

# 1 Introducción

Son varios los caminos que han llevado al desarrollo de este proyecto. En este capítulo tendré la oportunidad de poder hablar de cada uno de ellos. Desde empezar a estudiar ingeniería informática hasta la mentalidad que he ido adquiriendo a lo largo de estos años gracias a ella.

## 1.1 Motivaciones

La motivación principal ha sido la ingeniería informática y es que sin ella no sería quién soy ahora. Para bien y para mal, ha ido siendo una evolución de pequeños pasos empezando en primero de carrera hasta llegar a saber una pizca de todo este mundo de tecnología ya acabando cuarto. Todo empezó desde pequeño, desde que vi esos monitores gigantes, los disquetes y el Age of Empires 1 a la edad de cuatro años.

Mi madre tuvo un novio, Martín se llamaba, que nos trajo el mundo de la tecnología a la casa. Ella también era otra fan tecnológica, pues al haberme tenido con tan solo 17 años siempre tuvo esa mentalidad más abierta con todo el tema de nuevas herramientas. Creo que esto me acabó ayudando bastante.

Luego de esta pequeña época vino una un poco más oscura que consistía en una vida de cero tecnología. Influyó bastante el aspecto de vivir en Argentina y que el precio de los aranceles fuera bastante alto influyendo en los productos tecnológicos en su mayoría. Pasar al lado de los escaparates y ver esas pantallas brillantes con juegos, películas, canciones... era un martirio para mí.

Otro aspecto muy importante y el cual también he de agradecer fue la concesión que hizo la Junta de Andalucía allá por 2009 de aquellos pequeños ordenadores verdes. Esos sí que fueron gasolina para mis aspiraciones y para terminar hoy aquí, dentro del grado de Ingeniería Informática de la Universidad de Almería.

Mi segunda motivación principal para el desarrollo de este proyecto fue la beca extracurricular que publicó el Departamento de Informática de la Universidad de Almería, cuyo director en el momento de la concesión y en la fecha actual de redacción de este documento es Juan Francisco Sanjuan Estrada, el tutor de este proyecto.

Gracias a ella me sentí motivado para poder afianzar la actividad desempeñada en aquel pequeño trayecto de tiempo, unos seis meses, a la redacción de este trabajo.

El trabajo no se basa en lo que exactamente realicé en aquella beca, sino en una refactorización/transformación de aquel proyecto para poder convertirla en algo “fresco” podríamos llamarlo.

Y aquí pasamos al tercer y último punto de este apartado. La transformación del proyecto, el aplicar el concepto de ingeniería para poder transformar algo que era un software cerrado, con poca modularidad, difícil de entender y de ampliar en algo que merezca la pena tener. Un software que combine tecnologías y metodologías de hoy en día. Un ejemplo a seguir.

## 1.2 Objetivos

El objetivo principal de este proyecto es la creación de un sistema de gestión de inventario y creación de préstamos para el Departamento de Informática de la Universidad de Almería.

Este sitio web tiene que cumplir con unos requisitos principales, que son:

- Llevar un registro del inventario del Departamento de Informática ubicado en el edificio Científico Técnico III
- Que el estudiantado y el personal docente e investigador realicen solicitudes de préstamos para los distintos elementos ofertados dentro de la página
- Que los técnicos de servicio puedan gestionar estas solicitudes más el seguimiento del inventario dentro del edificio

Por estos puntos entendemos que la motivación principal de la herramienta es la de gestionar y organizar préstamos.

Luego de los requisitos principales que había que cumplir se nos presentaban los secundarios.

- La página web tenía que disponer de un modo adaptativo para las versiones móviles
- Había que incorporar la posibilidad de realizar un importado de datos gracias al procesamiento de una tabla de datos que es lo que antiguamente utilizaban los técnicos del departamento
- Había una serie de datos que tenían que almacenar los objetos añadidos a este sistema los cuales podían ser de tres tipos: inventarios, fungibles o kits
- Se permitiría ver un seguimiento de los préstamos realizados sobre los objetos.

