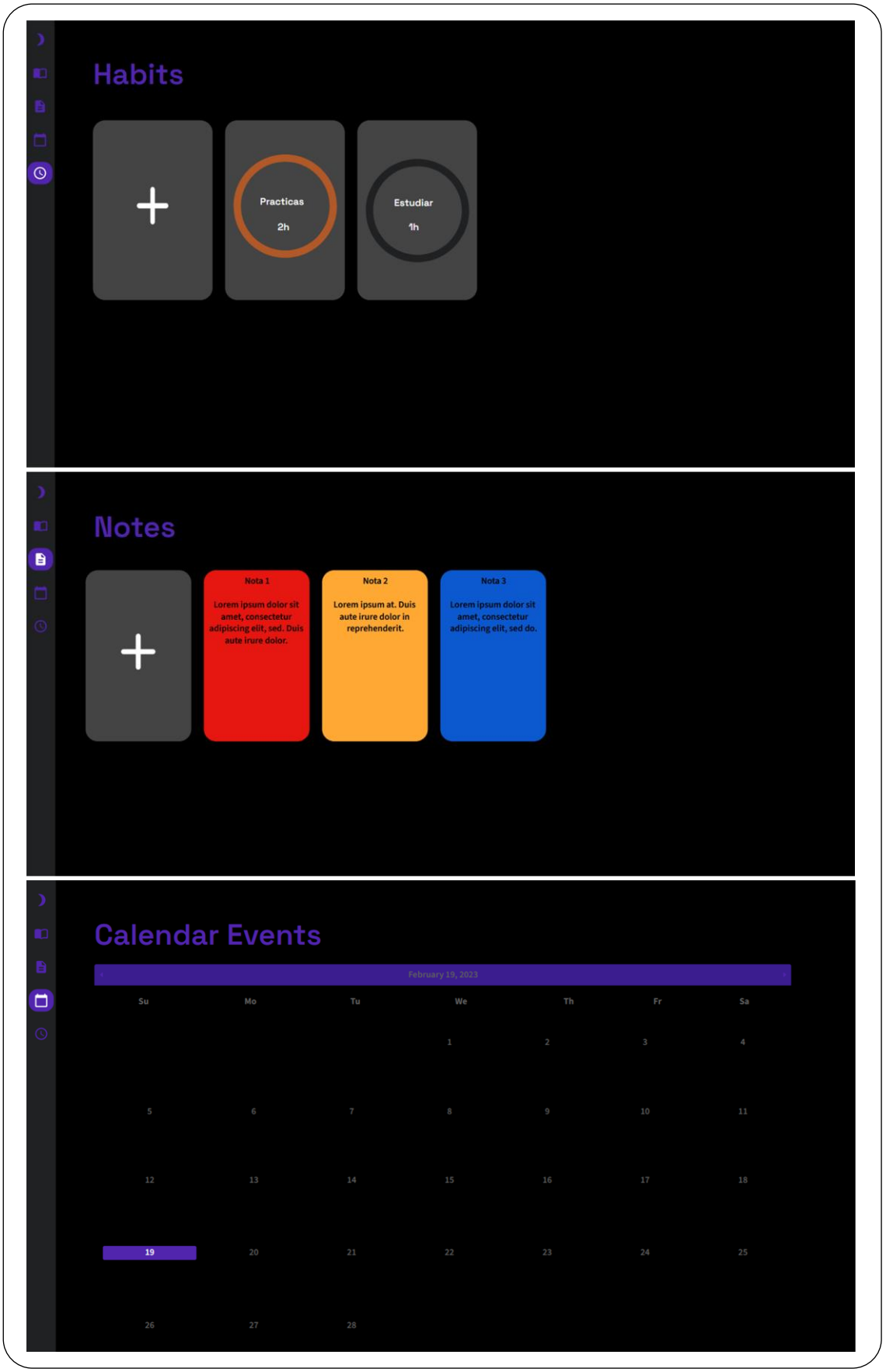


Ilustración 5. Diseño de los hábitos, notas y calendario

4. Implementación

4.1 Front-end

En este apartado vamos a entrar en profundidad en el funcionamiento y el uso de las librerías utilizadas para la parte visual de la aplicación.

4.1.1 Estructura del proyecto y navegación entre vistas

Una de las grandes ventajas de usar Next.js, es el enrutamiento automático de la web. En una aplicación con React sobre otro framework más simple, normalmente, es necesario hacer uso de alguna librería como *React router*[12] para ayudarnos a hacer el código para la navegación entre las distintas páginas (enrutamiento). Con Next.js no necesitamos escribir nada de código, simplemente necesitamos crear los diferentes archivos Javascript que contiene la vista de nuestra aplicación en la carpeta *./pages* del proyecto.

Esta carpeta es creada al iniciar el proyecto junto otros dos archivos, *_app.js* e *index.js*. *index.js* es la vista principal de la aplicación, la primera página que veremos al acceder a la URL de la web. *_app.js* define el punto de entrada de la aplicación web. A través de este componente se irán insertando las distintas páginas de nuestra plataforma. Por ejemplo, fijándonos en el bloque de código 2, nuestras paginas creadas como son el *home* o el *about*, dependiendo de la URL en la que nos encontremos, la etiqueta <*Component*/> será remplazada automáticamente por el contenido de *home.js* o de *about.js* creados en */pages*.

```
function MyApp({ Component, pageProps })
{
    return (
      <ChakraProvider theme={theme}>
        <Fonts />
        <UserProvider>
          <CommunitiesProvider>
            <NotesProvider>
              <HabitsProvider>
                <CalendarEventsProvider>
                  <SubjectsProvider>
                    <MainPageLayout>
                      <Component />
                    </MainPageLayout>
                  </SubjectsProvider>
                </CalendarEventsProvider>
              </HabitsProvider>
            </NotesProvider>
          </CommunitiesProvider>
        </UserProvider>
      </ChakraProvider>
    );
```

Bloque de código 2. _app.js

Este archivo, a su vez, nos permite envolver todo el contenido que se muestra en todas las páginas de la web en proveedores comunes, como los proveedores personalizados que crearemos, la librería Chakra UI, o en un layout general, en este caso, el que habíamos diseñado, <*MainPageLayout*> que contiene la barra lateral.

También creamos rutas dinámicas, es decir si por ejemplo creamos el archivo [id].js, bajo la carpeta *./note/[id].js*, nos enrutará la URL "*www.nuestraweb.es/note/52432"* y desde la vista ([id].js) podremos obtener ese *id* de la URL para obtener la información de la nota en concreto.

