



UNIVERSITAT
POLITÀCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Inarts, la red social de contacto entre artistas y editoriales

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Enrique Pons Ballester

Tutoras: Manoli Albert Albiol, Victoria Torres Bosch

2018-2019

Resumen

El Trabajo de Final de Grado que se ha desarrollado consiste en la creación de una red social de contacto entre dos roles, artista y editor. El sistema se trata de un portal web donde cualquier usuario puede valorar las obras de los artistas y de esta forma premiar con mayor visibilidad a aquellos que más reacciones positivas suscitan. Por otro lado, los editores podrán promocionar mediante un mecanismo de donativos a los artistas que deseen, siendo premiados de esta forma. El mecanismo de premiado consistirá en dos rankings diferentes, uno para cada rol.

El desarrollo del presente proyecto se ha llevado a cabo siguiendo una metodología en cascada, donde, en una primera etapa se ha diseñado la aplicación siguiendo técnicas de diseño centrado en el usuario como las *Personas* o *Casos de uso*. A continuación, se realizó la implementación utilizando *Node.js*, *JavaScript* y *Angular*. Por último, se han realizado pruebas en vivo lanzando la aplicación e invitando a una serie de usuarios para asegurarnos de que las funcionalidades programadas funcionaban correctamente.

Palabras clave: Red social, Diseño Centrado en el Usuario, Angular, Diseño web, API, Arte, Responsive Web Design.

Abstract

The Final degree work that has been developed consists in the creation of a social network of contact between two roles, artist and the editor. The system is a web portal where any user can evaluate the works of the artists and thus reward with greater visibility to those who more positive reactions arise. On the other hand, publishers will be able to promote through a mechanism of donations to the artists they wish, being rewarded in this way. The rewarding mechanism will consist of two different rankings, one for each role.

The project has been carried out through several processes, in a first stage the application has been designed following user-centric design techniques such as *People* or *Use cases*. The implementation was then implemented using Node, JavaScript and angular. Finally, live tests have been carried out launching the application and inviting several users *Testers* To make sure that the programmed features worked properly.

Keywords: Social Network, User-centered Design, Angular, Web Design, API, Art, Responsive Web Design.

Agradecimientos

Los agradecimientos se encuentran repartidos entre un sinfín de personas que han colaborado conmigo apoyándome en diferentes etapas del proyecto, grupo extenso en exceso por lo que resumiré a los principales actores.

Inmaculada Soler Ramos, mi pareja y principal apoyo en las interminables horas de diseño, artífice de mi autocrítica ante cualquier imperfección en el pixel de las esquinas, nunca hubiera imaginado que para un simple logo se necesitaran dos lluvias de ideas, semanas de pensamiento intenso, tratamiento de imagen al detalle e investigación y aprendizaje. Sin duda gracias por hacerme entender la diferencia entre logotipo, isotipo, imagotipo, isologo ...

A mi familia por aguantarme hasta en los momentos más difíciles. Cuando la desesperanza hacía acto de presencia siempre me animaron y estuvieron a mi lado y sin ellos jamás hubiera seguido adelante y completado el trabajo.

A mis profesores y tutores, Pedro Valderas y Victoria; el primero, con el cual inicié la andadura, me aconsejó sabiamente utilizar mis conocimientos previos. No le hice demasiado caso y, si bien la motivación principal del proyecto era emplear tecnologías desconocidas por mí al inicio, me arrepiento de mi desobediencia ya que hubiera ahorrado sin duda meses de investigación.

A Victoria, aun comenzando con ella ya avanzado el TFG, hizo una gran labor en la organización de la memoria y la planificación de la parte final.

Por último, mención especial a mi abuela, fallecida el 14 de marzo de 2018, una persona muy especial para mí, cuya vida me ha inspirado a trabajar duro y no dejar nada a medias.

Tabla de contenidos

1.	Introducción	9
1.1.	Prefacio	9
1.2.	Objetivo.	10
2.	Estado del arte	13
2.1.	Redes horizontales	13
2.2.	Redes verticales.	14
2.2.1.	Pinterest.	14
2.2.2.	Patreon	14
2.2.3.	Afactys	15
3.	Requisitos	17
3.1.	Requisitos funcionales	17
3.2.	Requisitos no funcionales.	18
3.2.1.	Eficiencia.	18
3.2.2.	Usabilidad.	18
3.2.3.	Seguridad.	19
4.	Contexto tecnológico	21
4.1	Entorno de trabajo.	21
4.1.1.	Visual Studio Code	21
4.1.2.	Google Chrome	22
4.1.3.	Postman	22
4.1.4.	GIT y GitHub	23
4.1.5.	MongoDB Compass Community Edition	23
4.1.6.	FileZilla Client	24
4.1.7.	Docker	24
4.2.	Tecnologías.	25
4.2.1.	Angular	25
4.2.2.	Node	26
4.2.3.	MongoDB	27

5.	Arquitectura	29
5.1	Esquema de componentes.	29
5.2.	Estructura de archivos común.	31
5.3.	Patrones arquitectónicos.	33
5.3.1.	Backend	33
5.3.2.	Frontend	35
5.4.	Arquitectura de base de datos.	36
6.	Diseño gráfico	39
6.1.	Nombrado.	39
6.2.	Logotipo.	39
6.3.	Estética general.	43
6.4.	Mockups	43
6.4.1.	Mockups web	44
6.4.2.	Mockups móviles	46
7.	Seguridad	49
8.	Implementación	53
8.1.	Registro.	53
8.2.	Correo electrónico.	54
8.3.	Login	55
8.4.	Gestión de errores	57
8.5.	Saneamiento de datos	58
8.6.	Mensajería instantánea	58
8.7.	Servicio de upload	59
8.8.	Servicio de donativos	60
8.9.	Métodos crud	62
9.	Conclusiones y trabajo futuro	63
10.	Bibliografía	65

1. Introducción

1.1. Prefacio

El Trabajo de Final de grado supone la última prueba de conocimientos previa a la terminación de los estudios de Ingeniería Informática.

Como tal, se pretende demostrar los conocimientos adquiridos dentro del grado y aplicarlos a un área en concreto de principio a fin, con un desarrollo completo y pasando por cada una de las fases necesarias para finalizar con éxito el proyecto.

Este sentido tan amplio, en mi caso fue concentrado en aquello que siempre me ha ayudado a avanzar, la curiosidad. Los diferentes lenguajes existentes en el ámbito informático y los avances diarios de estos siempre me han fascinado.

A tecnologías con las que había tenido contacto como PHP y algunos de sus frameworks como *Laravel* o *Codeigniter* les tengo cierto aprecio pues me permitieron tener un primer contacto con el mundo laboral y empezar a conocer el día a día de un informático.

Sin embargo, para este proyecto quería demostrarme algo a mí mismo, necesitaba comprender lo que implicaba realizar un proyecto largo, sin apenas conocimientos previos y con una utilidad real.

Por ello debía utilizar una tecnología que no hubiera usado, que no controlara ni entendiera en profundidad. De ahí surgieron las principales tecnologías empleadas, tanto *Node.js* por lo que me costó aprobar la asignatura “Tecnologías de sistemas en red” como *Angular*, por lo mucho que había escuchado de ella como tendencia de desarrollo frontend.

Por otra parte, durante los últimos tres años he sentido interés hacia el área del diseño web, esencialmente por las posibilidades que ofrece y la amplia oferta de trabajo que presenta.

Debido a esto fui acercándome a un mundo que me era totalmente ajeno. El arte es un ambiente diferente al informático, comprende un abanico enorme de campos que tratar, desde la ilustración hasta el cómic, pasando por el desarrollo digital o el diseño editorial.

Este acercamiento me ha permitido observar los retos que han de sobrellevar las personas dedicadas por completo a este ámbito; problemas como el alto coste de la producción de un portfolio y la escasa remuneración de la gran mayoría, han dado lugar al planteamiento de alguna forma de paliarlos.

De ahí surgió la idea de un sitio web que diera visibilidad a los trabajos, que permitiera conocer a los artistas y contactar con ellos para ofrecerles empleo. Una red de contactos al fin y al cabo para artistas y personas que deseen apoyarlos.

1.2. Objetivo.

El principal objetivo que se plantea en este proyecto consiste en desarrollar un sitio web que proporcione capacidades de interacción a artistas y editores, soportando las características de visualización y promoción que esto conlleva.

Para poder cumplir este objetivo, se pondrá especial énfasis en la usabilidad, la adaptabilidad a formato móvil, el diseño y la consistencia de las funciones, así como a la optimización de funciones y escalabilidad tanto a nivel de código como de infraestructura.

La facilidad de uso para el público general de una aplicación de este tipo es esencial, ya que debe entenderse sin la necesidad de un manual de instrucciones.

El diseño por su parte también goza de gran importancia dado el carácter artístico del público al que se dirige y la adaptabilidad a dispositivos móviles, ya hoy en día son tendencia en navegación.

Por otro lado, dado que las redes sociales manejan cantidades ingentes de datos, el sistema debe ser fácil de escalar en diversos sentidos:

El almacenamiento no puede ser un factor limitante debido a que una de las principales utilidades consiste en la subida de imágenes, por lo que la capacidad debe adaptarse de forma adecuada a los archivos subidos.

Tampoco debe resultar limitante la capacidad de la base de datos para soportar accesos concurrentes, dada la afluencia potencialmente masiva esperada.

Los usuarios tendrán las capacidades de ver, reaccionar y comentar publicaciones. También podrán chatear entre ellos, recibir notificaciones y visitar los perfiles de otros usuarios.

Además de estas funcionalidades, el rol de artista dispondrá de la posibilidad de crear publicaciones, mientras que los editores podrán promocionar a los artistas a través de donaciones.

Todas las interfaces han de ser rápidas y reactivas, centrándose en la usabilidad e interacción con el usuario y dando respuesta a los eventos de forma inmediata, sin esperas innecesarias.

Dada la previsión de crecimiento de la infraestructura conforme crezca la afluencia, tanto la ejecución de buenas prácticas a nivel programático como la estructura creada para la plataforma serán indispensables para que programadores nuevos en el proyecto puedan entender con facilidad las funciones y crear nuevas sin un tiempo de aprendizaje excesivo.

Todos estos apartados se pueden resumir en los siguientes objetivos específicos:

- El sitio web será adaptable a las resoluciones 1920x1080 (ordenador FHD), 1366x768 (ordenador HD), 720x1280 (móviles), 320x480 (móviles pequeños) y 768x1024 (IPAD).
- El costo mensual de alojamiento, almacenamiento en la nube, base de datos y sistemas asociados serán inferiores a 100 euros durante el desarrollo, el primer año tras este el costo aumentará a 1000 euros.
- A través de la comisión opcional mediante el sistema de donativos, el sitio generará los ingresos necesarios para cubrir el 50% del gasto nombrado en el requisito anterior a partir del segundo mes tras el lanzamiento.

2. Estado del arte

En el actual ecosistema web existen multitud de redes sociales utilizadas por artistas para mostrar su producción al público. Desde grandes infraestructuras como Facebook, hasta sitios dedicados a un ámbito más local, como *Afactys* en España.

En este punto realizamos una clasificación de todas estas con tal de encontrar el hueco en el mercado que podamos situarnos.

Basándonos en la clasificación realizada por Pablo Fernández [31], las redes sociales se clasifican en dos grandes grupos: Aquellas redes que se basan en la comunicación sin medios electrónicos, llamadas *Off-Line* y aquellas que hacen uso de estos medios y son llamadas *On-Line*. Estas son en las que nos centraremos.

Dentro de esta clasificación, las redes *On-Line* se pueden caracterizar como *horizontales*, que son las dirigidas al público general o como *verticales*, que se construyen en base a una temática establecida.

Los ejemplos más importantes dentro del primer grupo son *Facebook*¹, *Twitter*², *YouTube*³ e *Instagram*⁴, mientras que, del segundo grupo, dada la cantidad de sistemas que ofrecen este esquema, nos centraremos en las redes *Patreon*⁵, *Pinterest*⁶ (esta no se corresponde al completo con la definición de redes verticales, pero posee más afinidad como explicaremos a continuación) y *Afactys*⁷, pues cada una tiene ciertas características que las asemejan a lo que pretendemos elaborar.

2.1. Redes horizontales

Esta clase de redes no disponen de un objetivo específico y admiten múltiples clases de contenido. Debido a esto, disponen de una audiencia muy grande por lo que son las favoritas para comenzar sin ser conocidos y crear una base de seguidores.

El potencial de las redes de esta clase radica en su amplia gama de usuarios y el alto alcance que pueden llegar a tener. Sin embargo, a diferencia de estas, la propuesta del presente trabajo pretende crear una comunidad exclusiva de artistas, que, a pesar de tener un mercado más reducido presente un interés mucho más dirigido

Esto se pretende lograr mediante el sistema de donativos y puntuaciones que se explicará más tarde.

¹ <https://www.facebook.com/>

² <https://twitter.com/>

³ <https://youtube.com/>

⁴ <https://www.instagram.com>

⁵ <https://www.patreon.com/>

⁶ <https://www.pinterest.es/>

⁷ <https://afactys.com/>

De esta manera, no se entraría en competencia directa con esta clase de sistemas, pues el servicio ofrecido no pretende atraer gente externa a la comunidad del arte.

2.2. Redes verticales.

Las redes verticales se acercan más a nuestro modelo, de forma que analizaremos cada una de ellas de forma individual, pues a pesar de tener un enfoque más parecido a lo que pretendemos, argumentaremos que se encuentran diferencias notables.

2.2.1. Pinterest.

Pinterest es la red social de inspiración de referencia. Presenta organización de imágenes temática en lugar de cronológica, a diferencia de las redes más comunes. Además, emplea un sistema en abierto, lo cual implica que no necesitas ser un usuario registrado para poder visitarla.

Es debido al carácter puramente artístico de los trabajos mostrados que la hemos incluido como red vertical, a diferencia de *Instagram*, cuyo contenido se centra sobre todo en la fotografía de la vida diaria.

La principal diferencia de *Inarts* frente a Pinterest es la idea que subyace como base del proyecto.

Inarts es una red profesional con dos roles diferenciados, cada uno premiado por acciones diferentes, por lo que se encuentra centrada en los dos papeles representados, separándose de esta manera del concepto de apertura total de la red descrita, aun tomando la posibilidad de visitar el contenido sin estar registrado.

2.2.2. Patreon

Patreon responde a la necesidad de subvención de artistas para continuar con sus trabajos. Los artistas suben sus trabajos de forma privada, por lo que para poder acceder debes estar registrado y dispuesto a proporcionar medios económicos al autor.

A pesar de no estar exclusivamente dirigido a profesionales del sector artístico, sigue siendo uno de los mecanismos más utilizados, de él hemos extraído la idea de los donativos, pero a diferencia de este, en la plataforma creada, todo el contenido es público, por lo que no capamos las posibilidades de acceso y nos establecemos como mecanismo de difusión profesional y no puramente de mecenazgo.

2.2.3. Afectys

Afectys es sin duda la más similar a nuestra propuesta en cuanto al objetivo, pues se define a sí misma como la red social de profesionales del Arte y la Cultura.

Esta red posee un sistema similar de búsqueda y promoción, así como creación de castings o eventos. Dispone de un *feed* de publicaciones y multitud de características útiles para establecer vías de contacto entre artistas y las compañías promotoras.

El espacio que ocupan se encuentra muy cerca de nuestro enfoque, pero en su caso establecen como requisito para utilizar su sistema la creación de una cuenta. Este mecanismo resta efectividad a la hora de darle visibilidad a los trabajos publicados, una cuestión de gran relevancia en el objetivo que buscamos, aun no siendo lo principal.

3. Requisitos

En este apartado se detallan los requisitos funcionales y no funcionales del sistema. Mientras que los requisitos funcionales son aquellos que describen el comportamiento del sistema, los no funcionales, describen las características de dicho comportamiento, es decir, que trata sobre aspectos de la calidad del sistema.

3.1. Requisitos funcionales

Tras realizar un análisis sobre el comportamiento ofrecido por redes sociales verticales similares, a continuación, detallamos la lista de requisitos mínimos funcionales que el sistema debe satisfacer.

- El sistema ofrecerá un mecanismo de registro basado en el uso de un formulario, el cual notificará de manera automática al usuario sobre dicho registro.
- El sistema permitirá realizar el cambio de la contraseña.
- Se permitirá a usuarios modificar sus datos personales, así como decidir acerca de la privacidad de sus publicaciones e información.
- Se permitirá la visualización de obras a usuarios no registrados, pero de una forma limitada, impidiendo la interacción.
- El sistema proporcionará servicio de mensajería a sus usuarios registrados.
- Existirá un acceso de super administrador a la aplicación con capacidades para la creación, modificación y borrado de datos.
- Los artistas podrán recibir donaciones de editores.
- Se premiará a artistas y editores por su actividad de forma diferente en función de sus acciones.
- Los usuarios registrados podrán valorar y comentar las publicaciones de otros usuarios, así como seguir y dejar de hacerlo a otros usuarios.
- Se proporcionará un *feed* personalizado a los usuarios registrados en función de sus gustos.
- Se podrá proporcionar acceso a otras aplicaciones a los datos recogidos con él.