En base a estos objetivos principales y secundarios realizaremos la construcción de la aplicación.

### 1.3 Planificación

La planificación del proyecto se dividirá en cinco grandes grupos: reuniones iniciales con los clientes, planificación y elaboración del desarrollo del proyecto, preparación del entorno de trabajo, construcción de la aplicación, testeo y comprobación de la aplicación y comprobación de errores.

#### 1.3.1 Reuniones iniciales con los clientes

A pesar de basarse en la reconstrucción de una aplicación las reuniones con los clientes son iniciales. Estas se hicieron en un principio del desarrollo y se mantuvieron unas pocas más a lo largo del mismo. Terminando añadiendo nuevas funcionalidades o descripciones de los productos.

#### 1.3.2 Planificación y elaboración del desarrollo del proyecto

En el desarrollo del proyecto no solo se considera la planificación de las diferentes etapas para poder ver y realizar un seguimiento del proyecto sino que también se considera la redacción y especificación de las diferentes actividades realizadas en cada una de estas. Este apartado de la planificación se desarrolla junto a todo el resto de apartados.

#### 1.3.3 Preparación del entorno de trabajo

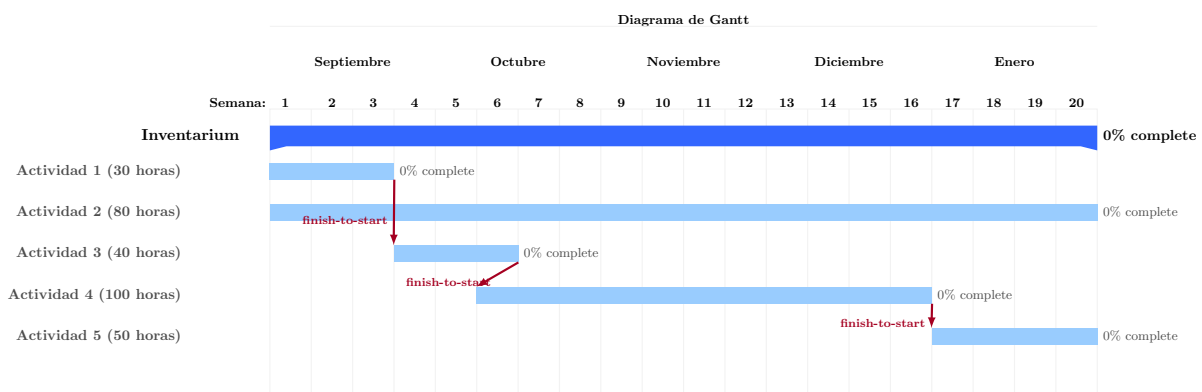
Este apartado me hace especial ilusión ya que considero que gran parte de la actividad que realizas depende del entorno donde te manejas. Los mayores casos de éxito tanto en producción como en mejoras de producto de una empresa se debe a la mejora de su entorno de producción. Gracias a algunas asignaturas del grado, a la investigación, a poder haber estado alrededor de un año en el entorno laboral y a la magnífica herramienta que es internet y todo lo que nos brinda he podido perfeccionar esta creación de entornos de trabajo y cada día siento que puedo ir mejorando un poquito más en cada aspecto que rodea a esta intencionalidad.

### 1.3.4 Construcción de la aplicación

Uno de los apartados más difíciles en cierto momento, por lo que conllevaba la creación de la lógica de la aplicación, las interacciones que realiza la interfaz con el usuario y las distintas funcionalidades que tiene que presentar la misma. Esta sección unos dos años atrás pudo haber sido de las que más tiempo podrían llevar pero gracias a los distintos frameworks que tenemos hoy en día para poder reutilizar componentes software y ahorrar pasos intermedios se ha ido volviendo más pequeña, que no quiere decir menos importante. Gracias a Angular 12 y a algunas tecnologías más que utilizaremos veremos como la construcción no es tan complicada como en un principio parecía.