[12] React Router

4.1.2 *Estado en React (useState)*

Los estados son un concepto fundamental de las aplicaciones en React. Los estados son una forma de procesador o guardar los datos de un componente de forma dinámica. Lo que hace especial a los estados es que, cuando cambia el valor y este valor afecta a un componente, React se encargará de actualizar la página donde se muestra el componente a tiempo real.

```
1- const Contador = () => {
2-     const [count, setCount] = useState(0);
3-     return (
4-       <div>
5-         <h2>{count} likes</h2>
6-         <span onClick={() => setCount(count + 1)}>👍</span>
7-         <span onClick={() => setCount(count - 1)}>👎</span>
8-       </div>
9-     );
10-  };
```

Bloque de código 3. Ejemplo de useState

Un *hook* es una pieza de código que nos permite manejar los estados sin necesidad de escribir ninguna clase extra, podemos personalizarlos para extraer lógica de nuestros componentes y compartirlos. Como podemos ver en el bloque de código 3, el hook *useState* se crea usando la función *useState()* (línea 2), donde su primer parámetro es el valor inicial. Al crear el estado nos define dos variables (*count* y *setCount*). La primera variable posee el valor del estado y nos permite renderizar el valor directamente en la aplicación (línea 5). El segundo parámetro definido, no es un parámetro si no una función. Con esta función, por ejemplo, al llamarla desde un evento producido por un clic (línea 6 y 7), modificará el valor del estado. La gran ventaja se encuentra en que al actualizar el estado desde el botón, automáticamente React actualizará el objeto que se muestra en la vista, manteniéndolo actualizado en todo momento sin nosotros tener que hacer ninguna gestión extra.

Este es uno de los mayores potenciales de usar React, con este sencillo código podemos crear una página con componentes dinámicos, donde React mantendrá la página actualizada, refrescando la vista de la forma más optima posible.

4.1.3 Implementación de proveedores propios (useContext)

En una aplicación de React, para compartir información entre componentes, se utilizan los llamados *props*, que son parámetros estáticos que pueden ser especificados en la definición de los componentes. A medida que crece la aplicación, la comunicación a través de *props* se puede resultar bastante farragosa. Por ejemplo, al depender del token JWT para hacer peticiones de usuario, necesitaríamos estar todo el rato pasándolo a cada componente de la aplicación y si le vamos agregando más objetos como notas, se puede convertir en un código muy confuso.

Una forma de resolver el problema es crear hooks personalizados, estos hooks contienen estados globales, también llamados contextos. Este conjunto de estado global (contexto) y hook, se le llaman proveedor. Los proveedores, envolverán a nuestros componentes y gracias a ello podremos compartir los diferentes estados de una forma global y síncrona.

Crearemos diferentes contextos para cada entidad de la aplicación como la información del usuario, sus notas, los objetivos, etc. Gracias a los proveedores cuando un componente se muestre en la vista, no hará falta hacer ninguna petición al servidor, simplemente se suscribirá al proveedor correspondiente de forma local y obtendrá los datos. Además, estos proveedores pueden definir funciones extras para crear o eliminar los objetos del estado y de la base de datos. Esto consigue unificar las funcionalidades, evita la duplicidad del código y al ser un estado global, afectará a todos los componentes inicializados en la vista al mismo tiempo.

A continuación, veremos la implementación de un proveedor completo.

```
1- const NotesContext = createContext();
2-
3- export const NotesProvider = ({ children }) => {
4-   const [currentUser] = useContext(UserContext);
5-   const [userNotes, setUserNotes] = useState([]);
6-
7-   useEffect(() => {
8-     const gettingUserNotes = async () => {
9-       await getUserNotes(currentUser.token).then((notesList) => {
10-        if (notesList) {
11-          setUserNotes(notesList);
12-        }
13-      });
14-    };
15-    gettingUserNotes();
16-  }, [currentUser.token]);
17-
18-  return (
19-    <NotesContext.Provider value={[userNotes, setUserNotes]}>
20-      {children}
21-    </NotesContext.Provider>
22-  );
23-};

24-export default NotesContext;
```

Bloque de código 4. Proveedor de notas

Para crear un proveedor en primer lugar, en el bloque de código 4, se crea el contexto global para almacenar las notas (línea 5), utilizando el contexto del usuario (línea 4) haremos una única la petición Rest que pide las notas al back-end, esta petición se hace con el hook *useEffect* (línea 7 a 16). Este otro hook, ejecutará su código definido (línea 8 a 15) solo cuando haya una modificación en las variables que pasamos al segundo parámetro del hook (currentUser.token) (línea 16), es decir cuando este se rellene o se modifique. Finalmente devolvemos el proveedor para que cuando un componente se subscriba, pueda acceder a la información (línea 18 a 22).

```
1- export function UseNotes() {
2-     const [currentUser] = useContext(UserContext);
3-     const [userNotes, setUserNotes] = useContext(NotesContext);
4-
5-     const onAddNote = async (title, content, color, important, subjectId) => {
6-       try {
7-         const savedNote = await createUserNote(currentUser.token, { title, content,
8-         color, important, subjectId  });
9-        if (savedNote) {
10-          setUserNotes(userNotes.concat(savedNote));
11-        }
12-      } catch (error) {}
13-    };
14-
15-    const onDeleteNote = async (idNote) => {
  ...
17-    };
18-    const editContentHandler = async (idNote, content) => {
  ...
20-    };
21-    const editImportantHandler = async (idNote, important) => {
  ...
23-    };
24-   return { userNotes, onAddNote, onDeleteNote, editContentHandler, editImportantHandler
25-          };
26- }
```

Bloque de código 5. Custom hook de notas

Para poder interactuar con el proveedor y agregar funcionalidad extra, creamos un hook personalizado que será utilizado por los componentes. Como podemos ver en el bloque de código 5, el hook se suscribe al contexto de notas (línea 3) creado en el anterior bloque, recibe el estado y creamos los diferentes métodos para crear, borrar o editar notas (linea 5, 15, 18 y 21). Estos métodos manejarán el estado global además de realizar las llamadas necesarias al back-end. Como he comentado estos cambios afectan de forma global a la aplicación.