3.2. Requisitos no funcionales.

A continuación, presentamos algunos requisitos no funcionales clasificados en función de las diferentes áreas que tratan.

3.2.1. Eficiencia.

- Todas las funciones deben responder en menos de 2 segundos, a excepción de las relacionadas con la subida y descarga de archivos que dependerá del tamaño del fichero.
- El sistema podrá operar sin problema con hasta 20.000 usuarios en tiempo real⁸.
- El almacenamiento no debe suponer una limitación en cuanto a capacidad y coste con hasta 20.000 usuarios realizando 10 publicaciones al día a partir del primer año.
- El sistema podrá soportar adecuadamente 100 peticiones por segundo.

3.2.2. Usabilidad.

- El tiempo de aprendizaje debe ser instantáneo para nativos digitales y corto para anteriores generaciones.
- Los errores cometidos por el usuario deberán proporcionar mensajes informativos útiles del fallo al usuario final.
- El diseño debe ser responsivo para las resoluciones más comunes de teléfono móvil, tableta y ordenador.
- El tiempo de respuesta será inmediato, proporcionando retroalimentación continuamente al usuario.
- La carga de la página no debe demorar más de 1 segundo en un ordenador de sobremesa o portátil y no más de 4 en móviles.

⁸ El sistema que soporta el servidor actualmente dispone de 1GB de RAM por lo que basándonos en la previsión de la consulta

<https://stackoverflow.com/questions/17448061/how-many-system-resources-will-be-held-for-keeping-1-000-000-websocket-open>

y mediante una división, este podría soportar unas 33.333 conexiones concurrentes, proporcionando un margen, concluimos que 20.000 podría ser un requisito aceptable por el momento

- El acceso a la plataforma debe estar disponible para el usuario incluso sin conexión de este a internet.

3.2.3. Seguridad.

- El sistema debe desarrollarse siguiendo estándares que refuercen la seguridad de los datos.
- Los sistemas de almacenamiento de datos deben encontrarse respaldados por replicas que guarden copias y se activen en caso de fallo.
- El acceso a las imágenes debe estar disponible un 99.99% del tiempo.

4. Contexto tecnológico

En la presente sección vamos a tratar las herramientas y lenguajes utilizados durante el desarrollo del sistema, así como su planificación y lanzamiento.

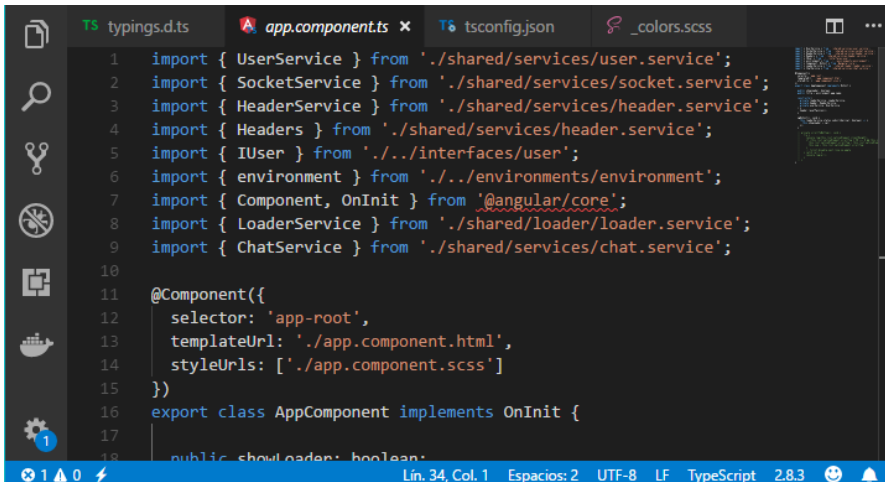
Se dividirá en tres apartados diferenciadas: en el primero, se presenta el entorno de trabajo utilizado durante el proyecto, el segundo trata sobre la planificación y, por último, un tercero correspondiente a las tecnologías directamente asociadas al funcionamiento de la aplicación.

4.1 Entorno de trabajo.

4.1.1. Visual Studio Code

El editor de texto utilizado ha sido Visual Studio Code (a partir de ahora VS Code) por ser gratuito, con amplio soporte para la mayoría de los lenguajes, soporte para Git, funcionalidades de depuración y un grado alto de personalización mediante extensiones, que transforman el editor en prácticamente un IDE. Las extensiones que hemos utilizado son:

- Angular Language Service.
- Angular 6 snippets, Beautify css/sass/scss/less.
- Bootstrap4, Font awesome 4, Font awesome 5 free & pro snippets.
- Docker.
- EditorConfig for VS Code.
- ESLint.
- Path Intellisense.
- Debugger for Chrome.



```
1 import { UserService } from './shared/services/user.service';
2 import { SocketService } from './shared/services/socket.service';
3 import { HeaderService } from './shared/services/header.service';
4 import { Headers } from './shared/services/header.service';
5 import { IUser } from '../interfaces/user';
6 import { environment } from '../environments/environment';
7 import { Component, OnInit } from '@angular/core';
8 import { LoaderService } from './shared/loader/loader.service';
9 import { ChatService } from './shared/services/chat.service';
10
11 @Component({
12   selector: 'app-root',
13   templateUrl: './app.component.html',
14   styleUrls: ['./app.component.scss']
15 })
16 export class AppComponent implements OnInit {
17   public showloader: boolean;
```

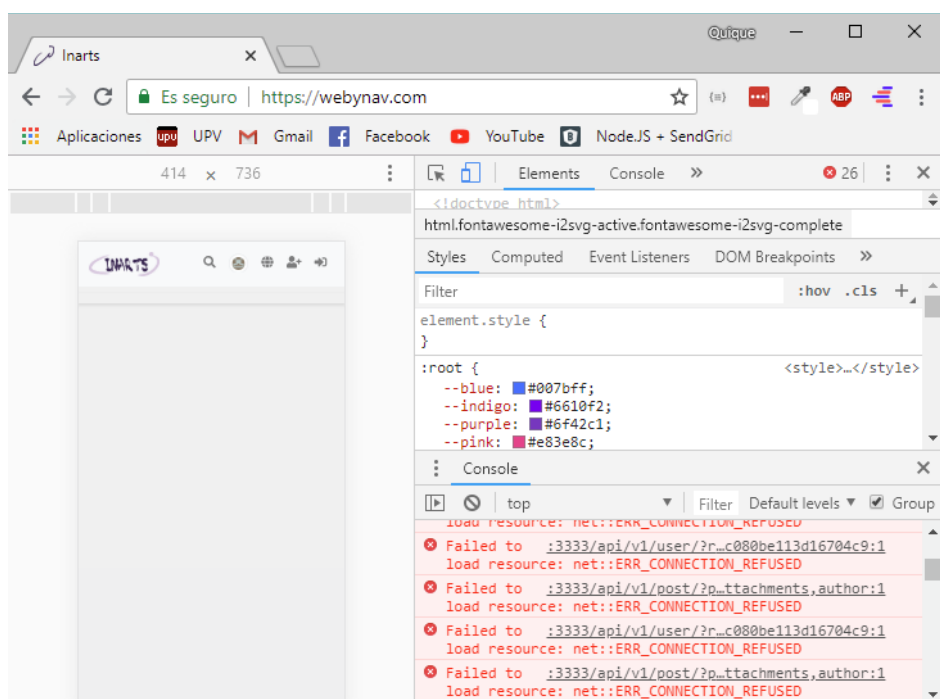
1 - Interfaz Visual Studio Code



4.1.2. Google Chrome

El navegador escogido para el desarrollo ha sido Google Chrome, pues junto a la extensión *Debugger for Chrome* de VS Code y las múltiples herramientas de desarrollo que posee (como la prueba de adaptación a diferentes tamaños (ilustración 2)), nos ha facilitado la tarea de encontrar multitud errores durante la programación del frontend. Además de lo anterior, hemos empleado dos extensiones para facilitar el diseño:

- *ColorZilla* para la extracción de colores dentro del navegador.
- *Keyframes* para el diseño de las animaciones CSS.

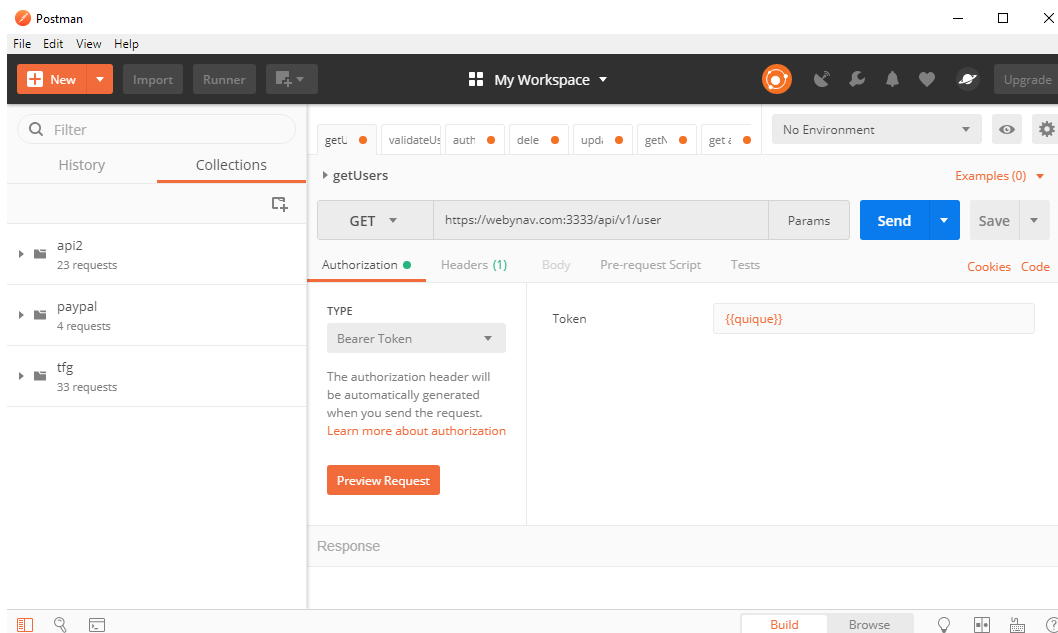


2 - Navegador Chrome con el inspector abierto durante el desarrollo

4.1.3. Postman

Postman se define como un entorno de desarrollo de API (ADE). El sistema posee múltiples funciones para ello: desde las pruebas de las consultas hasta la documentación automática, pasando por el diseño y monitorización de estas. En el proyecto hemos aprovechado sobre todo el lanzamiento de consultas http y la documentación, probando cada una de las posibles llamadas a la API.

A su vez, al poder gestionar múltiples colecciones nos ha permitido mantener una separación entre las consultas al entorno de desarrollo y al de producción.



4.1.4. GIT y GitHub^{9 10}

3 – Solicitud Postman API

Por último, necesitábamos un sistema de control de versiones y, con este propósito establecimos Git como mecanismo, pues como herramienta de código abierto no supuso coste alguno y por su popularidad.

Disponer de este sistema permite mantener cierto control de los cambios realizados, además de proporcionar la capacidad de retroceder en situaciones donde algún error desestabiliza la aplicación.

GitHub es una plataforma de desarrollo donde hemos alojado los repositorios de forma remota para mantener el código a salvo de cualquier posible percance.

4.1.5. MongoDB Compass Community Edition

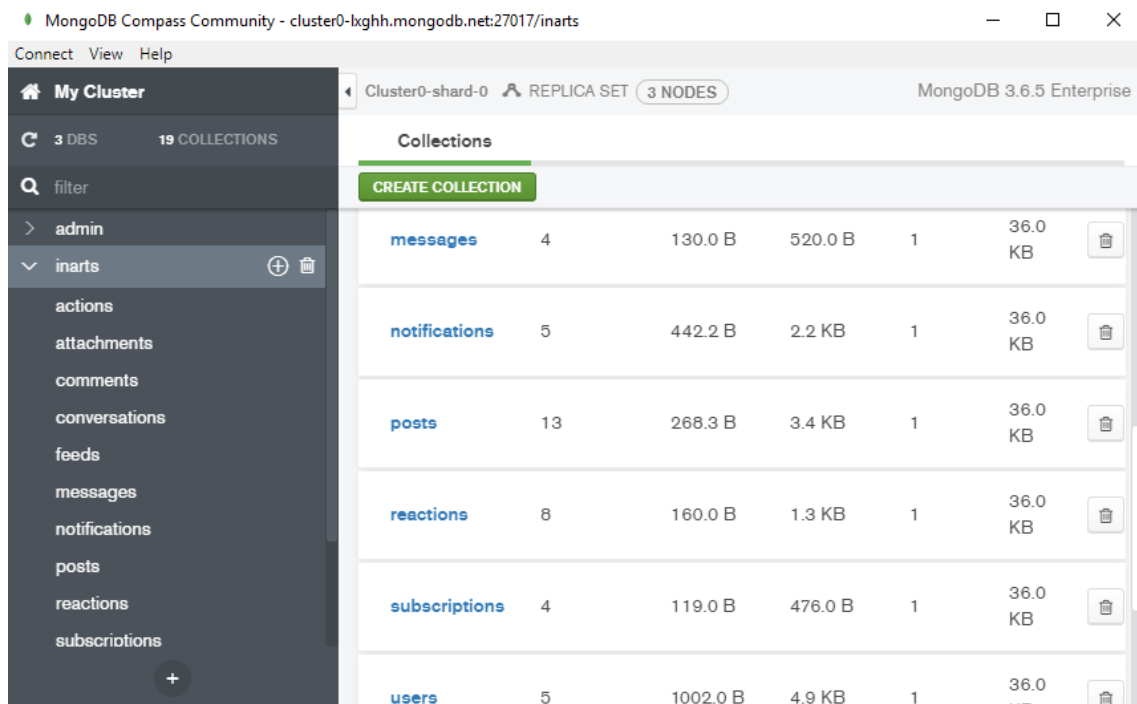
MongoDB Compass es un gestor gráfico de bases de datos MongoDB, este nos ha permitido interactuar con la base de datos alojada con MongoDB Atlas en servidores dispuestos por *Amazon Web Services*.

Este sistema nos ha servido durante el desarrollo para comprobar el estado de la base de datos tras la ejecución de diferentes funciones.

Las pruebas podrían haberse realizado mediante consultas lanzadas al servidor directamente, sin embargo, disponer de un entorno gráfico evita errores y facilita el trabajo.

⁹ <https://github.com/ticquique/frontend>

¹⁰ <https://github.com/ticquique/backend>



4 - MongoDB Compass visualización de las colecciones

4.1.6. FileZilla Client

FileZilla es un cliente ftp con capacidades para conexión SFTP (*SSH file transfer protocol*).

Esta es la herramienta que hemos utilizado para la gestión visual y sencilla de los ficheros en el servidor virtual alojado en DigitalOcean en producción.

4.1.7. Docker

Docker es el software de contenerización más utilizado globalmente. Este sistema nos permite el despliegue de aplicaciones emulando un entorno de producción y, al mismo tiempo, mediante Docker-compose podemos establecer un entorno de desarrollo ágil en combinación con los comandos del cli de angular.

4.2. Tecnologías.

En la siguiente sección presentamos las tecnologías empleadas para el funcionamiento de la aplicación, se comentará la principal utilidad dentro del trabajo y las razones que llevaron a su elección frente a sistemas similares.