### 1.3.5 Testeo y comprobación de la aplicación y comprobación de errores

¿Qué es de una aplicación bien planificada, con una interfaz bien elaborada y unos requisitos satisfechos que presente errores? No es nada, y es por ello que un testeo intensivo en el momento que la aplicación finalmente haya sido publicada hará que mejore en esos apartados que anteriormente podría haber presentado fallos.



- **Actividad 1** (30 horas) Reuniones iniciales con los clientes
  - Reunión inicial con el director del proyecto
  - Reunión con los técnicos para investigar el dominio de la aplicación
  - Presentar primer diseño de la aplicación junto a una interpretación del dominio
- **Actividad 2** (80 horas) Planificación y elaboración del desarrollo del proyecto
- **Actividad 3** (40 horas) Preparación del entorno de trabajo
  - Creación del entorno en la nube Google Cloud
  - Creación del repositorio en GitHub
  - Creación del entorno virtual en Visual Studio Code
  - Contenerización de los distintos sistemas que se utilizarán en el desarrollo de la aplicación
- **Actividad 4** (100 horas) Construcción de la aplicación
  - Elaboración y creación de la base de datos
  - Desarrollo de la API
  - Desarrollo del sitio web
- **Actividad 5** (50 horas) Testeo, comprobación de la aplicación y comprobación de errores

## **1.4 Estructuración del documento**

La estructura de este documento de trabajo fin de grado es la siguiente:

### **1.4.1 Herramientas utilizadas**

Donde haremos un pequeño resumen de todas las herramientas tanto físicas como tecnológicas utilizadas para la construcción de este software.

### **1.4.2 Desarrollo de las fases del proyecto**

Donde se describirán y detallarán todas las fases de desarrollo del proyecto. Esta es la más extensa.

### **1.4.3 Conclusiones y posibles mejoras**

Evaluaremos las distintas fases de desarrollo, su desempeño en ellas y posibles mejoras que se hubieran podido realizar.

### **1.4.4 Bibliografía**

Hablaremos sobre la bibliografía en la que se han fundamentado cada uno de los pasos de elaboración de este proyecto.

## 2 Herramientas utilizadas

En este capítulo hablaré sobre las herramientas utilizadas durante el desarrollo del proyecto. Me parece bastante interesante debido a la inclusión de nuevas herramientas que irán en conjunción a nuevas metodologías de trabajo.

Al igual que las tecnologías van avanzando las empresas también lo hacen, y la amplia gama de tecnologías gratuitas que están a nuestro alcance conlleva que el conocimiento de las mismas sea de vital importancia en el momento de construcción de una aplicación.

### 2.1 Hardware

Dentro del apartado de hardware disponemos de dos ordenadores. Su procesador no conlleva relevancia en el desarrollo de la aplicación debido a la utilización de servicios en la nube.

#### 2.1.1 Torre de PC

La cual contiene como procesador un Xeon E5-2620 V3. 16GB de RAM DDR3. Una tarjeta gráfica RTX 570 de 4GB DDR5. Tiene 256GB de memoria SSD y 1TB de memoria HDD.

#### 2.1.2 Un portátil

Es un MacBook Air M1 de 2020 con 8GB de RAM y 256GB de almacenamiento.

### 2.2 Software

#### 2.2.1 Entorno de desarrollo

Nuestro entorno de desarrollo y desde donde haremos casi absolutamente todo será desde Visual Studio Code. Este es un editor de código desarrollado por Microsoft que soporta varias distribuciones de sistemas operativos, entre ellas: Windows, Mac Os y Ubuntu.

Una de las características de esta herramienta que la hacen la predilecta de varios desarrolladores es el gran soporte que tiene por parte de la comunidad. Tiene un mercado de plugins bastante grande que apoya la creación continua de código para todos los desarrolladores.

Dispone de integraciones con Git, resaltado en errores de sintaxis, finalización de código y hasta conexión remota a otros entornos de trabajo mediante SSH.