Una de las principales ventajas de usar estos proveedores con contexto global y no un estado único por cada componente, es que cuando se renderiza el componente en la vista, solo necesitamos suscribirnos al contexto para cargar la información, y no necesitamos realizar una petición por cada vez que se renderiza el componente en la Web. Como hemos visto en el proveedor, el hook *useEffect*, que es el encargado de hace la llamada al back-end, solo se ejecuta cuando se modifica el token de usuario, es decir, cuando el usuario inicia sesión, esto hace que la velocidad de la aplicación sea mucho mayor y sea mucho más ágil y responsiva.

4.1.4 Implementación de componentes

Un gran atractivo del uso de React es la reutilización de los componentes a lo largo de las diferentes vistas. En nuestra plataforma el usuario puede crear notas simples, pero existe la posibilidad de crearlas asignándolas un objetivo. Cuando entremos al directorio o la vista de un objetivo, el componente que muestra las notas es el mismo y a través de los parámetros de entrada podemos adaptarlo para las dos situaciones.

Vamos a coger la vista de las notas y analizarla en profundidad para entender cómo es el funcionamiento del código.

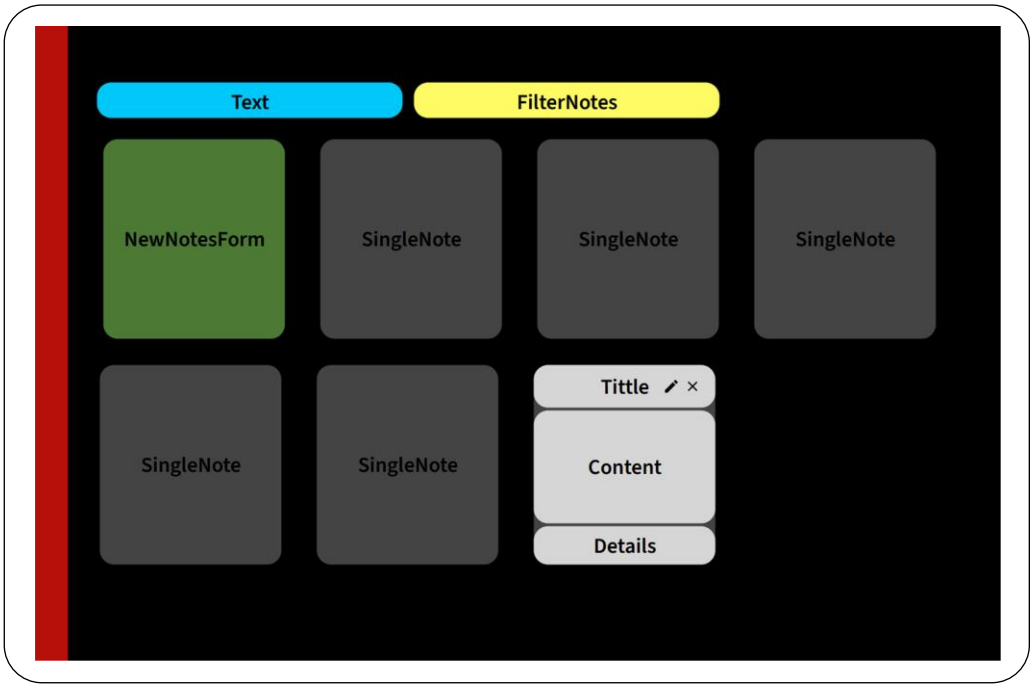


Ilustración 6. Esqueleto de la vista de notas

En la ilustración 6, podemos ver el esqueleto de la vista de las notas. Como podemos ver tenemos la barra lateral, la cual ya está implementada en la raíz de la aplicación. De forma efectiva el componente que vamos a realizar es todo el recuadro negro. En la primera fila encontraremos un componente de texto donde aparecerá el título de la sección (caja azul) y a continuación, un formulario para poder filtrar las notas (caja amarilla). En el hueco restante, tenemos una cuadricula donde se irán recuperando las notas creadas por el usuario, exceptuando la primera caja, que es un botón que se convierte en formulario para añadir notas nuevas. Por cada nota que se recupere se van insertando componentes *SingleNote* para representar cada nota y la estructura de este componente está compuesto por las cajas gris claras en la ilustración.

Entrando más en el codigo del componente, en primer lugar, definiremos la función que retorna el componente, dándole un nombre y luego permitiendo distintos parámetros de entrada. La definición la podemos ver en el bloque de código 6 a continuación y en la

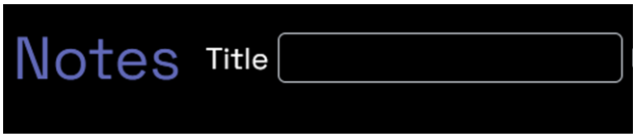
línea 1. Para el parámetro de entrada, en este componente, se acepta un *id* de un objetivo. Como hemos comentado, se utiliza este valor para cambiar el comportamiento del componente dependiendo desde donde sea accedido. Si al llamar a esta vista no especificamos este parámetro, tomará el valor por defecto *null*.

```
1- export default function UserNotes({ objectId = null }) {
2-     const {
3-       userNotes,
4-       onAddNote,
5-       onDeleteNote,
6-       editContentHandler,
7-       editImportantHandler,
8-     } = UseNotes();
9-
10-    const [titleFilter, setTitleFilter] = useState("");
```

Bloque de código 6. Inicio de la vista de notas

Este componente utilizará el hook personalizado que hemos creado para las notas. Al importar el hook, importamos el estado global que contiene todas las notas del usuario, además de las funciones para añadir notas nuevas, borrarlas o editarlas (línea 2 a 8). Por último, también definimos un estado en el componente (línea 10), que se utilizara para hacer el filtrado de las notas por su título. En este caso al ser un estado que carece de sentido fuera del componente, no es necesario crear un contexto global.

```
1- return (
2-   <Flex direction={"column"} width={"full"} gap={"10"}>
3-     <Flex>
4-       <Text textColor={"brand.purpura"} fontSize="5xl">
5-         Notes
6-       </Text>
7-       <FilterNotes
8-         setTitleFilter={setTitleFilter}
9-       />
10-    </Flex>
```



Bloque de código 7. Primera parte de la vista notas

La definición del componente continua en el bloque de código 7 y ya encontramos el retorno de los componentes de React que son renderizados en el navegador. El retorno lo podemos escribir utilizando sintaxis Javascript y podemos incluir tanto componentes React como directamente etiquetas HTML.

El componente al completo se envuelve en un Flex (línea 2 hasta el final de la definición del componente), que es un componente de Chakra, la librería gráfica que utilizamos, y viene a ser una caja HTML creada con las etiquetas <*div*> y es responsiva. Como todos

los componentes Chakra, podemos añadir distintos parámetros para personalizar su comportamiento.

