Universitat Oberta de Catalunya

Desarrollo de un CMS con Markdown notation basado en Node y litElement.

Autor: Jesús Martín Mejías

Tutor: Jordi Ustrell García

Profesor: Ferrán Adell Español

Grado en Diseño Multimedia

Ingeniería Informática

Fecha de entrega

Créditos/Copyright

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Desarrollo de un CMS con Markdown notation basado en Node y litElement*
Nombre del autor:	*Jesús Martín Mejías*
Nombre del colaborador/a docente:	*Jordi Ustrell García*
Nombre del PRA:	*Ferrán Adell Español*
Fecha de entrega (mm/aaaa):	*MM/AAAA*
Titulación o programa:	*Plan de estudios*
Área del Trabajo Final:	*Ingeniería informática*
Idioma del trabajo:	*Español*
Palabras clave	*CMS, Deno, Web Components, Mark Down, LitElement, Shadow Web*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo*

Este proyecto consiste en el desarrollo de una aplicación web CMS, enfocada a desarrolladores, que usa notación Markdown para la edición de los contenidos.

La finalidad es crear un CMS ágil, basado en nuevas tecnologías que aprovechan al máximo los estándares implementados en los navegadores, y competir con otros CMS basados en lenguajes y librerías más tradicionales.

El desarrollo del back-end se realiza en entorno Deno, creando una API para manejar los datos y guardarlos en una base de datos documental tipo MongoDb.

El desarrollo de la programación front-end se programa con la librería LitElement, usando web components y CSS haciendo uso del ShadowDOM, para agilizar la gestión de estilos y dependencias.

La metodología para el desarrollo de esta aplicación aplica el método SCRUM, en sprints de 15 días.

El resultado es una aplicación con una interfaz privada que permite publicar contenido, en forma de posts, con textos e imágenes, usando notación Markdown. La organización del contenido es jerárquica en categorías, con la posibilidad de usar organización transversal en tags. Permite a su vez, confeccionar una portada pública configurable con contenidos destacados.

En conclusión, este trabajo pretende demostrar que usando las últimas tecnologías se puede simplificar y usar más eficientemente los recursos nativos de los propios navegadores y que se puede crear un CMS usable, rápido y de fácil uso enfocado a usuarios habituados a escribir en notación Markdown.

Abstract (in English, 250 words or less):

This project consists of the development of a CMS web application, for developers, using Markdown notation to edit the contents and developed with modern technology.
The purpose of this work is to create an agile CMS, based on new technologies that take full advantage of the native standards implemented in browsers, and compete with other CMS based on traditional languages and libraries.
The back-end development is developed based in Deno library, making an API to handle the data and save it in a documental database as MongoDb.
The front-end programming development is programmed with the LitElement library, using web components and CSS, using ShadowDOM to speed up the management of styles and dependencies.
The methodology used for develop this application uses SCRUM method, in 15-day sprints.
The result will be a web app with a private admin panel that allows to publish content, in the form of posts, with texts and images, using Markdown notation. The content organization is hierarchical in categories, with the possibility of using transversal organization in tags. At the same time, it allows creating a configurable public homepage with featured content.
In conclusion, this work aims to demonstrate that using the latest technologies, it is possible to simplify and use the native resources of the browsers themselves more efficiently and that a usable, fast and easy-to-use CMS can be created for people used to Markdown notation.

Dedicatoria/Cita

Dedico este trabajo a las personas inquietas, que creen en el constante aprendizaje, a todos los profesores en mi vida, y a los profesores de la vida, que son mis amigos, mis familiares y mis desconocidos.

Agradecimientos

Agradezco al profesor que me descubrió *Deno* y *Web Components* Cesar Alberca, y a mi tutor Jordi Ustrell por dejarme hacer este proyecto y acompañarme en el mismo.

Notaciones y Convenciones

Código ejemplo:

```
<div class="wrapper"></div>
```

Estructura de archivos:

```
├── LICENSE
├── Procfile
├── README.md
├── api.json
├── deps.ts
├── favicon.ico
├── main.ts
├── public
│   └── uploads
├── src
│   ├── app.ts
│   ├── areas
│   │   ├── category
│   │   │   ├── category.area.ts
│   │   │   └── category.controller.ts
│   │   ├── image
│   │   │   ├── image.area.ts
│   │   │   └── image.controller.ts
│   │   ├── post
│   │   │   ├── post.area.ts
```

Índice

1 Introducción 13
1.1 Introducción/Prefacio 13
1.1.1 Los navegadores web 14
1.1.2 Web Components y los frameworks Javascript 15
1.1.3 Deno, NODE, Alosaur, usando Javascript para gestionar el back-end 17
1.1.4 Typescript vs Javascript 18
1.1.5 Notación Markdown 18
1.2 Descripción/Definición 19
1.2.1 Punto de partida y necesidades del proyecto 19
1.2.2 Relevancia del proyecto 19
1.2.3 El CMS 20
1.3 Objetivos generales 20
1.3.1 Objetivos principales 20
1.3.2 Objetivos secundarios 21
1.4 Metodología y proceso de trabajo 22
1.5 Planificación 22
1.5.1 Sprint 1 DOCUMENTACIÓN 22
1.5.2 Sprint 2 DISEÑO y DOCUMENTACIÓN 22
1.5.3 Sprint 3 y 4 BACK END 23
1.5.4 Sprint 5 y 6 FRONT END 1 23
1.5.5 Sprint 7 FRONT END 2 y publicación 23
1.6 Estructura del resto del documento 24
2 Análisis de mercado 26
2.1 Público objetivo (i.e. *target audience*) y perfiles de usuario 26
2.2 Antecedentes 27
3 Propuesta 31
3.1 Definición de objetivos/especificaciones del producto 31
4 Diseño 32
4.1 Análisis 32

4.1.1 Análisis de requisitos ... 32
4.1.2 Historias de usuario ... 33
4.1.3 Casos de uso ... 34
4.2 Arquitectura general de la aplicación ... 36
4.2.1 Arquitectura back-end ... 37
4.2.2 Arquitectura front-end ... 37
4.3 Back-end ... 38
4.3.1 Tecnologías ... 38
4.3.2 Organización del back-end ... 39
4.3.3 API REST ... 48
4.4 Base de datos y modelo de datos ... 54
4.5 Front-end ... 56
4.5.1 Tecnologías ... 56
4.5.2 Organización del front-end ... 59
4.5.3 Autenticación ... 81
4.5.4 Notificaciones ... 82
4.5.5 Títulos y meta descripciones ... 82
4.5.6 Destacados ... 83
4.6 Arquitectura de la información y diagramas de navegación ... 83
4.6.1 Parte pública ... 83
4.6.2 Parte privada ... 84
4.7 Diseño gráfico e interfaces ... 84
4.7.1 Estilos ... 84
4.7.2 Usabilidad /UX ... 88
4.8 Lenguajes de programación y *APIs* utilizados ... 94
4.8.1 Back-end ... 94
4.8.2 Front-end ... 94
4.8.3 Herramientas de diseño ... 94
4.8.4 Editor de código ... 95
4.8.5 Repositorio ... 95
5 Implementación ... 96
5.1 Requisitos de instalación ... 96
5.2 Instrucciones de instalación ... 96

5.2.1 Creación de base de datos *MongoDB* con Atlas 96
5.2.2 Instalación de *back-end* con *Deno* de forma local, servidor *Deno* 99
5.2.3 Instalación de back-end con Deno en Heroku 100
5.2.4 Instalación del front-end en entorno local 104

6 Demostración y guía de usuario 106
6.1 Instrucciones de uso 106
6.2 Prototipos 115
6.2.1 Prototipos *Lo-Fi* 115
6.2.2 Prototipos *Hi-Fi* 116
6.3 Ejemplos de uso del producto 122

7 Conclusiones y líneas de futuro 123
7.1 Conclusiones 123
7.2 Líneas de futuro 124

Bibliografía 126
Anexos 128

Figuras y tablas

Índice de figuras

Figura 1 Cuota de navegadores web mundial en el último año 14
Figura 2 Penetración de los Web Components o Custom Elements según Caniuse 16
Figura 3 Comparativa entre los principales CMS por Tecnowired 28
Figura 4 Comparación cualitativa de Joomla!, WordPress, y Drupal. *Martinez-Caro (2018)*. 29
Figura 5 Capas de la arquitectura front-end 37
Figura 6 Diseño de la base de datos documental 54
Figura 7 Diagrama de arquitectura front-end 75
Figura 8 diagrama de navegación de parte pública 83
Figura 9 Estructura de navegación privada 84
Figura 10 Logotipo del CMS 85
Figura 11 Escala de grises en **Deno**tate 85
Figura 12 Aplicación de tipografías en el CMS 86
Figura 13 Escala tipográfica titulares 87
Figura 14 Botones en **Denotate** 87
Figura 15 Iconos emoji utilizados 87
Figura 16 Diseño de la parte pública de una categoría del CMS 88
Figura 17 Diseño mobile de una categoría 89
Figura 18 Diseño de un post Desktop 90
Figura 19 Diseño post mobile 91
Figura 20 Diseño del administrador, vista de posts 91
Figura 21 Administrador versión mobile 92
Figura 22 Administrador vista de edición de post en modo *desktop* 93
Figura 23 Edición de entidad vista mobile 93
Figura 24 Creación de proyecto en Atlas 97
Figura 25 Free Cluster 97
Figura 26 *Interfaz* de configuración de conexión de *Atlas MongoDb* 98
Figura 27 Ventana con la URI para la conexión de la base de datos 98
Figura 28 Ventana que nos indica que el servidor se está ejecutando localmente en el entorno de dev 100
Figura 29 Crear nueva app en Heroku 101
Figura 30 sincronizamos el repositorio donde tenemos el back-end en GitHub 101
Figura 31 Añadiendo buildpack Deno en Heroku 102
Figura 32 Logs con dinosaurio que indica que el servidor y la *app* se ha iniciado correctamente 104
Figura 33 Formulario del primer usuario administrador 106
Figura 34 Formulario de Login 107
Figura 35 Pantalla inicial del administrados sin nada creado 107
Figura 36 Pantalla de configuración del site 108
Figura 37 Nueva categoría 109

Figura 38 Creación de un nuevo post con categoría deportes creada....110
Figura 39 Primer post creado....111
Figura 40 Administrador vista fondo oscuro listado de posts....112
Figura 41 Post con tema oscuro activado....113
Figura 42 Formulario de creación de nuevo usuario....114
Figura 43 Acciones en los listados del administrador....114
Figura 44 Primeros prototipos de estilo, fuentes, diagramas, colores en *Figma*....115
Figura 45 Primeras pantallas diseñadas....116
Figura 46 Diseño de la parte pública de una categoría del CMS....117
Figura 47 Diseño mobile de una categoría....118
Figura 48 Diseño de un post Desktop....119
Figura 49 Diseño post mobile....119
Figura 50 Diseño del administrador, vista de posts....120
Figura 51 Administrador versión mobile....120
Figura 52 Administrador vista de edición de post en modo *desktop*....121
Figura 53 Edición de entidad vista mobile....122

Índice de tablas

Tabla 1 Ranking de cuota de mercado de CMS en webs de internet según Kinsta.com....13
Tabla 2 Planificación....24

1 Introducción

1.1 Introducción/Prefacio

*"Un sistema de gestión de contenidos o **CMS** (del inglés content management system) es un programa informático que permite crear un entorno de trabajo para la creación y administración de contenidos, principalmente en páginas web, por parte de los administradores, editores, participantes y demás usuarios"* [1]

Los CMS llevan muchos años facilitando las labores de edición y publicación a creadores de contenidos, periodistas, documentalistas, publicadores web, profesores y otros profesionales.

En la actualidad existen más de 2000 o 3000 sistemas CMS, entre ellos *WordPress*[2], uno de los más conocidos, pero su evolución ha sufrido una enorme explosión en los últimos 20 años.

En los 70 nacen los primeros CMS ejecutados desde súper ordenadores, sin embargo, es en 1984 cuando se desarrollan los primeros CMS para editar y publicar contenido desde un ordenador personal. En la actualidad más del 45% de todas las páginas web del mercado están construidas con CMS, y de estas según W3techs en el artículo en Kinsta.com: "*Cuota de Mercado de WordPress*", el 63% de estas están, elaboradas con WordPress.

	% Todos los sitios web	% Mercado CMS
WordPress	37	63
Joomla	2.6	4.6
Drupal	1.7	3.0
Squarespace	1.5	2.7
Wix	1.3	2.3

Tabla 1 Ranking de cuota de mercado de CMS en webs de internet según Kinsta.com

Uno de los problemas principales de los CMS más conocidos, *WordPress*, *Joomla*[3], o *Drupal*[4], es el consumo de recursos, debido a su gran variedad de opciones, plugins y capacidades de customización,

[1] **Pérez-Montoto, Mario** (2006). *Gestión del conocimiento, gestión documental y gestión de contenidos***.**

[2] Wordpress, web oficial en español: https://es.wordpress.org/

[3] Joomla, web oficial: https://www.joomla.org/

[4] Drupal, web oficial: https://www.drupal.org/

que hace que sean necesarias librerías pesadas como jQuery y cantidad de código y de estructuración de datos complejos.

De esta problemática nace la idea de construir un CMS liviano, que consuma recursos nativos del navegador y no pesadas librerías, y que se ejecute de forma rápida, usando una tecnología moderna, con componentes independientes tanto en su HTML como en su estilo (HTML y CSS en *Shadow DOM*).

La tecnología usada en el desarrollo de este CMS tiene una relevancia diferenciadora, pero para entenderlo es necesario hacer un pequeño recorrido en el desarrollo *front-end* y *back-end* web, desde el punto de vista de la evolución de los navegadores web.

1.1.1 Los navegadores web

Actualmente la cuota de mercado de los navegadores web es la siguiente:

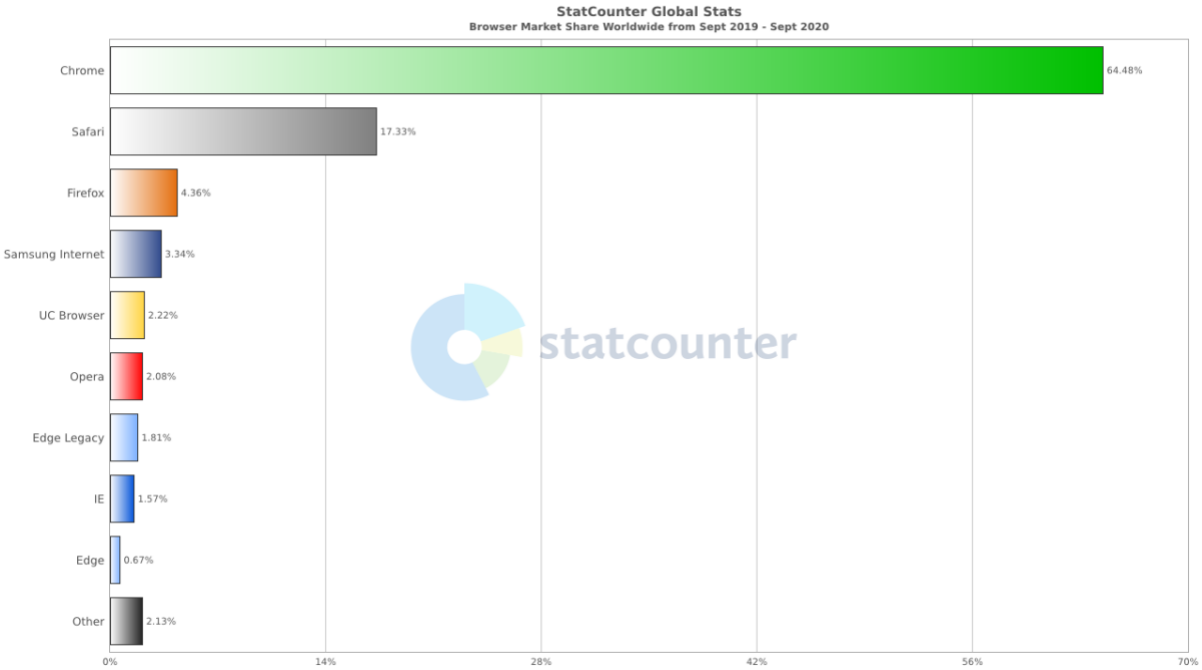

Figura 1 Cuota de navegadores web mundial en el último año

Como podemos observar existen variedad de navegadores y todos compiten por tener la mayor cuota de uso, siendo Chrome el más utilizado mundialmente en la actualidad. *Chrome* nació en 2008, y en poco tiempo ha desbancado a *Internet Explorer* que era el navegador más utilizado cuando nació, por delante también de *Firefox* o *Safari*.

Una de las cosas que ha impulsado *Chrome* como uno de los navegadores más usados, ha sido su integración de herramientas para desarrolladores, y la inclusión de nuevas propuestas en CSS, *Javascript* y otros lenguajes que se interpretan en el navegador.

Todos los navegadores usan un motor de renderizado que interpreta código que luego muestra en forma de aplicaciones web. Durante mucho tiempo *WebKit*[5] ha sido el motor predominante, usado en *Chrome*, *Safari*, y *Ópera* entre otros, hasta que *Chrome* integró su propio motor *Blink*.

La forma en la que un navegador interpreta el código hace que podamos usar determinadas versiones de código, tanto HTML, como CSS o *Javascript* entre otros. Así hay reglas CSS que no todos los navegadores interpretan, y otras que dependerán de la versión del navegador, ya que los navegadores están en constante actualización. De esta manera casi todos los navegadores modernos están actualizados e intentan incluir en sus motores de renderizado las versiones de código consensuadas y estables para que los desarrolladores puedan programar de una forma más moderna.

Una forma de documentarnos si un método, regla, atributo o funcionalidad es interpretable por un navegador, es usar la web *caniuse.com*[6], donde nos indican qué navegadores y versiones aceptan el uso de esos elementos.

1.1.2 Web Components y los frameworks Javascript

Desde hace años y tras el nacimiento de Node, han ido apareciendo librerías basadas en Javascript que trabajan por componentes, que comúnmente hemos llamado frameworks Javascript. Algunas de las más conocidas son Angular, React, Vue o Ember.

Todas ellas tienen en común el uso de componentes, siendo un componente un elemento que encapsula HTML, JS y CSS, pudiendo recibir parámetros (comúnmente llamadas props).

Todos estos frameworks requieren la instalación de sus librerías y otras dependencias, ya que en última instancia renderizan todo en un código compilado de HTML, JS y CSS que el navegador puede entender.

---

[5] Se puede consultar el proyecto *Webkit* en su página oficial: https://webkit.org/

[6] https://caniuse.com/

Este tipo de sistemas ha hecho posible el uso de un Virtual DOM en los navegadores, pudiendo realizar tareas en tiempo real en el cliente, sin necesidad de usar el servidor, pudiendo actualizar vistas sin salir de la misma página web.

La extensión de este tipo de desarrollos y sus aplicaciones en internet han sido exponenciales, estando presente actualmente en las webs relevantes, como *Facebook*, *Twitter*, *AirBnb*, *BBC*, *Netflix*, *Uber* etc…

Todos los navegares modernos ya pueden renderizar de forma nativa elementos en forma de lo que se ha llamado "*web components*", es por lo que actualmente muchos de estos *frameworks* empiezan a renderizar en estos elementos para ser más rápidos y eficientes.

Aquí podemos ver la penetración de web components en los diferentes navegadores:

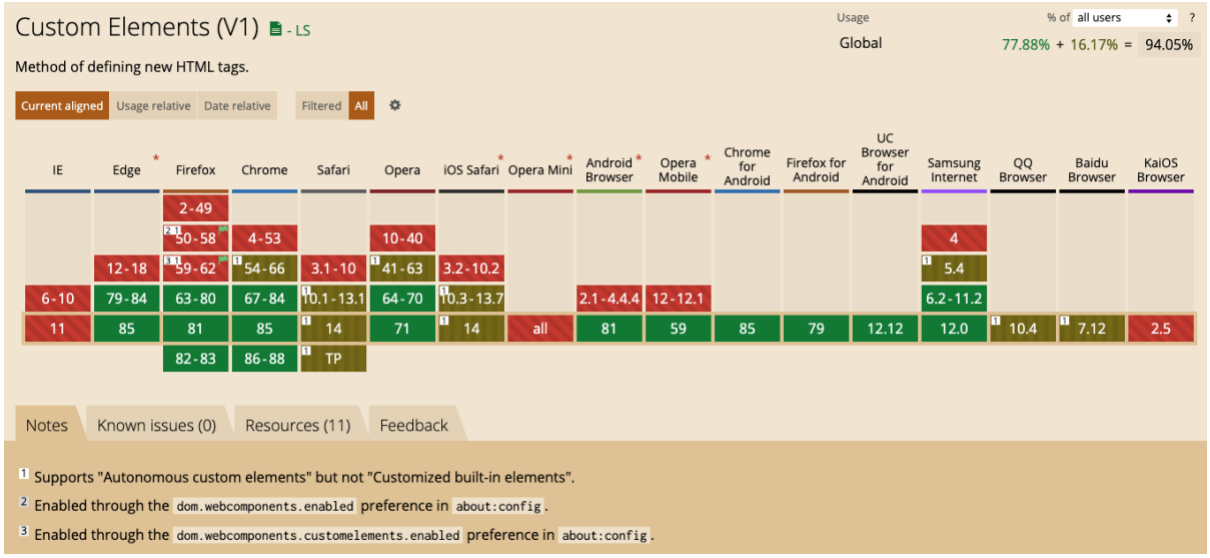

Figura 2 Penetración de los Web Components o Custom Elements según Caniuse

En este proyecto vamos a usar toda la potencia de los web components usando la librería *LitElement* que usa y renderiza directamente en estos elementos, con un peso comprimido de tan solo 6,4kb para toda la librería, lo que lo hace excepcionalmente ligero.

Los web components nativos tienen algunas peculiaridades que los hacen especiales de manejar en su programación y su estilado. Generan lo que se ha llamado "***ShadowDOM***"[7], que es un DOM virtual

[7] https://developer.mozilla.org/es/docs/Web/Web_Components/Using_shadow_DOM

encapsulado. A este dom no pueden acceder el resto de los elementos del DOM, ni los estilos. Puede estar abierto o cerrado, permitiendo, si está abierto, acceder a sus elementos a través de *Javascript*. Al ser elemento que los navegadores modernos puden interpretar, ahorran todas las librerías necesarias que usan los *frameworks* como *React* o *Angular* para renderizar sus componentes en *HTML* y *JS.*

1.1.3 Deno, NODE, Alosaur, usando Javascript para gestionar el back-end

Node es un entorno de programación *Javascript* del lado del servidor. Fue creado en 2009 por *Ryan Dahl*, y rápidamente revolucionó el mundo de la programación al usar *Javascript* para gestionar las peticiones a los servidores web. Aunque la propietaria inicial era la empresa *Joyent* donde trabajaba *Ryan*, la comunidad de programadores de *open source*, paralelamente, lanzó un *fork* amigable de *Node* para hacer avanzar este entorno, bajo el nombre io.js[8], y compatible con npm, lo que hizo que Node ganara muchas mejoras. En un momento determinado se crea *Nodejs Foundation* para reunificar las tecnologías en una sola, llamando al proyecto *Node.js Convergence*, sacando como resultado la versión estable 4.0.0 de Node en 2015 y evolucionando desde entonces de forma estable este entorno de desarrollo.

En 2018, *Ryan Dahl* en su conferencia “10 cosas de las que me arrepiento acerca de Node.js” anuncia Deno[9], un nuevo entorno para *Javascript* y *Typescript* [10]basado en el motor V8 de *Javascript*, más moderno, en el que el autor ha mejorado los fallos que considera que tiene Node.js y evitado las cosas de las que se arrepiente después de todos estos años de Node.js.

Este nuevo entorno soporta *Typescript* de forma nativa, está escrito en *Rust*[11], y no usa un repositorio privado como es NPM[12] en Node. Usa módulos require, algo que ya es nativo en *Javascript* ES6, impulsando el standard, haciendo que los paquetes se importen directamente con import, sin existir un package.json como en Node y que puedan guardarse en caché haciendo su uso más rápido.

La gestión de dependencias está descentralizada, a diferencia de Deno que utiliza un package.JSON, donde se referencian todas las librerías y dependencias que se utilizan en la aplicación, y se descargan

---

[8] https://github.com/nodejs/iojs.org

[9] La documentación oficial se puede consultar en: https://deno.land/

[10] Typescript es un superconjunto de Javascript más tipado: https://www.typescriptlang.org/

[11] Lenguaje de programación: https://www.rust-lang.org/

[12] Gestor de paquetes de Node: https://www.npmjs.com/

en una carpeta llamada node-modules al instalarse el proyecto. En el caso de Deno, la *URL* de descarga ataca directamente a la *URL* del repositorio de cada librería y se descarga la primera vez que carga quedando cacheada, pero nos ofrece la posibilidad de tener un *import map* para tener todas referenciadas, y tiene un objeto global Deno que nos da acceso a muchos módulos nativos que no necesitamos importar. Otra de las características mejoradas con respecto a *Node*, de *Deno*, es la seguridad, para evitar el *malware*, ejecutando los programas en su propio *sandbox*, siendo necesario solicitar permiso para acceder a ellos de forma explícita, por ejemplo para permitir escribir archivos de nuestra aplicación tenemos que poner el *flag –allow-write*.

Por todas estas características, Deno es un nuevo entorno para servidor, moderno y seguro, que usa los propios estándares de los navegadores, lo que hace que sea eficiente y rápido, por eso es mi elección para hacer este proyecto, en el que mi premisa es usar toda la potencia de los propios navegadores.

Para la creación del back-end en Deno, uso Alosaur [13], un framework para crear los controladores, modelos y API REST necesarios para este proyecto

1.1.4 Typescript vs Javascript

Typescript es un lenguaje de programación construido sobre *Javascript*, que dota a *Javascript* de características adicionales para escribir el código con menos errores y con mayor facilidad de uso, coherencia e integridad. Fue creado en 2012 por Microsoft y desde entonces ha crecido su uso por todos los desarrolladores *front-end*.

Tal como hemos referido anteriormente Deno ya incluye soporte *Typescript* de forma nativa y *LitElement* también.

1.1.5 Notación Markdown

Según la web markdown.es[14]: “*Markdown* nació como una herramienta de conversión de texto plano a HTML. Esta herramienta fue creada en 2004 por *John Gruber*, y se distribuye de manera gratuita bajo una licencia BSD”.