4.2.1. Angular

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  showImageUploader: string;
  showUploader:boolean = true;

  constructor() {
    private imageUploadService: ImageUploadService
  }

  ngOnInit(): void {
    this.imageUploadService.showImageEditor.subscribe(val => {
      this.showUploader = true;
      this.showImageUploader = val;
    });
  }
}
```

5 – Código typescript Angular declaración componente principal

Angular es un *framework* JavaScript mantenido por Google con una base de usuarios de entre las más amplias en el sector del desarrollo SPA (*single-page application* o aplicación de página única).

Como indica su nombre, una SPA es un sitio web que solo carga una página. Esto permite una experiencia fluida por parte del usuario, evitando tiempos de carga innecesarios.

En estas, los recursos o se cargan al principio o dinámicamente en función por norma general de la interacción del usuario. De esta manera, la navegación entre páginas lógicas no implica interacción con el servidor con lo que no existen esta clase de latencias.

Otra tecnología valorada para el desarrollo ha sido ReactJS.

ReactJS, aunque se define como una librería más que como un framework, tiene una comunidad de desarrolladores detrás tal que le permiten establecerse como digna rival.

A nivel funcional se puede equiparar y también esta mantenida por una gran compañía como es Facebook.

Angular, a diferencia de ReactJS utiliza Typescript de forma nativa.

Typescript es un superconjunto de JavaScript mantenido por Microsoft que añade tipado estático y objetos basados en clases. El código, luego, es compilado a JavaScript común.

Esta tecnología, al ser compilada da la posibilidad de establecer una versión destino de JavaScript, por lo que podemos emplear funciones avanzadas sin preocuparnos por la compatibilidad de estas, tales como *arrow functions*, *async/await*, etc.

A su vez, también incluye ReactiveJS, una librería de programación asíncrona útil para tratar secuencias de datos observables entre otras funciones.

ReactJS también puede disponer de estas características, pero mediante componentes externos al producto principal, lo cual nos hizo decantarnos hacia la tecnología de Google.

En conclusión, pensamos en Angular como un framework escalable y consistente dada la empresa que lo mantiene, que permite gracias a Typescript un código comprensible, requisito esencial ya que el trabajo no estaría destinado a ser trabajado por un único programador

4.2.2. Node

Node es un *framework* de JavaScript que permite su ejecución en el lado del servidor.

Para ello hace uso de un bucle de eventos no bloqueante que, a diferencia del modelo de concurrencia utilizado por la mayoría de los sistemas (como Apache o Nginx), es de único hilo.

El hecho de que utilice un solo hilo funciona de forma que el bucle de eventos principal recibe las llamadas a eventos de entrada/salida (input/output o IO) y las delega a subprocesos paralelos y, dada la naturaleza asíncrona y no bloqueante del sistema, el bucle principal continúa su ejecución.

Esto permite gestionar en un solo hilo principal todos los potenciales clientes, garantizando la escalabilidad de la aplicación y el aprovechamiento de los recursos disponibles.

Por otro lado, Node es una de las tecnologías que más auge están teniendo en los últimos años y debido a esto dispone de uno de los gestores de paquetes con mayor diversidad (npm o *node package manager*).

Esto permite la instalación de multitud de soluciones y librerías aumentando así las capacidades propias.

Las principales librerías usadas en el proyecto son Express, Mongoose y Socket.io.

4.2.2.1. Express

Express es una librería de Node que facilita el desarrollo de Apis. Para ello establece un sistema de middlewares que son atravesados por cada una de las solicitudes que llegan al servidor.

El enrutamiento que presenta separando para cada verbo HTTP resulta intuitivo y, al ser una de las bibliotecas más utilizadas (más de 4 millones de descargas por semana), dispone de multitud de paquetes que enrobustecen el sistema como Cors o Helmet, otros que incrementan funcionalidades, etc.

4.2.2.2. Mongoose

Mongoose es el mecanismo elegido para establecer conexión con la base de datos Mongo. Posee un constructor de consultas y cuenta con sistemas de validación o programa intermedio.

Una de las ventajas del uso de esta librería es que establece esquemas, lo cual favorece la adaptación al trabajo sin estructuras definidas para desarrolladores provenientes de SQL.

A su vez, al definir los modelos nos permite establecer el patrón modelo-vista-controlador (a partir de ahora MVC) que utilizaremos en la aplicación.

4.2.2.3. Socket.IO

Socket.IO es un sistema para el desarrollo de aplicaciones en tiempo real.

Este utiliza WebSockets como tecnología de transporte, aunque no es una implementación de estos, pues ni es compatible la conexión de un cliente Socket.IO con un servidor WebSockets ni viceversa.

En la bibliografía se encuentra un enlace a las especificaciones del protocolo.

Este mecanismo posee en realidad dos librerías diferenciadas, una para el lado del cliente, que se cargará en Angular y otra en el lado del servidor.

4.2.3. MongoDB

MongoDB es una de las bases de datos NoSQL más utilizadas.



Este sistema de base de datos se encuentra orientado a documentos, que son guardados en formato BSON, una representación binaria de JSON.

Tomamos este tipo de base de datos debido a que es un sistema muy fácilmente escalable, pues la replicación y el escalado horizontal (sharding) no resultan difíciles de configurar.

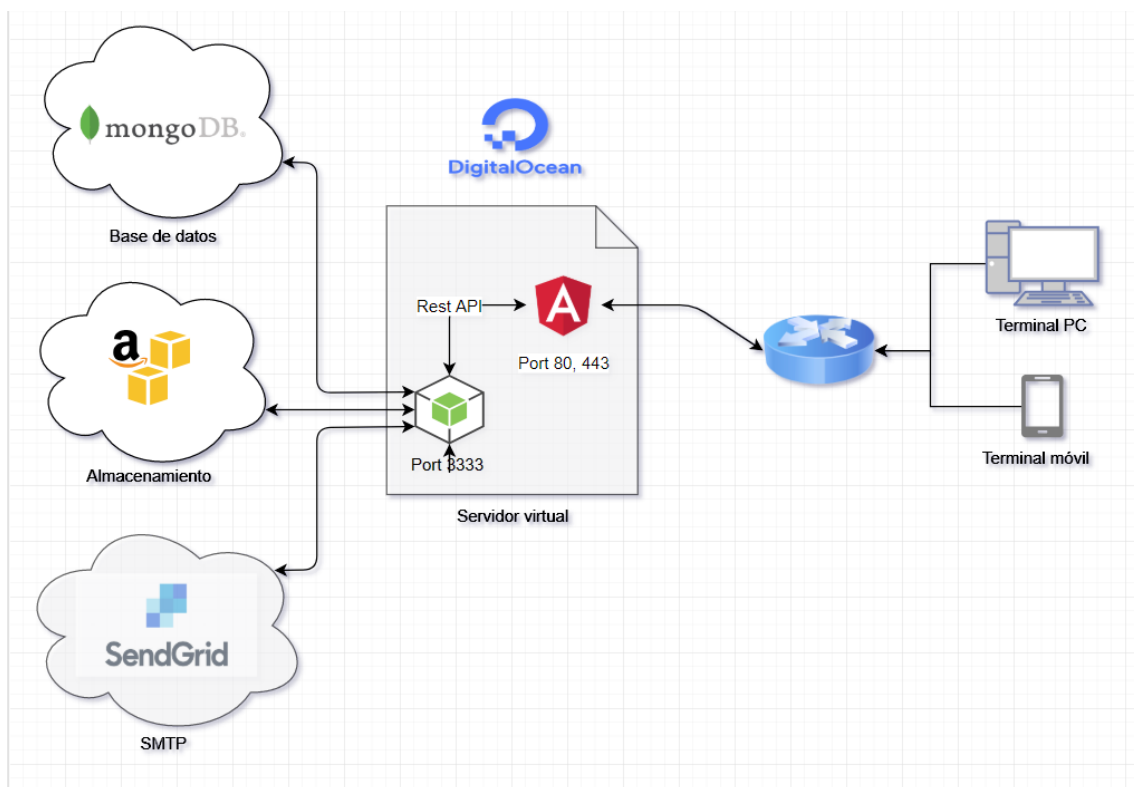
Sin embargo, en un futuro optaríamos por adaptarnos a una base de datos relacional, pues se echan en falta ciertas propiedades de este tipo de sistemas de almacenamiento como las operaciones JOIN o las transacciones.

Las tablas MySQL serían similares a las colecciones y las filas pasarían a llamarse documentos.

5. Arquitectura

En este apartado vamos a explicar los aspectos arquitectónicos del trabajo. Humberto Cervantes (2015) señala que la arquitectura software se refiere a “*las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos.*” [22]

5.1 Esquema de componentes.



6 – Diagrama de componentes de la aplicación.

El esquema que hemos utilizado se basa en el clásico cliente – servidor, pero de forma transparente para el usuario, el servidor se encuentra dividido en dos partes: por un lado, un servidor Nginx, el cual provee los archivos estáticos y por otro el servidor Node, que soporta el API REST.

El usuario, al realizar una primera conexión lanzará una petición al servidor Nginx y recibirá los ficheros JavaScript de Angular con los cuales se construirá la interfaz de usuario.

Estos se guardarán en cache y la sincronización con el servidor se llevará a cabo a través del registro del *service worker* correspondiente.

El concepto de *service worker* lo trataremos más tarde, pues posee gran relevancia a la hora de resolver tres de los requisitos no funcionales: el tiempo de carga, la disponibilidad sin conexión y el servicio para dispositivos móviles.

Una vez servidos los archivos estáticos la aplicación pasará a comunicarse con el API, a su vez nuestro API hace uso de servicios tres servicios externos:

MongoDB como sistema de base de datos se encuentran externalizado en 3 clústeres de MongoDB cloud.

Esto provee de seguridad de datos ante posibles pérdidas al mismo tiempo que evita que los datos se encuentren en la misma localización que los demás sistemas, evitando el problema del punto único de fallo, satisfaciendo el requisito no funcional número 3.

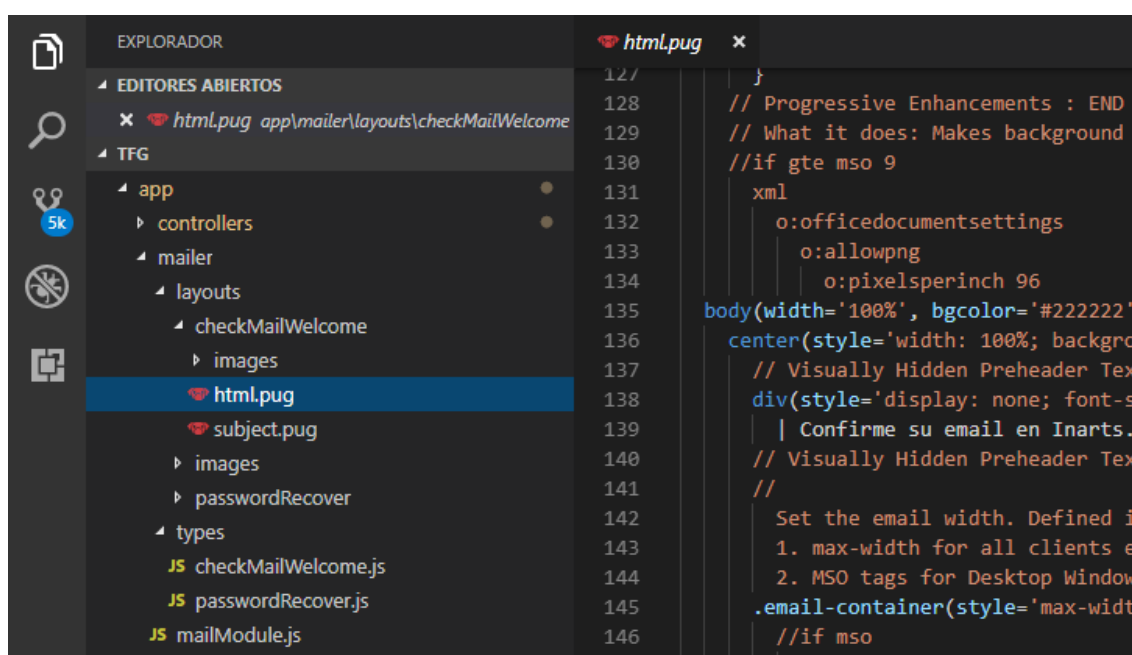
Amazon S3 es un sistema que permite el almacenamiento escalable de datos.

Como web destinada sobre todo a artes plásticas, para los artistas, uno de los recursos indispensables son las imágenes de sus obras, pues es la principal forma de exhibirse y darse a conocer.

Debido a esto, la disponibilidad de las fotos resulta esencial. Para ello, el almacenamiento de estas se ha derivado a este servicio de Amazon, ya que el servicio promete una durabilidad mayor del 99.99% y la capacidad es auto escalable.

A su vez, también se puede replicar los datos a otra región de AWS, lo cual dota de un acceso rápido desde cualquier lugar del mundo, dotando así de un acceso rápido a uno de los recursos que más pueden entorpecer la experiencia del usuario.

Por último, Sendgrid es el servicio de envío de emails que hemos utilizado. En un principio, no íbamos a emplear un sistema de este tipo, utilizando Nodemailer como módulo para mandar los mensajes directamente.



```
127 }
128 // Progressive Enhancements : END
129 // What it does: Makes background i
130 //if gte mso 9
131 xml
132   o:officedocumentsettings
133   o:allowpng
134   o:pixelsperinch 96
135 body(width='100%', bgcolor='#222222',
136 center(style='width: 100%; backgrou
137 // Visually Hidden Preheader Text
138 div(style='display: none; font-si
139 | Confirme su email en Inarts.
140 // Visually Hidden Preheader Text
141 //
142 Set the email width. Defined in
143 1. max-width for all clients ex
144 2. MSO tags for Desktop Windows
145 .email-container(style='max-width
146 //if mso
```

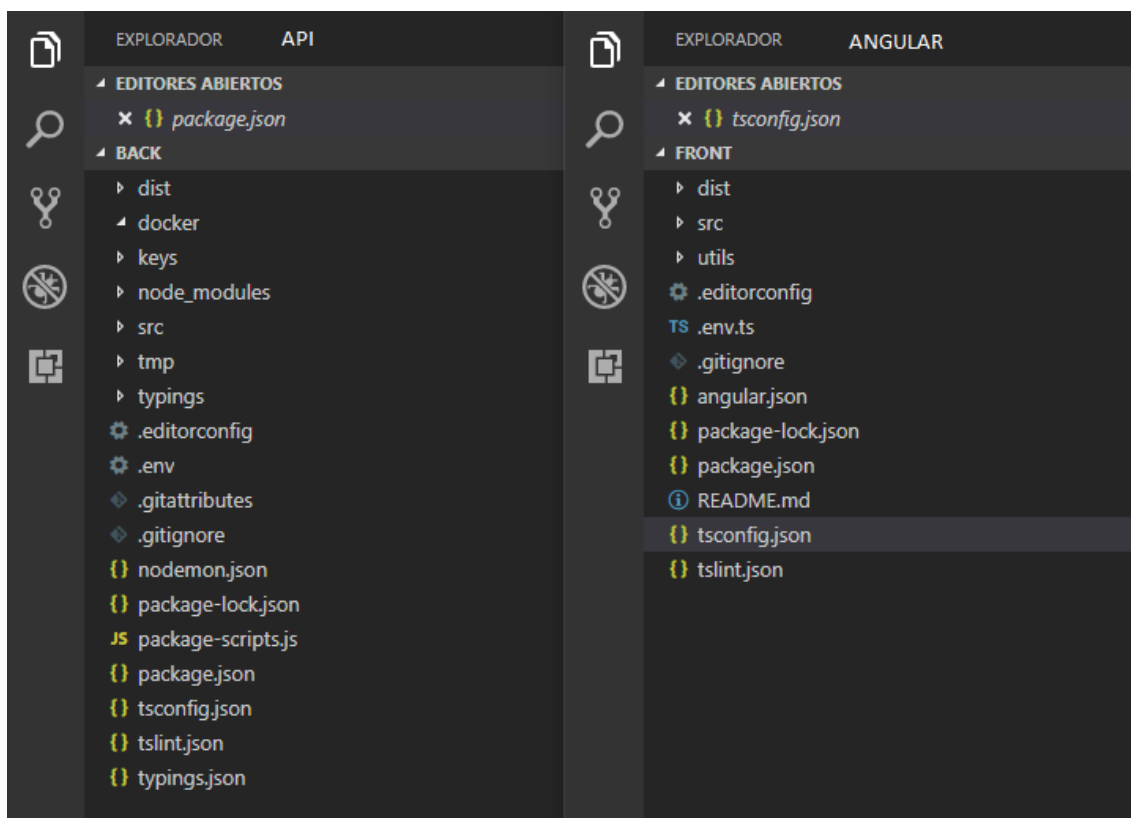
7 - Estructura de archivos con Nodemailer y Pug como sistema de renderizado

Sin embargo, la idea fue cambiada ya que Sendgrid nos ofrecía la capacidad de crear campañas sin problemas como la reputación online o el desarrollo del envío en caso de fallo de entrega.

Debido a esto, y a pesar de la compatibilidad de Nodemailer con el servicio, decidimos trasladar la implementación a llamadas a la API que proporciona esta plataforma.

5.2. Estructura de archivos común.

La estructura de archivos se encuentra dividida en dos directorios principales, por un lado, los archivos correspondientes al API y por otro, la estructura del proyecto Angular.



8 - Directorio raíz del API y Angular

Ambas estructuras, al estar escritas en typescript poseen archivos de configuración comunes (“*tslint.json*” y “*tsconfig.json*”), así como la carpeta “*src*” y “*dist*”.

También los archivos de configuración de Git (“*.gitattributes*” y “*.gitignore*”) y el “*.editorconfig*” que establece unos estándares en cuanto a decisiones de sangrado y cuestiones de estilo del código.

Tsconfig.json es el archivo que utilizará el compilador para la conversión de typescript. En él se define entre otras cosas la versión de JavaScript a la que compilar

(es6 para Node y es5 Angular), la ruta de salida o los alias para rutas en relación con la ruta base.

Tslint contiene la configuración de la herramienta de análisis de código typescript. Este mecanismo lo hemos incluido para mantener estándares en cuanto al desarrollo tanto del frontend como del API.

Las reglas escogidas han sido las de angular.

Src es la carpeta con el código fuente de la aplicación. En esta carpeta se ha llevado a cabo el desarrollo y es la que detallaremos más adelante para explicar el modelo utilizado, el cual se encuentra referenciado en tsconfig.

Dist es el directorio donde se guardan los archivos resultantes del compilado, listos para producción, también se encuentra referenciado en el fichero tsconfig.

Package.json contiene configuración específica para el sistema de paquetes de Node (npm). Aquí se encuentran definidas entre otras, las dependencias y el comportamiento de algunos comandos npm.

Las dependencias se instalan en la carpeta node_modules (presente en ambos proyectos).