Como hemos visto en el esqueleto, primero encontramos una fila con el nombre y un filtro, para ello, dentro de otra caja Flex (línea 3 a 10), renderizaremos el título de la sección (línea 4 a 6) utilizando otro componente de Chakra para escribir texto. Finalmente, agregamos el componente *FilterNotes* (línea 7 a 9), que es un componente que hemos creado, que contiene el formulario para filtrar las notas. A este componente le pasamos a través de los parámetros (props), la función que maneja el estado para filtrar las notas (titleFilter) y de esta forma poder recuperar la información y filtrar los resultados.

La parte final del componente que contiene la cuadricula de notas es la que aparece en el bloque de código 8.

```
1-  <SimpleGrid spacing={3} templateColumns="repeat(auto-fill, minmax(300px, 1fr))">
2-   <NewNotesForm subjectId={ objectId } onAddNote={onAddNote} />
3-   {userNotes == null ? (
4-    <OurSpinner />
5-   ) : (
6-    userNotes.filter((note) => titleFilter == "" || note.title.includes(titleFilter))
7-     .filter((note) => objectId == null || note.object?.id == objectId)
8-     .map((note, id) => {
9-         return (
10-          <SingleNote
11-            key={id}
12-            noteToShow={note}
13-            removeHandler={onDeleteNote}
14-            editContentHandler={editContentHandler}
15-            editImportantHandler={editImportantHandler}
16-          />
17-         );
18-     })
19-  )}
20- </SimpleGrid>
21-</Flex>
```

Bloque de código 8. Segunda parte de la vista de notas

Para conseguir una cuadrícula que contenga las notas y se vaya adaptando a la resolución de la pantalla, en primer lugar, añadimos el layout *SimpleGrid* de Chakra (línea 1 a 20), el cual, adaptará y colocará cada nota de forma ordenada en filas y columnas.

Dentro de la cuadrícula, en primer lugar, se añade un componente llamado *NewNotesForm* en la línea 2. Este componente permite añadir notas a través de un formulario.

El segundo componente se comprende dentro de una sentencia Javascript (línea 3 a 19). La sentencia entre llaves renderiza el primer o el segundo componente dependiendo de la condición *userNotes == null* (línea 3). Es decir, si el contexto global de notas del usuario es nulo renderizará un *Spinner* (Que sea nulo significa que no ha sido cargado desde el back-end, si el usuario no hubiera creado notas, estaría vacío).

Si ya se ha cargado la información de nuestra base de datos, filtramos esa lista de notas dependiendo del filtro de título (línea 6) que hemos creado en el bloque de código 7 y de si nos encontramos en la vista de un objetivo (línea 7), lo cual es conocido si el parámetro de entrada viene con valor.

La lista resultante después de pasar los filtros será iterada con la función *map()* (línea 8 a 18). Por cada nota que contiene la lista, iremos creando un componente *SingleNote*, a los cuales les pasaremos los parámetros necesarios para renderizar las notas correctamente. Como este componente internamente permite modificar o eliminar las notas, también le pasamos las funciones del hook propio para dicho cometido. El resultado final de la vista lo podemos ver en la ilustración 7.

Ilustración 7. Vista final de las notas

Por último, vamos a ver un ejemplo de cómo funcionan los formularios en la aplicacion. Para ello, usaremos una librería llamada *react-hook-form*[13] un hook personalizado. A continuación, veremos el formulario que es necesario para crear una nueva nota.

[13] React hook form

```
1- const {handleSubmit, register, formState: { errors, isSubmitting }, reset } = useForm();
2- ...
3- function onSubmit(values) {
4-  values.title
5- ...
6- }
7- ...
8- <form onSubmit={handleSubmit(onSubmit)}>
9-  <FormControl isInvalid={errors.content}>
10-    <Textarea
11-       id="content"
12-       placeholder="Note Content"
13-       color={"white"}
14-       {...register("content", {
15-         required: "Content is required",
16-         minLength: {
17-           value: 1,
18-           message: "Minimum length should be 1",
19-         },
20-       })}
21-     />
22-  <FormErrorMessage>
23-    {errors.content && errors.content.message}
24-  </FormErrorMessage>
25- </FormControl>
26- </form>
```

Bloque de código 9. Formulario para la creación de notas

Como es habitual para este tipo de herramientas, hemos de invocar el hook que nos proporciona la librería (línea 1). Como en el hook personalizado que hemos creado, un hook puede proporcionar varias funciones pero nos permite coger solo las que necesitamos, en este caso, utilizaremos las variables *handleSubmit*, el cual es la pieza que controla el flujo de respuesta cuando se acepta un formulario; *register*, el cual usaremos para añadir los campos del formulario; *formState*, que se usa para controlar los distintos estados de los campos y tener un apoyo visual a posibles errores y por ultimo *reset* que se llama para volver el formulario al estado iniciar.

En lo que resta de código podemos ver como se utilizan estas variables. En primer lugar, vemos la definición de la función que es llamada cuando se acepta el formulario (línea 3) y cómo podemos obtener los valores de los campos existentes en el formulario (línea 4). A la hora de retornar el componente React que contiene el formulario, es definido con la etiqueta HTML *form* (línea 8 a 26), iremos registrando cada campo. En este caso definimos un cuadro de texto (línea 10 a 21). Especificamos las validaciones que se van a aplicar al campo de texto (línea 14 a 20), en este caso un mínimo de longitud para el texto y que es obligatorio de rellenar. Por último, definimos un componente donde se renderizará el mensaje de error (línea 22 a 24) en caso de que no cumpla alguna de la validación descrita.

Podemos registrar todo tipo de inputs de usuario como casillas de verificación, fechas, archivos, etc. Y podemos validar de muchas formas incluso podemos crear funciones que llamen a servicios externos para comprobar si un valor introducido es correcto, en nuestra

aplicación se utiliza a la hora de crear un usuario para evitar duplicidad en los nombres de usuario.

4.1.5 Implementación de servicios y controlador Rest

La última pieza del front-end son los servicios que hacen las llamadas Rest al back-end de la aplicación. Estas llamadas se encapsulan a nivel de front-end en unos módulos llamados servicios, que no deben confundirse con los servicios web del back-end.

Los servicios actúan de puente entre los componentes y el servidor. Estos recibirán los parámetros necesarios, llamarán al servidor y normalizarán la respuesta que devuelven. Para su implementación hacemos uso de la librería Axios[14], la cual nos facilita la creación y envío de las peticiones HTTP (Get, Put, Post, Delete).