---

[13] https://deno.land/x/alosaur@v0.25.0

[14] https://markdown.es/

Actualmente se usa para hacer usualmente los documentos readme.md, y en aplicaciones de *microblogging*. Prácticamente todos los editores de código y de texto tienen *plugins* que permiten la implementación y visualización este lenguaje. Muchas plataformas como GitHub tienen incluidas implementaciones de *Markdown*.

Todo esta ha hecho que la mayoría de los desarrolladores conozcan este lenguaje y lo utilicen comúnmente para documentar sus proyectos o para crear algunos textos. Esta es una de las razones por las que quiero usar esta notación para crear el contenido de este CMS, ya que está enfocado principalmente a desarrolladores y gente tecnológica.

1.2 Descripción/Definición

1.2.1 Punto de partida y necesidades del proyecto

Este proyecto nace de la necesidad de hacer un CMS enfocado a desarrolladores o trabajadores digitales, que usa estándares y recursos nativos de los navegadores. Para conseguir este objetivo son necesarias varias premisas:

- Usar un entorno de edición de contenidos apetecible para desarrolladores, para ello se usa *Markdown*;
- Usar un diseño ligero pero efectivo que se centre en el contenido y no en el diseño;
- Usar un lenguaje de servidor que use los estándares y recursos nativos de los navegadores de forma moderna y actualizada, para ello se usa Deno;
- Usar un desarrollo *front-end* que use de forma eficiente los estándares y recursos nativos de los navegadores, para ello se usa *LitElement* gestionando web components;
- Realizar una aplicación modular que permita su escalado y la incorporación de nuevas funcionalidades. El uso de *web components* ya de por si permite esta compartimentalización.

1.2.2 Relevancia del proyecto

Actualmente conocemos muchos CMS en el mercado, pero no existe un CMS enfocado a desarrolladores, que usen *Markdown notation* como lenguaje de contenidos y que estén basados en tecnologías modernas como es Deno o *Web Components*. El hecho de usar estas tecnologías y de enfocarlo a desarrolladores hace que se pueda tomar ventaja con respecto al resto de CMS y que se pueda extender y usar más rápidamente entre esta comunidad.

1.2.3 El CMS

Esta aplicación consta de dos partes diferenciadas. El administrador privado, al que se puede acceder a través de un login, y la parte pública, que se inicia en una portada administrable desde la parte privada.

Los artículos o posts, se construyen con un editor *MarkDown* simple donde se introduce el contenido, con la posibilidad de usar un entorno *wysiwyg* más visual que la introducción directa del código.
Desde el panel de control se pueden crear categorías, y asociar los posts a las mismas desde la vista de posts. También se pueden crear tags y asociarlas desde los posts.

Estas categorías y tags pueden ser enriquecidas con contenido desde una nueva vista en el panel de administración donde se gestiona cada una por separado.

Todo el contenido se puede organizar en una portada general del CMS donde se puede destacar posts o categorías y navegable desde un menú.

1.3 Objetivos generales

Los objetivos de este trabajo se distribuyen como se explica a continuación

1.3.1 Objetivos principales

Objetivos de la aplicación:

- Ser un CMS rápido y ágil.
- Permitir la introducción de contenidos usando notación *Markdown*.
- Permitir la creación de un sistema taxonómico usando tags y categorías.
- Permitir que se use una temática oscura o clara en la parte publicada del CMS.
- Permitir diferentes configuraciones de visualización para mostrar los diferentes contenidos.
- Permitir incluir fotos y vídeos en el sistema de publicación de contenidos.
- Permitir la configuración y la publicación de una portada de contenidos que muestre contenidos destacados.

Objetivos para el cliente/usuario:

- Permitir que un usuario pueda registrarse como creador de contenido en el CMS.

- Permitir que un usuario pueda publicar, gestionar, categorizar y eliminar el contenido de este CMS.

Objetivos personales del autor del TF:

- Usar Deno como entorno de desarrollo para conectar el servidor y la base de datos.
- Usar *LitElement* con toda la potencia de los *web components*.
- Hacer uso del *Shadow DOM* para la gestión de estilos independientes y encapsulados en cada componente.
- Conocer nuevas tecnologías emergentes, como son Deno y *Litelement*,, en un lenguaje menos flexible como es *Typescript* que *Javascript*, siendo pionero en su implementación en una aplicación compleja, lo que supone un reto de altura en el área de la documentación, de la resolución de problemas y en el aprendizaje de nuevos lenguajes y entornos.

1.3.2 Objetivos secundarios

- Usar Principios *SOLID* en la construcción de los componentes y el código *back-end.*
- Desarrollar una API homogénea para la gestión de contenidos desde el *front-end*.
- Profundizar en todas estas nuevas tecnologías y anticiparnos a su evolución.

1.4 Metodología y proceso de trabajo

La metodología seguida para el desarrollo del código de este CMS se base en la aplicación del método SCRUM en *sprints* de 15 días. Para mantener el desarrollo de esta metodología se usa un gestor de tareas *Trello*[15], donde se organizan las tareas en tiempos y prioridades. Se construye el código en módulos, para confeccionar un mínimo producto viable, que se evolucionará en los diferentes *sprints*, hasta conseguir la aplicación deseada en diversas iteraciones, siguiendo los hitos marcados en la planificación.

La documentación necesaria para la elaboración de este proyecto se obtiene de las páginas oficiales de Deno[16], *litElement*[17], *Typescript*[18], *MongoDB*[19] así como otros recursos asociados que se encuentran en repositorios de GitHub.

El diseño UX / UI se elabora con la aplicación *Figma*.

El código se desarrolla con el editor *Webstorm*[20] de *JetBrains* usando los plugins de *Typescript* y Deno y se usa *GitHub* como repositorio de control de versiones.

1.5 Planificación

1.5.1 Sprint 1 DOCUMENTACIÓN

- Elaboración de la memoria del proyecto inicial;
- Análisis de la forma del proyecto;
- Obtención de información necesaria para el desarrollo.

1.5.2 Sprint 2 DISEÑO y DOCUMENTACIÓN

- Seguimiento de la documentación de la memoria;
- Diseño de las vistas públicas del CMS, portada, categoría, tag, post, login;

---

[15] Gestor de tareas online: https://trello.com/

[16] https://deno.land/

[17] https://lit-element.polymer-project.org/

[18] https://www.typescriptlang.org/

[19] https://docs.mongodb.com/

[20] https://www.jetbrains.com/es-es/webstorm/

- Diseño de las vistas del administrador: Panel de control, usuarios, categorías, posts, tags;
- Diseño de la base de datos en *Mongo*, estructuración de los datos;
- Diseño de la API y de las rutas 1.0.

1.5.3 Sprint 3 y 4 BACK END

- Configuración inicial de la parte Back Deno y del servidor;
- Creación de las rutas de la API en Deno que incluyan:
    - CRUD de portada
    - CRUD de categoría
    - CRUD tag
    - CRUD de post
    - CRUD usuario
    - CRUD imágenes
- Creación de modelos *MONGODB* en Deno para entidades portada, categoría, tag, post;
- Creación de la autenticación en Deno;
- Generación de *JSON Web Toquen* para rutas autenticadas;
- Documentación de la Api y testeo de funcionamiento con *PostMan*.

1.5.4 Sprint 5 y 6 FRONT END 1

- Configuración inicial de *LiteElement* y del FRONT END;
- Estructuración de componentes para vistas privadas: ADMIN.

1.5.5 Sprint 7 FRONT END 2 y publicación

- Estructuración de componentes para vistas públicas: portada, categoría, tag, post;
- Estilado, configuración de portada y de vistas de categoría, tag y post;
- Ajustes y mejoras;
- Testeo en entorno de preproducción.

		Nombre	Fecha inicio	Fecha Fin	PEC
Sprints	1	Documentación	21/09/2020	04/10/2020	2
	2	Diseño	04/10/2020	19/10/2020	3
	3 y 4	Back End	19/10/2020	16/11/2020	3 y 4
	5 y 6	Front End	16/11/2020	14/12/2020	4 y 5
	7 y 8	Front End 2, Testeo y publicación	14/12/2020	28/12/2020	5

Tabla 2 Planificación

1.6 Estructura del resto del documento

2 Análisis de mercado

En esta sección se encuentra el desarrollo de el público objetivo de este CMS, enfocado en los perfiles que he creído más interesantes para el uso del CMS, y los antecedentes de los CMS, donde he querido plasmar
la historia reciente, con los *CMS* más relevantes y sus características, como son Joomla, Wordpress o Drupal.

3 Propuesta

En esta sección se desarrollan los objetivos de la construcción de este *CMS*, que incluyen las especificaciones generales de cualquier CMS y las especificaciones tecnológicas.

4 Diseño

Dentro del apartado 4 se hace una breve explicación de las consideraciones tomadas a la hora de realizar el diseño de la aplicación, realizando previamente un análisis de requisitos, desarrollando unas historias de usuario y unos casos de uso que han ayudado al diseño de los requerimientos del software. Posteriormente en este misma sección se explican desarrolladamente la arquitectura de la aplicación separada en back-end y front-end, desarrollando la construcción de la base de datos, las tecnologías utilizadas y las estructuras de archivos en ambas partes.

A continuación se desarrolla el camino que se ha seguido en el diseño gráfico de la aplicación, desde los estilos iniciales, hasta las pantallas finales, y todas las herramientas y lenguajes utilizados tanto para el diseño como para la programación.

Para concluir se hace referencia a los dos repositorios de código del front-end y el back-end para su supervisión y descarga.

5 Implementación

En esta sección se explican las instrucciones para implementar este CMS, con pautas de implementación de forma local y en un servidor remoto tanto en front-end como en back-end, con un ejemplo de implementación en Heroku.

6 Demostración

Es esta sección se encuentran las instrucciones de uso, junto a los prototipos de la aplicación y un ejemplo práctico de uso, con el que se ilustra como se puede usar el CMS desde su primera instalación hasta la publicación de contenidos y donde se pueden ver algunas de las pantallas del CMS tanto públicas como del administrador.

7 Conclusiones y líneas de futuro

En esta sección se detallan las conclusiones del proyecto, logros, y bloqueos que me he encontrado al desarrollar este CMS y posibles líneas de desarrollo futuras, con un listado de las funcionalidades que están planteadas para poder seguir desarrollando este CMS tras este mínimo producto viable.

Glosario

En este anexo se pueden encontrar los términos utilizados en esta memoria y en este proyecto de especial interés para el lector.

2 Análisis de mercado

2.1 Público objetivo (i.e. *target audience*) y perfiles de usuario

Los CMS en general están enfocados a creadores y publicadores de contenidos, que pueden ser desde particulares, que publican contenidos en blogs personales o comerciales, a compañías, medios online u otras empresas creadoras de contenido.
Debido a la gran diversidad de publicadores de contenidos, el mercado de los CMS se ha diversificado, ofreciendo multitud de opciones y plantillas.

En el caso de este CMS, el público objetivo o *target*, es un **público tecnológico**, principalmente personas que trabajan en el entorno del diseño y del desarrollo, y *webmasters*. El hecho de utilizar ***Markdown***[21] *notation* puede ser un atrayente para personas que conocen este lenguaje de contenidos y que sienten preferencia por el mismo.

Algunos perfiles tipo son:

Webmaster

- Persona entre 26 y 55 años.
- Encargado de instalación y mantenimiento de *websites* para empresas y/o particulares.
- Clase media-alta.
- Nivel académico universitario.
- Programador, diseñador UX, diseñador UI o ingeniero.
- Cultura tecnológica.
- *Heavy user* de Internet.
- Usa *MarkDown*.

Content Manager

- Persona entre 18 y 45 años.
- Encargado de la gestión contenidos web para empresas y/o particulares.
- Clase media.
- Nivel académico intermedio, FP o universitario.

[21] Se puede ver la sintaxis en esta guía de Markdown: https://www.markdownguide.org/basic-syntax/

- Periodista, relaciones públicas, filólogo, autodidacta, traductor o estudiante.
- Cultura tecnológica intermedia.
- *Heavy user* de Internet.
- Puede conocer *Markdown*.

Programador

- Persona entre 24 y 55 años.
- Programador, propietario de un blog o divulgador de contenido tecnológico en otros blogs y redes sociales. Usuario habitual de *GitHub* y otros entornos que usan *Markdown*.
- Clase media-alta.
- Nivel académico universitario.
- Programador o ingeniero.
- Cultura tecnológica alta.
- *Heavy user* de Internet.
- Usa *MarkDown* de forma habitual.

2.2 Antecedentes

Los sistemas de gestión de contenidos aparecieron principalmente en las décadas de los 80 y 90 ante la necesidad de publicar contenidos de forma habitual por empresas inicialmente y por particulares de forma posterior.

Fundado en 1985, ***FileNet*** se considera el primer sistema que fue un sistema de gestión de contenido real. En 1995, *FileNet* introdujo un conjunto completo de programas integrados de administración de documentos con imágenes y flujos de trabajo. ***Vignette***[22] entró en escena a fines de 1995 con el objetivo de hacer que la publicación web sea más accesible y personalizada, y comúnmente se le atribuye el origen del término "**sistema de administración de contenido**". Un año después, *Vignette* presentó *StoryBuilder*. A partir de ese momento muchos CMS empresariales comenzaron a aparecer, incluidos *Interwoven* (1995), *Documentum* (1996), *FatWire* (1996), *FutureTense* (1996), *Inso* (1996) y *EPiServer* (1997). [23]

[22] Vignette fue comprada por OpenText en 2009: https://www.opentext.com/products-and-solutions/products/opentext-product-offerings-catalog/rebranded-products/vignette-is-now-opentext

[23] Fuente: https://www.contentstack.com/blog/all-about-headless/content-management-systems-history-and-headless-cms

A partir de ese momento los CMS evolucionarían, hasta la aparición de los CMS más utilizados en la actualidad y con más penetración: ***WordPress***, seguido de ***Joomla***, ***Drupal***, ***Spacesqueare*** y ***Wix***, aunque sistemas como los ***wikis***[24] también son considerados CMS.

Un CMS, por definición, cuenta principalmente con la capacidad de crear, **editar, agrupar, publicar y mantener contenidos**; pero unos difieren de otros por sus características técnicas, de diseño, de seguridad, precio etc.

Se pueden observar las valoraciones de algunas de estas características en las siguientes tablas comparativa de los CMS más utilizados actualmente:

	Drupal	Joomla	WordPress
Precio	Gratuito	Gratuito	Gratuito
Comunidad activa	* * * *	* * * *	* * * *
Características técnicas	* * * *	* * *	* * * *
Seguridad	* * * *	* * *	* * * *
Diseño personalizable	* * *	* * *	* * * *
Facilidad de uso	* *	* * *	* * * *
Plugins disponibles	* * *	* *	* * * *
Mantenimiento y Soporte	* *	* * *	* * * *
Edición de Contenidos	* * *	* * *	* * * *
Administración del Sitio	* * * *	* * * *	* * * *

Figura 3 Comparativa entre los principales CMS por Tecnowired

[24] Wiki: https://es.wikipedia.org/wiki/Wiki

Feature	Joomla!	Drupal	WordPress
Main content type	Websites, online apps	Blog	Blog, e-commerce, online apps
Extension availability	High	Middle	High
Functionality range	High	Middle	High
Extension repository	Distributed	Centralized	Distributed
Documentation	Excellent	Good	Excellent
User community	Very active	Limited	Very active
Ease of use	Simple	Complex	Simple
User role personalization	Middle	Very High	Middle
Manual SEO positioning	Yes	Yes	Yes
Automatic SEO positioning	Extensions	Modules	Plugins and tools

Figura 4 Comparación cualitativa de Joomla!, WordPress, y Drupal. *Martinez-Caro (2018)*.

Como podemos observar, *WordPress* es actualmente el que tiene cualidades mejor valoradas y el que más opciones tiene.

Algo que tienen en común todos estos CMS es la cantidad de opciones, variedad de diseños, de *plugins* de terceros y diferentes implementaciones. El CMS que desarrollamos en esta memoria pretende ser una **solución sencilla**, con opciones claras que cumplan los objetivos de creación, edición, mantenimiento y publicación de los contenidos, de una forma minimalista y presentados con un diseño *responsive* centrado en el contenido que responde a las necesidades de lectura y de consumo en diferentes dispositivos.

Otro punto interesante que quiero resaltar es la introducción de los contenidos. *WordPress* recientemente ha actualizado este sistema con una tecnología o paquete que ha llamado ***Gutemberg***[25], basado en bloques, que complejiza más que facilita la inclusión de los contenidos. De forma habitual se suelen utilizar editores ***WYSIWYG***[26] para la introducción de contenidos. El CMS que estoy desarrollando pretende usar también un editor *WYSIWYG*, pero con notación ***Markdown***, de tal manera que una persona que esté habituada a esta notación de contenido no tendrá la necesidad de usar los botones del propio editor, sino que podrá incluir estas marcas de forma directa, haciendo mucho más rápida la introducción de contenidos. Si nos enfocamos en las personas que no conocen *Markdown*, o que no están habituados a él, el hecho de tener también un editor *WYSIWYG* facilita la amplitud de usuarios que pueden utilizar este CMS.

Para finalizar esta comparativa quiero hacer referencia a la tecnología y la performance al servir los contenidos de estos CMS. El hecho de que utilicen PHP, como los indicados en las figuras 3 y 4, con

[25] https://wordpress.org/support/article/wordpress-editor/

[26] https://es.wikipedia.org/wiki/WYSIWYG

multitud de opciones y de librearías *Javascript* entre otras, *plugins*, variedad de hojas de estilo CSS no estancas y otras prácticas pesadas, los convierte en aplicaciones pesadas, que a veces no son tan buenas en la performance, y hace que el CMS que planteo ofrezca una mejora cualitativa en su simplicidad tecnológica que acelera esa performance de forma considerable.

3 Propuesta

3.1 Definición de objetivos/especificaciones del producto

Este CMS cumple con todas las cualidades inherentes a cualquier CMS:

- Creación de contenido.
- Agrupación de contenido.
- Edición de contenido.
- Mantenimiento de contenido.
- Publicación de contenido.

Además, cumple con las siguientes características tecnológicas:

- Edición de contenidos con notación *Markdown.*
- Uso de *web components* en el *front-end.*
- Uso de *Deno* como entorno de servidor.
- Uso de Alosaur como framework de Deno para construir la API REST
- Uso de *MongoDB* como base de datos documental.
- CSS compartimentalizadas, con uso de *ShadowDom.*

Y con las siguientes características cualitativas

- Rapidez al cargar los contenidos en el navegador.
- Simplicidad de uso y de gestión.
- Opciones minimalistas necesarias para la publicación del contenido.
- Diseño sencillo que consuma el menor número de recursos.
- Login para editores y administradores a través de formulario, *Google* o *Facebook*.
- Opciones mínimas de personalización.
- Diseño responsive adaptado a todos los dispositivos.

4 Diseño

4.1 Análisis

4.1.1 Análisis de requisitos

Funcionales

1. **Gestión de usuarios (sólo administrador)**
    a. Permite crear, ver, editar o borrar usuarios de la base de datos.

2. **Gestión de blog (sólo administrador)**
    a. Permite gestionar la configuración de *site* de la base de datos.

3. **Gestión de categorías (sólo administrador)**
    a. Permite crear, editar, modificar, destacar o borrar categorías.

4. **Gestión de tags (sólo administrador)**
    a. Permite crear, editar, modificar, destacar o borrar *tags*.

5. **Gestión de *posts***
    a. Permite borrar *posts* (Administrador y creadores de los *posts*).
    b. Permite destacar *posts* (Administrador).
    c. Permite crear, o modificar *posts* (Todos).
    d. Permite crear y asignar *tags* desde el *post* (Todos).
    e. Permite asignar *categorías* desde el *post* (Todos).

No funcionales

1. **Requisitos de *UX* y *UI***
    a. La web tiene interfaces gráficas responsive tanto en la parte de administración como en la parte pública que se ven correctamente en todos los dispositivos.

2. **Requisitos de performance**
    a. La web es rápida y robusta, debe cumplir los estándares de performance, con respecto a imágenes, estáticos y otras buenas prácticas.

3. **Requisitos de seguridad**
    a. Las contraseñas están cifradas en todo momento a través de *Bcrypt*[27].
    b. La validación del usuario se hace por *JSW (JSON Web Token)*[28].

4. **Requisitos legales**
    a. La web cumple con la actual ley de RGPD con un aviso de cookies

4.1.2 Historias de usuario

En este CMS tenemos 3 actores principales:

- Administrador
- Editor
- Usuario no registrado

En base a los requisitos podemos definir historias de usuario que nos ayudan a la definición de las funcionalidades que ha de tener el CMS y los casos de uso.

Las historias de usuario las podemos dividir entre sus actores principales de la siguiente manera:

Historias de administrador

Gestión de la web

Como administrador quiero poder dar de alta el *CMS*, su título, su dominio, y su administrador inicial.

Gestión de usuarios

Como administrador quiero ver el listado de usuarios, y crear, editar o borrar los usuarios con sus características y roles.

Gestión de categorías

[27] https://deno.land/x/bcrypt@v0.2.4

[28] https://jwt.io/

Como administrador quiero ver el listado de categorías, y poder crearlas, editarlas, borrarlas o destacarlas.

Gestión de *tags*

Como administrador quiero ver el listado de *tags*, y poder crearlas, editarlas, borrarlas o destacarlas.

Gestión de *posts*

Como administrador quiero ver el listado de *posts*, poder crearlos, editarlos, borrarlos o destacarlos

Historias de editor

Gestión de *posts*

Como editor puedo ver el listado de *posts*, crearlos, o editarlos.

Como editor puedo borrar los *posts* que he creado.

Historias de usuario no registrado

Como usuario quiero poder navegar desde la portada entre los diferentes posts, tags y categorías destacadas.

Como usuario quiero poder navegar desde la cabecera entre las diferentes categorías de la web.

4.1.3 Casos de uso

Los casos de uso principales son los de administración para el *back-end*. Expongo el caso de uso del administrador que engloba todas las acciones de gestión. El editor puede realizar este mismo caso de uso, pero solo limitado a *posts*.

- **Caso de uso:** Gestionar usuarios, categorías, *tags* y *posts*
- **Actor principal:** Administrador
- **Ámbito:** Administración, parte privada
- **Nivel:** General

Escenarios principales de éxito

1. El usuario hace *login* como administrador en el sistema.
2. El sistema ofrece el menú de administración con usuarios, *tags*, categorías, *posts* y configuración.
3. El usuario elige la información que quiere gestionar.
4. El sistema muestra la página de administración de esa información, donde puede crear nuevos elementos o elegir uno de los que existen.
5. El usuario elige crear, borrar o editar uno de los elementos.
6. Opcionalmente el usuario elige otro tipo de información y volvemos al paso 4.
7. El sistema muestra la interfaz de edición o de creación del elemento.
8. El usuario introduce los datos y envía la información al sistema.
9. El sistema guarda la información.
10. El usuario reitera en los pasos 3 a 8 hasta considerarlo oportuno.

Escenarios alternativos

5 Crear un elemento

1. El usuario elige crear un nuevo elemento, sea usuario, *posts*, categoría o *tag.*
2. El usuario introduce los datos y envía los datos al sistema.
3. El sistema guarda los datos y avisa al usuario.
4. El usuario puede terminar la acción o volver al paso 1.

5 Borrar un elemento

1. El usuario elige borrar un elemento, sea usuario, *posts*, categoría o *tag* y envía la elección al sistema.
2. El sistema borra ese elemento y avisa al usuario.
3. El usuario puede terminar la acción o volver al paso 1.

5 Editar un elemento

1. El usuario elige editar un elemento, sea usuario, *posts*, categoría o *tag.*
2. El sistema ofrece la interfaz de edición de ese elemento.
3. El usuario introduce los datos y envía los datos al sistema.
4. El sistema guarda los datos y avisa al usuario.

5. El usuario puede terminar la acción o volver al paso 1.

- **Caso de uso:** Consultar un post de una categoría
- **Actor principal:** Usuario
- **Ámbito:** Web, parte pública
- **Nivel:** General

Escenarios principales de éxito

1. El usuario entra a la web desde la portada.
2. El usuario accede a una categoría desde la cabecera.
3. El sistema devuelve la página de esa categoría con el listado de *posts* de dicha categoría.
4. El usuario elige el post que quiere consultar.
5. El sistema devuelve el post que ha solicitado el usuario.
6. El usuario puede repetir cualquier paso del 2 al 5.

Escenarios alternativos

2 El usuario accede a una categoría desde la portada.

2 El usuario accede a un post desde la portada.

4.2 Arquitectura general de la aplicación

Este CMS se compone de dos desarrollos separados y diferenciados, el *front-end* construido en componentes con *LitElement*, que hace consultas al *back-end,* construido en Deno, con Alosaur, a través de su *API REST*, el cual obtiene y gestiona los datos a través de sus controladores, haciendo consultas a la **capa de datos** , la base de datos *MongoDB* y los devuelve a la capa *front-end* en forma de respuesta.

Los elementos de estas 3 capas son:

- **Base datos documental** tipo ***MongoDB***.
- ***Back-end,*** **servidor** que ofrece una *API* con diferentes *URIS* elaborado con ***Deno y Alosaur***.
- ***Front-end*** **cliente**, elaborado con ***LitElement*** en *web components*. A su vez se divide en dos partes:
    - **Parte pública**: El contenido navegable que se ha creado con el CMS.

- **Parte privada**: Administrador del CMS.

Además cada aplicación tiene su propia arquitectura.

4.2.1 Arquitectura back-end

El back-end está organizado en:

- **Modelos**: Interfaces que definen los modelos de las entidades
- **Servicios**: Servicios o métodos que comunican las entidades con la base de datos
- **Áreas:** con controladores: son los que gestionan todas las acciones, hacen uso de los servicios, para gestionar y construir las llamadas de la API y las respuestas.