Otra enorme ventaja que presenta es el consumo de memoria que tiene, bastante pequeño. Es un programa para ordenadores de todos los tamaños y precios, un software gratuito y una herramienta increíblemente potente al alcance de todos.

#### 2.2.2 Redacción del documento

Estas líneas están siendo escritas ahora mismo desde LaTeX. LaTeX es un sistema de composición de textos que está formado mayoritariamente por órdenes construidas a partir de comandos TeX. En un principio no estaba seguro de qué herramienta utilizar, ya que la posición de varios profesores respecto a esta herramienta era bastante férrea pero Word siempre había ido agarrado a mi mano desde comienzos del instituto.

Luego de pasar de Word a LaTeX y de LaTeX a Word bastantes veces no fue hasta que mi

profesora Rosa, en una de mis visitas matinales a su despacho me dijo: Yo hice mi TFG en LaTeX.

No me lo podía creer y al comprobar la fecha de publicación de este programa de procesamiento de textos me sorprendí al ver que su lanzamiento oficial fue en 1980. “Si el programa ha durado tanto es que algo de importante tendrá” pensé. Y aquí me hallo redactando este documento con un programa que facilita el control de versiones de Git de una manera asombrosa. Facilita también los procesos de documentación y disposición de las diferentes subsecciones. Y, lo que más me gusta sin lugar a dudas, que puedo realizar una separación de cada capítulo por documentos separados y es que a mí, el tener las cosas descompuestas, me puede.

### 2.2.3 Diseño y creación de la base de datos

La base de datos ha sido diseñada con MySQL Workbench. Esta es una herramienta visual de diseño de bases de datos, capaz de administrar, diseñar, gestionar y mantener bases de datos. Su primera versión fue publicada en 2005.

En un principio la base de datos fue exportada para que el deploy también se realizará en un servidor MySQL pero el versionado de estos scripts para la creación de las bases de datos me llevan dando problemas tres años. Grata fue mi sorpresa al descubrir MariaDB y que ofrece una compatibilidad perfecta con MySQL, además, es software libre.

MariaDB es un sistema de gestión de bases de datos derivado de MySQL con licencia GPL. Fue escrito por el mismo creador que MySQL, y esto ¿por qué? Porque vendió su producto (MySQL) a Oracle, dejando este de ser software libre.

Para la gestión de la base de datos he utilizado PHPMyAdmin. Un gestor de bases de datos que se utiliza desde páginas web. Facilita cualquier tipo de inspección que un técnico tenga que realizar por no poder hacerlo desde la aplicación web de inventarium.

### 2.2.4 Diseño del sitio web

Para realizar el diseño de la aplicación utilicé Adobe XD. Adobe XD es un editor de gráficos vectoriales desarrollado y publicado por Adobe Inc para diseñar un prototipo de la experiencia del usuario para páginas web y aplicaciones móviles.

Adobe XD apoya a los diseño vectoriales y a los sitios web wireframe creando prototipos simples e interactivos con un solo click.

### 2.2.5 Diseño de diagramas

El diseño del funcionamiento de la aplicación y el análisis de requisitos fue hecho mediante Visual Paradigm Professional. Visual Paradigm es una herramienta CASE, Ingeniería de Software Asistida por Computación, que nos brinda un gran abanico de herramientas para la ayuda del diseño, creación, planificación, análisis, documentación y un largo etcétera más de proyectos informáticos.

Esta herramienta ha sido concebida para soportar el ciclo de vida completo del proceso de desarrollo de un software a través de la representación de todo tipo de diagramas.

### 2.2.6 Diseño y modificación de imágenes

Para el diseño y modificación de imágenes utilizaremos Adobe Photoshop 2021. Adobe Photoshop es un editor de fotografías desarrollado por Adobe Systems Incorporated. Usado principalmente para el retoque de fotografías y gráficos.