En el bloque de código 10, vemos un ejemplo de un servicio que obtiene las notas de un usuario. para ello simplemente se la pasa el token de usuario y se llamaría a la API.

```
export const getUserNotes = async (user) => {
    try {
      const notes = await getUserNotesAPI(
        user
      );
      return notes;
    } catch (err) {
      logService.captureError(err);
    }
    return null;
  };
```

Bloque de código 10. Ejemplo servicio

El controlador API, que vemos en el bloque de código 11, se encarga de generar la petición HTTP para comunicarse con el servidor, en este caso las notas del usuario y devolver la respuesta. Para ello escribe en la cabecera el token de usuario y la variable *baseUrl*, que es la dirección donde está alojando nuestro servidor back-end.

```
export async function getUserNotesAPI(userToken) {
    const config = {
      headers: {
        Authorization: `Bearer ${userToken}`,
      },
    };

    const { data } = await axios.get(baseUrl, config);
    return data;
```

Bloque de código 11. Ejemplo controlador API

[14] Axios

Las sentencias *async* y *await* son declaraciones que hacemos a nuestras funciones para declarar que la función es asíncrona. Cuando el hilo que ejecuta la función llega al *await*, este no bloquea los demás hilos, sino que simplemente espera a que se ejecute la petición liberando la carga de la CPU para otras tareas, como resultado, la aplicación Web no se verá bloqueada para el usuario final.

4.2 Back-end

4.2.1 API y base de datos

Aunque ya hemos visto cómo funciona el back-end, como está estructurado y como son los esquemas de la base de datos, vamos a comentar un poco cómo funciona el código de la API Rest. La API Rest esta creada con Express en Javascript. Express es un framework para el desarrollo de aplicaciones Web ligero y flexible que funciona sobre Node.js,

En primer lugar, el back-end se conecta con la base de datos. Para este proyecto estamos usando una base de datos MongoDB y la propia compañía nos ofrece un servicio de alojamiento de nuestra base de datos con replicación en la nube. Al ser un volumen de datos y operaciones pequeña, el servicio es gratuito. En el bloque de código 12 podemos ver el proceso para conectarnos utilizando una variable de entorno, también controlamos la desconexión en caso de error inesperado capturando las trazas de error.

```
const mongoose = require("mongoose");
const { MONGO_DB_URI, MONGO_DB_URI_TEST, NODE_ENV } = process.env;
const connectionString = NODE_ENV === "test" ? MONGO_DB_URI_TEST : MONGO_DB_URI;

mongoose
  .connect(connectionString)
  .then(() => {
    console.log("Database connected");
  })
  .catch((err) => {
    logService.captureError(err);
  });

process.on("uncaughtException", (exception) => {
  logService.captureError(exception);
  mongoose.disconnect();
});
```

Bloque de código 12. Conexión a la BBDD Mongo

En segundo lugar, definimos los controladores dedicados a las distintas entidades de nuestra aplicación que compondrán las peticiones HTTP que puede recibir nuestra aplicación. Para ello, en el archivo *index.js* del back-end, asignamos a cada URL la ruta del archivo que da soporte a dicha URL. Por ejemplo, para el caso de las notas, emparejamos la URL /api/notes con el archivo ./controladores/notes.js, el cual contiene los métodos para dar soporte a las peticiones. HTTP sobre esta URL.

A continuación, definimos los *middlewares*, que es el nombre que reciben en Node.js las funciones que reencaminan el flujo de ejecución de la aplicación, cuando se da alguna circunstancia excepcional en ella. Por ejemplo, en nuestra aplicación, tenemos definido un middleware, el cual cuando se esté resolviendo cualquier petición, si hay algún error no esperado, el código salte a este middleware y lo gestione, analizando el error que es, escribiendo un log con el problema para el servidor y generando una respuesta HTTP con el error correspondiente.

Por último, tenemos definido en la lógica del back-end, un extractor de usuarios, el cual, cuando un endpoint lo requiera, llamará a esta función antes de ejecutar su propio código. El extractor simplemente leerá el token de usuario que ha sido enviado en la cabecera y lo verificará. De esta manera nos ahorramos duplicar el código por cada controlador o endpoint para resolver el token.

A continuación, vamos a ver tres endpoints de ejemplos, el que utiliza el usuario para acceder, la obtención de notas y el extractor de usuarios.

```
1- loginRouter.post("/", async (request, response) => {
2-     const { body } = request;
3-     const { username, password } = body;
4-
5-     const user = await User.findOne({ username });
6-
7-     let passwordCorrect = null;
8-     try {
9-       passwordCorrect =
10-        user === null ? false : await bcrypt.compare(password, user.passwordHash);
11-   } catch (err) {
12-     return response.status(500).end();
13-   }
14-
15-   if (!passwordCorrect) {
16-     return response.status(401).json({
17-       error: "invalid user or password",
18-     });
19-   }
20-   const userForToken = {
21-     id: user._id,
22-     username: user.username,
23-   };
24-
25-   const token = jwt.sign(userForToken, process.env.JWT_SECRET, {
26-     expiresIn: "7d",
27-   });
28-
29-   response.send({
30-     name: user.name,
31-     username: user.username,
32-     email: user.email,
33-     userPic: user.userPic,
34-     token,
35-   });
```

Bloque de código 13. Endpoint acceso de usuario a la plataforma

En el bloque de código 13, vemos el endpoint utilizado para acceder a la plataforma y recibe en el cuerpo de la petición un nombre de usuario y una contraseña (línea 3). La contraseña del usuario se comparada con el hash que tenemos guardado (línea 10) del

usuario que intenta acceder. Cuando se crean un usuario guardamos su contraseña como hash, de esta forma en la base de datos no se guardarán las contraseñas de los usuarios en texto plano, evitando problemas. Si la contraseña es válida, generaremos el token (línea 20 a 23) que utilizará el usuario para hacer las siguientes peticiones. Este token se genera con *id* y el nombre de usuario, luego al resolverlo de vuelta, se podrá rescatar esta información. La forma de hacer seguro este token es utilizando una cadena de caracteres secreta que nosotros creamos y guardamos de forma segura (línea 25 a 27). En la parte final del endpoint, si no hay ningún error, se compone la respuesta HTTP con información necesaria que se enviará al host que ha hecho la petición de inicio de sesión (línea 29 a 35).

```
1- module.exports = (request, response, next) => {
2-     const authorization = request.get("authorization");
3-
4-     let token = "";
5-     if (authorization && authorization.toLocaleLowerCase().startsWith("bearer")) {
6-       token = authorization.substring(7);
7-     }
8-
9-     let decodedToken = null;
10-    try {
11-      decodedToken = jwt.verify(token, process.env.JWT_SECRET);
12-    } catch (error) {
13-      console.error(error);
14-    }
15-
16-    if (!token || !decodedToken.id) {
17-      return response.status(401).json({ error: "token missing or invalid" });
18-    }
19-
20-    const userId = decodedToken.id;
21-    request.userId = userId;
22-
23-    next();
24-  };
```