4.2.2 Arquitectura front-end

La arquitectura del front-end está organizada en capas de la siguiente manera:

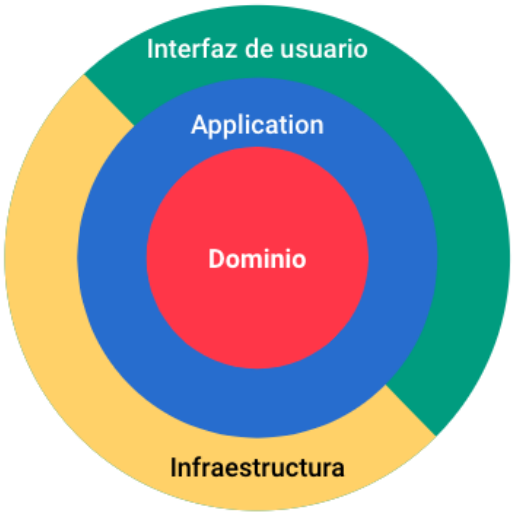

Figura 5 Capas de la arquitectura front-end

Hay varias capas:

- Capa interfaz de usuarios, es la capa de presentación, son los componentes visuales.
- Capa Infraestructura: métodos de interacción con interfaz y aplicación.
- Capa Aplicación: comunicación entre dominio e interfaz e infraestructura,
- Capa dominio: Interfaces de entidades front y de repositorios con lógica y definiciones de negocio.

4.3 Back-end

4.3.1 Tecnologías

Deno

Entorno de desarrollo principal de todo el back-end, similar a NODE pero más moderno y avanzado.

Página oficial y repositorio: https://deno.land/

Alosaur

Es un framework sobre Deno con utilidades como decoradores, middlewares, openAPI, y otras utilidades. Se puede ver toda la documentación en su página oficial y repositorio.

Documentación oficial y repositorio: https://deno.land/x/alosaur@v0.26.0

MongoDB

Base de datos relacional, con respositorio en la nube.

Documentación oficial: https://docs.mongodb.com/

Librerías utilitarias

Bcrypt

Encriptado y desencriptado de contraseñas.

Documentación oficial y repositorio : https://deno.land/x/bcrypt@v0.2.4/mod.ts

DJWT

Creación de JSON web *tokens* y *desencriptación* de los mismos para autenticación.

Documentación oficial y repositorio: https://deno.land/x/djwt@v2.0

Mongo

Es un driver para conectar *Mongo* en *Deno*.

Documentación oficial y repositorio: https://deno.land/x/mongo@v0.20.1

Dotenv

Librería para manejar variables de entorno [29]en *Deno*

Documentación oficial y repositorio: https://deno.land/x/dotenv@v2.0.0

4.3.2 Organización del back-end

Organización de archivos en back-end

```
├── LICENSE
├── Procfile
├── README.md
├── api.json
├── deps.ts
├── favicon.ico
├── main.ts
├── public
│   └── uploads
├── src
│   ├── app.ts
│   ├── areas
│   │   ├── category
│   │   │   ├── category.area.ts
│   │   │   └── category.controller.ts
│   │   ├── image
│   │   │   ├── image.area.ts
```

[29] https://es.wikipedia.org/wiki/Variable_de_entorno

```
│   │   │   └── image.controller.ts
│   │   ├── post
│   │   │   ├── post.area.ts
│   │   │   └── post.controller.ts
│   │   ├── site
│   │   │   ├── site.area.ts
│   │   │   └── site.controller.ts
│   │   └── user
│   │       ├── token.controller.ts
│   │       ├── user.area.ts
│   │       └── user.controller.ts
│   ├── config
│   │   ├── db.ts
│   │   └── env.ts
│   ├── middlewares
│   │   └── log.middleware.ts
│   ├── models
│   │   ├── category.ts
│   │   ├── env.ts
│   │   ├── id.ts
│   │   ├── image.ts
│   │   ├── middleware-target.ts
│   │   ├── post.ts
│   │   ├── site.ts
│   │   ├── tag.ts
│   │   ├── token.ts
│   │   └── user.ts
│   ├── services
│   │   ├── category.service.ts
│   │   ├── image.service.ts
│   │   ├── post.service.ts
│   │   ├── site.service.ts
│   │   ├── tag.service.ts
│   │   ├── token.service.ts
│   │   └── user.service.ts
│   ├── settings.ts
│   └── utils
│       ├── index.test.ts
│       ├── index.ts
│       ├── openApi.ts
│       └── verifyToken.ts
├── tsconfig.app.json
```

└── tsconfig.json

El *back-end* está construido con *Deno*, y con *Alosaur* en *Typescript*. Parte de un archivo ***main.ts***, desde donde importamos al archivo **app.ts**. Todos los modelos, controladores, configuraciones (conexión con la base de datos), archivo de dependencias y middlewares se encuentran en la carpeta **src**.

Existe un archivo de configuración para *Typescript:* **tsconfig.ts,** que a su vez extiende el archivo **tsconfig.app.ts** donde necesitamos la siguiente configuración para que *Alosaur* funcione correctamente, para el uso de decoradores en *Typescript*:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

Las dependencias que utilizamos las englobamos en el archivo deps.ts:

```
export { MongoClient, Database, Collection } from
"https://deno.land/x/mongo@v0.13.0/mod.ts";
export * as flags from 'https://deno.land/std/flags/mod.ts';
export { genSalt, hash, compare } from
"https://deno.land/x/bcrypt@v0.2.4/mod.ts";
export * from "https://deno.land/x/djwt@v1.9/mod.ts"
export * as colors from "https://deno.land/std@0.77.0/fmt/colors.ts";
export * from "https://deno.land/std@0.77.0/fs/mod.ts";
export * from "https://deno.land/x/alosaur@v0.25.0/mod.ts";
export { AlosaurOpenApiBuilder } from
"https://deno.land/x/alosaur@v0.25.0/openapi/mod.ts";
```

Desde este archivo importamos y exportamos las dependencias, de esta manera están centralizadas, exportando solo lo que nos interesa de cada una de ellas. En este archivo se pueden ver las librerías necesarias para la conexión con la base de datos *MongoDB*, *Bcrypt,* para la encriptación de las contraseñas, las librerías propias de *Deno*, como *filesystem*, y *colors*; las librerías del *framework Alosaur,* o la librería del constructor de la documentación de la *API* de *Alosaur AlosaurOpenApiBuilder*, entre otras.

Carpeta SRC

La estructura de la carpeta *src* es la siguiente:

.

```
├── app.ts
├── areas
├── config
├── middlewares
├── models
├── services
├── settings.ts
└── utils
```

Esta carpeta contiene el código funcional de toda la aplicación, normalmente se suele llamar carpeta src, de source a la carpeta que contiene el código que después se va a compilar en la aplicación real, fuera de esta carpeta deberían ir solamente archivos de configuración, complementarios o externos al nucleo de la aplicación. Comprende archivos y carpetas:

app.ts

Es el archivo principal de la *app* de *Alosaur*, donde la iniciamos, de la siguiente manera:

```
import { App } from '../deps.ts'
import { appSettings } from './settings.ts'

export const app = new App(appSettings)
```

settings.ts

Es el archivo donde hacemos las llamadas a las áreas y middlewares

```
import { AppSettings } from '../deps.ts'
import { PostArea } from './areas/post/post.area.ts'
import { Log } from './middlewares/log.middleware.ts'
import { TagArea } from './areas/tag/tag.area.ts'
import { CategoryArea } from './areas/category/category.area.ts'
import { UserArea } from './areas/user/user.area.ts'
import { ImageArea } from './areas/image/image.area.ts'

export const appSettings: AppSettings = {
  areas: [PostArea, CategoryArea, TagArea, UserArea, ImageArea],
  middlewares: [Log],
  logging: true
}
```

Config

En esta carpeta tenemos el archivo **db.ts**, donde configuramos la conexión con la base de datos de MongoDB, que exportamos en la constante **db**.

```
import { MongoClient, Database } from '../../deps.ts'
import env from './env.ts'

// await init();

class DB {
  private client: MongoClient
  constructor(private dbName: string, private dbUri: string) {
    this.dbName = dbName
    this.dbUri = dbUri
    this.client = {} as MongoClient
  }

  connect() {
    const client = new MongoClient()
    console.log(this.dbUri)
    client.connectWithUri(this.dbUri)
    this.client = client
  }

  get getDatabase(): Database {
    return this.client.database(this.dbName)
  }
}

const db = new DB(env.dbName, env.dbUri)

db.connect()

export default db
```

env.ts

En este archivo encapsulo y defino por defecto las diferentes variables de entorno utilizadas en la *app* en una clase que llamo *envBuilder*, que se usarán en caso de no haber declaradas variables de entorno.

Las variables de entorno están en el archivo de la raíz *.env* y son:

- **Deno_ENV:** entorno, si es desarrollo uso DEV si es producción uso PROD. Hay un método en *utils* que pintará un dinosaurio de un tipo o de otro en consola cuando se inicia dependiendo de la variable si es desarrollo o producción.
- **Deno_HOST**: host
- **Deno_PORT:** puerto para el servidor.

- **DB_NAME:** nombre de la base de datos
- **DB_URI**: URI de la conexión de la base de datos
- **SECRET**: palabra secreta que usamos para la encriptación

Models

```
.
├── category.ts
├── env.ts
├── id.ts
├── image.ts
├── middleware-target.ts
├── post.ts
├── site.ts
├── tag.ts
├── token.ts
└── user.ts
```

Son los modelos de las entidades. Para definirlo creamos una interfaz para cada uno de ellos, por ejemplo para categoría:

```
import { ObjectID } from './id.ts'

export interface Category {
  title: string
  brief: string
  description: string
  img: string
  posts: ObjectID[]
}

export type CategoryDoc = ObjectID & Category
```

ObjectId es una interfaz especial que determina como ha de ser un ID.

Los modelos que tenemos son

- **category.ts** para categorías.
- **env.ts** para las variables de entorno que encapsulamos.
- **id.ts** para los *IDs*.
- **image.ts** para las imágenes.
- **middleware-target.ts** para la clase log en *middlewares*.
- **post.ts** para los *posts.*

- **tag.ts** para las *tags.*
- **token.ts** para los *tokens* de la autenticación.
- **user.ts** para los usuarios.

Areas

Son las áreas de *Alosaur.* Un área, tal como se explica en la documentación del propio framework, es un módulo de la aplicación, es decir una agrupación con entidad propia. Cada área tiene un archivo definidor del área y un controlador. Tenemos las siguientes áreas:

```
.
├── category
├── image
├── post
├── site
└── user
```

Un ejemplo de área, el de categoría, es el siguiente:

```
import { Area } from '../../../deps.ts'
import { CategoryController } from './category.controller.ts'

@Area({
  baseRoute: '/category',
  controllers: [CategoryController]
})
export class CategoryArea {}
```

En ella se importa el eobjeto *Area* de *deps.ts*, y el controlador, se define el área con el decorador *@Area*, donde indicamos la ruta base, en este caso */category*, que determinará la ruta base de la *URI* de la *API*.

Se finaliza exportándolo como clase *CategoryArea*.

Las carpetas de áreas, con definición de *Area* y controlador dentro son:

- **category**, para las categorías.
- **image**, para las imágenes.
- **post**, para los *posts*.

- **user**, para los usuarios.

Controlador

En cada carpeta en áreas tenemos, además del área, el controlador. En este controlador están los diferentes métodos que construyen la API, a través de decoradores de *Alosaur*, con llamadas *get*, *posts*, *put* y *delete*.

El controlador es declarado como una clase, con su constructor, donde indicamos además al servicio:

```
@Controller()
export class CategoryController {
  constructor(private readonly service: CategoryService) {}
```

Un ejemplo de un método con decorador para obtener una categoría por id sería el siguiente:

```
@Get('/:id')
async getCategory(@Param('id') id: string, @Res() response: Response,
@Req() request: Request) {
  if (!isId(id)) {
    return new NotFoundError('Category Not Found...')
  }
  try {
    const document: CategoryDoc = await this.service.findCategoryById(id)

    if (document) {
      return Content(document, 200)
    }

    return new NotFoundError('Category Not Found...')
  } catch (error) {
    console.log(error)

    throw new InternalServerError("Failure On 'findCategoryById' !")
  }
}
```

Este controlador hace una llamada al servicio, que se conectan a la base de datos en cada caso y a través de un parámetro id que hemos pasado por la URL, nos devuelve el contenido de la categoría, o un error en caso de que ocurra algún error en la llamada.

Services

```
.
├── category.service.ts
├── image.service.ts
```

```
├── mail.service.ts
├── post.service.ts
├── site.service.ts
├── tag.service.ts
├── token.service.ts
└── user.service.ts
```

La carpeta *services*, tiene los diferentes servicios que conectan con la base de datos, estos servicios son llamados desde el constructor de cada entidad.

El servicio de categoría es de la siguiente manera:

```
import { Injectable } from '../../deps.ts'
import { Category, CategoryDoc } from '../models/category.ts'
import db from '../config/db.ts'

@Injectable()
export class CategoryService {
  private collection: any

  constructor() {
    const database = db.getDatabase
    this.collection = database.collection('category')
  }

  async findAllCategoriesByUser(user: string): Promise<CategoryDoc[]> {
    return await this.collection.find({ user: user })
  }

  async findCategoryById(id: string): Promise<CategoryDoc> {
    return await this.collection.findOne({ _id: { $oid: id } })
  }

  async findCategoryByQuery(query: string): Promise<CategoryDoc> {
    return await this.collection.findOne({ query })
  }

  async insertCategory(post: Category): Promise<any> {
    return await this.collection.insertOne(post)
  }

  async updateCategoryById(id: string, post: Partial<Category>):
Promise<number> {
    const { modifiedCount } = await this.collection.updateOne({ _id: {
$oid: id } }, { $set: post })
    return modifiedCount
  }

  async deleteCategoryById(id: string): Promise<number> {
    return await this.collection.deleteOne({ _id: { $oid: id } })
  }
```

```
  async getPostsFromCategory(id: string): Promise<any> {
    const category = await this.collection.findOne({ _id: { $oid: id } })
    console.log(category)
    return category.posts
  }
}
```

Los servicios son:

- **category.service.ts** para categorías.
- **image.service.ts** para imágenes.
- **post.service.ts** para *posts*.
- **token.service.ts** para los *tokens*.
- **user.service.ts** para los usuarios.

Middlewares

En esta carpeta están los *middlewares* que se llaman desde *settings*. Solo hay una clase log que utilizo para los *logs* en el entorno de desarrollo.

Utils

Son diferentes métodos utilitarios, por ejemplo para pintar el dinosaurio en el arranque de la app, o para saber si un id es un id en MongoDb, o para transformar *Bearer Token* en *token*, quitando *bearer*, o para formatear fechas.

4.3.3 API REST

Alosaur incluye una herramienta para crear la documentación de la *API REST*, llamada *OpenAPIAlosaur*, con un archivo de configuración que está localizado en utils:

```
import { AlosaurOpenApiBuilder } from "../../deps.ts";
import { appSettings } from "../settings.ts";

AlosaurOpenApiBuilder.create(appSettings)
  .addTitle("Denotate")
  .addVersion("1.0.0")
  .addDescription("Denotate OpenApi")
  .addServer({
    url: "https://denotate-back.herokuapp.com/",
```

```
    description: "Backup Denotare Server"
  })
  .saveToFile("./api.json");
```

Con esta herramienta se puede crear un archivo *JSON*, que sirve de base para crear la documentación de la API REST en *Swagger*[30]. Este archivo es *api.json* y se encuentra en la raíz.

En el caso de esta app la API generada es de uso interno, por lo que no es necesario documentarla para uso público o de terceros.

Para documentar la memoria reflejo las diferentes *URIS* que se crean, y qué devuelven en cada caso. Estas URIS hacen peticiones *HTTP*[31] con diferentes métodos. Yo he usado los siguientes:

- **GET**: Hace una petición de datos, es un método más inseguro por lo que sólo deben recuperar datos
- **POST:** Envían entidades a un recurso, causando a veces cambios en el servidor.
- **PUT:** Reemplaza representaciones con las enviadas en la petición
- **DELETE:** Borra recursos específicos que se solicitan en la petición.

Estas llamadas a la API se realizan a través de una URI, que puede contener parámetros, o *queyParams.* Los queryParams se escriben en la URL con **URL?=QueryParam** y los parámetros van directamente al final de la URL **URL/Parámetro**

A continuación detallo las diferentes **URLs** de consulta a la *API* construida y que datos se envían y se devuelven. Todas irían precedidas de la URL del servidor donde está alejado el back-end.

Todas las respuestas que contienen datos están enviadas en formas de objetos o *arrays* de los mismos desde el servidor para que sean gestionados por el *front-end*. Todas las llamadas pueden devolver un error si la petición no se puede realizar por algún tipo. Estos errores son:

- **GET** si el dato solicitado no existe. 404
- **POST** si se rompe alguna regla de integridad, acciones no permitidas, 403, o usuario no logueado on no administrador si es requerido 401
- **PUT** las mismas restricciones que en POST

---

[30] https://swagger.io/

[31] Métodos de petición HTTP: https://developer.mozilla.org/es/docs/Web/HTTP/Methods

- **DELETE** si los elementos a eliminar no existen 404 o si el usuario no está logueado o tiene que ser administrador 401 o intenta borrar algo de manera no adecuada 403

Devolverá también códigos 200 en caso de acciones satisfactorias, y un error generalizado 500 si hay un error no contemplado anteriormente. Esto se controla en la App principal:

```
app.error((context: Context<any>, error: Error) => {
  context.response.result = Content(
    "There is a 500 server error",
    (error as HttpError).httpCode || 500
  );
  context.response.setImmediately();
});
```

En todas las peticiones enviamos el token del usuario como Bearer token [32]en el header, para que las rutas que requieran autenticación o saber si es administrador puedan comprobarlo con ese token.

Si alguna respuesta es diferente lo indico en cada caso:

Ruta /site

Es la ruta que en la que se gestionan los métodos que manejan el **modelo site.**

GET "/site"

- **Respuesta**: datos del ***site*** en un objeto JSON o 404.

POST "/site"

- **Envío**: datos en *body* para crear *site.*
- **Respuesta**: datos del *site* creados 201 o error 400 si el site ya existe.

PUT "/site"

- **Envío**: datos en *body* para cambiar *site.*
- **Respuesta**: datos del *site* modificados 201 o si no se ha modifcado nada 204 u otro error 500.

[32] https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

Ruta /users

GET /users

- **Respuesta**: Devuelve un listado con los usuarios creados.

GET /users/:userid

- **Restricción**: Usuario logueado
- **Parámetros:** userid
- **Respuesta**: Devuelve los datos del usuario que coincide con ese userid o un error 404.

GET /users/thisislogged

- **Respuesta**: Devuelve true si el usuario está logueado correctamente, 200, o 403 si no lo está.

GET /users/thisisadmin

- **Respuesta**: Devuelve true si el usuario es *administrador*, 200, o 403 si no lo es.

GET /users/isadmin

- **Respuesta**: Devuelve true si existe un usuario *administrador* en la base de datos, 200, o 403 si no existe. Se usa para la creación del primer *administrador* nada más entrar en el *site* por primera vez.

DELETE /users/:id

- **Parámetros:** id
- **Restricción:** *administrador*, o devuelve error 403
- **Acción**: Bora el usuario pasado por parámetro.
- **Respuesta**: Devuelve los datos del usuario borrado, o si no lo encuentra un 404

POST /users/login

- **Envío:** Se pasa *login* y *password* por *body*
- **Acción:** *Loguea* el usuario que pasamos por el *body*
- **Respuesta:** Nos devuelve el *token* si ese usuario existe, o nos da error 404 si no lo encuentra o 403 si la contraseña es incorrecta

GET /users/logout

- **Restricción:** logueado o devuelve error 403
- **Acción:** Desloguea el usuario enviando un *token* vacío si encuentra al usuario en la base de datos.

PUT /users/pswd

- **Restricción:** logueado o devuelve error 403
- **Envío**: Contraseñas antigua y nueva a través del body
- **Acción:** Cambia la contraseña del usuario
- **Respuesta:** 200 si cambiada

POST /users/

- **Restricción:** administrador o devuelve error 403
- **Envío**: datos del usuario través del body
- **Acción:** Crea un nuevo usuario
- **Respuesta:** 200 y token del usuario

PUT /users/:id

- **Restricción:** administrador o devuelve error 403
- **Envío**: datos del usuario través del body
- **Acción:** Actualiza un nuevo usuario
- **Respuesta:** 200 y datos del usuario

Ruta Categories

GET /categories

- **QueryParams: user, post, cat, title**
- **Respuesta:** 200 y array de categorías según parámetros enviados o todas

GET /categories/:id

- **Parámetros:** id
- **Respuesta:** datos de la categoría o un 404 si no la encuentra.

GET /categories/:id/posts

- **Parámetros:** id
- **Respuesta:** 200 devuelve los posts que contiene esa categoría

POST /categories/

- **Restricción:** administrador o devuelve error 403
- **Envío**: datos de la categoría través del body
- **Acción:** Crea una nueva categoría
- **Respuesta:** 200 y datos de categoría creada

PUT /categories/:id

- **Restricción:** administrador o devuelve error 403
- **Envío**: datos de la categoría través del body
- **Acción:** Modifica una categoría
- **Respuesta:** 200 y datos de categoría modificada

Ruta Images

POST /images

- **Envio:** título de la imagen y datos
- **Acción:** Crea una imagen y la guarda en una carpeta
- **Respuesta:** URL de la imagen creada

DELETE /images

- **QueryParam:** name
- **Acción:** Borra una imagen por la el nombre enviado como parámetro
- **Respuesta:** 200 y mensaje de imagen borrada

4.4 Base de datos y modelo de datos

El diseño de la base de datos es sencillo. Este diseño se basa en objetos o documentos, al ser una base de datos documental *MongoDB,* siguiendo el siguiente diseño documental:

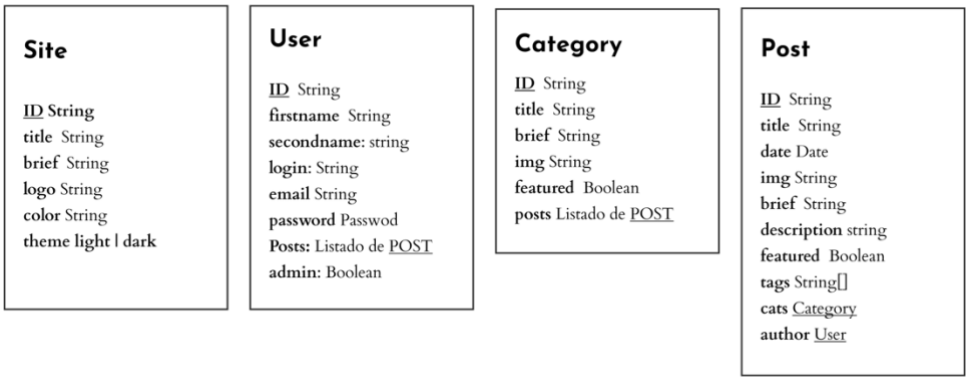

Figura 6 Diseño de la base de datos documental

Todos los documentos tienen un **ID** único, **ObjectID**[33], que es el ID que tienen en *MongoDb* y que *MongoDb* asigna a cada uno al ser creados.

Site

Es el documento que contiene los datos del site, como son:

- **logo:** imagen del logotipo
- **color:** color principal
- **title**: título
- **brief**: descripción
- **theme**: tema claro u oscuro

[33] https://docs.mongodb.com/manual/reference/method/ObjectId/

User

Es el documento que contiene los datos del usuario. Se usa la autenticación para acceder al administrador. Tiene los siguientes atributos:

- **firstName:** es el nombre del usuario
- **secondName:** Apellidos del usuario
- **login:** login del usuario, atributo único.
- **email:** es el email del usuario. Es un atributo único.
- **password:** es la password de acceso. Este atributo está cifrado.
- **posts:** es el listado de posts que ha creado el autor.
- **administrador:** es booleano, si el usuario es administrador es true, sino es false.

Category

Son las categorías. Agrupan *posts*. Tiene los siguientes atributos:

- **title:** es el nombre de dicha categoría, es único, no se puede repetir.
- **brief** es una breve descripción de la categoría.
- ***img*** es la *URL* de la imagen de la categoría. Sólo tiene una por categoría, la podemos sustituir desde el administrador de categorías.
- ***featured*** es un atributo booleano, que indica si la categoría es destacada.
- ***posts*** son los *posts* que tienen esa categoría.

Post

Son los *posts*. Tiene los siguientes atributos:

- **title:** es el título del post, es único
- **date**: es la fecha de creación del *post*.
- **img:** es la imagen destacada del *post*.
- **brief:** es la descripción corta del *pos o entradillat*.
- **description:** es el cuerpo del *post*.
- ***featured*** es un atributo *booleano*, que indica si la categoría es destacada.
- **cats:** es el listado de categorías a las que pertenece el *post*.
- **tags:** es el listado de *tags* a las que pertenece el *post*.

- **author:** es el autor del post. Se considera autor al primer creador.

4.5 Front-end

4.5.1 Tecnologías

Web Components

Según la documentación oficial[34]: *"Los Componentes Web son un paquete de diferentes tecnologías que te permiten crear elementos personalizados reutilizables — con su funcionalidad encapsulada apartada del resto del código — y utilizarlos en las aplicaciones web"*

Es una tecnología que todos los navegadores modernos interpretan, como capsulas de *HTML*, *CSS* Y JS que se encapsulan en un componente padre, declarado en el *DOM*, que contiene un **ShadowDOM** (árbol *DOM* sombra) con esos elementos. El ***shadowDOM*** puede ser open o closed, en *LitElement* por defecto es open, esto hace que podamos acceder a los diferentes elementos a través de *JS*.

Las ventajas de usar estos componentes frente a *frameworks JS* es la rapidez y que son nativos de *HTML* y por lo tanto no necesitan de otras librerías para interpretarse.
La gran desventaja es que el *shadowDOM* encapsula estilos, solo deja pasar las variables *CSS*, por lo puede ser tedioso la gestión de *CSS* generalizados, teniendo que replicar a ve es la misma *CSS* para diferentes componentes. Para ello explico más adelante como lo he resuelto en este proyecto.

Documentación oficial: https://developer.mozilla.org/es/docs/Web/Web_Components

Litelement

Según su propia definición: "*LitElement es una clase base simple para crear componentes web rápidos y ligeros que funcionan en cualquier página web con cualquier framework".*

[34] https://developer.mozilla.org/es/docs/Web/Web_Components

Tal como indica su propia definición, Litelement es realmente ligero (6,8kb), y compila en *web components*, que los navegadores modernos interpretan directamente sin intermediarios ni compiladores.

Documentación oficial: https://lit-element.polymer-project.org/guide

Webpack

Es un compilador de módulos, que compila HTML, JS Y CSS. Litelement se puede compilar con Rollup [35]o con Webpack. Yo he usado Webpack.