```
1  {
2    "name": "inarts",
3    "version": "1.0.0",
4    "description": "awesome RESTful API with NodeJs & TypeScript for Inarts",
5    "main": "src/app.ts",
6    "scripts": {
7      "start": "nps",
8      "test": "npm start test",
9      "build": "npm start build",
10     "presetup": "yarn install",
11     "setup": "npm start config && npm start setup.script"
12   },
13   "engines": {
14     "node": ">=8.0.0"
15   },
16   "repository": "git+ssh://git@github.com/ticquique/backend",
17   "keywords": [],
25   [],
26   "homepage": "https://github.com/ticquique/backend/readme.md",
27   "author": "ticquique <ticquique@gmail.com>",
28   "license": "ISC",
29   "devDependencies": {
52   },
53   "dependencies": {
137  }
138 }
139
```

9 - Package.json del API

5.3. Patrones arquitectónicos.

Un patrón o arquetipo software establece un esquema a la hora de relacionar los diferentes subsistemas de un desarrollo.

En este punto pasaremos a describir los patrones de arquitectura seguidos por los dos bloques principales de desarrollo, por un lado, el backend accesible a través de una API y por otro el frontend Angular.

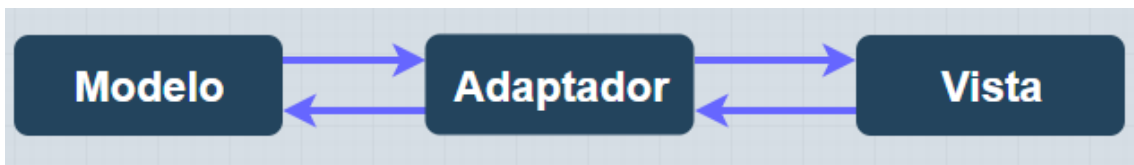
El hecho de separar ambos conceptos se debe a una aproximación a un modelo de microservicios, administrando mediante una única interfaz de usuario uniforme, las llamadas de la API.

5.3.1. Backend

En el desarrollo del Backend resultaba esencial mantener los conceptos bien diferenciados, ya que las características de este, con el tiempo, abarcarán más cuestiones y posibilidades muy heterogéneas entre sí, y sin una jerarquía marcada, podría complicarse el desarrollo.

Por esta razón decidimos implementar un modelo – vista – adaptador o “*Mediating-controller*”.

Este tipo de esquema es una evolución del clásico modelo – vista – controlador (mvc a partir de ahora), pero a diferencia del anterior, no forma el triángulo, ya que la vista jamás se comunica directamente con el modelo, estableciendo al adaptador como único conocedor de, tanto la vista como el modelo.



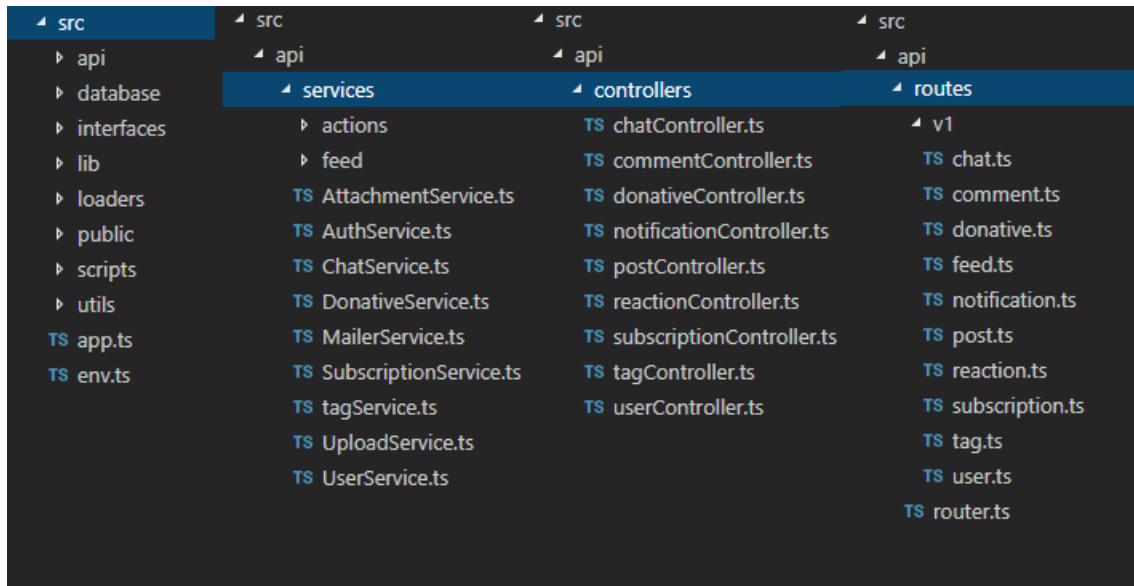
10 - Modelo - Vista – Adaptador



11 - Modelo - Vista – Controlador

Este arquetipo abstrae la comunicación entre vista y modelos, permitiendo a través de un solo adaptador acceder a multitud de modelos, algo necesario para la cohesión entre todas las funcionalidades que deberá ofrecer.

Para lograr este objetivo, se ha estructurado la aplicación en los siguientes apartados:



12 - Estructura API

En la anterior ilustración se muestran los tres directorios que tratan de seguir este modelo.

Por un lado, se encuentra “*services*”, la cual, contiene la comunicación con la base de datos, por lo que constituye los modelos.

Por otro lado, en “*controllers*” se encuentran los adaptadores, pues se encarga de dar soporte a las peticiones recibidas y mediante el uso de los servicios generar una respuesta, que pasará a “*routes*”.

Esta última actuaría como la “*vista*” pues contiene los *endpoints* de la API.

Cada uno de los anteriores actores se encargará de las funciones específicas que hemos descrito, generando un ecosistema accesible e intuitivo para nuevos desarrolladores, dado que el tratamiento de las importaciones se abstrae del código, estableciendo un marco para los ficheros generados en cada directorio.

A su vez, hemos implementado una especie de patrón “*singleton*” con los servicios y adaptadores para mantener la referencia de la instancia, ocultando el constructor (privado) tras el método estático “*getInstance ()*”

```
private constructor(ioServer?: io.Server) {
  if (ioServer) {
    this.authService = AuthService.getInstance();
    this.userService = UserService.getInstance();
    this.actionService = ActionService.getInstance();
    this.notificationService = NotificationService.getInstance();
    this.logger = new Logger(__dirname);
    this.server = ioServer;
    this.onConnect(this.listeners);
  }
}

public static getInstance(ioServer?: io.Server): ChatService {
  if (!ChatService.instance) {
    ChatService.instance = new ChatService(ioServer || null);
  }
  return ChatService.instance;
}
```

13 - Implementación de patrón singleton en typescript del servicio de chat

5.3.2. Frontend

Angular utiliza un patrón de inyección de dependencias. Este patrón consiste en que las clases solicitan recursos en lugar de crearlos. Para ello, las dependencias deben declararse en el constructor y las clases inyectables deben ser llamadas en los proveedores del módulo.

Para dar forma a esto, decidimos crear un módulo llamado “*shared*”, el cual se encargaría de inyectar aquellos servicios, “*pipes*” y componentes comunes en distintos módulos de la aplicación.

El otro módulo que destacar es el llamado “*errors*”, que se encuentra en la carpeta “*core*”. En él se encuentran los módulos que requieren de una única instancia en toda la aplicación, de forma que cada módulo cargado de forma diferida o “*lazy loaded*” no disponga de su propia instancia del singleton.

El sistema de captura de errores intercepta respuestas de error y tras reintentar la petición dos veces redirige a una página dedicada a proporcionar retroalimentación del fallo al usuario.

Lo particular del módulo es su constructor, dado que en este radica la particularidad de los módulos ubicados en “*core*”.

```
export class ErrorsModule {
  constructor( @Optional() @SkipSelf() parentModule: ErrorsModule) {
    throwIfAlreadyLoaded(parentModule, 'ErrorsModule');
  }
}
```

14 - Constructor especial de errors

Finalmente, el resto de las carpetas corresponden a diferentes componentes individuales, encargándose de las distintas interfaces con las que el usuario interactúa con el sistema.

5.4. Arquitectura de base de datos.

El almacenamiento, como ya hemos comentado con anterioridad, utiliza el servicio MongoDB ubicado en MongoDB cloud.

Este sistema no obliga a mantener una estructura en sus colecciones o documentos. Sin embargo, para el mantenimiento del código pensamos que era necesario establecer un modelo que defina la forma de los documentos de cada colección.

Para llevarlo a cabo, creamos la carpeta *database* en *src* del *backend* donde se encuentran definidos los esquemas mediante *Mongoose*.