Bloque de código 14. Extractor de usuarios

El bloque de código 14, no es un endpoint si no una función middleware. Cuando otro endpoint lo necesita, cambia el flujo normal de ejecución del código y se ejecuta esta función antes. Entrando en detalle en el middleware, la función se encargará de resolver el token de usuario que es necesario añadir en la cabecera de las peticiones HTTP. Para ello simplemente formatea la cabecera (línea 5 a 7), coge el token y lo verifica con la misma cadena de texto con la que generamos el token (línea 11). Al resolver el token y obtener la información de usuario, lo dejamos insertado en la propia petición inicial (línea 20 y 21), por lo que los endpoints ya iniciaran la ejecución del código suponiendo que el *id* del usuario se encuentra en la petición (*request.userId*) que le llega. La última línea *next()*, devuelve el flujo del código al endpoint original.

```
1- notesRouter.get("", userExtractor, async (request, response, next) => {
2-     try {
3-       const user = await User.findById(request.userId).populate({
4-         path: "notes",
5-         select: "title content color important date objective",
6-         populate: {
7-           path: "objective",
8-           select: "name",
9-         },
10-      });
11-      response.json(user.notes);
12-    } catch (error) {
13-      next(error);
14-    }
15-  });
```

Bloque de código 15. Endpoint Get para las notas de usuario

Por último, en el bloque de código 15 vemos un endpoint básico que devuelve las notas que un usuario tiene guardadas. La primera línea define que el endpoint es de tipo *Get*, que la ruta de la URL es: *urlServidor* + *urlControlador* y que antes de ejecutar el código de la función del endpoint (línea 2 a 14), pasara primero por el extractor de usuarios (userExtractor) visto en el bloque de código 14.

La función en sí, con el usuario resuelto gracias al middleware, accede a la base de datos para buscarlo con el id (línea 3) y poblará la lista de notas que tiene asignadas. Podemos definir que queremos recuperar de las entidades Nota (línea 5) e incluso podemos poblar un segundo nivel (línea 6 a 8), en este caso también recupera el nombre de los objetivos asignados. Por último, la función devuelve en la respuesta la lista de notas que serán convertidas a JSON (línea 11) y podemos ver que, si la sentencia try/catch recoge algún error, no lo controlará directamente si no que con *next()* (línea 13) enviará el error al middleware encargado y hará las acciones que sean necesarias.

5. Pruebas y despliegue

5.1 Pruebas

A lo largo del proyecto se han desarrollado dos tipos de pruebas. Para la parte del back-end se ha utilizado ocasionalmente la técnica de TDD (Desarrollo guiado por pruebas), y para la parte del front-end se han hecho pruebas end-to-end.

Las pruebas TDD se han creado en el back-end para probar tanto la base de datos como los distintos endpoints, el conjunto de pruebas tiene el siguiente aspecto:

```
beforeEach(async () => {
    await Note.deleteMany({})
    const notesObjects = initialNotes.map(note => new Note(note))
    const promises = notesObjects.map(note => note.save())
    await Promise.all(promises)
  })

  test('test note can be added', async () => {
    const newNote = { content: 'added note', important: false }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(200)
      .expect('Content-Type', /application\/json/)

    const notes = await getAllContentFromNotes()
    expect(notes.response.body).toHaveLength(initialNotes.length + 1)
    expect(notes.contents).toContain(newNote.content)
  })

  test('note without content is not added', async () => {
    const newNote = { important: false }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(400)

    const response = await api.get('/api/notes')
    expect(response.body).toHaveLength(initialNotes.length)
  })

  test('a note can be deleted', async () => {
    const { response } = await getAllContentFromNotes()
    const noteToDelete = response.body[0]

    await api
      .delete(`/api/notes/${noteToDelete.id}`)
      .expect(204)

    const notesAfter = await getAllContentFromNotes()
    expect(notesAfter.response.body).toHaveLength(initialNotes.length - 1)
    expect(notesAfter.contents).not.toContain(noteToDelete.content)
  })

  afterAll(() => {
    mongoose.connection.close()
    server.close()
  })
```

Bloque de código 16. Pruebas TTD para endpoint del back-end

Como vemos en el bloque de código 16, tenemos un ejemplo de las pruebas que han sido creadas a la vez que se programa la aplicación. Se define un método para que se ejecute antes de cada prueba (línea 1 a 6), que se encarga de dejar la base de datos en un estado conocido para realizar las pruebas y al final del código tenemos un método que se ejecutará al acabar todas las pruebas (línea 47 a 50). Estas pruebas se realizadas contra una base de datos de prueba y no la de producción.

Las pruebas en sí son sencillas ya que simplemente iremos realizando acciones con los endpoints creados y verificaremos el resultado de la acción mirando el estado de la base de datos. Son pruebas de creación de notas, borrado que luego se van sumando las distintas funcionalidades como la edición o filtrado.

Las pruebas end-to-end, han sido usadas para probar principalmente el front-end de la aplicación y para ello se ha usado la herramienta Cypress. Estas pruebas son automáticas y están prueban la aplicación desde la propia interfaz que ve un usuario, utilizando sus mismas entradas. De esta forma podemos verificar de forma directa que todo reaccione de la forma deseada y que el resultado sea el esperado.

Las pruebas de Cypress pueden ser escritas en Javascript y para crearlas iremos definiendo las acciones que se producen en la interfaz, simulando el uso normal del usuario. Para ello utilizaremos los *ids* de los componentes de la aplicación.