Documentación oficial: https://webpack.js.org/

NPM

Es un gestor de paquetes de instalación de NODE (Node package manager). Usamos Este gestor para instalar algunas dependencias utilizadas en el proyecto.

Documentación oficial: https://www.npmjs.com/

Librerías utilitarias

Vaadin router

Router para gestionar web components del lado del cliente. Con esta librería genero todas las rutas públicas del proyecto

Documentación oficial: https://vaadin.com/router

Axios

[35] https://rollupjs.org/guide/en/

Cliente HTTP basado en promesas[36], para navegadores y node.js. Lo utilizo para generar las llamadas a la API.

Documentación oficial: https://github.com/axios/axios

Dotenv

Librería node para gestionar variables de entorno

Documentación oficial y repositorio: https://www.npmjs.com/package/dotenv

Markdown-it

Parseador y renderizador de marcación Markdown

Documentación oficial y repositorio: https://github.com/markdown-it/markdown-it

SimpleMDE Markdown Editor

Es un editor Markdown

Documentación oficial y repositorios: https://simplemde.com/

highlight.js

Marcador de sintaxis para los contenidos del editor markdown

Documentación oficial y repositorio: https://highlightjs.org/

Web components externos

Color picker element

36

Componente *custom* para seleccionar color. Lo utilizo para la página de administración de configuración del *site*.

Documentación oficial y repositorio: https://www.npmjs.com/package/color-picker-element

Switcher web component

Web component custom para originar un switcher con estilo.

Documentación oficial y repositorio: https://www.therogerlab.com/webcomponents/switch.html

Uploader web component

We component custom que genera un componente para subir fotos con drag and drop

Documentación oficial y repositorio: https://www.therogerlab.com/webcomponents/uploader.html

4.5.2 Organización del front-end

Organización de archivos en front-end

```
.
├── LICENSE
├── Procfile
├── README.md
├── index.js
├── package-lock.json
├── package.json
├── src
│   ├── app-lit.ts
│   ├── assets
│   │   └── img
│   │       ├── check.svg
│   │       ├── favicon.ico
│   │       └── logo.svg
│   ├── core
│   │   ├── components
```

```
│   │   │   ├── buttons
│   │   │   │   ├── button.ts
│   │   │   │   ├── index.ts
│   │   │   │   └── styles.ts
│   │   │   ├── extra
│   │   │   │   ├── index.ts
│   │   │   │   ├── switch.js
│   │   │   │   └── uploader.js
│   │   │   ├── forms
│   │   │   │   ├── container-form.ts
│   │   │   │   ├── index.ts
│   │   │   │   ├── inputs
│   │   │   │   │   ├── index.ts
│   │   │   │   │   ├── input-base.ts
│   │   │   │   │   ├── input_styles.ts
│   │   │   │   │   ├── option.ts
│   │   │   │   │   └── option_styles.ts
│   │   │   │   └── styles.ts
│   │   │   ├── index.ts
│   │   │   ├── markdownEditor
│   │   │   │   ├── md.ts
│   │   │   │   ├── mdEditor.ts
│   │   │   │   └── styles.ts
│   │   │   ├── modulos
│   │   │   │   ├── index.ts
│   │   │   │   ├── modulo.ts
│   │   │   │   └── styles.ts
│   │   │   ├── notice
│   │   │   │   ├── index.ts
│   │   │   │   ├── notice.ts
│   │   │   │   └── styles.ts
│   │   │   └── slider
│   │   │           ├── index.ts
│   │   │           ├── slider.ts
│   │   │           └── styles.ts
│   │   ├── index.ts
│   │   └── pages
│   │           ├── admin
│   │           │   ├── admin-container-styles.ts
│   │           │   ├── admin-container.ts
│   │           │   ├── menu.ts
│   │           │   └── menu_styles.ts
```

```
│   │       ├── index.ts
│   │       └── public
│   │           ├── body-container-styles.ts
│   │           ├── body-container.ts
│   │           ├── footer.ts
│   │           ├── header.ts
│   │           ├── header_style.ts
│   │           ├── index.ts
│   │           ├── not-found.ts
│   │           ├── public-container-full.ts
│   │           ├── public-container.ts
│   │           └── special-container.ts
│   ├── featured
│   │   ├── categories
│   │   │   ├── domain
│   │   │   │   ├── category-repository.ts
│   │   │   │   └── category.ts
│   │   │   ├── infrastructure
│   │   │   │   ├── category-dto-to-category-mapper.ts
│   │   │   │   ├── category-dto.ts
│   │   │   │   ├── category-http-repository.ts
│   │   │   │   ├── category-repository-factory.ts
│   │   │   │   └── category-to-category-dto-mapper.ts
│   │   │   └── ui
│   │   │       ├── admin-list.ts
│   │   │       ├── edit.ts
│   │   │       ├── home.ts
│   │   │       ├── list.ts
│   │   │       └── new.ts
│   │   ├── images
│   │   │   ├── domain
│   │   │   │   ├── image-service.ts
│   │   │   │   └── image.ts
│   │   │   └── infrastructure
│   │   │       └── image-http-service.ts
│   │   ├── posts
│   │   │   ├── domain
│   │   │   │   ├── post-repository.ts
│   │   │   │   ├── post-service.ts
│   │   │   │   └── post.ts
│   │   │   ├── infrastructure
│   │   │   │   ├── post-dto-to-post-mapper.ts
```

```
│   │   │   │   ├── post-dto.ts
│   │   │   │   ├── post-http-repository.ts
│   │   │   │   ├── post-repository-factory.ts
│   │   │   │   └── post-to-post-dto-mapper.ts
│   │   │   └── ui
│   │   │       ├── admin-list.ts
│   │   │       ├── edit.ts
│   │   │       ├── form-styles.ts
│   │   │       ├── home.ts
│   │   │       ├── list.ts
│   │   │       └── new.ts
│   │   ├── shared
│   │   │   ├── auth
│   │   │   │   ├── auth-guard.ts
│   │   │   │   ├── authorization-service.ts
│   │   │   │   └── page-enabled.ts
│   │   │   ├── emptyObjects
│   │   │   │   └── index.ts
│   │   │   ├── http
│   │   │   │   └── http.ts
│   │   │   └── id
│   │   │       └── id.ts
│   │   ├── site
│   │   │   ├── domain
│   │   │   │   └── site.ts
│   │   │   ├── infrastructure
│   │   │   │   └── site-service.ts
│   │   │   └── ui
│   │   │       ├── home.ts
│   │   │       └── update-site.ts
│   │   ├── tags
│   │   │   ├── domain
│   │   │   │   ├── tag-repository.ts
│   │   │   │   └── tag.ts
│   │   │   ├── infrastructure
│   │   │   │   ├── tag-http-repository.ts
│   │   │   │   └── tag-repository-factory.ts
│   │   │   └── ui
│   │   │       └── home.ts
│   │   └── users
│   │       ├── domain
│   │       │   ├── user-repository.ts
```

```
│   │       │   ├── user-service.ts
│   │       │   └── user.ts
│   │       ├── infrastructure
│   │       │   ├── user-dto-to-user-mapper.ts
│   │       │   ├── user-dto.ts
│   │       │   ├── user-http-repository.ts
│   │       │   ├── user-http-service.ts
│   │       │   ├── user-repository-factory.ts
│   │       │   └── user-to-user-dto-mapper.ts
│   │       └── ui
│   │           ├── admin-list.ts
│   │           ├── edit.ts
│   │           ├── first-user.ts
│   │           ├── list.ts
│   │           ├── login.ts
│   │           └── new.ts
│   ├── index.html
│   ├── main.ts
│   ├── routes
│   │   ├── admin
│   │   │   ├── categories.ts
│   │   │   ├── index.ts
│   │   │   ├── posts.ts
│   │   │   └── users.ts
│   │   ├── index.ts
│   │   ├── others
│   │   │   └── index.ts
│   │   └── public
│   │       ├── categories.ts
│   │       ├── index.ts
│   │       ├── posts.ts
│   │       ├── tags.ts
│   │       └── users.ts
│   ├── styles
│   │   ├── admin-styles.ts
│   │   ├── general.ts
│   │   ├── public.ts
│   │   └── theme.ts
│   └── utils
│       ├── env.ts
│       ├── envModel.ts
│       └── utils.ts
```

```
├── tsconfig.json
├── web-test-runner.config.mjs
├── webpack.config.js
└── webpack.prod.js
```

El Proyecto se divide en su directorio principal en 3 directorios secundarios:

- **dist:** es donde se compila el proyecto con Webpack [37]cuando se hace build.
- **src:** es el directorio principal del proyecto, más adelante detallo su contenido.
- **styles:** es donde se encuentran los estilos generales del proyecto.
- **tests:** carpeta de tests.

Además de estas carpetas hay algunos archivos de configuración necesarios:

Además de los archivos de configuración y el README del proyecto:

```
├── LICENSE
├── README.md
├── custom-elements.json
├── package-lock.json
├── package.json
├── tsconfig.json
├── web-test-runner.config.mjs
└── webpack.config.js
```

Webpack-config.js

Es el archive de configuración de webpack, donde está configurada la compilación en entorno de desarrollo y de producción:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const CopyWebpackPlugin = require("copy-webpack-plugin");
const TerserPlugin = require("terser-webpack-plugin");
const Dotenv = require("dotenv-webpack");

module.exports = (env, argv) => ({
  mode: "development",
  entry: {
```

[37] https://webpack.js.org/api/compilation-object/#root

```
    app: "./index.ts"
  },
  devServer: {
    contentBase: path.join(__dirname, "dist"),
    historyApiFallback: true
  },
  devtool: argv.mode === "production" ? "none" : "inline-source-map",
  plugins: [
    new CleanWebpackPlugin(),
    new Dotenv(),
    new HtmlWebpackPlugin({
      template: "./index.html"
    }),
    new CopyWebpackPlugin([
      {
        context: "node_modules/@webcomponents/webcomponentsjs",
        from: "**/*.js",
        to: "webcomponents"
      },
      {
        from: "./src/assets/img/*",
        to: "./",
        flatten: true
      }
    ]),

    new TerserPlugin({
      parallel: true,
      terserOptions: {
        ecma: 6,
        output: {
          comments: false
        }
      }
    })
  ],
  output: {
    filename: "[name].bundle.js",
    path: path.resolve(__dirname, "dist")
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use:
          argv.mode === "production"
            ? [
                {
                  loader: "minify-lit-html-loader"
                },
                {
                  loader: "ts-loader"
                }
              ]
            : [
                {
                  loader: "ts-loader"
```

```
                }
              ],
        exclude: /node_modules/
      },
      {
        test: /\.css$/,
        include: path.resolve(__dirname, "styles"),
        //include: /stylesheets|node_modules/,
        use: ["style-loader", "css-loader"]
      },
      {
        test: /\.s[a|c]ss$/,
        use: [
          { loader: "style-loader" },
          { loader: "css-loader" },
          { loader: "sass-loader" }
        ]
      },
      {
        test: /\.(png|gif|jpg|cur)$/i,
        loader: "url-loader",
        options: { limit: 8192 }
      },
      {
        test: /\.woff2(\?v=[0-9]\.[0-9]\.[0-9])?$/i,
        loader: "url-loader",
        options: { limit: 10000, mimetype: "application/font-woff2" }
      },
      {
        test: /\.woff(\?v=[0-9]\.[0-9]\.[0-9])?$/i,
        loader: "url-loader",
        options: { limit: 10000, mimetype: "application/font-woff" }
      },
      {
        test: /\.(ttf|eot|svg|otf)(\?v=[0-9]\.[0-9]\.[0-9])?$/i,
        loader: "file-loader"
      }
    ]
  },
  resolve: {
    extensions: [".tsx", ".ts", ".js", ".css"]
  }
});
```

tsconfig.json

Es el archive de configuración de typescript

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "experimentalDecorators": true,
    "strict": false,
```

```
    "strictPropertyInitialization": false,
    "sourceMap": true,
    "module": "es2020",
    "target": "es6",
    "moduleResolution": "node",
    "jsx": "react",
    "allowJs": true
  }
}
```

index.html

Es el archivo HTML inicial que contiene todo el proyecto. Llama a las *CSS* de fuentes externas, otros estilos, y contiene un div con id outlet, donde el router va a inyectar todos los componentes en las rutas.

He incluido una etiqueta base, ya que al compilarse el router si no ponemos esta base genera algunas rutas secundarias mal construidas.

```
<!doctype html>
<html lang="es">
<head>
  <base href='/'>
  <!-- Mobile Specific Metas
    ------------------------------------------------- -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0,
user-scalable=no, minimum-scale=1.0,maximum-scale=1.0" />
  <meta charset="utf-8">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link
href="https://fonts.googleapis.com/css2?family=Cardo&family=Josefin+Sans&di
splay=swap" rel="stylesheet">
  <style>
    *{margin: 0;
    padding: 0}
  </style>
  <script src="webcomponents/webcomponents-loader.js"></script>
</head>
<body>
<div id="outlet"></div>
</body>
</html>
```

Index.ts

Es el archive que inicia el proyecto realmente. Desde él se llama al router y se generan las rutas. Todo el proyecto se renderiza en un index.html, que contiene un div con id #outlet tal como se muestra en este archivo

```
import { Router } from "@vaadin/router";
import { routes } from "./src/routes/index";
export const router = new Router(document.getElementById("outlet"));
router.setRoutes(routes);
```

.env

Variables de entorno. Solo usamos una que es la URL de la API.

package.json y package.json.lock

Son los archivos de dependencias de npm con todas las librerías instaladas entre ellas Litelement

Src

Es la carpeta donde se encuentran los directorios operativos del proyecto.

```
├── app-lit.ts
├── assets
├── core
├── featured
├── routes
└── utils
```

app-lit.ts

Componente: app-lit

Es el componente inicial del proyecto. Desde el se llama a todos los demás componentes, exceptuando los que se llaman desde el router que genera las vistas. Dentro está también el contenedor general de toda la aplicación, que además controla si el tema es claro u oscuro.

Para mostrar el tema en colores claros u oscuros la gestión del tema claro u oscuro se hace comprobando en la base de datos que tema ha elegido el administrador, pero también tiene un listener que observa un evento lanzado desde el switcher de la página de configuración del tema, para que el cambio sea instantáneo.

```
public static styles = [
    theme,
    general,
    css`
      #wrapper {
        background: var(--background-wrapper);
        background-size: var(--background-wrapper-size);
        min-width: 100%;
        min-height: 100vh;
        overflow: auto;
        font-family: var(--body-font);
      }
      @import
url("https://fonts.googleapis.com/css2?family=Cardo&display=swap");
      @import
url("https://fonts.googleapis.com/css2?family=Josefin+Sans&display=swap");
    `
  ];

  render() {
    return html`
      <div id="wrapper" class="${this.theme}">
        <style>
          #wrapper {
            --main-color: ${this.color} !important;
          }
        </style>
        <slot></slot>
      </div>
    `;
  }
  async connectedCallback() {
    super.connectedCallback();
    const siteService = new SiteService();
    const site = await siteService.getSite();

    if (site) {
      this.color = site.color;
      this.theme = site.theme;
    } else {
      Router.go("/newsite");
    }
  }

  _handleChangeTheme = e => {
    this.theme = e.detail.theme;
  };
```

```
  async firstUpdated() {
    await new Promise(r => setTimeout(r, 0));
    this.addEventListener("change-theme", this._handleChangeTheme);
  }
  disconnectedCallback() {
    this.removeEventListener("change-theme", this._handleChangeTheme);
    super.disconnectedCallback();
  }
}
```

Assets

Es la carpeta donde están los estáticos del proyecto, como el logotipo u otras imágenes.

Core

```
.
├── components
├── index.ts
└── pages
```

En esta carpeta se encuentran los componentes no funcionales, es decir la interface que luego se reutilizará en vistas y componentes funcionales

Están divididos en dos grupos: components y pages.

Pages

```
├── admin
│   ├── admin-container-styles.ts
│   ├── admin-container.ts
│   ├── menu.ts
│   └── menu_styles.ts
├── index.ts
└── public
    ├── body-container-styles.ts
    ├── body-container.ts
    ├── footer.ts
    ├── header.ts
    ├── header_style.ts
    ├── home.ts
    ├── index.ts
```

```
├── not-found.ts
├── public-container.ts
└── special-container.ts
```

Distribuidos en dos carpetas *administrador* y *public* contiene:

- Los componentes contenedores de las vistas, de parte pública y privada (administrador)
- Los menús de la parte pública y privada (administrador)
- El footer de la parte pública
- La vista para página no encontrada o 404
- Los estilos para cada componente

Components

```
.
├── buttons
│   ├── button.ts
│   ├── index.ts
│   └── styles.ts
├── extra
│   ├── index.ts
│   ├── switch.js
│   └── uploader.js
├── forms
│   ├── container-form.ts
│   ├── index.ts
│   ├── inputs
│   │   ├── index.ts
│   │   ├── input-base.ts
│   │   ├── input_styles.ts
│   │   ├── option.ts
│   │   └── option_styles.ts
│   └── styles.ts
├── index.ts
├── markdownEditor
│   ├── md.ts
│   ├── mdEditor.ts
│   └── styles.ts
├── modulos
```

```
│   ├── index.ts
│   ├── modulo.ts
│   └── styles.ts
├── notice
│   ├── index.ts
│   ├── notice.ts
│   └── styles.ts
└── slider
    ├── index.ts
    ├── slider.ts
    └── styles.ts
```

Son los componentes visuales. Cada componente puede tener un archivo de estilos separado para hacer el componente menos extenso.

Los componentes visuales que hay son:

- **Buttons** : botones
- **Switch** : switcher
- **Uploader** : subidor de imágenes con drag and drop
- **container-form**: contenedor de formularios
- **input-base:** input base, para todos los inputs que no sean radiobuttons o checkbox
- **option**: input radiobutton o checkbox
- **markdownEditor**: Editor MarkDown
- **modulo**: Módulo de evento para listados
- **notice:** Banda de notificaciones de usuario
- **slider:** slider de portada de categorías destacadas

Cada carpeta contiene un *index.ts* donde se importan sus componentes. A su vez de forma anidad todos los index se importan en un index.ts en la carpeta componentes. Este *index.ts* es el que se importa en *app-lit.-ts* haciendo que todos los componentes visuales estén disponibles para todo el DOM en todo momento.

Routes

```
.
├── admin
│   ├── categories.ts
│   ├── index.ts
```

```
│   ├── posts.ts
│   └── users.ts
├── index.ts
├── others
│   └── index.ts
└── public
    ├── categories.ts
    ├── index.ts
    ├── posts.ts
    ├── tags.ts
    └── users.ts
```

En esta carpeta se gestionan las rutas del proyecto, usando la librería Vaadin Router[38]

La gestión de esta carpeta es también con un index.ts, desde donde a su vez se importan los index de las carpetas public, administrador, y others.

Dentro de cada carpeta hay otro index que gestiona otras rutas, principalmente de cada entidad:

```
    ├── index.ts
    ├── posts.ts
    ├── tags.ts
    └── users.ts
```

Cada ruta es un objeto, con diferentes opciones:

```
{
  path: "/:category",
  action: async () => {
    await import("../../featured/categories/ui/home");
  },
  component: "category-home-c"
},
```

Path: es la ruta pública.

Action: son acciones o métodos que queremos que se ejecuten. En muchos casos, importamos el componente de la vista desde esta función, o gestionamos las autenticaciones o validaciones.

Component: es el componente que se va a renderizar en esta ruta.

[38] https://vaadin.com/router

En ocasiones es necesario hacer una redirección y no necesitamos componentes:

```
{
  path: "/",
  redirect: "/"
},
```

Redirect: Es la ruta a la que debe redireccionar esta ruta.

Todas las rutas están organizadas jerárquicamente para que una ruta padre pueda contener un componente que englobe a los demás, por ejemplo para gestionar cabeceras o elementos comunes.

Utilizo el método *setTitleDescriotion* en algunas rutas para setear el *title* la meta *description* de la página.

Featured

```
.
├── categories
├── images
├── posts
├── shared
├── site
├── tags
└── users
```

En cada entidad de la carpeta featured encontramos las siguientes subcarpetas:

```
.
├── domain
├── infrastructure
└── ui
```

Que hacen referencia a las capas explicadas anteriormente, de arquitectura de software utilizada, de interfaz de usuario, infraestructura y dominio:

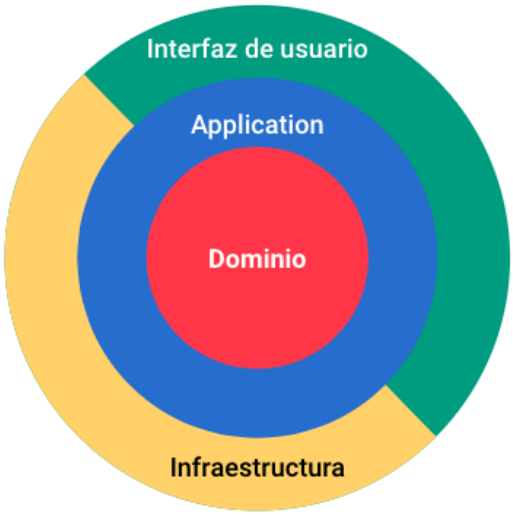

Figura 7 Diagrama de arquitectura front-end

A continuación se detalla que contiene cada carpeta:

Domain

```
.
├── entidad-repository.ts
└── entidad.ts
```

Se corresponde con Dominio. Es donde está la lógica de negocio. Dentro tengo las interfaces que definen la entidad y la gestión del repositorio de esa entidad. Esta gestión es uan interfaz, luego tendrá que ser implementada por otras clases.

Infrastructure

```
.
├── entidad-dto-to-entidad-mapper.ts
├── entidad-dto.ts
├── entidad-http-repository.ts
├── entidad-repository-factory.ts
└── entidad-to-entidad-dto-mapper.ts
```

Contiene:

- **Entidad-dto-to-entidad-mapper:** Método que convierte las entidades que vienen del back-end en entidades para el front-end definidas en domain.

- **Entidad-to-dto-entidad-mapper:** Método que convierte las entidades del front en entidades adecuadas para enviar al back-en. Este método y el anterior hace posible separar ambas entidades de tal manera que si en algún momento la entidad del back-end o la del front-end cambia sea sencillo implementarlo cambiando solo estos dos métodos.
- **Entidad-dto**: Es la interfaz de la entidad que viene del back-end
- **Entidad-http-respository:** Implementa los métodos definidos en la interfaz interfaz-repository. En este caso como usamos una api, esta implementación se hace con llamadas http, en otros casos podría consultarse un archivo json, o implementarlo de otras maneras, separando la lógica de negocio de la lógica de implementación.
- **Entidad-repository-factory**: Clase Factory que aglutina los dos *mappers* y el *http-repository*. Es la que llamaremos para gestionar las entidades.

UI

```
.
├── admin-list.ts
├── edit.ts
├── home.ts
├── list.ts
└── new.ts
```

Son los componentes que se van a llamar en las diferentes vistas, con contenido funcional.
A su vez contienen componentes visuales de *core* y hacen las gestiones con la factoría de la entidad para pintar los datos que se obtienen desde el servidor. En el esquema de arquitectura, sería UI, pero en este caso hacen de conexión entre la interfaz visual y la infraestructura ya que gestionan las factorías.

Están organizados de la siguiente manera:

- **Admin-list:** componente y gestión de factoría del listado de entidades en el administrador.
- **Edit.ts:** Componente y gestión de factoría de formulario de edición del administrador.
- **Home.ts**: Componente y gestión de factoría de la vista principal de la entidad en la parte pública**.**
- **List.ts:** Componente y gestión de factoría de la vista pública de listado de la entidad si la hay.
- **New.ts:** Componente y getión de factoría del formulario de nuevo elemento en el admin

Cada uno contiene un web component creado con Litelemente. Un ejemplo:

```
import { LitElement, html, customElement, property } from "lit-element";
import { getId } from "../../../utils/utils";
import { CategoryRepositoryFactory } from "../infrastructure/category-
repository-factory";
import { Category } from "../domain/category";
import { general } from "../../../../styles/general";
import { PostRepositoryFactory } from "../../posts/infrastructure/post-
repository-factory";
import { publicStyles } from "../../../../styles/public";
import { Router } from "@vaadin/router";

@customElement("category-home-c")
export class CategoryHome extends LitElement {
  categoryRepository = CategoryRepositoryFactory.build();
  postRepository = PostRepositoryFactory.build();
  @property() category: Partial<Category>;
  @property() posts = [];
  public static styles = [general, publicStyles];

  render() {
    return this.category && this.category.title
      ? html`
          <div>
            <h1>${this.category.title}</h1>
            ${this.category.img
              ? html`
                  <div class="img-container featured">
                    <img
                      src="${process.env.API_URI +
                        "/uploads/" +
                        this.category.img}"
                      alt="${this.category.title}"
                      title="${this.category.title}"
                    />
                  </div>
                `
              : null}
            <div class="brief">${this.category.brief}</div>
            <div class="description">${this.category.description}</div>
            ${this.posts.length > 0
              ? html`
                  <ul class="modules-container">
                    ${this.posts.map(post => {
                      return html`
                        <li><post-module-c .post="${post}"></post-module-
c></li>
                      `;
                    })}
                  </ul>
                `
              : ""}
            <slot></slot>
          </div>
        `
      : null;
  }
```

```
  async connectedCallback() {
    super.connectedCallback();
    await new Promise(r => setTimeout(r, 0));
    const id = getId();
    !id
      ? Router.go("/not-found")
      : await this.categoryRepository.getById(getId()).then(response => {
          this.category = response;
          if (response.posts.length > 0) {
            response.posts.map(async post => {
              this.postRepository.getById(post).then(response => {
                this.posts = [...this.posts, response];
              });
            });
          }
        });
  }
}
```

A continuación, detallo la estructura de un componente utilizando en componente **category-home-c** como ejemplo[39]:

La primera parte son las importaciones necesarias, de librerías necesarias, todas las que empiezan con import, el objeto a importar y la ruta desde donde se importa. Por ejemplo **import { PostRepositoryFactory } from "../../posts/infrastructure/post-repository-factory",** importa la clase PostRepositoryFactory desde la URL que sigue a from.