```
// Reaction Schema
const ReactionSchema = new Schema({
  type: {
    type: String,
    enum: reactions,
  },
  user: {
    type: String,
    required: true,
    ref: 'User',
  },
  related: {
    type: Schema.Types.ObjectId,
    required: true,
    refPath: 'reference',
  },
  reference: {
    type: String,
    required: true,
    enum: references,
    default: 'Post',
  },
}, schemaOptions);
export const Reaction = model<IReactionModel>('Reaction', ReactionSchema);
```

15 - Esquema Mongoose de la colección reactions

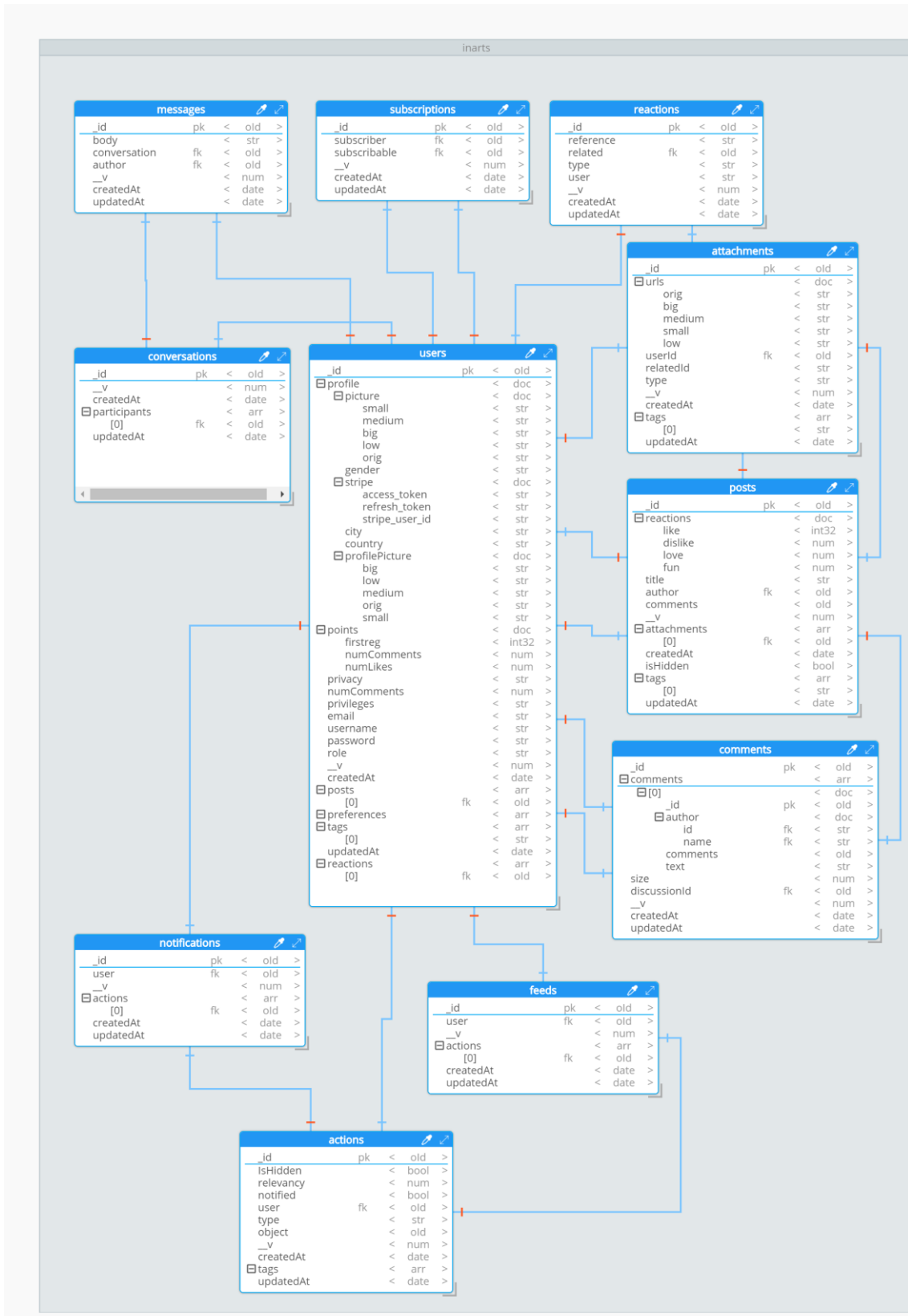
La base de datos se ha diseñado pensando en la optimización de las consultas a realizar.

Al ser Mongo una base de datos orientada a documentos nos permite crear anidaciones de estos como campo de otro. Esta capacidad resulta interesante a la hora de optimizar operaciones habituales sobre relaciones de uno a muchos como listar las publicaciones de un usuario o las reacciones de este.

Sin embargo, este sistema provoca un esfuerzo extra a la hora de programar los controladores de los métodos de inserción y borrado, ya que hay que tomar en cuenta cada una de las referencias al documento afectado.

A continuación, se muestra el esquema completo de la base de datos generado mediante la herramienta *Hackolade* (imagen 16).

Inarts, la red social de contacto entre artistas y editoriales



16 - Hackolade esquema base de datos

6. Diseño gráfico

En este apartado vamos a hablar de las pautas que hemos seguido en el proceso de nombrado del proyecto, cómo surgió el logotipo, las pruebas de color realizadas, etc.

6.1. Nombrado.

Fueron varios los conceptos surgidos en la lluvia de ideas para decidir un nombre adecuado que representara la esencia y la funcionalidad de nuestra red social. En un principio se valoraron opciones como *Netart*, siempre asociando el arte con la conectividad.

A primera vista no presentaba ningún problema, sin embargo, realmente esta web trata de conectar gente que ya se encuentra en el campo del arte con posibles agentes editoriales en una redacción que puede acabar en un proyecto de trabajo. Por tanto, quedó descartada al no mostrar la inclusividad que se buscaba.

Retomamos nuestra búsqueda para encontrar otro nombre, llegando a *Webart*, nombre que incluía conceptos que, a pesar de distanciarse de lo anterior, mantenía una esencia básica, la idea de la web como lugar de encuentro.

Esto se plasmaba a la perfección, sin embargo, tampoco fue elegido dado que ya existía una agencia con ese nombre relacionada con marketing, por lo que ganar posicionamiento frente a ellos y hacerse visibles no sería tarea fácil.

Por último, combinando las palabras arte y “dentro” en inglés (*in*) se llegó a *Inarts*. Este nombre fue un resultado muy interesante, ya que transmitía de forma precisa lo que tenía por definición.

Derivando *in* se transformó en el lema con “Inclusividad”, “interacción”, “dentro” y *arts* completó el sentido, culminando la búsqueda.

6.2. Logotipo.

Como primera referencia se tomó el logotipo de la empresa de comida Taco Bell, pues observamos que las formas del interior recordaban a una arroba, símbolo directamente relacionado con el mundo de la web.

Con esta idea de jugar con las formas curvas, llevamos a cabo una lluvia de ideas con conceptos relacionados con el arte, los editores y la informática. De esto surgieron figuras como arrobas, pinceles, lápices, libros.

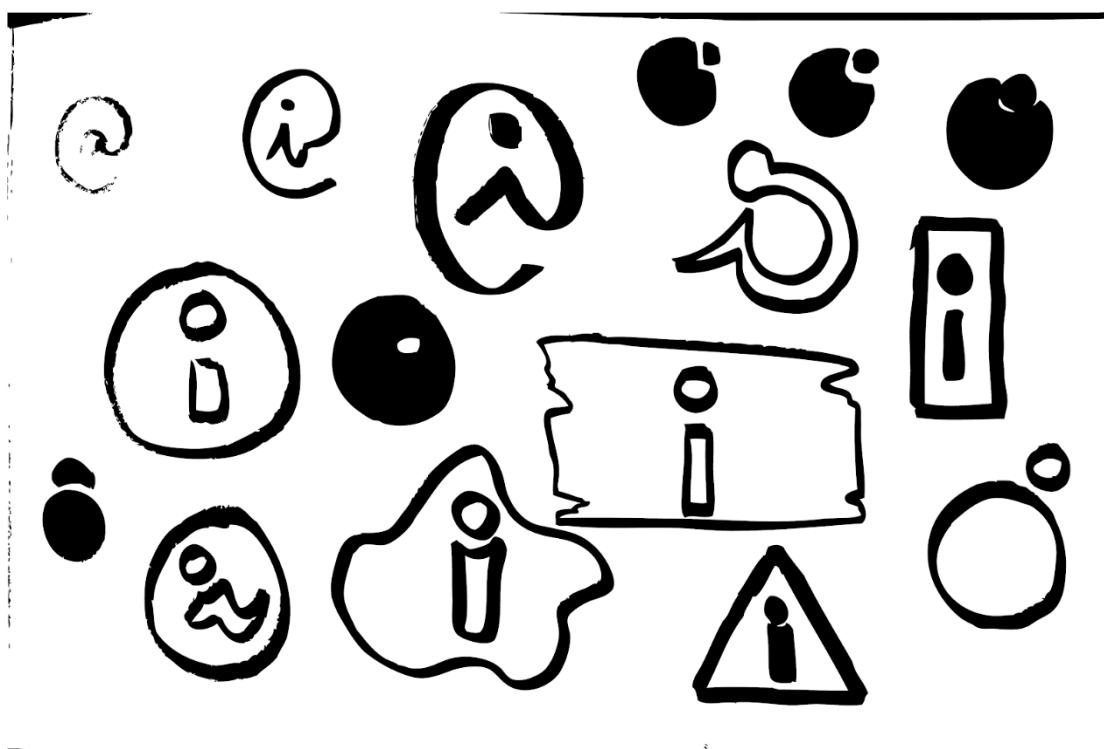
Empezamos a jugar primero con estas formas, juntándolas, deformándolas, recortando partes... También jugamos, en unos bocetos muy básicos, con la letra “I”, inicial del nombre de este proyecto.

Pusimos la inicial sobre fondos circulares, rectangulares a modo de tablón de madera, de triángulo, etc.

Estas ideas fueron descartadas porque no aportaban nada al significado principal del logotipo, no identificaba la marca y, por tanto, no tenía sentido dejarlo así.

Posteriormente, empezamos a trabajar con la idea de la “i” y una arroba, símbolo comentado anteriormente.

Hicimos diferentes versiones en las cuales el círculo estaba más o menos deformado, se abría de una manera o de otra y la letra se combinaba de maneras distintas.



17 - Bocetos del logo de Inarts en blanco y negro

En algunos bocetos, la forma de la arroba era mucho más cerrada en torno a la letra, cosa que posteriormente se cambió.

Fuimos modificando la dirección, el ángulo y añadiendo a modo decorativo una línea por detrás, que atravesara el elemento principal. Con esta línea tratamos de dotar al símbolo de mayor cercanía al arte, ya que el registro, aunque se realizara de forma digital, iba a ser semejante a un trazo de lápiz.

En este punto ya comenzamos a trabajar con el programa Adobe Illustrator, ya que era mucho más cómodo para cambiar rápidamente los elementos y borrar o añadir nuevos.

En las primeras pruebas, el elemento principal se mantuvo, pero variamos el aspecto de la línea trasera, aumentando su amplitud, su largo o su dirección.

Con este mismo archivo trabajamos distintas pruebas de color, intentando siempre guiarnos por cierta psicología del color. En sus primeras versiones tratamos sistemáticamente de combinar dos colores complementarios, cosa que le daba potencia a la imagen representativa de nuestra marca.



18 - Bocetos a color

Se realizaron combinaciones de naranja y verde, naranja y morado, borgoña con verde claro y este mismo verde con morado oscuro, junto con pruebas de azul claro y fucsia.

Dado que ninguna de estas combinaciones representaba lo suficiente a la marca de “Inarts”, se realizó otra prueba en morado y amarillo.

Esta fue la escogida finalmente, dado que el morado representa ese sentido elegante y ecléctico tan particular del mundo del arte, donde se reúne un sinfín de disciplinas, mientras que el color amarillo, se asocia a la diversión, la espontaneidad y la felicidad.

Con todo esto construimos un logo muy enriquecido y que concordaba con los valores de la marca en cuestión.

Para llegar al arte final sometimos el logo a una prueba de reducción para ver si era funcional y podría adaptarse a desde los formatos más reducidos a los más grandes.

Desgraciadamente, en este punto nos dimos cuenta de que la imagen de la “i” se asemejaba en exceso a una “g” y, además, en los formatos más reducidos como el *favicon* para la parte superior de la pestaña, se perdía la línea amarillenta al ser una forma rugosa como el trazo de una cera y no sólida.

Por tanto, cambiamos a una forma más pulida tanto del elemento principal como de la franja de color.

Alargamos el gancho de la i, que parte de una forma creada manualmente basándonos en las fuentes con serifa, hasta deformarlo completamente hacia la izquierda.

Repetimos este proceso también en la parte derecha, elevándolo hasta juntarlo con el punto de la “i”, buscando así que el elemento se transformara hasta estar más cercano a la arroba que a la letra misma.

Ya que el peso visual, aunque estuviera medianamente equilibrado, se centraba en la parte izquierda, pusimos el decorado de la línea amarilla a modo de pincelada desde el centro de la composición hacia la parte superior derecha.

También probamos a darle sentido rugoso a la letra en sí, pero esto quedó automáticamente descartado al haber un exceso de textura en la imagen.

La imagen de la letra fue resuelta con vectores, dejando así un trazo uniforme el cual movilizamos con el uso de la pluma.

El último punto fue eliminar finalmente el toque de trazo amarillo, ya que causaba impedimento en la reducción y por lo tanto ensuciaba la imagen en los formatos más pequeños de aplicación del logo.



19 - Arte final de Inarts

La imagen final de la marca fue un isologo con la inicial de “Inarts” combinado con la arroba, en color morado y un acabado vectorial que podemos ver en la imagen 19. Al guardarlo generamos los diferentes tamaños de favicon para una adaptabilidad perfecta a toda clase de dispositivos, desde 16x16 hasta 512x512 pasando por 32x32, 64x64 o 96x96.

6.3. Estética general.

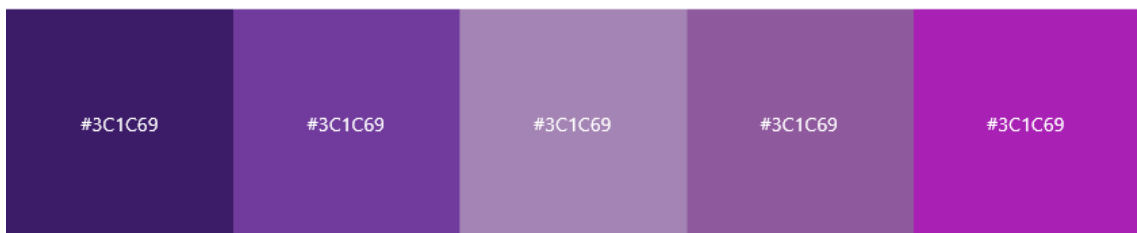
En el apartado que viene a continuación se explicará todo aquello relacionado con el diseño de la página en sí.

La estética general se basa en los conceptos del diseño flat y, por tanto, se emplean colores y formas planas tratando de favorecer la mayor usabilidad y funcionalidad posibles.

Se busca hacer un diseño con la menor cantidad de elementos posibles, eliminando lo superfluo y haciendo que pasar a *responsive* sea muchísimo más sencillo.

Guiándonos por este estilo, buscamos también simplicidad en cuanto a la gama de colores utilizados.

Para esto, lo primero fue fijarse en el isologo realizado con color morado. A partir de aquí, y con el uso de paletas generadoras de color como Adobe Color CC, nos quedamos con cerca de unos 5 tonos de color, de tonalidades más claras a las más oscuras, sobre las que poder combinar el texto y otros elementos sin impedimentos visuales en cuanto al color.



20 - Paleta de color

Al ir realizando diferentes bordes y sombras, nos dimos cuenta de que nuestro estilo se acercaba mucho más al *Material Design*.

Este es un estilo fue popularizado por Google y la inclusión de luz y sombras permite la jerarquía de elementos, dando sensación de profundidad y orden.

6.4. Mockups

Como referencia se tomaron las principales redes sociales hoy en día: Facebook y Twitter, y, en menor medida, Instagram y YouTube.

En la primera etapa del trabajo se realizaron bocetos a lápiz o *wireframes*. Esto se realizó para establecer una posible distribución de elementos en algunas de las páginas, así como para decidir qué iba a contener cada pantalla, de forma que cumplieran los requisitos funcionales establecidos.

En los primeros bocetos se diseñó el login y se pusieron anotaciones sobre las diferentes partes de este, incluyendo un apartado para presentar a la marca, el contacto o las redes sociales.

Con las otras funcionalidades realizamos el mismo procedimiento, dejando claros los apartados de cada página y anotando los puntos en común con el perfil de artista o el de editor.

Establecimos los elementos por un sistema de columnas y de líneas horizontales para cuadrar los elementos en el espacio, facilitando posteriormente el trabajo con el framework css que íbamos a utilizar (Bootstrap).

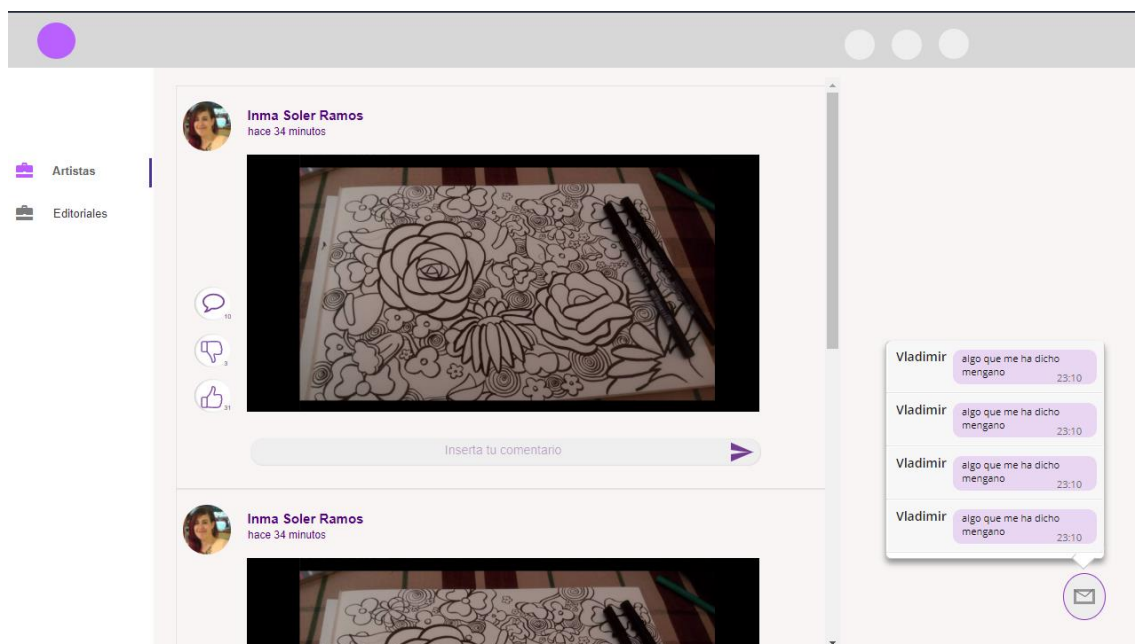
Una vez las páginas estuvieron claras, pasamos a trabajar con un software de prototipado llamado "Atomic.io". Con él se realizaron los mockups mediante los cuales fuimos alcanzando una idea más cercana al resultado final en cuanto formas y la paleta de color antes descrita.

A continuación, se presentan una selección de mockups realizados con esta herramienta y pasaremos a hablar del diseño móvil.

6.4.1. Mockups web

Primero vamos a tratar de unos cuantos realizados para web.

Esta selección contiene aquellos diseños que comprenden varios casos de uso, y los describiremos posteriormente a su visualización.



21 - Mockup feed, chat y post

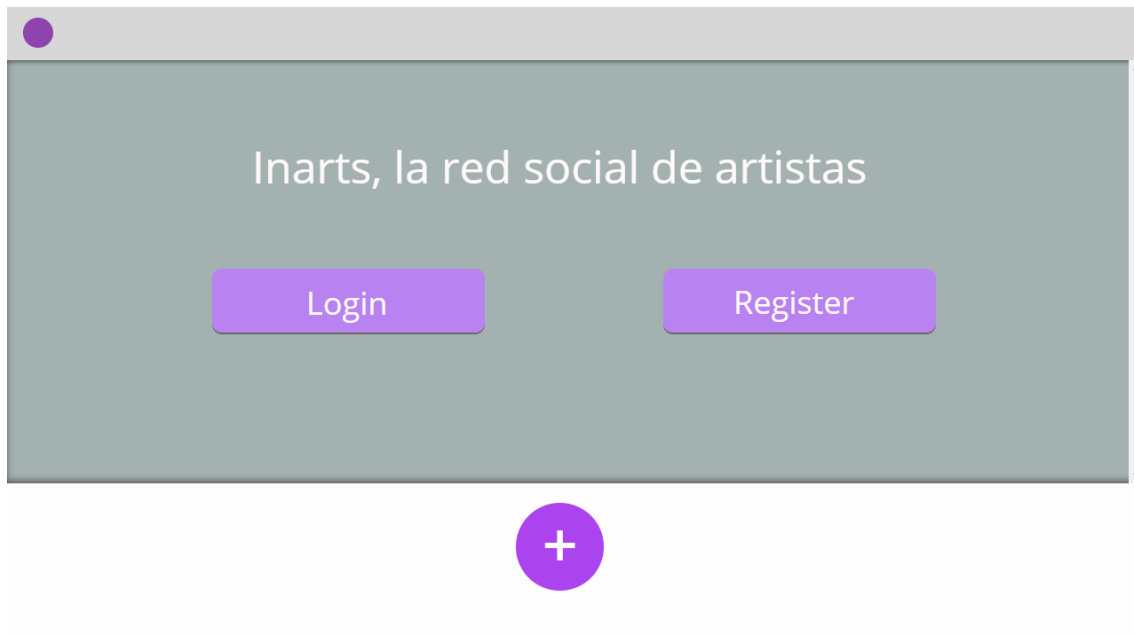
En esta primera imagen podemos observar el aspecto general de la página.

Primero, en la parte superior se encuentra el menú, que estará fijado en esa posición para mantener el acceso a los diferentes sitios del sistema en cualquier momento.

En la parte izquierda se puede observar un filtro para ver solo aquellas publicaciones de uno u otro rol.

A continuación de los filtros, en el centro y dominando la visión, se encuentra el listado de posts, el cual contiene un resumen del contenido de este y a su vez, desde esta interfaz podremos valorarlo y comentarlo.

Para la carga de más contenido utilizamos la técnica del *scroll infinito* de forma similar a Facebook.

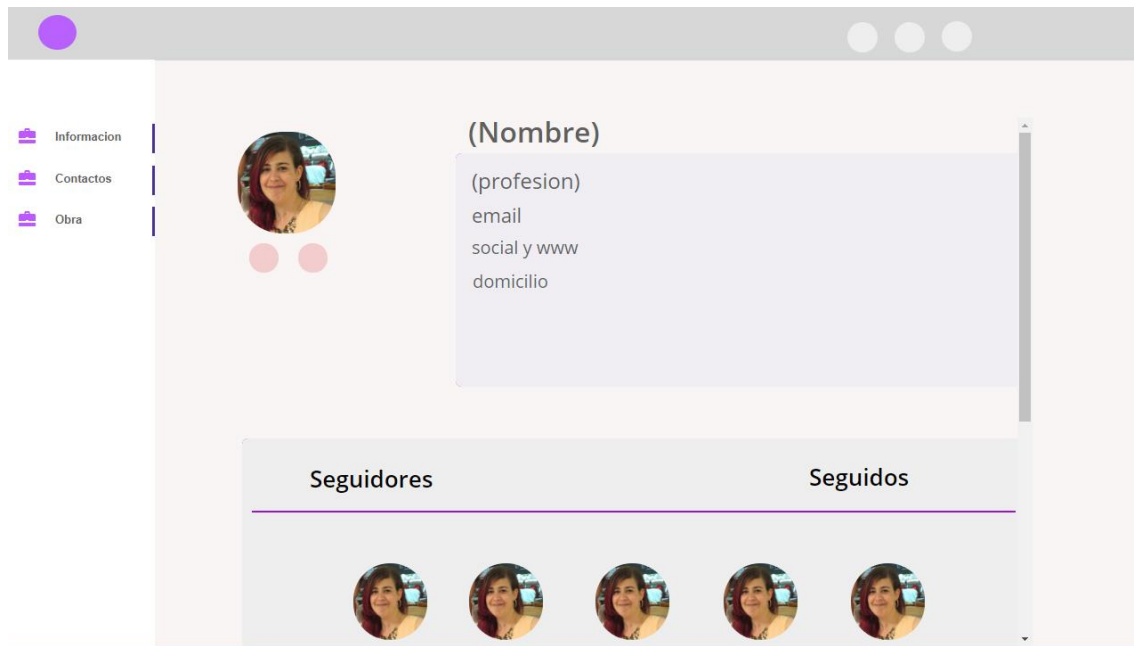


22 - Landing page

La *landing* como indica su nombre es la página de aterrizaje, es decir, la primera que van a observar los usuarios al entrar en tu sitio.

Debido a esto, debe resultar llamativa y contener una breve descripción de lo que van a encontrar, así como las acciones que pueden realizar.

En este diseño, a modo de concepto introducimos los botones de iniciar sesión y registro, así como uno más que, al pasar el ratón por encima, mostrara una descripción de quienes somos y la razón del proyecto



23 - Mockup del perfil de usuario

El último planteamiento que vamos a mostrar es el del perfil, donde se mostrará la información personal de cada usuario, desde los datos personales que decida introducir o sus publicaciones hasta los seguidores y seguidos que posee.

6.4.2. Mockups móviles

El diseño de la aplicación fue una búsqueda para sintetizar la web haciéndola en un formato que se pudiera adaptar a la vista en móvil.

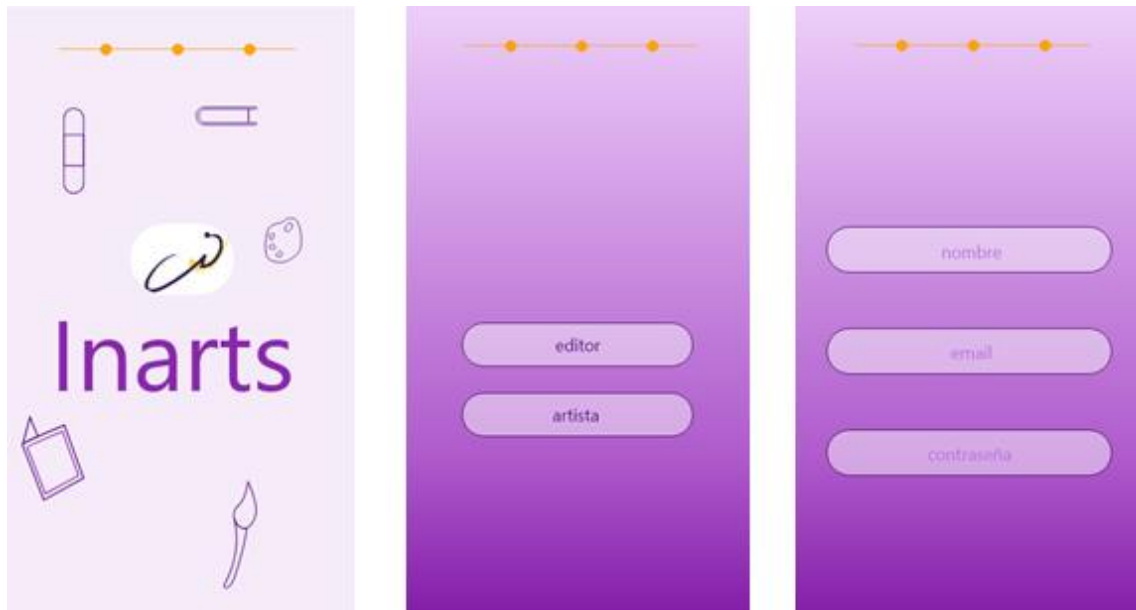
Volvimos a trabajar sobre la misma paleta de color que en los mockups y se añadieron recursos gráficos como el desenfoque sobre las imágenes del feed.

En el caso de la aplicación, se utilizó un degradado sobre el fondo, ya que, aunque se sale un poco de las normas del diseño flat, nos sirvió para romper con el problema de una excesiva simplicidad y les dio un ligero relieve a los elementos.

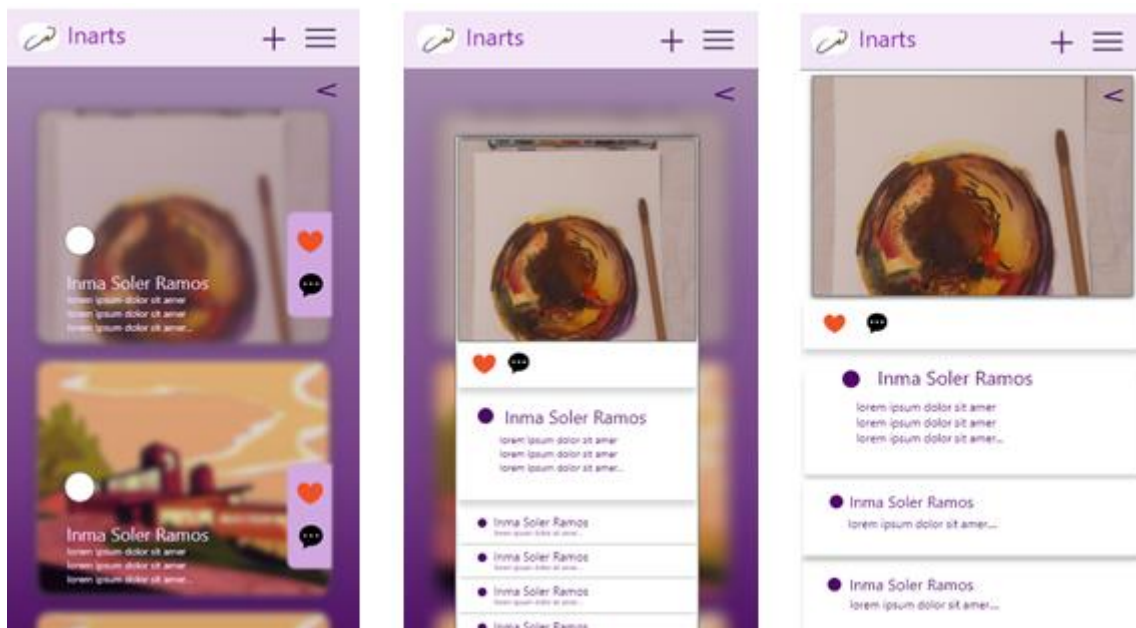
En cuanto al diseño de la pantalla de carga añadimos iconos vectoriales diseñados previamente, para decorar esta pantalla y dinamizarla.

En esta etapa Adobe anunció que su herramienta de diseño pasaría a ser gratuita, como ya había sido durante su etapa beta. *Experience Design* era un programa con el que ya teníamos experiencia por lo que la decisión de utilizarlo fue inmediata.

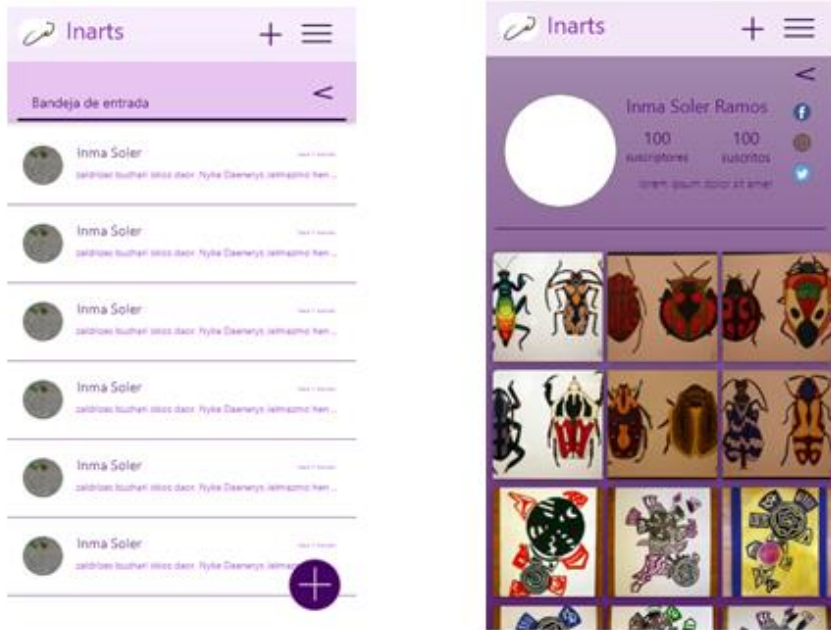
A continuación, se presenta otra serie de mockups relacionados para el diseño de la aplicación móvil.



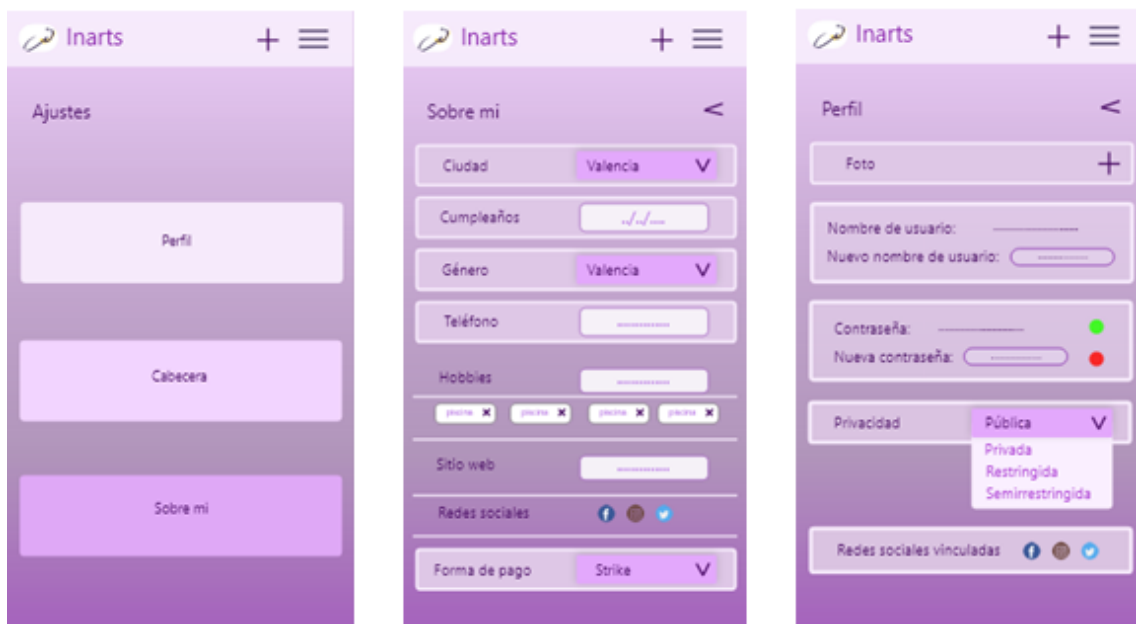
24 - Proceso de registro



25 - Mockup listado de publicaciones, previsualización y publicación individual



26 - Mockups bandeja de entrada del chat y mockup de perfil



27 - Diseño de ajustes de usuario

7. Seguridad

En este apartado nos centraremos en los componentes empleados para conseguir que nuestra arquitectura posea un carácter relativamente seguro para un entorno de producción de forma sostenible.

Citando a Chema Alonso en el ESET Security Day en Valencia, “Do the basics”. Esto significa que antes de elaborar soluciones complejas, se debe manejar lo más básico.

La seguridad al fin y al cabo consiste en gestionar los riesgos, de forma que se elaboró la siguiente declaración, con los elementos a cubrir y la manera de realizarlo.

Todas las llamadas al API han de pasar por una serie de filtros para la comprobación de datos.

Por un lado, el propio framework de angular crea una primera capa de seguridad evitando (a través de control sobre modificaciones del árbol DOM) potenciales ataques contra el servicio.

Por otro lado, mediante este mismo framework nos encargamos de sanear los datos que pasan al servidor a modo de segunda capa de protección, evitando ataques XSS.

Justo después de recibir la petición a través de la ruta establecida, se debe comprobar que los datos provistos por el cliente son los requeridos para la solicitud.

En el lado del servidor, la solicitud es filtrada y transformada para que sea acorde a los requisitos de la base de datos (everything lowercase entre otras cuestiones).

Tras esto, se procesa la solicitud con las llamadas correspondientes a los modelos de Mongoose que se requieran. Los modelos de Mongoose se utilizan al mismo tiempo para comprobar de nuevo la corrección de los datos introducidos, establecen un esquema para la base de datos no relacional y generando algunas restricciones.

De esta manera, se hace posible cierto grado de control sobre el carácter de los documentos a guardar.

Todo este proceso de procesamiento de datos a diferentes niveles permite tratar de forma segura las entradas que recibe el sistema, robusteciendo la arquitectura con mecanismos que permitan tratar una amplia variedad de ataques.

A su vez, el servicio web y el API se encuentran tras un proxy inverso Nginx, el cual abstrae el API express y permite generar un certificado SSL único mediante Letsencrypt dotando de un único punto de acceso al sistema mediante conexión segura.

Por otro lado, para proteger datos especialmente sensibles como las contraseñas, estos se guardan encriptados utilizando la librería bcrypt con un elemento Salt generado con un factor de coste de 10 (lo cual implica 2^{10} iteraciones de la función de derivación de clave). Este mecanismo evita ataques de tipo attack vector.

Un punto importante es la cuestión de la autenticación, pues establece los datos accesibles por cada usuario.

En una arquitectura basada en REST API, debe respetarse el principio de que esta información debe mantenerse en el lado del cliente, evitando el uso de sesiones, por lo que es el cliente el que en cada petición enviara su identidad.

Para esto, se ha implementado un mecanismo de autenticación mediante JSON web tokens. Este sistema consiste en que, tras la validación de credenciales realizada por la API, esta genera un token que envía al cliente.

Este estándar de autenticación se encuentra definido en la TFC7519 [30], donde se indica que el token debe contener tres campos:

El *payload*, que contiene información útil para el usuario, el *header*, que indica el tipo de dato enviado (JWT) y el algoritmo de encriptación (SHA-256 en nuestro caso) entre otros parámetros y finalmente la *Signature* o firma que se utiliza para validar el token y consiste en el propio *header* y *payload* codificados en base64, concatenados mediante un punto y finalmente cifrados mediante el algoritmo de encriptación descrito en el *header*

El cliente guarda este token y lo adjunta con cada solicitud en la cabecera AUTHORIZATION de http mediante el esquema Bearer (Authorization: Bearer <token>).