La prueba del bloque de código 17, es una prueba diseñada para crear una nota y poder probar que se añaden correctamente:

```
1- it('01.01.01 - Create Note', () => {
2-     cy.login('user', '123456')
3-     cy.get('[href="/notes"]').click()
4-     cy.get('#createNoteButton').click()
5-     cy.get('#title').type('testtitle')
6-     cy.get('#content').type('testcontent')
7-     cy.get('.css-1q0i1xd').click()
8-     cy.get('.chakra-checkbox__control').click()
9-     cy.get('#addNoteButton').click()
10-    cy.get('note').should('contain', 'testtitle')
11-  })
```

Bloque de código 17. Prueba Cypress

La prueba comienza desde la pantalla de inicio de sesión, como si entrásemos a la aplicación de forma normal. Para ello se pueden crear unas funciones personalizadas (línea 2), para poder ser reutilizada, que en este caso introducirá los parámetros de sesión y la iniciará.

Una vez la sesión está iniciada, simplemente vamos indicando como si fuéramos un usuario normal los pasos que deberíamos recrear para crear la nota, es decir, ir a la página

de notas (línea 3), pulsar el botón de crear (línea 4), rellenar el título (línea 5), etc. Por último, debemos incluir condiciones para comprobar que la nota se ha creado correctamente, en este caso podemos comprobar que en la página ha aparecido el texto que hemos introducido en el formulario para el título de la nota (línea 10).

Una de las ventajas de Cypress es su interfaz, disponemos de aplicación web donde podemos ver todas las pruebas creadas para el proyecto y ejecutarlas. Mientras la prueba se ejecuta, vemos visualmente todos los pasos que realiza la prueba, si al ejecutarla vemos que falla podemos recrear todo el recorrido para analizar y detectar donde se produce error.

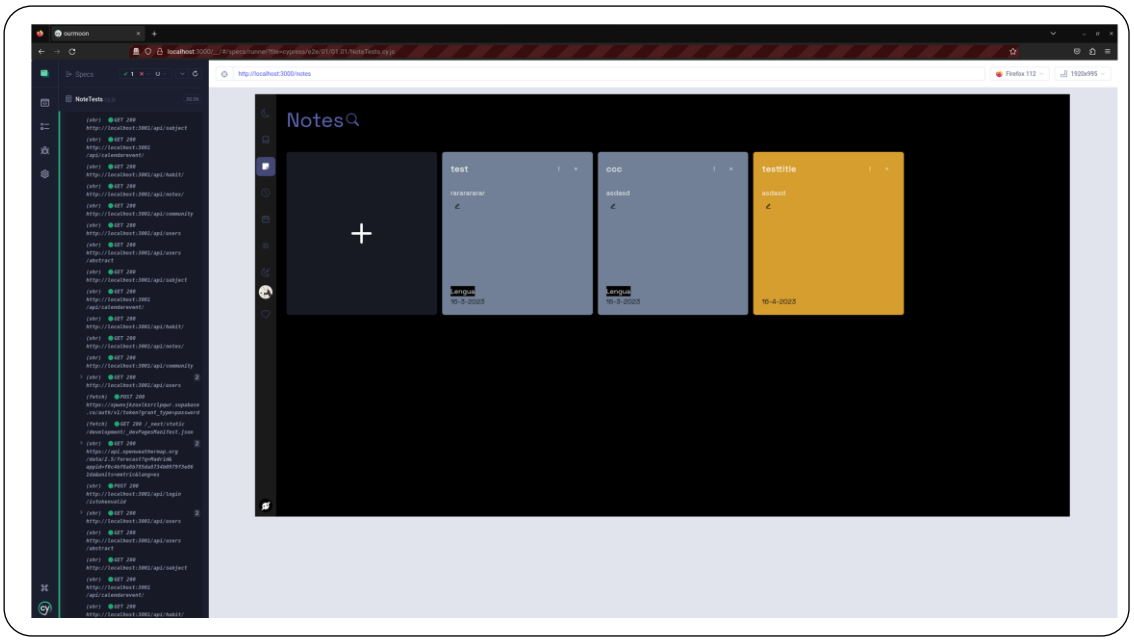


Ilustración 8. Ejecución de una prueba en Cypress

5.2 Integración continua y despliegue

Para el despliegue automático de la aplicación, se ha utilizado Vercel. Vercel es una herramienta que pertenece a los creadores de Next.js y que permite desplegar aplicaciones desarrolladas, utilizando esta tecnología de manera muy cómoda y sencilla. Además, también soporta el despliegue de aplicaciones basadas en Node.js.

Para desplegar un front-end desarrollado con Next.js en Vercel, simplemente debemos indicar el repositorio Git donde se encuentra nuestra aplicación y en qué rama se alojará el código que deseamos desplegar en producción. De esta forma, aautomáticamente cuando detecte algún cambio en la rama, redesplegará la página web con los nuevos cambios.

Para la parte del back-end, al ejecutarse sobre Node.js directamente deberemos crear un archivo *vercel.json* en la raíz del proyecto y especificar en el sobre qué plataforma debemos compilar nuestro proyecto y cuál es la ruta del archivo de entrada. Con este archivo añadido a nuestro proyecto, a la hora de importarlo a Vercel, automáticamente lo leerá y cambiará la configuración del servidor haciendo que se despliegue correctamente.

Un último aspecto fundamental a la hora de desplegar nuestro proyecto son las variables de entorno. Las variables de entorno son variables que contienen información sensible o secreta. Estas variables bajo ningún concepto pueden subirse al repositorio del proyecto ya que pueden ser contraseñas, llaves de API o claves para generar hashes seguros. En nuestro proyecto al ejecutarse en local, debemos crear un archivo para escribir todas estas variables, pero al ejecutarse desde Vercel, esta información deberá ser descrita en la configuración del proyecto, escribiendo el par de nombre de la variable y valor.

En la ilustración 9 podemos ver el proyecto desde Vercel y como nos arroja información sobre su estado. En la ilustración 10 vemos la configuración de las variables de entorno en las opciones del proyecto y por el ultimo en la ilustración 11 vemos como hemos configurado el Git del proyecto y como definimos la rama que marca el redespliegue de la aplicación cuando se produce algún cambio.