La segunda parte es la declaración del componente que extiende de litElement. Se declara con el decorador @customElemente que hemos importado anteriormente en esta línea: **import { LitElement, html, customElement, property } from "lit-element".** Con ese decorador definimos el nombre del componente: “category-home-c” y lo iniciamos con el nombre de la clase que le damos: **CategoryHome,** extendiendo de la clase LitElemente que hemos importado.

```
@customElement("category-home-c")
export class CategoryHome extends LitElement {
```

Dentro, hay otras 3 partes diferenciadas:

Declaración de variables, con llamadas a clases factorías:

```
categoryRepository = CategoryRepositoryFactory.build();
postRepository = PostRepositoryFactory.build();
```

[39] En la documentación de LitElement se puede consultar la estructura de los componentes de LitElement similar a lo que se ha implementado en esta aplicación

Declaración de propiedades del componente, se hace con el decorador property:

```
@property() category: Partial<Category>;
@property() posts = [];
```

Declaración de estilos en la variable pública estática *styles*

```
public static styles = [general, publicStyles];
```

El método que renderiza el html con el html incluyendo los contenidos dinámicos que se utilicen en las vistas, con llamadas *JS* (para incluir contenidos javascript se usa ` ${contenido js}`):

```
render() {
```

Otros métodos, en los que cabe resaltar los métodos del ciclo de vida de los componentes de litElement[40], principalmente connectedCallback que se ejecuta cuando se carga el componente, y requestUpload() que solicita una actualización del componente

```
async connectedCallback() {
  super.connectedCallback();
  await new Promise(r => setTimeout(r, 0));
  const id = getId();
  !id
    ? Router.go("/not-found")
    : await this.categoryRepository.getById(getId()).then(response => {
        this.category = response;
        if (response.posts.length > 0) {
          response.posts.map(async post => {
            this.postRepository.getById(post).then(response => {
              this.posts = [...this.posts, response];
            });
          });
        }
      });
}
```

Shared

.
├── auth
├── emptyObjects
├── http

[40] https://lit-element.polymer-project.org/guide/lifecycle

```
└── id
```

Además de las carpetas de cada entidad, hay una carpeta shared, que contiene:

- **Auth**: Métodos que contrlan la autenticación
- **http:** métodos para gestionar la autenticación http.
- **Empty-objects**: Definiciones de objetos vacíos de cada entidad que usamos en la carga inicial vacía de cada componente.
- **Id:** interface que define un id.

Utils

En esta carpeta incluyo métodos utilitarios que uso en toda la aplicación:
Contiene:

- **seralizeForm**: Para obtener los datos de un formulario y devolverlos en forma de objeto JSON
- **getID**: método que obtiene el id de la URL en forma de parámetro
- **getAdminUrl**: método que trasforma la URL en una URL para el administrador de la misma entidad
- **countErrors**: método que contabiliza los errores de validación de un formulario
- **setTitleDescription:** setea el title y la descripción de una página en función de las variables que le damos.

Styles

```
├── admin-styles.ts
├── general.ts
├── index.ts
├── public.ts
└── theme.ts
```

En esta carpeta he incluido los estilos generales de la aplicación (general.ts), los estilos de la parte pública (public.ts) y de la parte privada (admin-styles.ts) y las variables CSS[41] (theme.ts).
Estas variables a su vez tienen variables generales y varibales diferenciadas por tema claro u oscuro de tal manera que cuando en el componente principal se cambia la clase de light a dark o viceversa, se aplican estas variables y el tema cambia de look claro a oscuro o viceversa.

[41] https://developer.mozilla.org/es/docs/Web/CSS/Using_CSS_custom_properties

4.5.3 Autenticación

Este CMS consta de una parte pública y otra privada. La parte privada o de administración necesita asegurar que el usuario está registrado en la aplicación. Hay rutas como gestionar nuevos usuarios, gestionar categorías o gestionar *site* que exigen que el usuario sea administrador. Si no se cumplen los requerimientos estas rutas redirigen a la portada.

En cada ruta que enviamos al *back-end* se incluye el *token* del usuario, guardado en la memoria local, si existe, para que el servidor nos gestione las autorizaciones y nos devuelva una respuesta en cada caso.

Para enviarlo en todas las rutas genero el método http, con axios que incluye ese bearer token

```
import axios from "axios";
import { AuthorizationService } from "../auth/authorization-service";

const server = process.env.API_URI;

const authorizationService = new AuthorizationService();
const token = authorizationService.getToken();

export const http = axios.create({
  baseURL: server,
  headers: {
    Authorization: "Bearer " + token
  }
});
```

Este *token* se guarda tras hacer *login*, para ello tenemos un servicio en los servicios del usuario que conectan la ruta del back-end del *login*, enviando el usuario y contraseña y devolviendo el *token* del usuario si el usuario está registrado o un mensaje de error si no lo está. Una vez recogido el *token*, sabemos que el usuario está logueado, usamos ese *token* en determinadas rutas securizadas del *administrador* para comprobar los privilegios necesarios.

Esta autenticación se gestiona con lo métodos que hay en la carpeta **auth**, dentro de **featurad**, en **src.**

Tenemos un servicio con dos métodos:

- **pageEnabled**: controla que el usuario está logueado, sino redirecciona a la portada
- **padeAdminEnabled**: controla que el usuario es administrador, sino redirecciona a la portada

Estos dos métodos a su vez llaman a **userService**, donde tenemos dos métodos, **thisIsLogged**() y **thisIsAdmin**() que se comunican con el servidor para comprobar si el usuario está logueado o es administrador.

4.5.4 Notificaciones

Las notificaciones de usuario de realizan mostrando el componente que se encuentra en notice.ts en la carpeta notice en los componentes visuales.

Para hacer estas notificaciones dinámicas, he usado una función que se encuentra en utils.ts que he llamado notify:

```
export const notify = (type: string, message: string, e?: HTMLElement |
null) => {
  const ev = new CustomEvent('notification-event', {
    detail: { message: message, type: type },
    bubbles: true,
    composed: true
  })
  return e ? e.dispatchEvent(ev) :
document.body.querySelector('div')!.dispatchEvent(ev)
}
```

Esta función emite un evento custom, con el mensaje a notificar y el tipo. Este evento es recogido desde app de esta manera:

```
this.addEventListener('change-theme', this._handleChangeTheme)
```

Y le pasa los parámetros a dos propiedades, que a su vez son reactivas en el componente notificación, que reacciona al cambio de mensaje, mostrándose durante dos segundos.

4.5.5 Títulos y meta descripciones

Los títulos y los meta descriptions de las páginas públicas se setean ejecutando la función que he llamado *setTitleDescription*, cuyos parámetros son el *title* y el *description*, en la página de portada, de post y de categoría.

```
export const setTitleDescription = (title: string, description: string) =>
{
  document.title = title
  let allMetaElements = document.getElementsByTagName('meta')
  for (var i = 0; i < allMetaElements.length; i++) {
    if (allMetaElements[i].getAttribute('name') == 'description') {
```

```
        allMetaElements[i].setAttribute('content', description)
        break
      }
    }
}
```

4.5.6 Destacados

Todos los posts y categorías se pueden destacar. Las categorías destacadas, aparecen en el slider de portada, y debajo aparecen los posts destacados. Esta funcionalidad se encuentra en los formularios de edición y creación de posts y categoría y se usan para filtrar los posts y categorías de la portada

4.6 Arquitectura de la información y diagramas de navegación

La aplicación en el entorno de cliente consta de una parte pública y otra privada para la que es necesario hacer login. Para explicar mejor la arquitectura de la información ilustro ambas partes por separado:

4.6.1 Parte pública

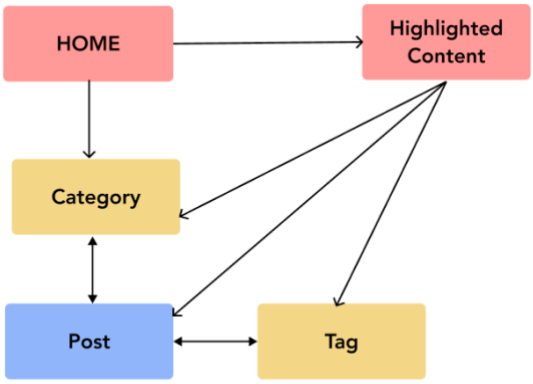

Figura 8 diagrama de navegación de parte pública

En la parte pública, partimos de una portada, con contenido destacado, en donde a través de la cabecera podremos navegar entre diferentes categorías, y a su vez podremos navegar de forma directa a las diferentes partes destacadas seleccionadas para la portada, *tags*, categorías o *posts*.
En cada post a su vez podremos navegar hacia sus *tags* o su categoría.

4.6.2 Parte privada

La parte privada del administrador sigue la siguiente estructura de navegación:

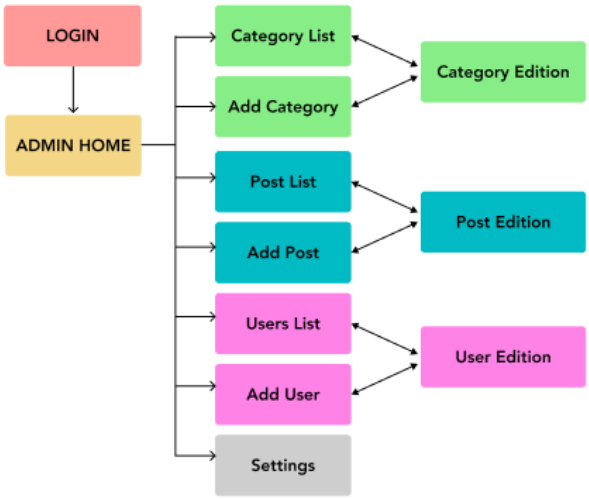

Figura 9 Estructura de navegación privada

4.7 Diseño gráfico e interfaces

4.7.1 Estilos

El estilo del tema principal de este CMS en la parte pública es **simple y minimalista**. Inspirado en webs como ***Medium***[42], con fondo blanco, cuerpo contenido, y con toda la importancia visual delegada en las tipografías. Este CMS abre la posibilidad de utilizar otro tema en un futuro para hacerlo más flexible.

[42] https://medium.com/

En la **parte privada** usa un estilo también minimalista, pero con una gama de grises más propia de un panel de administración inspirado ligeramente en *WordPress*, pero con identidad propia.

Logotipo

Denotate

Figura 10 Logotipo del CMS

El nombre de este CMS es ***Denotate***, y la web pública donde se promociona se aloja en ***denotate.me***

Este nombre está construido haciendo alegoría a *Deno*, *notation* (como *MarkDown notation*) y denote (denotar en inglés), con la extensión me.

Para su construcción he usado dos tipografías ***Google Fonts***[43]:

- *Josefina Sans*
- *Cardo*

Colores

La paleta de colores es muy minimalista. El contenido de la parte pública usa blanco de fondo y gris oscuro y negro para el texto. Concretamente el gris #333 y el negro #000

Figura 11 Escala de grises en **Deno**tate

[43] https://fonts.google.com/

He implementado una funcionalidad que permite elegir un color principal que será el que se use para determinados elementos.

Tipografías

Denotate usa las tipografías que componen su logotipo, ***Josefine Sans***[44] y ***Cardo***[45].

Su combinación confiere simplicidad y elegancia.

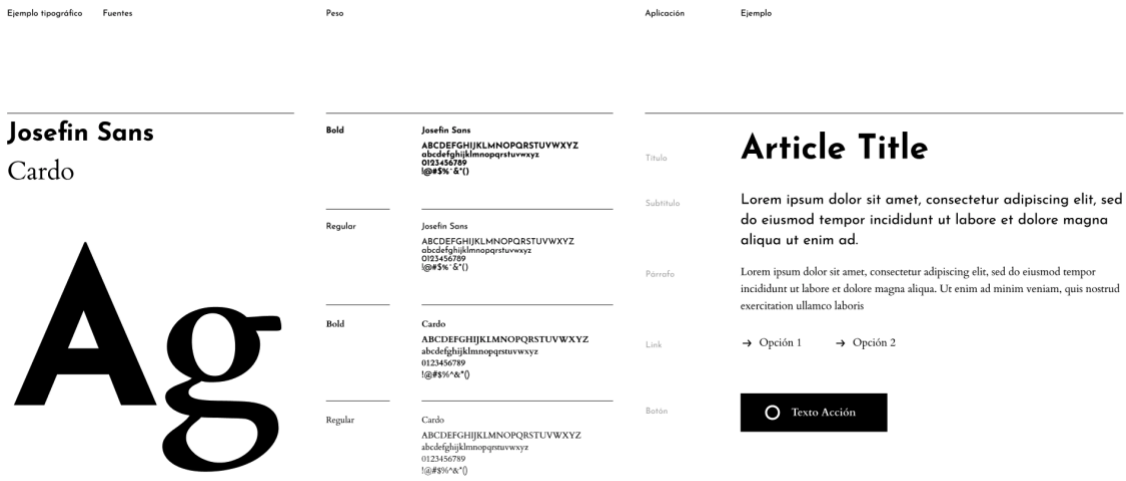

Figura 12 Aplicación de tipografías en el CMS

Denotate usa una escala tipográfica para estructurar los títulos tipo ***Perfect Four***[46]

---

[44] https://fonts.google.com/specimen/Josefin+Sans#standard-styles

[45] https://fonts.google.com/specimen/Cardo?query=Cardo

[46] Se puede ver esta escala por ejemplo en Typescale: https://type-scale.com

Lorem ipsum dolor (0.75em)

Lorem ipsum dolor (1em)

Lorem ipsum dolor (1.333em)

Lorem ipsum dolor (1.777em)

Lorem ipsum dolor (2.369em)

Lorem ipsum dolor (3.157em)

Lorem ipsum dolor (3.157em)

Figura 13 Escala tipográfica titulares

Botones

Los botones son transparentes y usan el color principal, con fuente Josefin, el estado hover invierte los colores, con el color principal de forma solida ocupando toda la caja:

Figura 14 Botones en **Denotate**

Iconos

He optado por usar emojis[47] como iconos, debido a que todos los navegadores los interpretan, son universales y todo el mundo está acostumbrado a ellos, son nativos como el resto de la aplicación y no tienen peso.

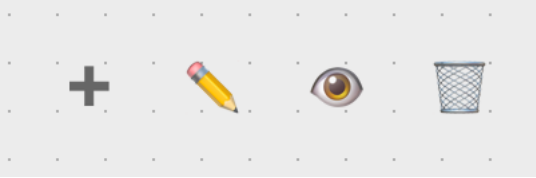

Figura 15 Iconos emoji utilizados

[47] https://getemoji.com/

4.7.2 Usabilidad /UX

La usabilidad en la **parte pública** se asienta en la facilidad de lectura, contraste de colores y organización del contenido.

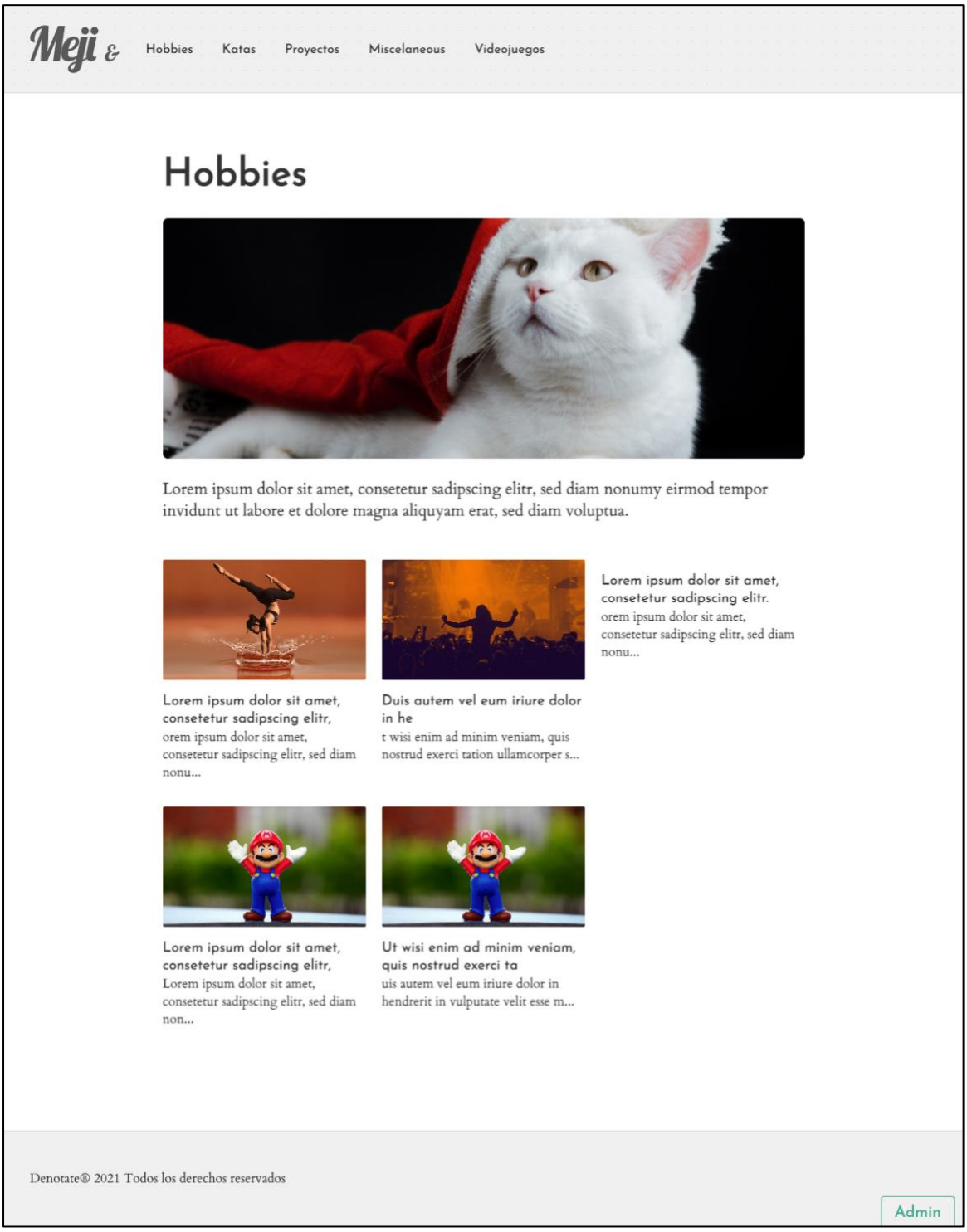

Figura 16 Diseño de la parte pública de una categoría del CMS

Los **textos del cuerpo** están contenidos en una columna no más grande de 800px de ancho para facilitar una lectura cómoda. Los párrafos y títulos están separados, y se usa una escala de espacios basada en 8px (8 - 16 - 24 - 32 - 40 - 48 - 56 - 64 - 72).

El **menú** se encuentra en la parte superior, y acompaña en todo momento al hacer *scroll*, haciendo accesible la navegación, con estados diferentes para *mouse hover* y categoría activa.

Figura 17 Diseño mobile de una categoría

En **la versión móvil este menú** pasa a ser un menú con icono hamburguesa, que se abre a toda pantalla para aprovechar todo el espacio.

Los **botones** son minimalistas, con el color principal seleccionado, pero fondo transparente que se invierte en el hover.

Los ***links*** se subrayan al hacer *mouse hover* y usan una tipografía diferente para identificarlos, además de usar el color negro, más resaltado que el gris oscuro de los textos.

POST

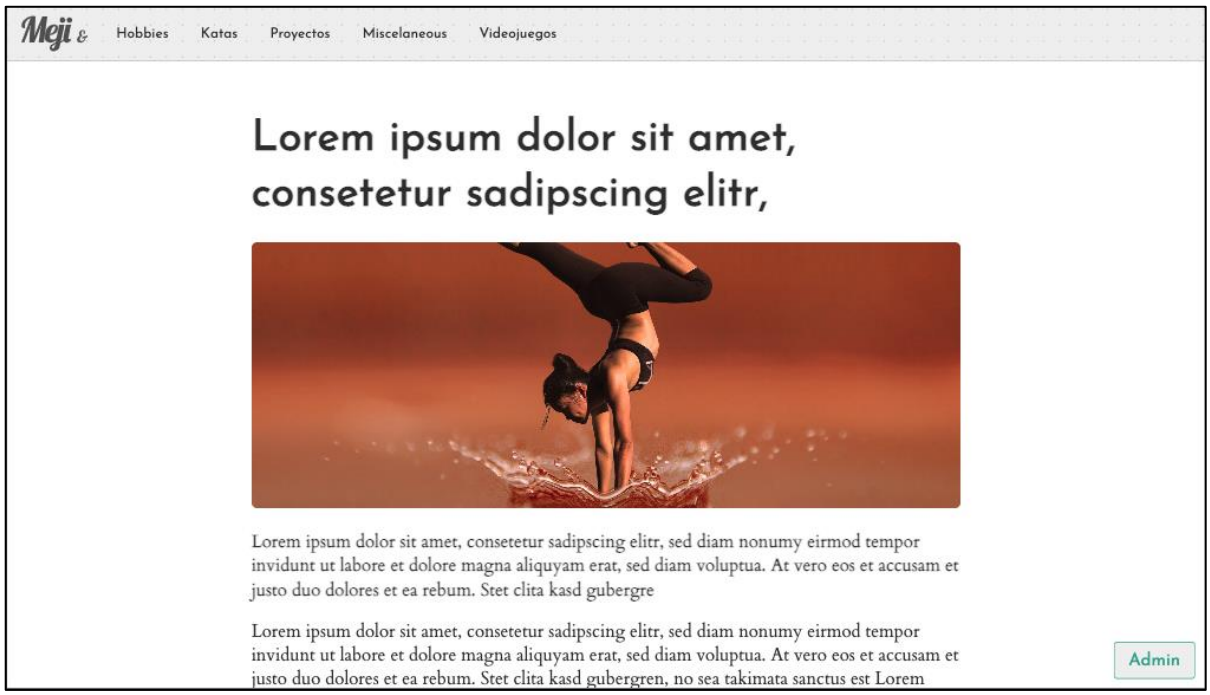

Figura 18 Diseño de un post Desktop

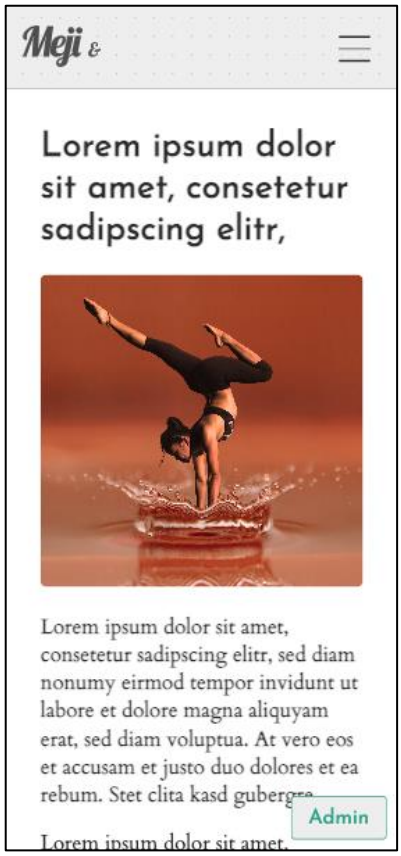

Figura 19 Diseño post mobile

ADMIN - Entidades

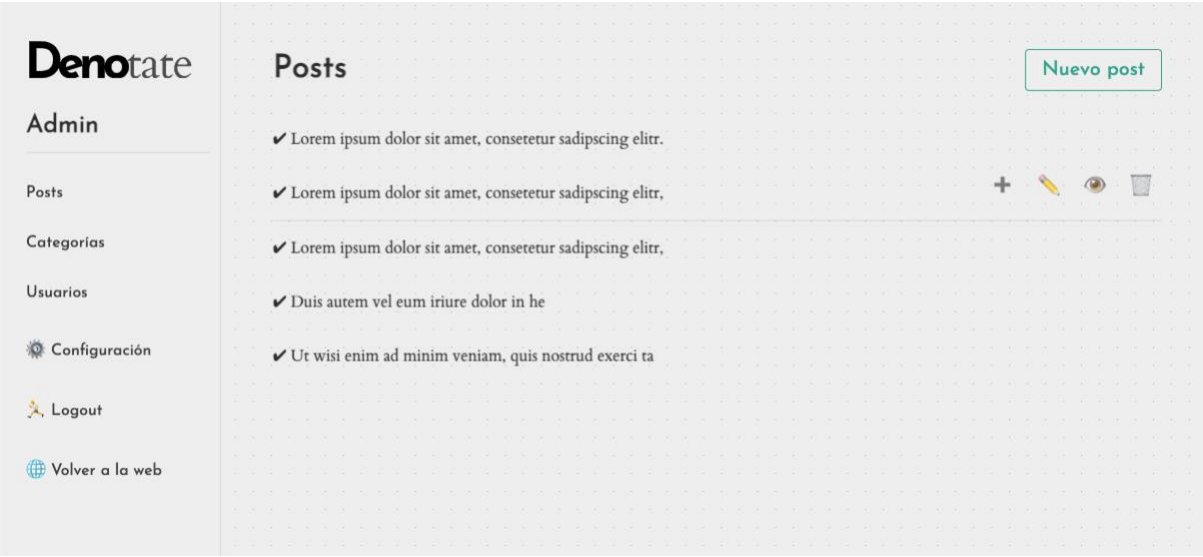

Figura 20 Diseño del administrador, vista de posts

Figura 21 Administrador versión mobile

En la **parte privada** he replicado la fórmula que utiliza *WordPress*: en desktop el menú está a la izquierda, y la edición de contenidos a la derecha. En *Mobile* este menú está bajo un menú hamburguesa que se abre lateralmente, ya que en la parte privada no nos interesa perder el contexto en todo momento.