```

public createToken = (user: IUserModel): string => {
  const payload: AuthToken = {
    // iss: 'my.domain.com',
    sub: user._id,
    privileges: user.privileges,
    role: user.role,
    iat: moment().unix(),
    exp: moment().add(7, 'days').unix(),
  };
  const secretOrKey = env.app.secret;
  const token = jwt.sign(payload, secretOrKey, { algorithm: 'HS256' });
  return token;
}

public validateToken = (token: string): Promise<AuthToken> => {
  return new Promise((resolve, reject) => {
    if (typeof token === 'string') {
      const parts = token.split(' ');
      if (parts.length === 2) {
        const scheme = parts[0];
        const credentials = parts[1];
        if (/^Bearer$/i.test(scheme)) {
          jwt.verify(credentials, env.app.secret, {
            algorithms: ['HS256']
          }, (err: any, decoded: any) => {
            if (err) { reject(err.message); } else {
              if (env.auth.revoquedTokens.indexOf(decoded.sub) > -1) {
                reject(new Error('User banned for some reason, check your email'));
              } else { resolve(decoded); }
            }
          });
        } else { reject(new Error('Format is Authorization: Bearer [token]')); }
      } else { reject(new Error('Invalid credentials')); }
    } else {
      reject(new Error('Invalid token provided'));
    }
  });
}

```

28 - Función de creación y validación del token