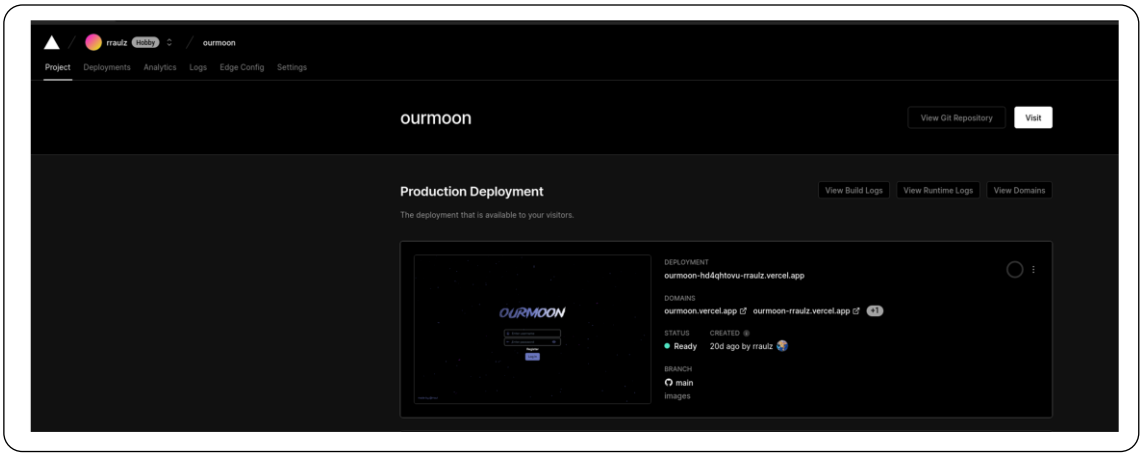


Ilustración 9. Proyecto Vercel

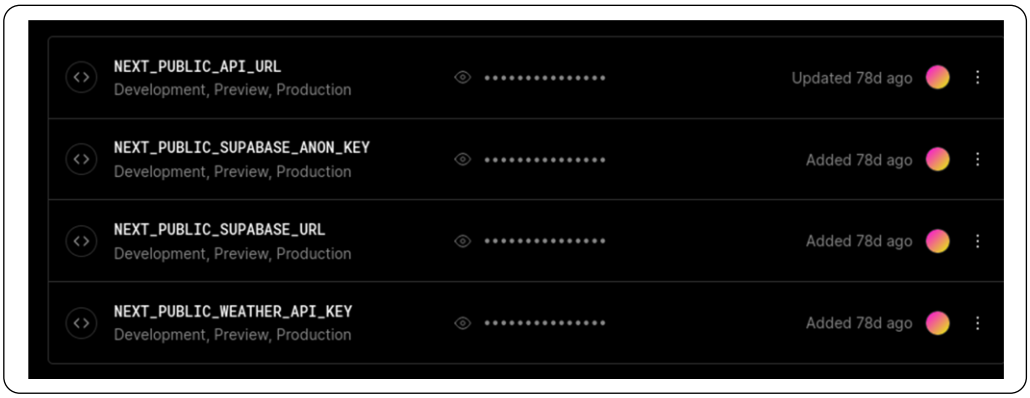


Ilustración 11. Variables de entorno en Vercel

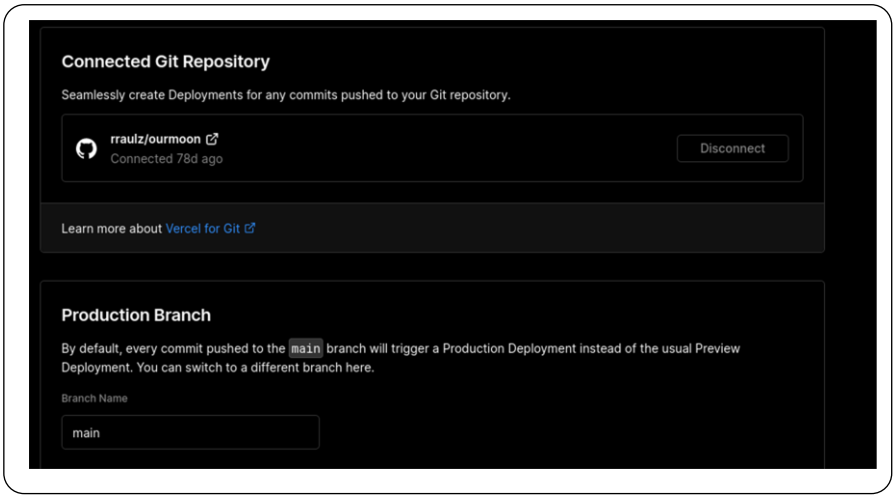


Ilustración 10. Configuración Git en Vercel

6. Conclusiones

En conclusión, el desarrollo de una aplicación de gestión de tareas y hábitos utilizando JavaScript tanto en el front-end como en el back-end ha sido una experiencia enriquecedora y de gran aprendizaje durante mi trabajo de fin de máster. A lo largo de este proyecto, he podido profundizar en el mundo de la programación web y adquirir conocimientos valiosos sobre estas tecnologías.

Una de las principales lecciones aprendidas ha sido comprender la importancia de un diseño de software bien estructurado y modular. Mediante el uso de JavaScript, pude crear componentes reutilizables en el front-end, lo que facilitó el desarrollo y la mantenibilidad del código. Además, la utilización de frameworks y bibliotecas populares, como React.js y Next.js, me permitió agilizar el desarrollo y aprovechar las ventajas que ofrecen para la construcción de aplicaciones web modernas.

Durante el proceso de desarrollo, también me enfrenté a desafíos relacionados con la sincronización de datos entre el front-end y el back-end. Aprendí a utilizar tecnologías como API REST para establecer una comunicación eficiente entre ambas partes de la aplicación. Esta experiencia me brindó una comprensión más profunda de los principios de diseño de API y la importancia de una arquitectura bien definida para garantizar un intercambio de datos fluido.

Al implementar funcionalidades de gestión de tareas y hábitos, tuve la oportunidad de explorar y aplicar diferentes técnicas de manejo de estados, como el uso de hooks en el front-end y proveedores. Esto me permitió gestionar de manera efectiva la información y las interacciones en la aplicación, mejorando así la experiencia del usuario.

Además, durante la creación de esta aplicación de gestión de tareas y hábitos, he adquirido conocimientos valiosos sobre el proceso típico de desarrollo de software. Desde la etapa inicial de análisis de requisitos hasta la planificación y el diseño.

En resumen, el desarrollo de esta aplicación utilizando JavaScript en el front-end y el back-end me ha proporcionado un sólido conjunto de habilidades en programación web. He adquirido conocimientos prácticos sobre el desarrollo de aplicaciones modernas y he aprendido a utilizar herramientas y tecnologías relevantes en la industria. Estoy satisfecho con los resultados obtenidos y confío en que estos conocimientos me serán de gran utilidad en mi carrera profesional como desarrollador de software.

7. Índice de ilustraciones

8. Índice de bloque de códigos

9. Índice de tablas