ADMIN – Edición de entidad

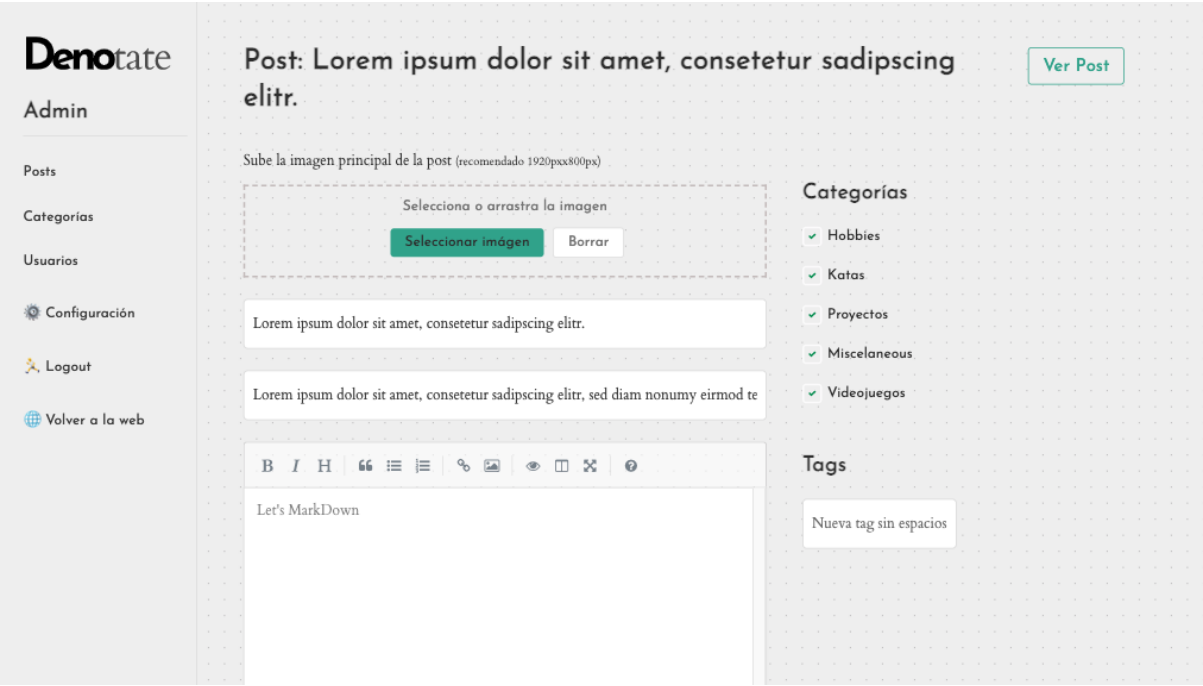

Figura 22 Administrador vista de edición de post en modo *desktop*

Figura 23 Edición de entidad vista mobile

4.8 Lenguajes de programación y *APIs* utilizados

En el front-end y en el back-end el lenguaje principal de programación utilizado para el desarrollo es Typescript, que a su vez es una versión tipada de Javascript. Al ser un lenguaje que no se interpreta directamente en el navegador, se compila en Javascript.

4.8.1 Back-end

Este CMS se construye sobre el entorno ***Deno***, usando el ***framework Alosaur*** para crear el back-end y la *API REST* que hemos explicado en la sección de diseño. La elección de este entorno de programación, como ya se ha explicado anteriormente en esta memoria, se fundamenta en la simplicidad de este entorno, y en el uso directo de los recursos y estándares ya implementados en los navegadores modernos.

La base de datos que usa este CMS es documental, concretamente tipo ***MongoDB***[48]. Para crear los modelos usaremos una librería que se llama ***DenoDB***[49], compatible con bases de datos relacionales también. Esto nos da la flexibilidad de poder modificar la aplicación posteriormente si fuera necesario, incluso de poder hacerla flexible ofreciendo poder conectarse a diferentes tipos de bases de datos.

4.8.2 Front-end

El desarrollo de las vistas y de la funcionalidad *front-end* se realiza usando la librería ***LitElement***[50]. Su elección se fundamenta en la ligereza y simplicidad de esta librería, pero principalmente en el uso que hace de *web components*, usando directamente estos componentes que el motor de navegadores renderiza de forma directa, sin el uso de librerías o renderizados intermedios como otras librerías, tipo *Vue* o *React*; además ofrece la utilización de *ShadowDOM* para compartimentalizar las CSS en los diferentes *web components.*

4.8.3 Herramientas de diseño

Todos los diseños, de diagramas, de los elementos que configuran el diseño de esta aplicación, de la usabildad y las tipografías y de todos los elementos gráficos, se han elaborado con ***Figma***[51].

---

[48] https://www.mongodb.com/

[49] https://eveningkid.github.io/denodb-docs/

[50] https://lit-element.polymer-project.org/

[51] https://www.figma.com/

4.8.4 Editor de código

La edición del código para el desarrollo de esta aplicación se ha realizado con el editor ***Webstorm***[52]. Existe una versión gratuita para estudiantes, y otra gratuita, early-bird con las últimas novedades de este editor. Además, he usado el *plugin* de *Deno* y de *LitElement* para *Webstorm*.

4.8.5 Repositorio

Todo el código utilizado para el desarrollo de este CMS se encuentra en dos repositorios de ***GitHub***[53]

El *back-end* Se puede consultar y descargar desde:

https://github.com/meji/denotate

El *front-end* se puede consultar y descargar en:

https://github.com/meji/denotatefront

[52] https://www.jetbrains.com/es-es/webstorm/

[53] https://github.com/meji

5 Implementación

5.1 Requisitos de instalación

Para poder instalar este CMS necesitamos:

- Crear una base de datos documental MongoDB y obtener su URI
- Subir a un servidor el back-end con entorno Deno, seteando las variables de entorno
- Subir a un servidor el front-end con entorno NODE seteando las variables de entorno

Se pueden utilizar servidores robustos, que permitan usar un entorno de desarrollo Deno, y NODE como Heroku, o AWS [54]entre otros. En los siguientes puntos explico como se configura esta subida a un servidor en Heroku y como se crea una base de datos en Atlas Mongo obteniendo la URI que tenemos que indicar en la variable de entorno del back-end

5.2 Instrucciones de instalación

5.2.1 Creación de base de datos *MongoDB* con Atlas

Para usar este *CMS* necesitamos una base de datos tipo documental *MongoDB*. Podemos crear esta base de datos en *Cloud Atlas*[55] de *Mongo.*

Para ello creamos una cuenta en Atlas, podemos hacerlo rellenando los datos o con *login* de *Google.*

Una vez creada tenemos que crear un proyecto:

[54] https://aws.amazon.com/es/

[55] https://www.mongodb.com/cloud/atlas

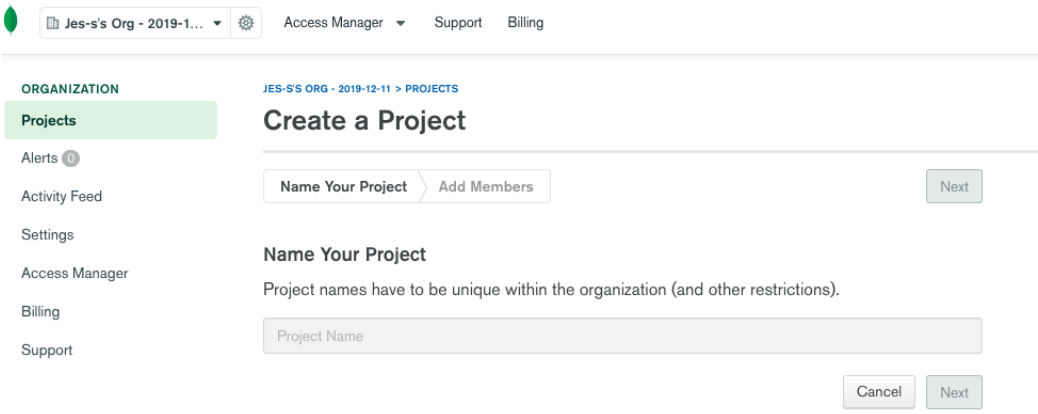

Figura 24 Creación de proyecto en Atlas

Tras crear el proyecto, crearemos un *cluster*. *Atlas MongoDB* nos permite un *cluster* gratis con algunas limitaciones, elegiremos ese.

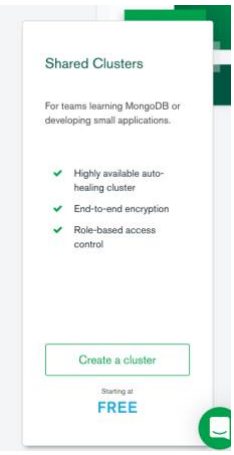

Figura 25 Free Cluster

Necesitamos la *URI* de la base de datos, para ello una vez creado el *cluster*, hacemos click en *connect*, y tendremos que configurar primero los permisos, le daremos a “Allow Access from Anywhere” y después crearemos un usuario para la base de datos:

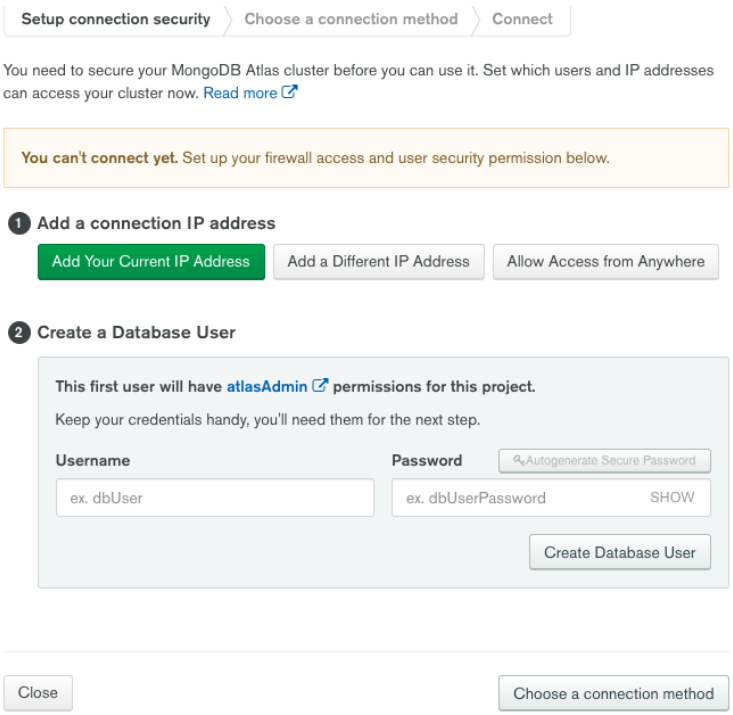

Figura 26 *Interfaz* de configuración de conexión de *Atlas MongoDb*

Una vez introducido elegiremos *connect your aplication*, en esa ventana nos dará la *URI*, con dos parámetros a rellenar, que son el *user* y la *password* que creamos en el caso anterior:

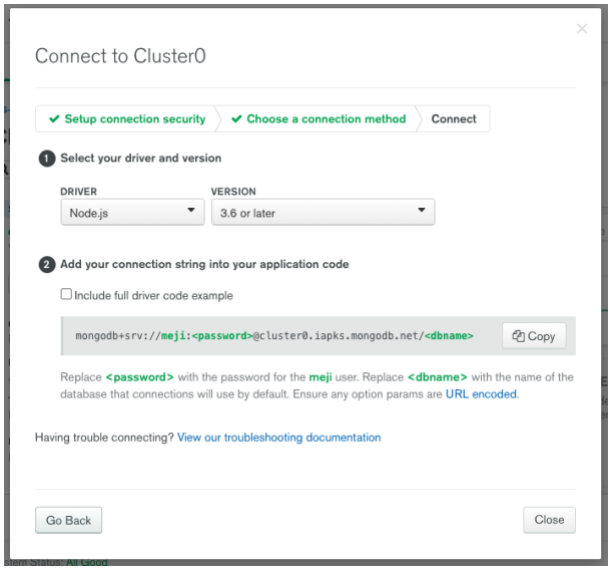

Figura 27 Ventana con la URI para la conexión de la base de datos

Con esta *URI* completada ya tendríamos lo que necesitamos para iniciar la conexión a la base de datos del *back-end*

5.2.2 Instalación de *back-end* con *Deno* de forma local, servidor *Deno*

Para instalar el *back-end* en un servidor local, debemos instalar[56] previamente *Deno*, para ello desde la terminal escribimos:

```
curl -fsSL https://deno.land/x/install/install.sh | sh
```

Una vez instalado *Deno* tenemos que configurar nuestras variables de entorno:

```
Deno_ENV="DEV"
Deno_HOST="localhost"
DB_NAME="nombre de la base de datos"
DB_URI="uri de mongo"
SECRET="palabra secreta que queramos"
```

La *db_uri* la obtenemos de *Mongo*, tal como vimos en el apartado anterior

Iniciamos la aplicación desde la carpeta del proyecto con ***deno run***, pero debemos tener en cuenta que *Deno* se ejecuta con los permisos que le demos, por lo que el comando correcto será:

```
deno run --allow-net --allow-env --allow-read --allow-write --
allow-plugin --config tsconfig.app.json --cached-only --unstable
main.ts
```

[56] Podemos ver más opciones de instalación en https://deno.land/#installation

--allow-net --allow-env --allow-read --allow-write --allow-plugin son las flags que determinan los permisos necesarios. **–config** es la flag que determina el archivo de configuración de Typescript. **–cached-only y –unstable** son flags de caché y para librerías inestables aún, y **main.ts** es el archivo principal de la app.

Una vez ejecutado este comando en la terminal veremos el dinosaurio que he configurado para el entorno de desarrollo (variable de entorno dev) en nuestra terminal:

Figura 28 Ventana que nos indica que el servidor se está ejecutando localmente en el entorno de dev

5.2.3 Instalación de back-end con Deno en Heroku

En mi caso he usado *Heroku*[57]. Hay que tener algunas consideraciones al instalar Deno en *Heroku*, ya que de forma predefinida no tiene un bundle de instalación para Deno.

Primero tenemos que hacernos una cuenta en Heroku, y crear una nueva APP:

[57] https://www.heroku.com/

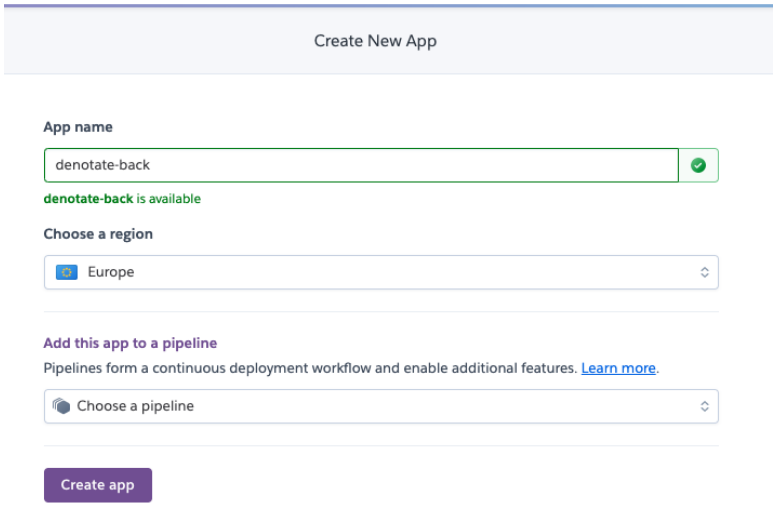

Figura 29 Crear nueva app en Heroku

Vamos a usar nuestro repositorio donde tenemos alojado el *back-end* para sincronizarlo con nuestro servidor de *Heroku*, para ello conectaremos nuestra cuenta de *Github*, y elegiremos el repositorio concreto:

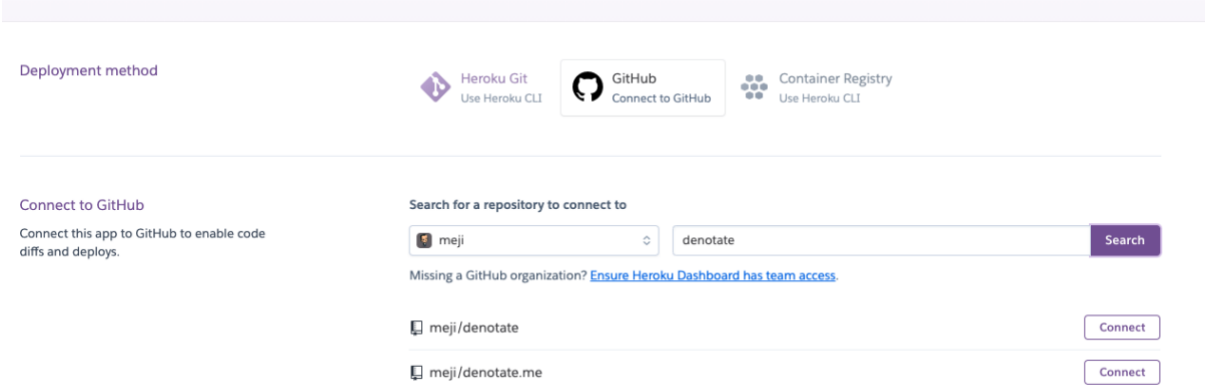

Figura 30 sincronizamos el repositorio donde tenemos el back-end en GitHub

Desde la terminal podemos sincronizar el repositorio, para ello primero tenemos que instalar *Heroku* en nuestro sistema con:

```
npm i -g heroku
```

Después tendremos que sincronizar nuestra app de *Heroku* con nuestro repositorio local, indicando en la ruta donde estamos cual es la app de *Heroku* a sincronizar (nombre-app):

```
heroku git:remote -a nombre-app
```

De esta manera podremos usar los comandos de *Heroku* desde la terminal para gestionar nuestro repositorio en *Heroku*.

Algo muy importante antes de hacer deploy de una aplicación Deno en *Heroku*, es que tenemos que configurar el *buildpack* tipo *Deno* en *Heroku,* para ello iremos a *settings*, y en la zona *buildpacks* tenemos que añadir el *buildpack* de *Deno* que es el siguiente*:*

https://github.com/chibat/heroku-buildpack-deno.git

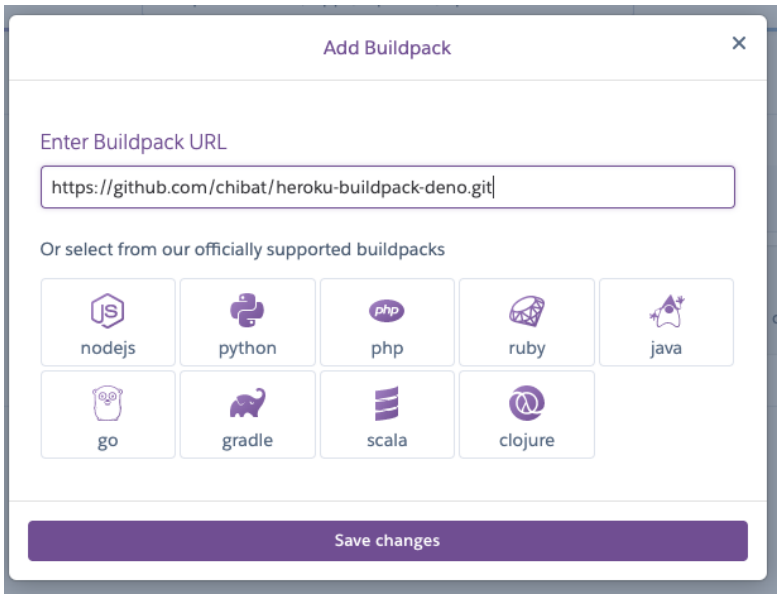

Figura 31 Añadiendo buildpack Deno en Heroku

Heroku nos da un puerto, que necesitaremos usar en vez del puerto que poníamos en nuestra aplicación cuando lo vemos de forma local. Para ello usamos una flag[58] y en nuestro *main.ts* cambiaremos la configuración del puerto para que si hay un puerto de *Heroku* coja ese.

[58] https://deno.land/std@0.79.0/flags

Para ello primero añadimos en nuestro archivo *deps* la librería *parse*:

```
export { parse } from 'https://deno.land/std/flags/mod.ts';
```

Y añadimos configuración en nuestro main.ts para que coja el puerto de *Heroku* si está puesto con la flag o el que está en las variables de entorno:

```
import { parse } from "./deps.ts";


const { args } = Deno;
const argPort = parse(args).port;

await app.listen(`${env.denoHost}:${argPort ? Number(argPort) :
env.denoPort}`);
```

En el back-end tenemos **variables de entorno** en el archivo .env, con variables de configuración. Para configurar estas variables en Heroku, vamos a *settings*, a *reveal config vars* y allí
Configuramos nuestras variables de entorno:

```
Deno_ENV="PROD"
Deno_HOST="URL de nuetro back"
DB_NAME="nombre de la base de datos"
DB_URI="uri de mongo"
SECRET="palabra secreta para usar en las codificaciones"
```

Para la subida en *Heroku*, crearemos un *Procfile*, o archivo de configuración para *Heroku*, donde pondremos como se inicializa la aplicación *Deno* con la *flag* del puerto al final y el resto de *flags* que ya necesitábamos para permisos de la siguiente manera:

```
web: deno run --allow-net --allow-env --allow-read --allow-write
--allow-plugin --config tsconfig.app.json --cached-only --
unstable main.ts --port=${PORT}
```

Podemos ver los *logs Heroku* desde el menú.

Si todo ha ido correctamente, veremos un dinosaurio correspondiente a producción en los logs:

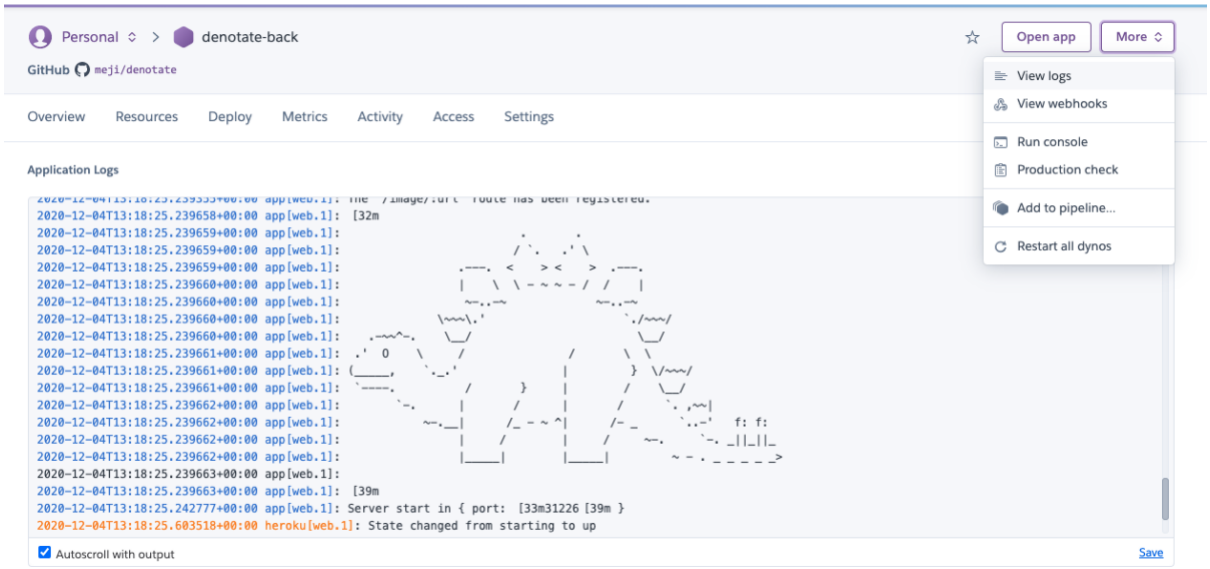

Figura 32 Logs con dinosaurio que indica que el servidor y la *app* se ha iniciado correctamente

5.2.4 Instalación del front-end en entorno local

Para instalar el front-end de forma local tenemos que descargar el repositorio, y desde la raíz del proyecto y con npm instalado, ejecutar **npm i** para que npm instale las dependencias del proyecto en la carpeta node_modules.

Una vez instaladas, debemos configurar nuestras variables de entorno, crearemos un archivo .env en la raíz del proyecto, y pondremos la URL del servidor de back-end de la siguiente manera:

```
API_URI = "http://localhost:8000"
```

En este caso está escrito el servidor de back-end local.

Para iniciar el proyecto, y con todos los pasos anteriores realizados, desde la raíz del proyecto ejecutamos: **npm run start** esto iniciará nuestro proyecto y abrirá una ventana del navegador con la primera pantalla del proyecto.

Para instalar el proyecto en Heroku hay que seguir los mismos pasos que para instalar el back en Heroku, y poner nuestra variable de entorno del servidor de back-end.

6 Demostración y guía de usuario

6.1 Instrucciones de uso

La primera vez que instalamos el CMS en un servidor, y accedemos a él, la aplicación detecta que no hay *site* creado ni usuario *administrador*, y nos lleva a una ventana donde nos solicita crear el primer usuario que será el usuario administrador:

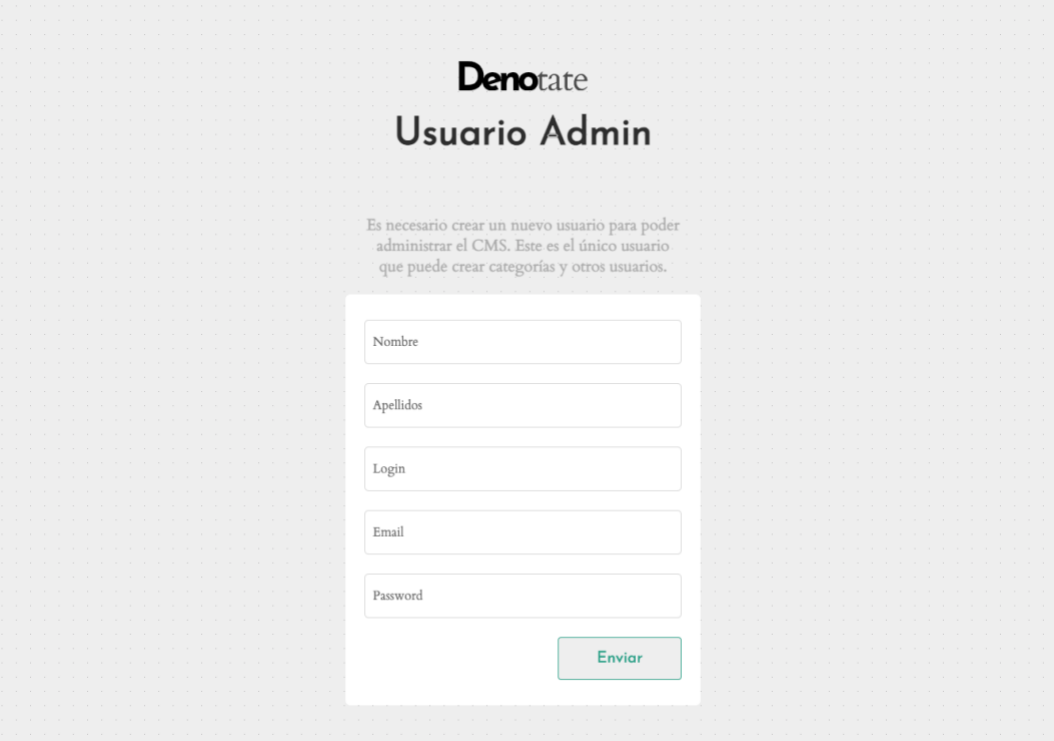

Figura 33 Formulario del primer usuario administrador

Una vez rellenado nos lleva al login para que introduzcamos el login y la contraseña

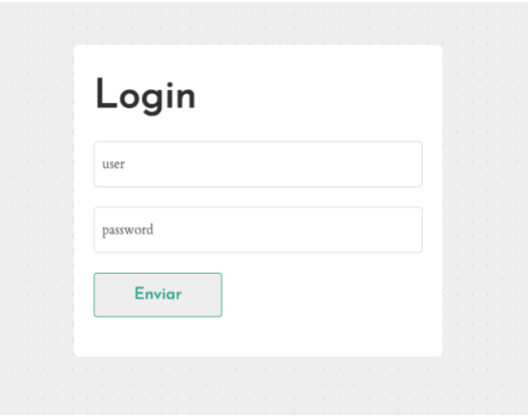

Figura 34 Formulario de Login

Cuando accedemos a la aplicación, después de hacer login, aparecemos en la vista del listado de posts, al no tener ninguno nos indica que configuremos el site o que creemos el primer post o la primera categoría.