```

export const authMiddleware = (
  level: 'min' | 'basic' | 'admin' | 'super'
) => (req: express.Request, res: express.Response, next: express.NextFunction) => {

  const authService = AuthService.getInstance();
  new Promise((resolve, reject) => {
    if (req.headers && req.get('api_key')) {
      ApiKey.find({ key: req.get('api_key') }, (err, doc) => {
        if (err || !doc.length) { reject(); } else {
          if (doc[0].key_name === 'admin') { resolve(); } else { reject(); }
        }
      });
    } else { reject(); }
  }).then(isApi => {next(); }).catch(() => {
    if (level === 'basic' || level === 'min') {
      if (req.headers && req.get('authorization')) {
        authService.validateToken(req.get('authorization'))
          .then((decoded) => {
            res.locals.user = decoded;
            next();
          })
          .catch((err) => {
            const e = new HttpError(env.api.error, err.message);
            next(e);
          });
      } else {
        if (level === 'basic') {
          const e = new HttpError(env.api.error, 'Log in to continue');
          next(e);
        } else {
          next();
        }
      }
    } else {
      const e = new HttpError(env.api.error, 'Permission denied');
      next(e);
    }
  });
};

```

29 - Middleware de detección del token

Como últimas cuestiones que se han de tener en cuenta es el envío de datos por el usuario al servidor y el uso de claves públicas para los otros servicios externos.

Los datos enviados por el usuario siempre hacen uso del método POST de http, lo que junto al uso de https evita que los datos puedan ser extraídos por un ataque de tipo *man in the middle* haciendo privado el intercambio de datos.

Finalmente, las claves para el uso de estos microservicios de email o almacenamiento en nube se encuentran en el servidor Node, lo que evita que sean compartidas con el usuario final, protegiendo estas comunicaciones.

A modo de resumen, sabemos que no existe la seguridad completa en el mundo de la informática y que todo entorno debe llegar a un acuerdo entre coste y beneficio, sin



Inarts, la red social de contacto entre artistas y editoriales

embargo, los puntos más básicos y esenciales se encuentran cubiertos implementando los mecanismos mencionados.



8. Implementación

En este apartado vamos a explicar los detalles de cómo se ha realizado cada una de las funcionalidades del proyecto. También explicaremos como se establecen algunos flujos de datos y la razón de esto.

8.1. Registro.

El flujo de datos del registro comienza cuando el cliente hace clic en el botón de registro de la *landing page* o si entra desde el menú.

Una vez iniciada la ventana modal correspondiente, se le solicita al usuario el nombre, el correo electrónico y el rol (a elegir entre artista y editor).

Estos datos son saneados en el lado del cliente antes de enviarse a través del API (posteriores saneamientos serán omitidos ya que a efectos prácticos son los descritos en el apartado 7 relacionados con la seguridad).

Una vez en el servidor, se comprueban las restricciones asociadas a esos datos.

En el caso del registro se determinará la unicidad del nombre y del correo introducidos, respondiendo al cliente y mostrando bajo el formulario la causa del error en caso de no cumplir dichas restricciones.

Tras comprobar la validez de la información obtenida, esta se procesa creando un documento en la base de datos de tipo *Valid*, el cual guarda la información asociada al nuevo cliente.

Tras este procesado, se responde al cliente con un código 200 y un mensaje también bajo del formulario indicando que se ha procesado correctamente y que en breves recibirá un correo con la información necesaria para acabar con la creación de su cuenta.

Los datos almacenados en el *Valid* consisten principalmente en los introducidos por el usuario, junto con una contraseña generada aleatoriamente mediante un identificador único universal (uuidV4) y un token también aleatorio.

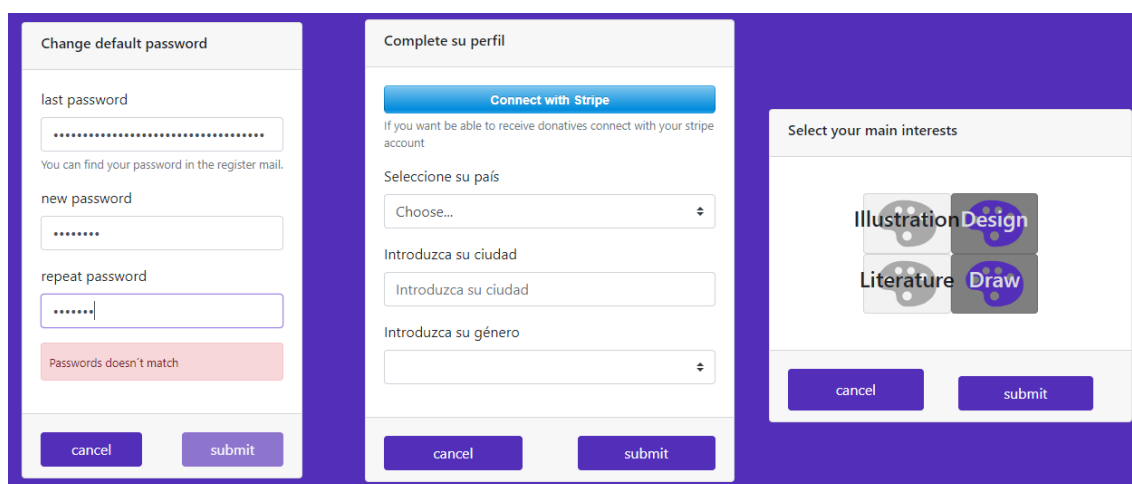
Finalmente se cierra la modal al cabo de 2 segundos mediante un efecto de *fade*, dándole al usuario una sensación de fluidez y devolviéndolo a la versión inicial de la web.

Cabe destacar que los mensajes de error presentes informan de forma diferenciada si el error proviene del email o del correo utilizado. Esto podría plantearse como un error de seguridad, ya que en caso de un atacante pretendiera realizar una suplantación de identidad, este recibiría una información de importancia.

Sin embargo, por el momento se ha valorado como una buena práctica dado que realmente se informa con precisión del elemento a modificar.

Continuando el proceso, en el correo recibido por el usuario se encuentra la contraseña provista por el sistema junto con un enlace.

Este enlace contiene una extensión con el token generado. Cuando el usuario sigue el *link*, entra en un formulario en el que se le solicita la contraseña del correo y se le pide que introduzca una nueva.



30 - Proceso de primer cambio de contraseña y modales para completar el perfil

Una vez validada la corrección de la contraseña introducida, finalmente, el usuario se crea en la base de datos, el cliente angular recibe y guarda en almacenamiento local el token que adjuntará en posteriores solicitudes al API (el cual servirá como mecanismo de autenticación) y se le redirige a la ventana de inicio de la aplicación donde se permitirá completar la información asociada a su perfil.

De esta manera, a partir de ese instante este usuario podrá realizar el proceso de login con las credenciales nombre de usuario y contraseña.

8.2. Correo electrónico.

Para el envío de correos hemos utilizado *sendgrid* como ya se mencionó en el apartado 5.1 en el esquema de componentes.

Para comunicarnos con esta solución se ha implementado un servicio específico que es el encargado de realizar las solicitudes correspondientes a través de la librería que dispone [13].

En concreto, como se muestra en la imagen 29, se han implementado dos tipos de mensajes disponibles en la función `newMail` del servicio, por un lado, el tipo 'welcome' y 'lostPassword'.

```

public newMail = (type: 'welcome' | 'lostPassword', to: IUserModel | IValidModel, data?: {
  url?: string,
  password?: string
}) => {
  const message: IEmail = {
    from: {
      email: env.sendgrid.accounts,
      name: env.app.name,
    },
    to: {
      email: to.email,
      name: to.username,
    },
    substitutions: {
      fromName: env.app.name,
      toName: to.username,
    },
  };
  if (type === 'lostPassword') {
    message.templateId = this.type.lostPassword;
    message.substitutions.url = data.url;
  }
  if (type === 'welcome') {
    message.templateId = this.type.welcome;
    message.substitutions.url = data.url;
    message.substitutions.password = data.password;
  }
  return this.sendMail(message);
}

```

31 – Función para el envío de emails

El primero se utiliza para el proceso de recuperación de contraseña y el segundo, como se comenta en el apartado anterior, para el registro del usuario.

Las substitutions son los elementos que serán reemplazados en el template del email y el templateId contiene un identificador del layout que se referencia.

8.3. Login

El proceso de Login hace uso del mismo subsistema de autenticación que el de registro, el usuario.

En la pantalla de inicio pulsa sobre el botón de Login y se abre una ventana modal para introducir los datos, los cuales se validan en el cliente y tras esto se envían al servidor donde son saneados y comprobados de nuevo.

En caso de encontrar un usuario con nombre y contraseña correctos se responde al cliente con un mensaje de éxito y se le redirige a la página del hilo de publicaciones.

Los servicios que requieren del usuario para ser utilizados se encuentran suscritos a un observable RXJS del servicio de usuario, el cual a su vez está suscrito un observable del servicio de autenticación. Este observable se actualiza cuando un token es guardado en almacenamiento local o de sesión, obteniendo de este los datos esenciales guardados y sincronizándolo, haciendo la aplicación reactiva a este cambio.



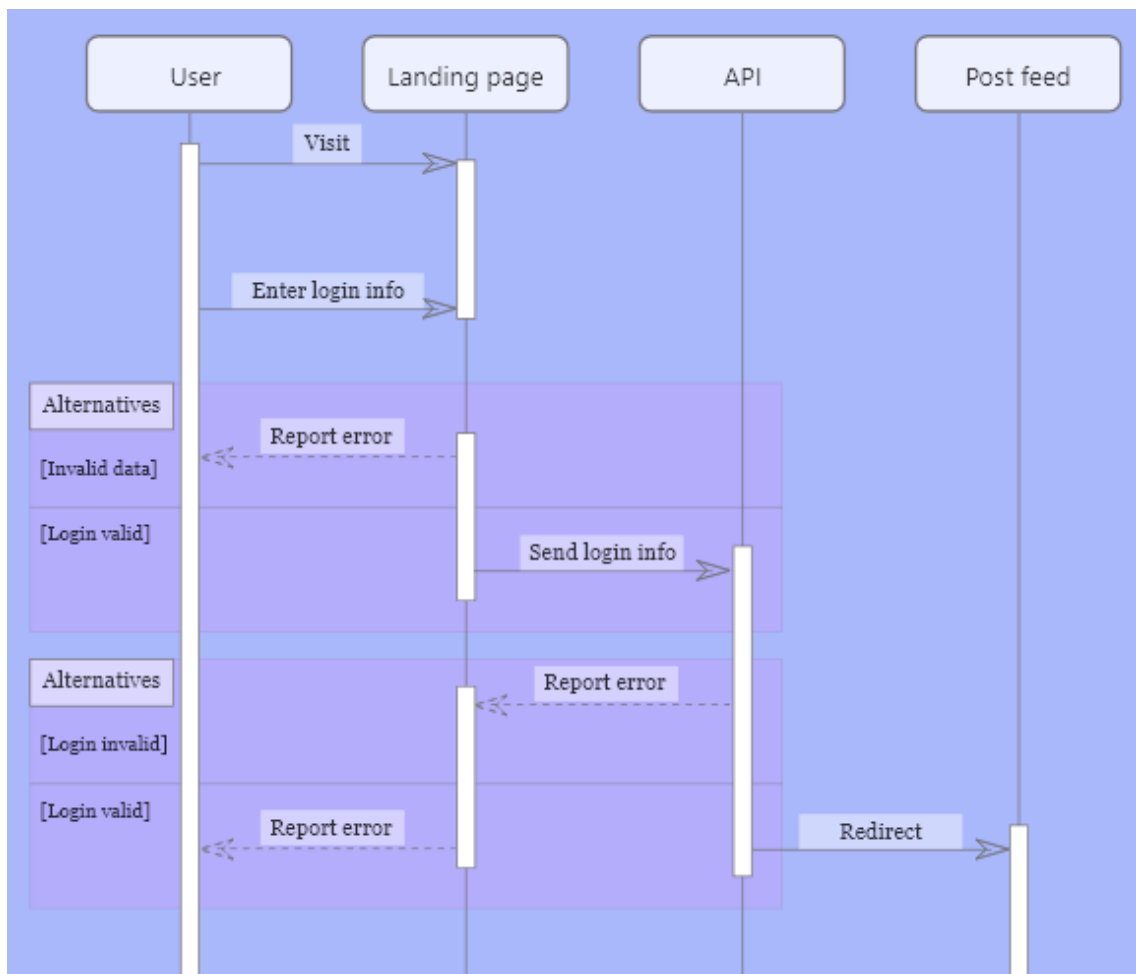
El observable utilizado es un *BehaviorSubject*, que, a diferencia de los observables normales este debe ser inicializado (a null, por ejemplo) y en el momento de suscribirse el subscriptor recibe el valor almacenado.

```

this.authService.authToken.subscribe(tokenVal => {
  if (tokenVal) {
    const token = this.authService.decodeToken(tokenVal);
    if (token) {
      const find: IFind = { resource: `_id-${token.sub}` };
      this.getUser(find).subscribe(val => {
        if (val.length) {
          this.user.next(val[0]);
        } else { this.user.next(null); }
      });
    } else { this.user.next(null); }
  } else { this.user.next(null); }
});

```

32 - Suscripción del servicio usuario al servicio de autenticación



33 - Esquema del flujo de datos del mecanismo de inicio de sesión.

8.4. Gestión de errores

La gestión de fallos por parte del servidor se hace a través de un *middleware* especial proporcionado por *express* que recibe todas aquellas peticiones que dispongan de un cuarto argumento extra (el cual añadiremos en los middlewares origen del fallo).

El argumento este contendrá información útil del tipo de fallo y se mandará al cliente con un código de error.

```
export const ErrorHandlerMiddleware = (
  error: HttpError, req: express.Request, res: express.Response, next: express.NextFunction
) => {
  const log = new Logger(__dirname);
  res.status(error.statusCode || env.api.error);
  res.json({
    name: error.name,
    message: error.message,
  });
  log.error(error.message);
};
```

34 - Middleware de error personalizado

Por su parte, en el cliente, en caso de recibir respuestas de error no gestionadas, es redirigido a un módulo de error particular que intenta de nuevo la petición. Posteriormente, en caso de recibir de nuevo el código de error, se carga un servicio que imprime en consola la información asociada. Por otra parte, si el fallo no proviene de una respuesta http se carga una página que muestra información más específica.

Este módulo de Angular que se carga hace uso de un elemento interceptor combinado con un servicio mediante el que se decide si cargar el componente o simplemente realizar el log.

```
@Injectable()
export class ErrorsHandler implements ErrorHandler {
  constructor(private injector: Injector) { }

  handleError = (error: Error | HttpErrorResponse) => {
    const router = this.injector.get(Router);
    let errorString = '';

    if (error instanceof HttpErrorResponse) {
      if (!navigator.onLine) {
        errorString = `No internet connection`;
      }
      if (error.error.message) {
        errorString = `${error.error.message}`;
      }
      console.error(errorString);
    } else {
      console.error(error);
      router.navigate(['/error'], { queryParams: { error } });
    }
  }
}
```

35 - Servicio manejador de errores

8.5. Saneamiento de datos

El proceso de saneamiento de datos lo realiza el servidor y hace uso de la librería *Joi* a modo de *middleware* para comprobar los datos provistos. A su vez, esta librería permite la limpieza de espacios en blanco a inicio y fin o el paso de mayúsculas a minúsculas. En caso de error, este se manda al sistema de gestión de errores.

```
const getPost = Joi.object().keys({
  page: Joi.number().min(1),
  perPage: Joi.number().min(1).max(100),
  resource: Joi.string(),
  sort: Joi.string(),
  filter: Joi.string(),
  partial: Joi.bool(),
  populate: Joi.string(),
  tags: Joi.string(),
});

export const getPostMiddleware = (
  req: express.Request, res: express.Response, next: express.NextFunction
) => {
  Joi.validate(req.query, getPost, (err, value) => {
    if (err) {
      const e: HttpError = new HttpError(401, err.message);
      next(e);
    } else {
      next();
    }
  });
};
```

36 - Middleware de saneamiento de petición para la búsqueda de publicaciones

8.6. Mensajería instantánea

El chat es el sistema de mensajería en vivo creado para el contacto directo entre usuarios de la aplicación.

La tecnología que hemos utilizado para establecer este sistema es *socket.io* que, como ya explicamos, emplea *WebSockets* cuando es posible para mantener abierto un canal de comunicación entre cliente y servidor.

Para implementar esto hemos tenido que crear un servicio Angular que se encargue de establecer la conexión con el socket del servidor node.

Este servicio se encuentra suscrito al servicio de usuario, de forma que sin realizar el *login* no se encuentra disponible. Una vez se encuentra conectado cambiamos el estado a online y se lanza una petición al servidor solicitando las 20 últimas conversaciones este haya mantenido.

El servidor responde con la lista de las 20 conversaciones, mas aquellas no hayan sido leídas.

Respondida la solicitud, de cada conversación, se pedirán al servidor los primeros mensajes de cada una, y de forma adicional, también los primeros 20 de la primera conversación.

Esto es así para mantener un rendimiento óptimo sin un exceso de datos a priori innecesarios en memoria.

```
export interface IConversation {
  id?: string;
  participants?: IUser[];
  ip?: string;
  createdAt?: Date;
  updatedAt?: Date;
  lastMessage?: IMessage;
  listMessages?: IMessage[];
}

export interface IMessage {
  id?: string;
  conversation?: string;
  body?: string[];
  author?: string | IUser;
  createdAt?: Date;
  updatedAt?: Date;
}
```

37 - Interfaces de conversación y mensaje en el frontend

Se actualizan en este momento el observable de *chatService conversationList*, que guarda un array con todas las conversaciones recibidas.

Relacionado con este servicio se encuentran dos componentes: el primero en la barra de navegación y el segundo en el *layout* general.

De esta manera, al estar suscritos ambos sistemas a los mismos observables, las actualizaciones no suponen un problema de ningún tipo puesto que se realizan de forma automática.

Este sistema es similar al empleado por las notificaciones, pues aprovechan el mismo servicio de socket io, pero a través de una llamada diferente.

8.7. Servicio de upload

El servicio de subida de imágenes consta de dos sistemas: por un lado, en el frontend se permite el recorte y centrado de la imagen mediante la librería *cropperjs*

Por otro lado, como comentamos con anterioridad, la carga de imágenes es vital dado que la aplicación se basa esencialmente en esto.

De esta forma, se ha implementado en el backend un mecanismo para generar diferentes copias de la imagen subida para cada una de las resoluciones.

Estas imágenes posteriormente se subirán a través del SDK de Amazon web services a la plataforma Amazon s3, con las ventajas que esto conlleva.

La gestión de la subida se ha llevado a cabo a través de *busboy*, librería que dota a Express de soporte para la subida de archivos y que nos permite limitar el número de subidas y su tamaño, así como el formato permitido.

El feed es adaptativo a la resolución, esto se realiza añadiendo un listener a las propiedades de tamaño del navegador. Debido a esto, en móviles se mostrará las imágenes con src con prefijo “small” y en resoluciones mayores con “big”.

Todo el sistema ha resultado complejo de implementar dado que se debía gestionar la subida múltiple y por tanto ha hecho falta iterar sobre cadenas de promesas.

Esta función se encuentra disponible por el momento únicamente para cambiar la foto de perfil y para la creación de publicaciones.

```
busboy.on('finish', () => {
  req.unpipe(busboy);
  busboy.removeAllListeners();
  const hidden = (answer.fields.hidden && answer.fields.hidden === 'true') || false;
  const userId = req.get('api_key') ? answer.fields.userId : res.locals.user.sub;
  const tags = answer.fields.tags ? answer.fields.tags.split(',') : [];
  const validator = createPostMiddleware(answer.fields.title, hidden, userId, tags);
  if (validator === true) {
    const post = new Post({ title: answer.fields.title, author: userId, attachments: [], comments: null, tags });
    const comments = new Comment({ discussionId: post._id });
    this.commentService.create(comments).then(comment => {
      post.comments = comments._id;
      if (promisesUpload.length > 0) {
        Promise.all(promisesUpload).then((values) => {
          const promisesAttachments: Array<Promise<IAttachmentModel>> = [];
          answer.files = values;
          answer.files.forEach(value => {
            let type = 'book';
            if (value.params.mimetype && value.params.mimetype.startsWith('image')) {
              type = 'photo';
            }
            const attachment = new Attachment({ userId, relatedId: post._id, urls: value.urls, type, tags });
            promisesAttachments.push(this.attachmentService.create(attachment, userId));
          });
          Promise.all(promisesAttachments).then((attachments) => {
            attachments.forEach(attachment => {
              post.attachments.push(attachment._id);
            });
            this.postService.create(post, userId, null, { hidden, tags }).then((newPost) => {
              res.status(env.api.success).json(newPost);
              next();
            }).catch(e => next(e));
          }).catch(e => next(e));
        }).catch((err) => {
          next(err);
        });
      } else {
        this.postService.create(post, userId).then((newPost) => {
          res.status(env.api.success).json(newPost);
          next();
        }).catch(e => next(e));
      }
    }).catch(e => next(e));
  } else {
    next(validator);
  }
}
```

38 - Gestión de la subida de posts

8.8. Servicio de donativos

Este servicio se divide en dos partes principalmente: por un lado, se trata la creación del cliente en Stripe y por otro se gestionan las donaciones.

En esta plataforma debemos gestionar que, para realizar un pago, el artista debe tener cuenta de Stripe. Para esto, en el frontend, dentro del propio perfil del artista se encuentra un botón para registrarse similar al utilizado en las ventanas modales del registro.

Este botón redirige a la plataforma de Stripe, donde se solicitan los datos necesarios para la creación del usuario. Una vez registrado, el cliente es devuelto a la página en la que se encontraba donde los datos recibidos de la plataforma Stripe para la recepción de pagos. Los datos de Stripe son almacenados en la base de datos propia en un campo del perfil del usuario.

```
public createCustomer(id: string, code: string, apiKey?: string): Promise<IUserModel> {
  return new Promise<IUserModel>((resolve, reject) => {
    request.post('https://connect.stripe.com/oauth/token', {
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
      },
      body: `client_secret=${env.payments.secretKey}&code=${code}&grant_type=authorization_code`,
      json: true,
    })
    .then(val => {
      if (val.error) { reject(val.error_description); }
      const updates: any = {};
      updates['profile.stripe.stripe_user_id'] = val.stripe_user_id;
      updates['profile.stripe.refresh_token'] = val.refresh_token;
      updates['profile.stripe.access_token'] = val.access_token;
      this.userService.update(id, updates, apiKey || null).then((user) => {
        resolve(user);
      }).catch((err: Error) => {
        reject(err);
      });
    })
    .catch(e => reject(e));
  });
}
```

39 - Mecanismo de creación de usuarios a través de Stripe

Cuando se pulsa el botón de realizar donación situado en el perfil del artista (característica única de estos) se abre una nueva ventana modal gestionada por la librería del cliente de Stripe, en ella se le solicitan los datos al usuario para realizar la donación de 5 euros.

```
public openHandler() {
  if (this.user && this.user.profile && this.user.profile.stripe && this.user.profile.stripe.customer_token) {
    this.createDonative(this.profileUser.profile.stripe.stripe_user_id, this.amount, this.donate);
  } else {
    this.handler.open({
      name: 'inarts',
      description: 'Support author',
      currency: 'eur',
      amount: this.amount,
    });
  }
}
```

40 - Manejador de la apertura de la ventana modal de Stripe

Esta solicitud devuelve un token, el cual es recibido y se llama a la función para crear el donativo a el artista, por supuesto solicitando la confirmación de la transacción.

```

public createDonative(account: string, amount: number, donate?: boolean, token?: string, email?: string): any {
  const alertRef = this.createAlert('Are you sure you want to make the 5€ donative?', 'options', true);
  (<AlertComponent>alertRef.instance).answer.subscribe(val => {
    const accept = val;
    if (accept !== null) {
      if (accept) {
        alertRef.destroy();
        this.loaderService.display(true);
        this.donativeService.createCharge(account, amount, donate, token, email).subscribe(val => {
          this.loaderService.display(false);
          const alertSuccess = this.createAlert('Successfully completed donation', 'confirm', true);
          setTimeout(() => {
            alertSuccess.destroy();
          }, 1500);
        });
      } else {
        alertRef.destroy();
      }
    }
  });
}

public createAlert(text: string, type: 'confirm' | 'options' | 'deny', fullScreen: boolean): ComponentRef<AlertComponent> {
  this.alertContainer.clear();
  const data = { fullScreen, text, type };
  const factory = this.resolver.resolveComponentFactory(AlertComponent);
  const componentRef = this.alertContainer.createComponent(factory);
  (<AlertComponent>componentRef.instance).data = data;
  return componentRef;
}

```

41 - Componente de alerta creado de forma dinámica y llamada al servicio que comunica con nuestra API

Todo el sistema se completa en el backend, el cual realiza la llamada pertinente a la API de Stripe para llevar a cabo el cargo y finalizar el proceso con un mensaje de confirmación al cliente de que la transacción ha sido realizada con éxito.

8.9. Métodos CRUD

Todos los modelos de la base de datos tienen sus correspondientes llamadas CRUD a través del API. Muchos de estos solo son accesibles por el rol del super administrador que a diferencia de los usuarios no dispone del mecanismo de autenticación por JWT descrito, sino que utiliza una API key para atravesar dicho mecanismo.

Esta clave solo puede ser creada manualmente y solo dispondrá de ella el administrador del sistema.

Otras tantas operaciones disponen de mecanismos para comprobar que el usuario que las ejecuta es el propietario. Por ejemplo, solo el artista que ha publicado un post puede eliminarlo, al igual que la propia cuenta, los comentarios o mensajes del chat.

Por otro lado, las dependencias se encuentran gestionadas por el controlador. Un ejemplo de este funcionamiento es el borrado de un post, el cual eliminará, de forma similar a una estrategia en cascada SQL, los attachments, comentarios y reacciones asociadas.

Al gestionarse de forma individual, no nombraremos cada uno dado que las implementaciones son similares a las descritas, cada cual mantiene sus particularidades, pero con un tratamiento lo más uniforme posible.

9. Conclusiones

En este apartado resumiremos el desarrollo junto con los problemas asociados a este y a continuación se describirá posibles soluciones a estos.

En un principio se planificó el diseño en base a las diferentes aplicaciones ya existentes y se trató de dotar de un sistema similar, añadiendo el registro, la creación de posts, el chat y la comunicación de los componentes de frontend con el api de una forma sencilla pero funcional, tratando de conseguir un sistema útil aun siendo rudimentario.

Esta primera versión accesible a través de *GitHub* también disponía de los mecanismos de *likes* y *dislikes* o los mecanismos de suscripción.

Sin embargo, a medida que el proyecto crecía, se hacía más patente la necesidad de emplear un método más estricto en la programación a la hora de tipar los datos para evitar múltiples errores.

Tomando como base el frontend que ya utilizaba *typescript*, se migró todo el *backend* a este *superset* de JavaScript.

Esta operación resultó costosa en cuestión de tiempo, pero establecimos una mayor relación con uno de los objetivos del trabajo que era mantener una estructura definida y accesible para desarrolladores (Apartado 2, página 10).

Una vez terminado lo básico, se mejoró el sistema de subida de archivos conectándolo con Amazon Web Services al tiempo que se desarrolló el sistema de creación de copias en diferentes resoluciones y se creó un mecanismo de *hooks* internos para que, al ser invocados se ejecutaran determinadas secciones de código.

Estos *hooks* se enlazaron con las operaciones CRUD y se iban a emplear para el sistema de puntuación. Comenzando el desarrollo de este mecanismo nos dimos cuenta de la dificultad que entrañaba el hecho de crear un ranking equilibrado.

Por un lado, para los artistas era mucho más sencillo conseguir puntos, dado que se valoraba más la cantidad de obra que la calidad en sí, para solucionar esto, se podría establecer que los *likes* aumentaban en mayor medida la valoración.

Sin embargo, establecer una valoración de forma heurística no pensamos que llevara a ninguna parte y debido a esto comenzamos a investigar otros algoritmos posibles para llevarlo a cabo.

Finalmente, desplazamos este elemento de la aplicación para el futuro dada la necesidad de un estudio más amplio en una dirección que discurre paralela a los objetivos del proyecto.

En resumen, podemos concluir que el sistema, aunque puede ser mejorado, se encuentra en un estado en el que la mayor parte de los objetivos se han cumplido de forma exitosa.

Se han investigado de forma amplia multitud de ámbitos previamente desconocidos

para nosotros y se ha tenido que lidiar con situaciones como la comunicación en tiempo real o el enlazado de variables en el *frontend* que han supuesto un verdadero reto.

9.1. Trabajo futuro

Existen múltiples posibilidades de ampliación para este proyecto, pero sobre todo una de las carencias principales dado que estaba entre los objetivos es una implementación del algoritmo de ranking.

Para esto es necesario el estudio a medio plazo del sistema una vez activo, estudiando los registros de este.

Otra cuestión es la migración a una base de datos relacional como MySQL u orientada a grafos como Neo4j.

El sistema NoSQL empleado en la implementación actual posee ciertas desventajas a la hora de interactuar con la base de datos dado que no dispone de operaciones JOIN de forma nativa ni restricciones de integridad como tal.

Esta clase de operaciones en una aplicación dedicada especialmente a las relaciones entre objetos como es la tratada en esta memoria resultan muy útiles, y aunque sea posible emularlas sin excesiva pérdida de rendimiento, se desplaza la responsabilidad sobre la consistencia de las operaciones al programador.

10. Bibliografía

- [1] ANGULAR. *Angular.io*. <<https://angular.io/docs>> [Consulta: 29 de noviembre de 2017]
- [2] NODE. *Node.js*. <<https://nodejs.org/dist/latest-v6.x/docs/api/>> [Consulta: 20 de noviembre de 2017]
- [3] MONGODB. *Mongodb*. <<https://docs.mongodb.com/>> [Consulta: 22 de noviembre de 2017]
- [4] MONGOOSE. *Mongoose*. <<https://mongoosejs.com/docs/>> [Consulta: 02 de diciembre de 2017]
- [5] SOCKET.IO. *Socket.io*. <<https://socket.io/docs/>> [Consulta: 20 de enero de 2018]
- [6] HUSSAIN, A. (2017). *Angular 2 From Theory To Practice*. Recuperado de <https://codecraft.tv/courses/angular/>
- [7] BROWN, E. (2014). *Web Development With Node and Express*. EE. UU.: Sebastopol
- [8] NGINX. *Nginx*. <<https://nginx.org/en/docs/>> [Consulta: 12 de julio de 2018]
- [9] CHACON, S. Y STRAUB, B. (2014) *Pro-Git*. Recuperado de <https://github.com/progit/progit2/releases/download/2.1.84/progit.pdf>
- [10] DOCKER. *Docker*. <<https://docs.docker.com>> [Consulta: 1 de agosto de 2018]
- [11] STRIPE. *Stripe*. <<https://stripe.com/docs/api>> [Consulta: 25 de marzo de 2018]
- [12] AWS. *Amazon web services*. <<https://aws.amazon.com/es/documentation>> [Consulta: 4 de mayo de 2018]
- [13] SENDGRID. *Sendgrid*. <<https://sendgrid.com/docs/api-reference/>> [Consulta: 2 de noviembre de 2017]
- [14] TYPESCRIPT. *Typescript*. <<https://www.typescriptlang.org/docs>> [Consulta: 29 de noviembre de 2017]
- [15] SASS. *Sass*. <<https://sass-lang.com/guide>> [Consulta: 29 de noviembre de 2017]



- [16] SNOOK, J. (2012) *Scalable and Modular Architecture for CSS*. Canadá: Ottawa
- [17] MULTER. *Multer*. <<https://github.com/expressjs/multer>> [Consulta: 25 de febrero de 2018]
- [18] EXPRESS. *Express*. <<http://expressjs.com/es/4x/api.html>> [Consulta: 21 de noviembre de 2017]
- [19] EXPRESS. *Sharp*. <<http://sharp.pixelplumbing.com/en/stable>> [Consulta: 5 de marzo de 2018]
- [20] MICROSOFT. *Visual Studio Code*. <<https://code.visualstudio.com/>> [Consulta: 20 de octubre de 2017]
- [21] CERVANTES, H. (2015, 1 de febrero). *Arquitectura de Software*. *SG Software Guru*. Recuperado de <<https://sg.com.mx/revista/27/arquitectura-software>> [Consulta: 1 de octubre de 2017]
- [22] POSTMAN. *Postman*. <<https://www.getpostman.com/docs/v6/>> [Consulta: 22 de enero de 2018]
- [23] OUTFUNNEL. *Nodemailer*. <<https://nodemailer.com>> [Consulta: 1 de noviembre de 2017]
- [24] PUGJS. *Pug*. <<https://pugjs.org>> [Consulta: 1 de octubre de 2017]
- [25] TSLINT. *Tslint*. <<https://palantir.github.io/tslint/>> [Consulta: 16 de mayo de 2018]
- [26] HACKOLADE. *Hackolade*. <<https://hackolade.com/help/>> [Consulta: 29 de agosto de 2018]
- [27] GOOGLE. *Material design*. <<https://material.io/design>> [Consulta: 4 de octubre de 2017]
- [28] ADOBE. *Experience design*. <<https://helpx.adobe.com/es/support/xd.html>> [Consulta: 10 de mayo de 2018]
- [29] JSONWEBTOKEN. *JsonWebToken*. <<https://www.npmjs.com/package/jsonwebtoken>> [Consulta: 29 de septiembre de 2017]
- [30] JSONWEBTOKEN. *Rfc7519*. <<https://tools.ietf.org/html/rfc7519>> [Consulta: 29 de septiembre de 2017]
- [31] FERNANDEZ, P. (2009) *Pablofb*. Recuperado de <https://www.pablofb.com/2009/03/clasificacion-de-redes-sociales>