Figura 35 Pantalla inicial del administrados sin nada creado

Debemos configurar el site para que se muestre el título y la descripción del site en nuestra portada y demás páginas, allí elegiremos también el color principal que dará color a muchos elementos en la web.

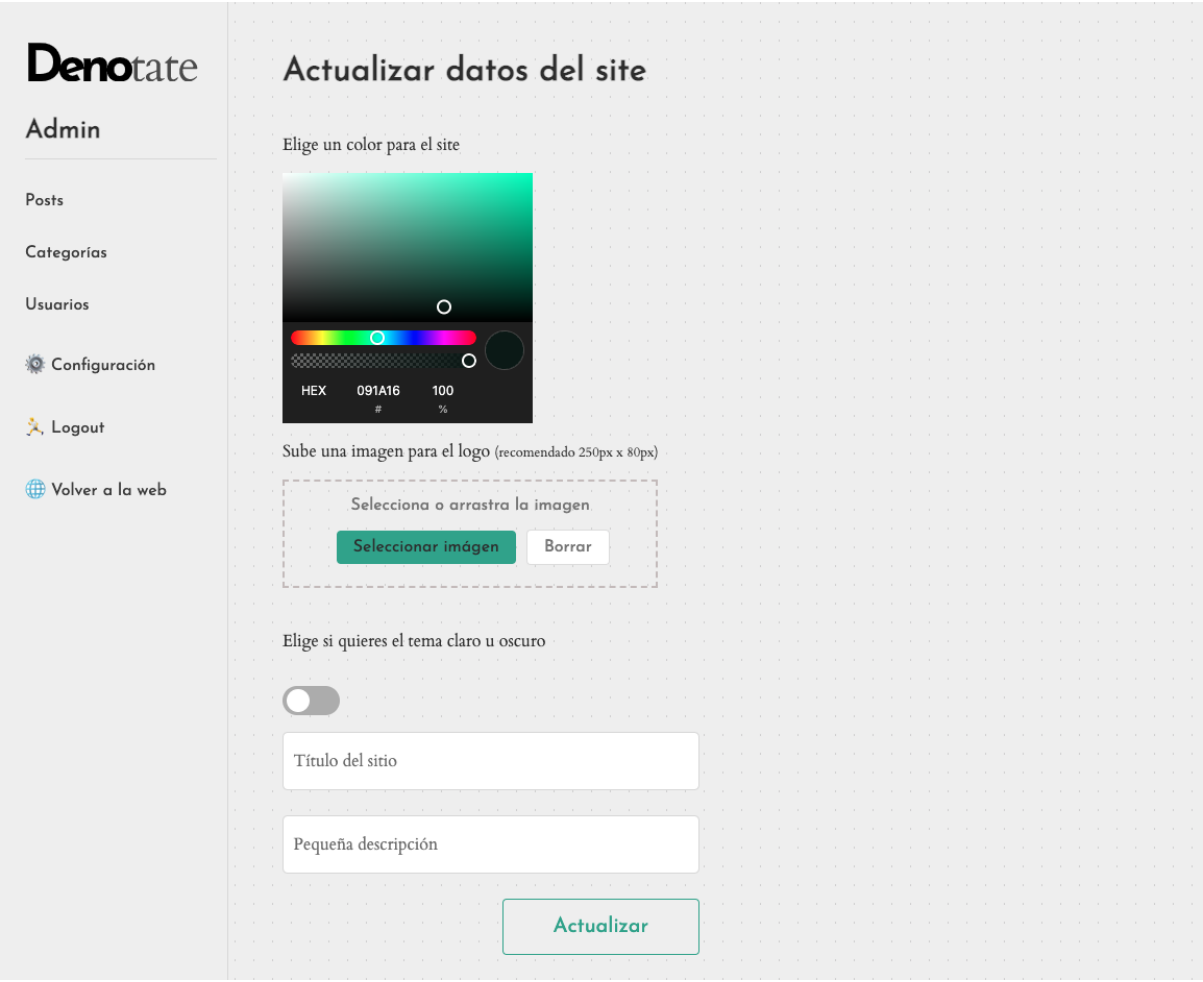

Figura 36 Pantalla de configuración del site

Una vez configurado el site, es recomendable crear las primeras categorías que conformarán el menú de navegación y serán elegibles desde los posts.

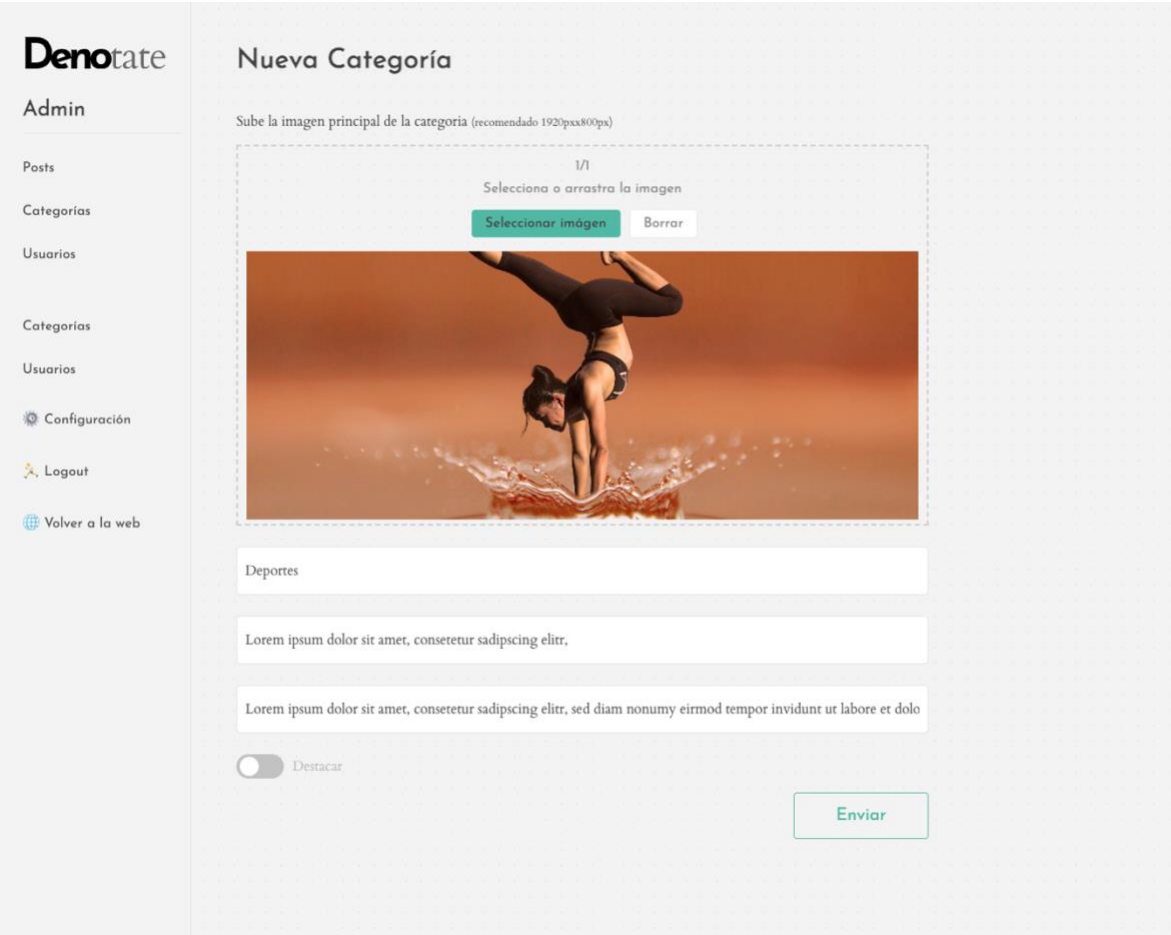

Figura 37 Nueva categoría

Con las categorías creadas, ya podemos crear los posts de la aplicación que aparecerán en las diferentes categorías.

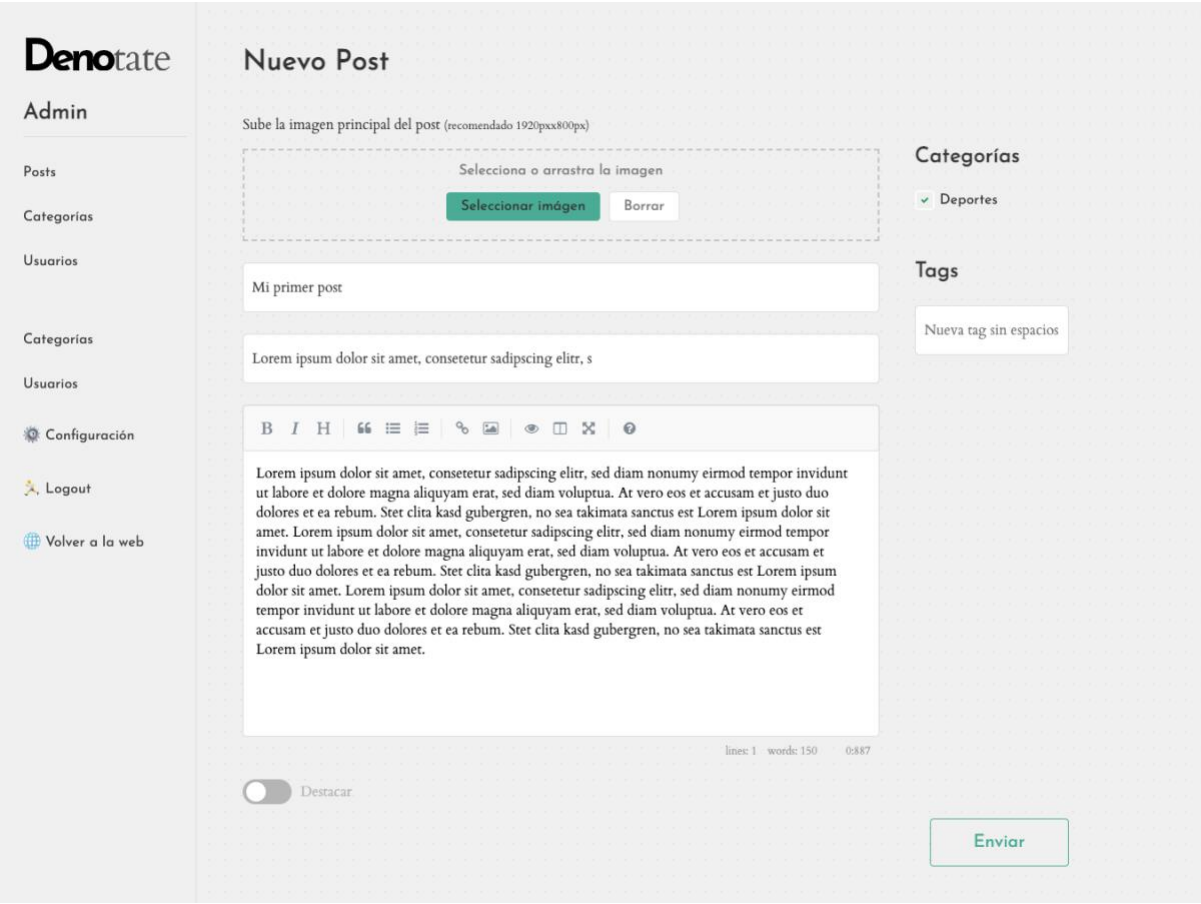

Figura 38 Creación de un nuevo post con categoría deportes creada

En la pantalla de creación o edición de un post, además del contenido del post y las categorías, tenemos dos elementos más:

Tags:

Son las tags asociadas con el post. Las tags que añadamos al post aparecerán en el post y podremos navegar hasta ellas para ver todos los posts que se tienen esa tag asociada.

Destacar

Los posts o categorías que destaquemos aparecerán en la portada.

Una vez creado nuestra primera categoría y nuestro primer post podemos verlos, bien desde el botón de ver post o categoría del administrador en cada caso:

Figura 39 Primer post creado

Hay una funcionalidad extra en la configuración del site que es el tema que queremos si es claro u oscuro. Si elegimos oscuro el administrador y la parte pública se verán en modo oscuro

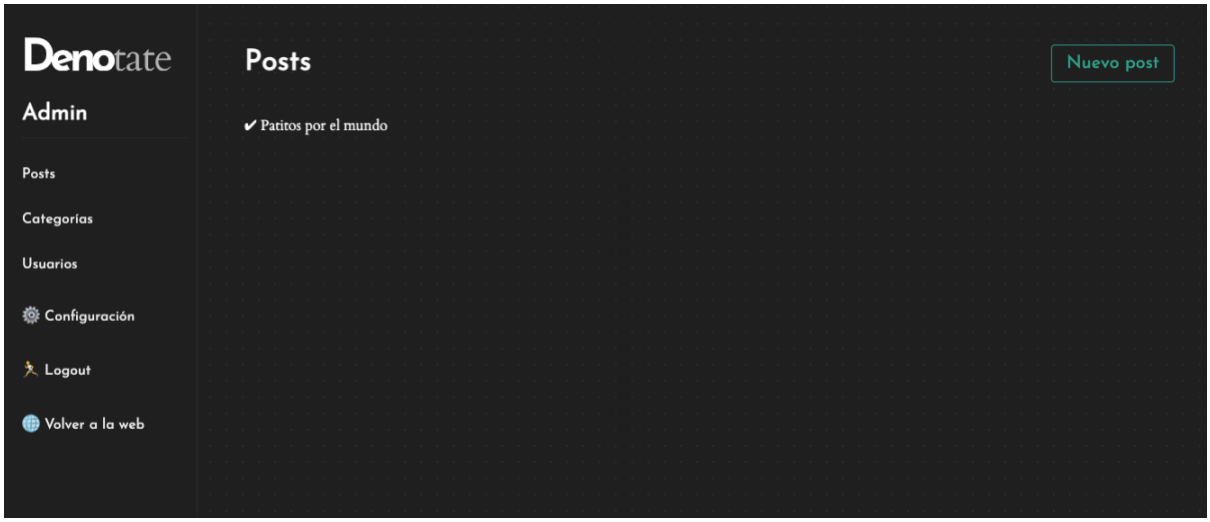

Figura 40 Administrador vista fondo oscuro listado de posts

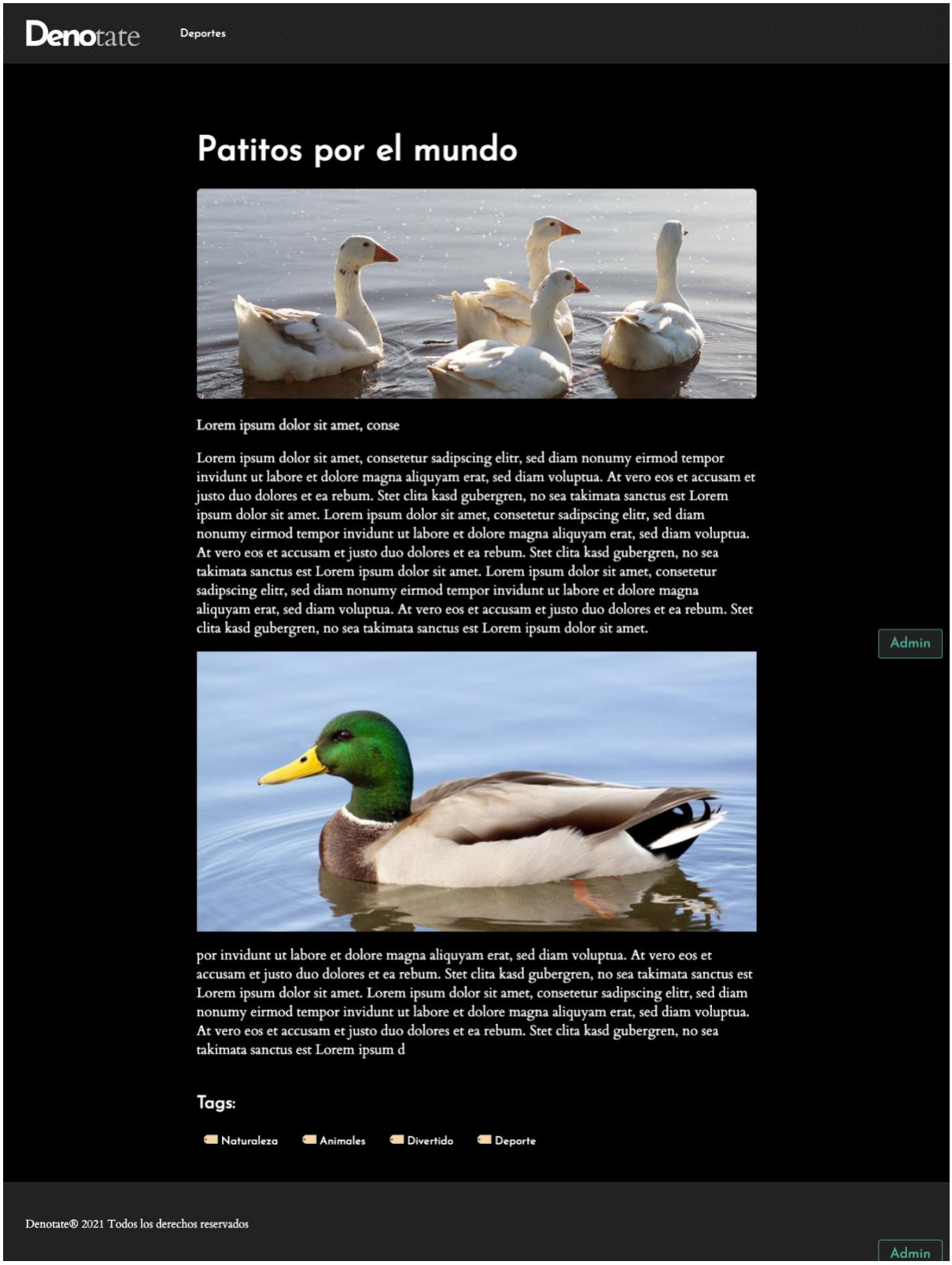

Figura 41 Post con tema oscuro activado

Además de poder crear posts, categorías, cambiar el tema del sitio, gestionar los contenidos o subir un logo propio, podemos crear otros usuarios que podrán escribir artículos, pero no crear más usuarios o categorías, desde la pestaña usuarios del administrador.

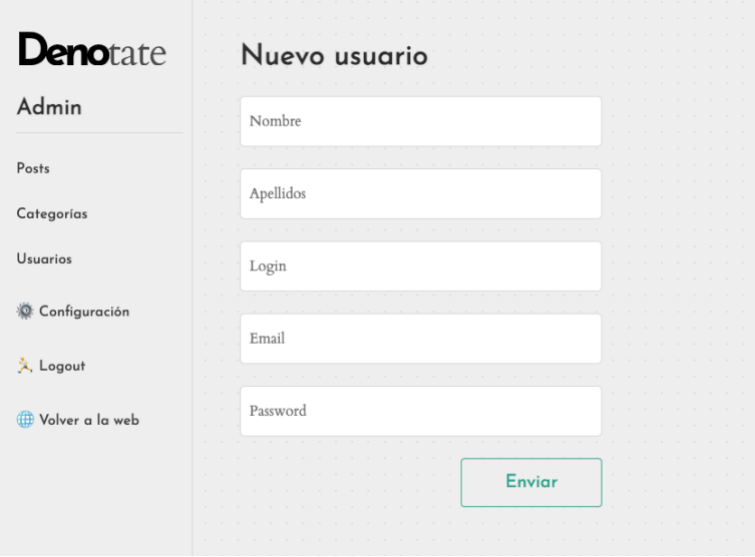

Figura 42 Formulario de creación de nuevo usuario

Todos los listados de categorías, de posts, o de usuarios del administrador tienen 4 acciones disponibles:

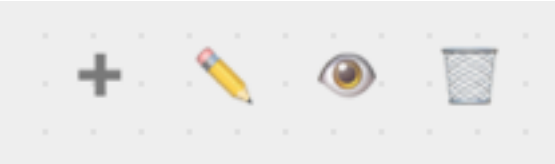

Figura 43 Acciones en los listados del administrador

1. Ampliar información
2. Editar
3. Ver elemento
4. Borrar

Desde la vista pública, si estamos logueados, hay un botón desde el que se accede al administrador, nos llevará al post si estamos en un post, a la categoría si estamos en una categoría o a la portada del administrador (listado de posts).

Admin

6.2 Prototipos

6.2.1 Prototipos *Lo-Fi*

Los prototipos en la primera fase de la construcción de este CMS, comprenden el desarrollo del estilo, el logotipo y las primeras pantallas.

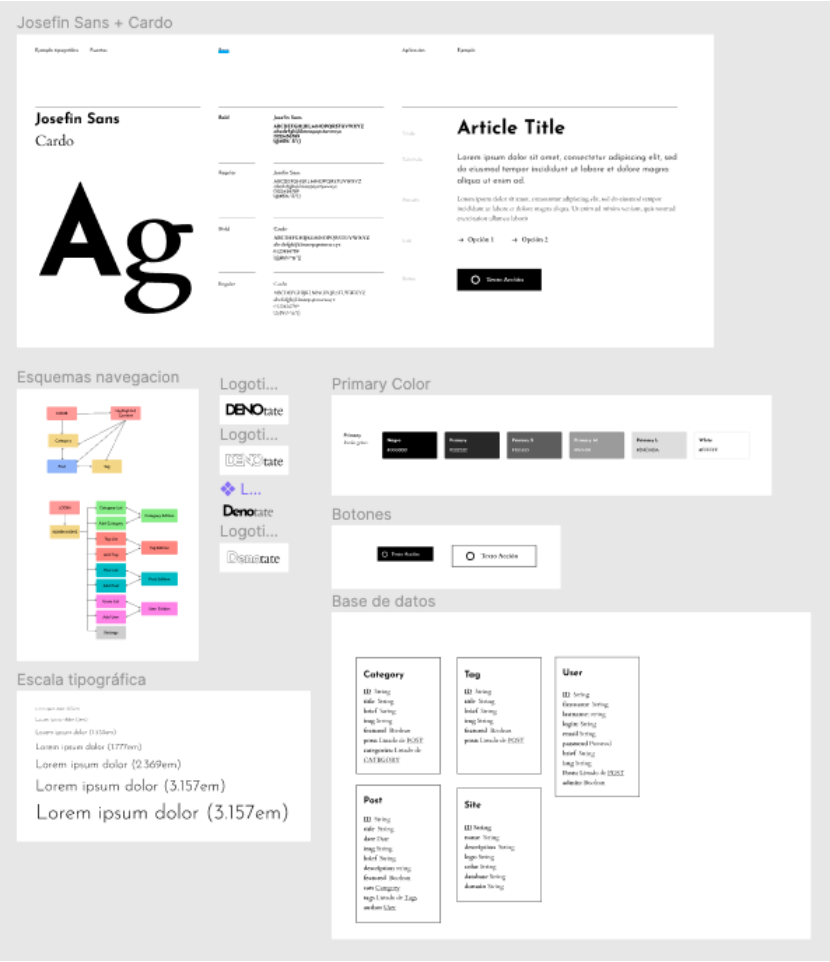

Figura 44 Primeros prototipos de estilo, fuentes, diagramas, colores en *Figma*

Se puedes consultar estos diseños en *Figma* en la siguiente URL:
https://www.figma.com/file/NC7jhTIh2gSiHfZqH0EDqM/DenoTATE.me?node-id=0%3A1

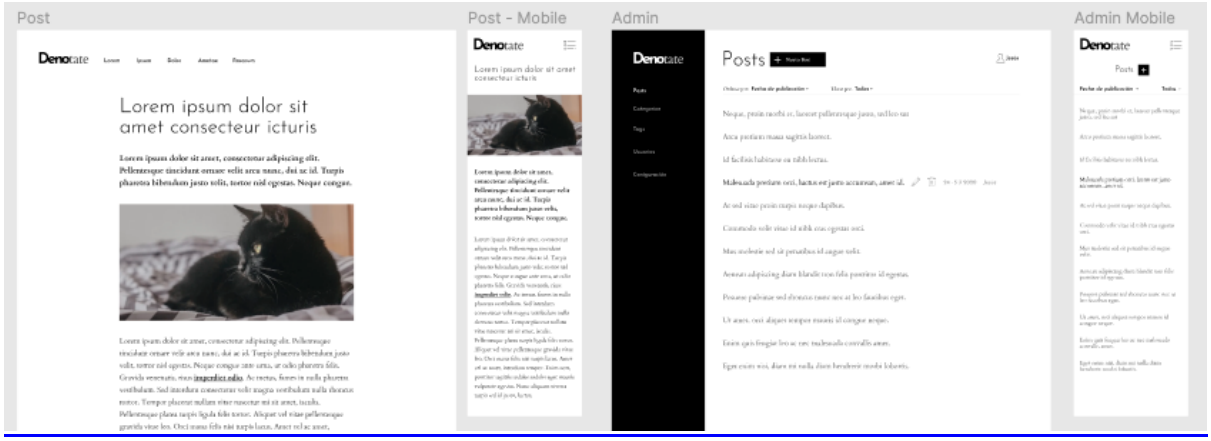

Figura 45 Primeras pantallas diseñadas

6.2.2 Prototipos *Hi-Fi*

Los prototipos *HI-Fi* se dividen en la parte pública y privada, y en cada parte en *mobile* y *desktop*.

CATEGORÍA

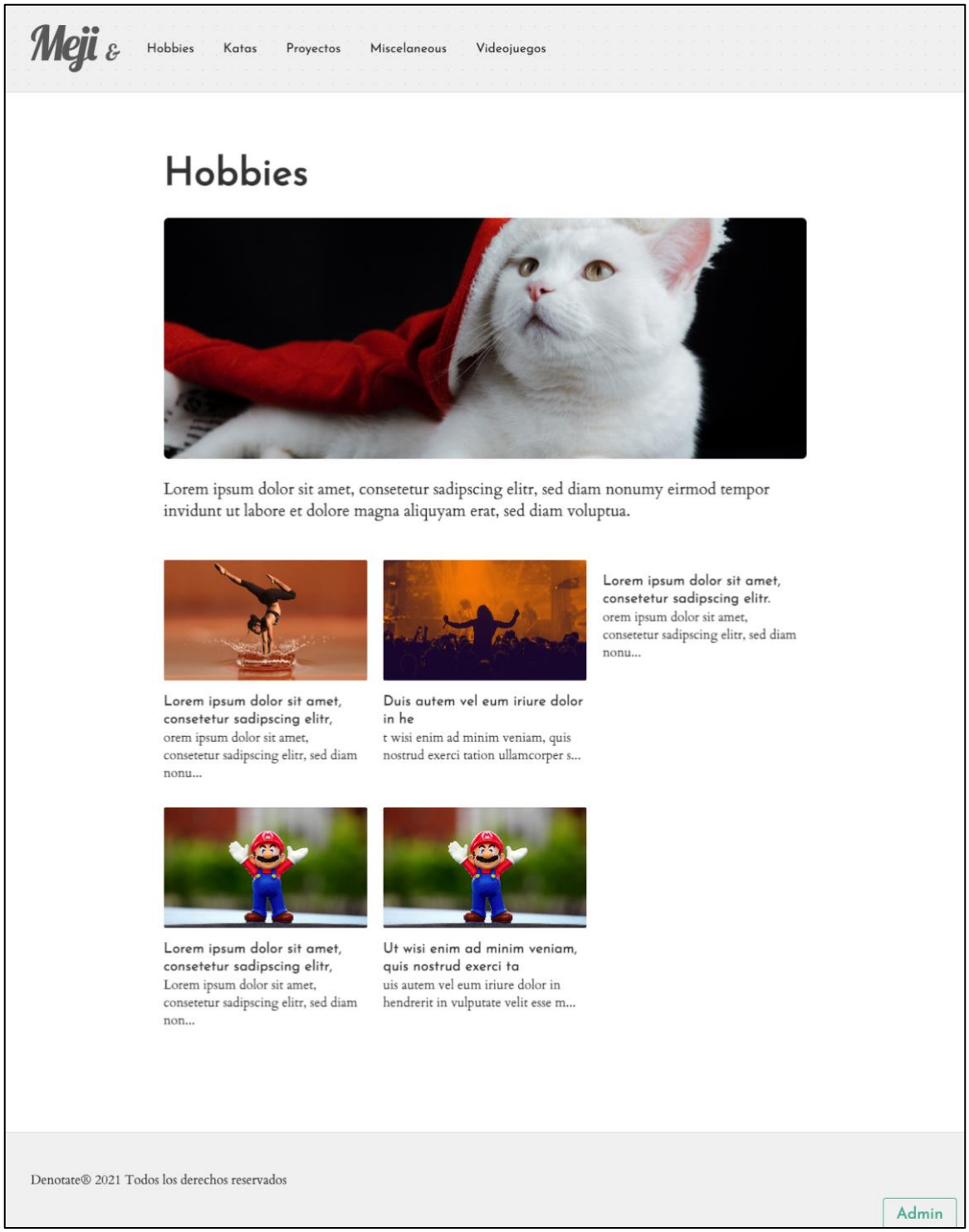

Figura 46 Diseño de la parte pública de una categoría del CMS

Figura 47 Diseño mobile de una categoría

POST

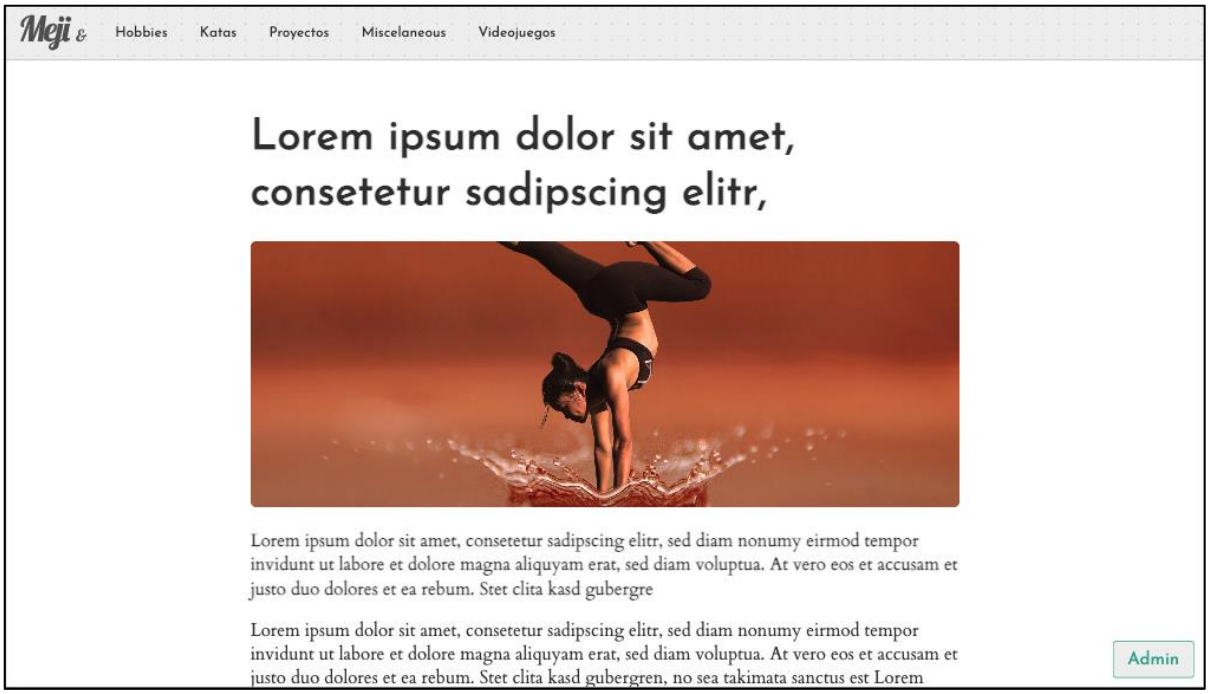

Figura 48 Diseño de un post Desktop

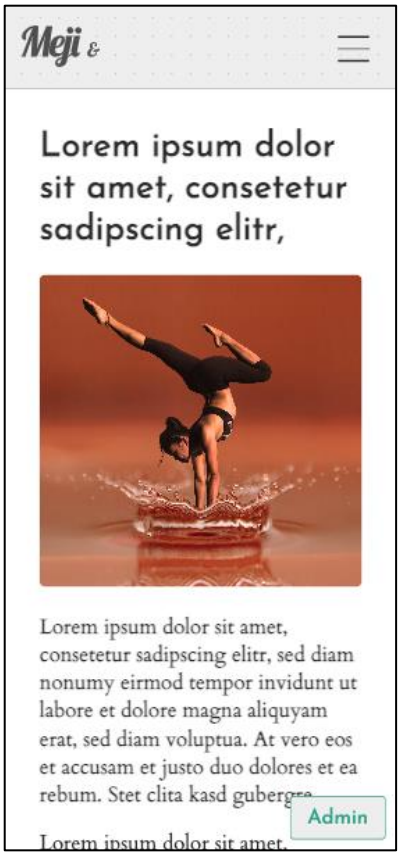

Figura 49 Diseño post mobile

ADMIN - Entidades

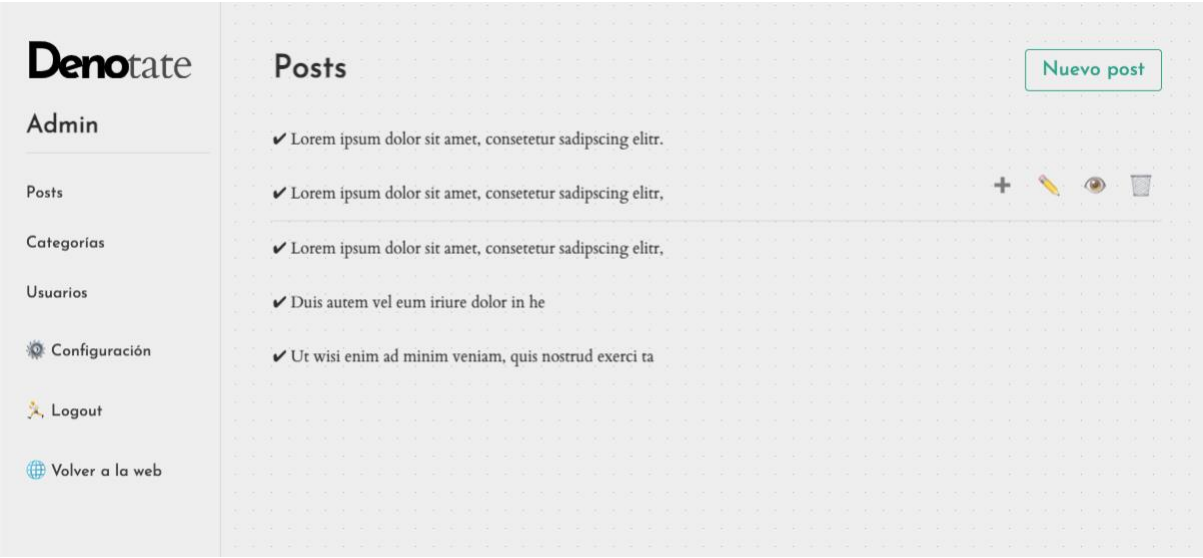

Figura 50 Diseño del administrador, vista de posts

Figura 51 Administrador versión mobile

ADMIN – Edición de entidad

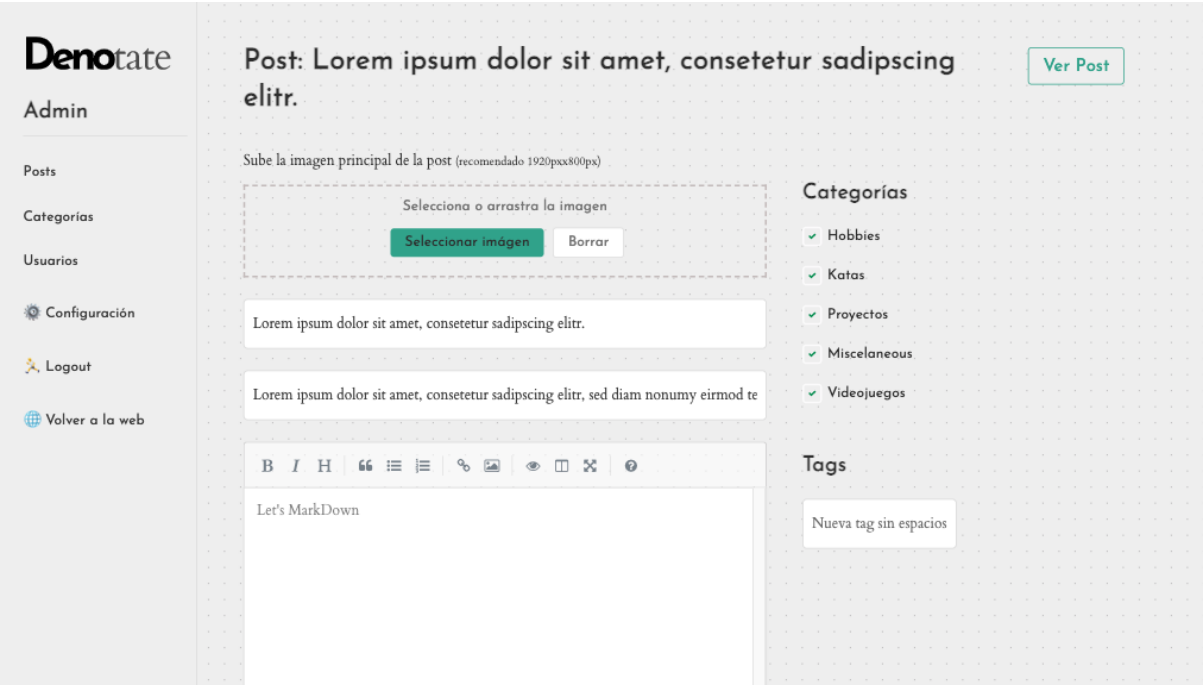

Figura 52 Administrador vista de edición de post en modo *desktop*

Figura 53 Edición de entidad vista mobile

6.3 Ejemplos de uso del producto

Este CMS se puede utilizar de diversas maneras, para publicar contenido que esté disponible online para usuarios no registrados, con un administrador en el que poder dar de alta creadores de contenido dinámico que enriquecaq ese CMS.

Un caso de uso práctico es una web personal que sirva por ejemplo de escaparate del traajo de una persona o de una empresa. Para ilustrar este ejemplo he creado mi propia página personal con este CMS que he subido en la url: https://meji.es/

En ella se pueden observar el uso de un logotipo personalizado, de elementos en la cabecera personalizados, y de artículos finales por ejemplo para desarrollar diferentes proyectos.

7 Conclusiones y líneas de futuro

7.1 Conclusiones

En esencia la realización de este trabajo me ha servido para utilizar la mayoría de los conocimientos adquiridos durante la formación de mi grado, en diferentes disciplinas:

- **Gestión de proyectos**: aptitud requerida para la definición del proyecto, la gestión de expectativas, errores, organización del tiempo, y en general evolución del producto a través de una metodología ágil, por sprints.
- **Desarrollo de proyectos online, inglés, documentación:** aptitud requerida para la documentación del proyecto. Las tecnologías utilizadas son relativamente recientes y la mayoría de la documentación requiere tener conocimientos de inglés avanzado, y soltura en medios online.
- **Programación, orientación a objetos, ingeniería del software, lenguaje y estándares web:** son todas ellas aptitudes nucleares en el camino abordado para construir esta aplicación. A través de la ingeniería he construido las historias de usuario y he organizado el trabajo, separado en front-end y back-end, convirtiendo dichas historias en funcionalidades. He separado las capas de desarrollo en vistas, modelos y controladores, con capa de dominio, infraestructura y componentes visuales, utilizando patrones de diseño, creando factorías o clases, con los conocimientos adquiridos en diseño y programación orientada a objetos, para estructurar el código desarrollado.
- **Usabilidad, diseño de interacción, arquitectura de la información:** Aptitud que he utilizado para conceptualizar y diseñar el diseño de la parte visible de la aplicación, utilizando diseño responsive adaptado a dispositivos.
- **Bases de datos:** conocimientos utilizados en la creación de los modelos, de los documentos y atributos en la base de datos, y en la relación entre las entidades.

Además de esta relación directa con los estudios y las aptitudes adquiridas durante el grado, la realización de este trabajo ha supuesto un reto personal, ya que aunque tenía las bases de conocimientos, quise enfrentarme a una tecnología completamente desconocida para mí y relativamente reciente, como es Node, Typescript o Litelement, apostando por los estándares web, y la implementación de tecnologías que los navegadores interpretan directamente, antes que por el uso de frameworks estables y conocidos. Este reto ha tenido ventajas y desventajas. La ventaja es el poder aprender otros entornos de programación, con superación y empeño, buscando documentación en muchos casos inexistentes, que me han obligado a concluir con soluciones alternativas. Uno de estos

casos ha sido por ejemplo la implementación de los formularios para los que he tenido que volver a incluir los elementos de formulario de los web components en el formulario inicial. La gran desventaja es la poca cantidad de desarrollos complementarios que existen alrededor de estas tecnologías aún, o el estado en muchos casos inicial de los mimos. Así por ejemplo he echado de menos el uso de librerías como Express, para crear las rutas, o Passport, para la autenticación, existentes en Node pero no en Deno.

Otro de los grandes problemas que me he encontrado durante el proyecto ha sido enfrentarme a la encapsulación del ShadowDOM, donde los estilos no se pueden heredar entre componentes, lo que hay hecho que tenga que diseñar unos estilos más encapsulados. Esto a la vez me hace pensar profundamente en el futuro de la web y del diseño, en la que cada vez tendemos a objetos más encapsulados.

La envergadura del proyecto, ha causado que en muchos puntos haya requerido un exceso de tiempo para resolver ciertos problemas, y ha hecho que algunas de las funcionalidades, que me gustaría como opcionales al proyecto, no hayan podido ser realizadas, como un gestor de recursos multimedia, un gestor del menú, o un gestor de la portada, pero si me siento satisfecho con el producto realizado, porque sienta las bases para poder desarrollar esas funcionalidades con posteridad.

El factor de desconocimiento de la tecnología y de la previsión real de mi tiempo libre para buscar documentación o desarrollarla ha hecho que en muchas ocasiones haya tenido que utilizar mucho más tiempo del presupuestado inicialmente, y que haya tenido que aumentar la velocidad de desarrollo en las etapas finales, siendo coherentes, debería haber previsto mucho más margen de error por este mismo motivo.

7.2 Líneas de futuro

Como he indicado en el punto anterior, el producto realizado sienta las bases para un producto mucho más complejo, en el que desarrollar una amplitud de funcionalidades y mejoras, pero con entidad propia como CMS para poder publicar contenidos. Las futuras ampliaciones pueden ser infinitas, pero cabe destacar:

- Refactorización de código de formularios para aligerar la carda de código y las funcionalidades compartidas en esos componentes.
- Gestión de envío de emails, para confirmar registro de usuarios, sistema SMTP.
- Gestión de menús, para incluir otros links que no sean categorías.
- Gestión de estado en borrador o publicado de posts.

- Gestor de portadas, con más elementos decorativos o estructuras de contenidos.
- Gestión de mensajes y comunicaciones con el usuario.
- Gestión de titles y descriptions para todas las páginas.
- Paquetización del producto en un solo repositorio.
- Gestión de CSS para poder elegir más opciones de diseño, además del color principal o de tema claro u oscuro.
- Creación de una web corporativa donde poder alojar el producto, promocionarlo y descargarlo.

Bibliografía

Pérez-Montoto, Mario (2006). Gestión del conocimiento, gestión documental y gestión de contenidos.

Tecnowired: https://www.tecnowired.com/programacion/wordpress-drupal-joomla-mejor-cms/ consultado 02/10/2020

Statcounter website: https://gs.statcounter.com/browser-market-share#monthly-201909-202009-bar, consultado 02/10/2020

Webkit website: https://webkit.org/ consultado 02/10/2020

Caniuse website: https://caniuse.com/ consultado 02/10/2020

Trello website: https://trello.com/ consultado 04/10/2020

Lit Element website: https://lit-element.polymer-project.org/ consultado 04/10/2020

Typescript website: https://www.typescriptlang.org/ consultado 04/10/2020

MarkDown notation sintaxis website: https://www.markdownguide.org/basic-syntax/ consultado 20/10/2020

Opentext website: https://www.opentext.com/products-and-solutions/products/opentext-product-offerings-catalog/rebranded-products/vignette-is-now-opentext consultado 22/10/2020

Wikipedia website: https://es.wikipedia.org/wiki/Wiki consultado 22/10/2020

https://wordpress.org/support/article/wordpress-editor/ consultado 22/10/2020

Wikipedia website: https://es.wikipedia.org/wiki/WYSIWYG consultado 22/10/2020

Medium website: https://medium.com/ consultado 26/10/2020

Google Fonts website: https://fonts.google.com/ consultado 26/10/2020

Type Scale website: https://type-scale.com/?size=16&scale=1.333&text=Escala%20tipogr%C3%A1fica&font=Josefin%20Sans&fontweight=600&bodyfont=body_font_default&bodyfontweight=400&lineheight=1.75&backgroundcolor=%23ffffff&fontcolor=%23000000&preview=false consultado 28/10/2020

GitHub website: https://github.com/oakserver/oak consultado 28/10/2020

DeboDB website: https://eveningkid.github.io/denodb-docs/ consultado 30/10/2020

Webstorm website: https://www.jetbrains.com/es-es/webstorm/ consultado 30/10/2020

The noun Project website: https://thenounproject.com/throwawayicons/collection/actions/ consultado 31/10/2020

Figma (Denotate) website: https://www.figma.com/file/NC7jhTIh2gSiHfZqH0EDqM/DenoTATE.me?node-id=8%3A10 consultado 31/10/2020

Martinez-Caro, J., Aledo-Hernandez, A., Guillen-Perez, A., Sanchez-Iborra, R., & Cano, M. (2018). A Comparative Study of Web Content Management Systems. *Inf., 9*, 27.

Deploy your Deno apps to Heroku: https://dev.to/ms314006/deploy-your-deno-apps-to-heroku-375h consultado 1/12/2020

Bcrypt: https://deno.land/x/bcrypt@v0.2.4 consultado 04/12/2020

Swagger: https://swagger.io/ consultado 05/12/2020

Mozilla: https://developer.mozilla.org/es/docs/Web/Web_Components/Using_shadow_DOM consultado 26/12/2020

Mozilla: https://developer.mozilla.org/es/docs/Web/HTTP/Methods consultado 26/12/2020

Webpack: https://webpack.js.org/api/compilation-object/#root consultado el 26/12/2020

Wikipedia: https://es.wikipedia.org/wiki/Variable_de_entorno consultado el 27/12/2020

NPM: https://www.npmjs.com/ consultado el 27/12/2020

Mozilla: https://developer.mozilla.org/es/docs/Web/Web_Components consultado el 28/12/2020

Mozilla: https://developer.mozilla.org/es/docs/Web/CSS/Using_CSS_custom_properties consultado el 28/12/2020

Github IO.js: https://github.com/nodejs/iojs.org consultado el 2/01/2021

Rust: https://www.rust-lang.org/ consultado el 2/01/2021

Anexos

Glosario

Alosaur: Framework para Deno, con múltiples decoradores y métodos utilitarios.

API REST: Representation Estate Transfer API. API que usa en comunicaciones HTTP sin estado, que usa métodos GET, POST, PUT y DELETE, para la gestión de hipermedia.

Bcrypt: Es una función de hash de contraseña. Se usa para encriptar y desencriptar contraseñas.

Compilar: Traducir un lenguaje de alto nivel, en uno o varios mñodulos, a un lenguaje de máquina para su ejecución.

DELETE: Método de petición HTTP, utilizado principalmente para solcitar borrado de entidades.

Deno: Entorno de desarrollo en tiempo de ejecución Javascript, que soporta también Typescript, y que está construido en Rust y usa el motor V8.

Drupal: CMS competencia de Joomla o Wordpress.

DOM Modelo de ojetos de documento, es una interfaz estándar que proporciona representación de ojetos para XML, XHTML y HTML.

Evento: Es una acción u ocurrencia que ocurre en el sistema programado que se puede identificar, para realizar una acción o ejecución consecuente a dicho evento.

Express: Librería Javascript para Node, enfocada a la construcción de un servidor con rutas.

Factoría: Patrón de diseño de programación, método u objeto, que se usa para crear otros objetos.

Figma: Programa de diseño vectorial, utilizado en diseño UX UI, enfocado en objetos reutilizables y librerías.

Framework: Es un esquema o patón, abstraído a un nivel superior sobre un lenguaje de programación, que se utiliza para la implementación y/o desarrollo de una aplicación.

GET: Método de petición HTTP, utilizado principalmente para solicitar datos.

Heroku: Plataforma de servidores en la nube con integración de diferentes lenguajes, propiedad de Salesforce.

Interfaz: Propiedades y métodos que un ojeto expone de forma pública.

Joomla: CMS basado en PHP que compite con Wordpress.

JSON: Acrónimo de Javascript Object Notation, Es un formato de texto que se usa en el intercambio de información, que guarda la estrutura similar a la de los objetos Javascript.

Listener: método manejador de un evento, sobre un evento u ojeto cncreto que se ejecuta al realizarse dicho evento.

Litelement: Framework basado en una clase base que se extiende para crear componentes web de forma ligera y sencilla.

Markdown: Lenguaje de marcado ligero, que tiene una entrada sencilla de comprender y escribir.

MongoDB: MongoDB es un sistema de base de datos documental NoSQL, orientado a documentos y de código abierto.

Node: Entorno en tiempo de ejecución de desarrollo javascript ideado para comunicaciones con servidor.

Npm: Acrónimo de Node Package Manager. Es un gestor de paquetes de programación desarrollados para ejecutarse en Node.

Passport: Librería desarrollada en Node utilizada para autenticación.

POST: Método de comunicación HTTP utilizado comúnmente para el envío de entidades o datos al servidor.

PUT: Método de comunicación HTTP utilizado comúnmente para el envío de entidades o datos al servidor que provocan un cambio en la entidad de la base de datos.

Router: En este proyecto, router se refiere a la programación que genera las diferentes URLS o routas de la aplicación, tanto en el back-end, rutas de la API REST, o en el front-end, rutas de navegación.

ShadowDOM: DOM encapsulado en un componente, que encapsula HTML, estilos y Javascript.

Slot: Es una etiqueta propia de los web components, que se usa para delimitar un espacio en el que un componente contendrá el resto del HTML que se encuentre entre su apertura y cierre cuando este es utilizado.

Swagger: Es una serie de librerías, métodos y herramientas reunidas en una aplicación que se usan para crear y documentar APIs REST.

Trello: Aplicación online utilizada para la gestión y documentación de proyectos y tareas,
Typescript: Lenguaje de programación superconjunto basado en Javascript, desarrollado por Microsoft, pero open source.

URI: Acrónimo de Uniform Resource Identifier, identificador de recursos uniformes.

Web component: *Los Componentes Web son un paquete de diferentes tecnologías que te permiten crear elementos personalizados reutilizables — con su funcionalidad encapsulada apartada del resto del código — y utilizarlos en las aplicaciones web***.**[59]

Webkit: Motor open source, plataforma que ha servido de base para la creación de navegadores web.

Webpack: Paquete de módulos Javascript, que compila diferentes módulos Javascript para su uso en navegadores, aunque también HTML y CSS.

WebStorm: Programa de edición de código, HTML, JS y CSS de Jet Brains.

[59] Según la documentación oficial de MDN