### 2.2.7 Contenerización del sistema

¿Qué significa contenerización del sistema? Contenerización es la práctica de transportar mercancías en contenedores con forma y tamaño uniformes. Y con la contenerización del sistema se pretende hacer lo mismo pero para sistemas operativos informáticos.

Esto lo podemos desarrollar gracias a Docker. Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

Para explicarlo de una forma que se entienda gracias a Docker podemos desplegar y borrar máquinas de forma fácil y eficiente. Modularizando su usabilidad y evitando que si ocurre algún error en una de ellas no se rompan las demás.

#### Docker Compose

Docker Compose es una herramienta de Docker que nos sirve para poder levantar varios contenedores a la vez y de forma simultánea. Gracias a esta utilidad podemos desplegar entornos web en tiempos cortos. Además nos brinda utilidades como si una de las máquinas se apaga Docker Compose la vuelve a encender.

### 2.2.8 NodeJS

NodeJS es un entorno de tiempo de ejecución de JavaScript. Este entorno de tiempo de ejecución en tiempo real incluye todo lo que se necesita para ejecutar un programa escrito en JavaScript. Gracias a él podremos utilizar varias herramientas que explicaré a continuación.

### 2.2.9 Levantamiento la Interfaz de Programación de Aplicaciones(API)

El servicio Web se levanta desde Express. Express es framework back-end para NodeJS que está diseñado para levantar sitios webs y APIs. Es la herramienta predilecta para levantar servicios web dentro del entorno de Node.

### 2.2.10 Levantamiento del sitio web

Nginx es un servidor web/proxy de código abierto. Su primer lanzamiento fue en 2004. Presenta ventajas en el momento de recibir un número elevado de solicitudes concurrentes y es utilizado por compañías de alto perfil tecnológico como Google, Facebook o Twitter.

### 2.2.11 Desarrollo del sitio web

El desarrollo del sitio web se realizará con Angular 12. Angular es sin duda el punto más importante de esta subsección.

Angular es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles. Angular se basa en clases tipo Componentes, cuyas propiedades son las usadas para hacer el binding de los datos.

Angular es la evolución de AngularJS (la versión de Angular que usaba JavaScript) aunque incompatible con su predecesor.

Este framework para mí ha sido como el descubrimiento de América. Y es que en la anterior versión de la aplicación había utilizado únicamente PHP, Javascript y CSS. Es más, había conseguido desarrollar un modelo basado en jQuery para que el sitio web se ubicara en una única página y fuera refrescando los datos de forma interactiva y fluida.

Pero Angular fue un cambio total de mentalidad, la gestión de componentes es lo que le da la vida en su totalidad y es que, ¿qué sería un desarrollo software sin que presente una modularidad en sus componentes? Angular te lo ofrece, y te da más. Podemos definir todo el dominio de la aplicación dentro de ella y crear los servicios para que estén listos para usarse para contactar con la API. Luego podemos definir el sistema de rutas que deba tener la aplicación junto al resto de componentes que tienen que interactuar con el usuario.

No podría imaginarme un desarrollado planificado, con sus diagramas de requisitos, de base de datos, diseño de componentes y estructuración de vistas sin este framework.

## 2.3 Servicios

### 2.3.1 Sistema de control de versiones

El sistema de control de versiones se realizará gracias a GitHub. GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de aplicaciones.

El software que opera GitHub fue escrito en Ruby on Rails. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc. Anteriormente era conocida como Logical Awesome LLC. El código de los proyectos alojados en GitHub se almacena normalmente de forma pública.

El 4 de junio de 2018 Microsoft compró GitHub por 7500 millones de dólares. Al principio, el cambio de propietario generó preocupaciones y la salida de algunos proyectos de este repositorio, sin embargo no fueron representativos. GitHub continúa siendo la plataforma más importante de colaboración para proyectos Open Source.

### 2.3.2 Gestión y creación de máquinas virtuales

Este apartado no creí que lo llegaría a utilizar. En la Universidad de Almería tenemos OpenStack un servicio que nos ofrece el poder crear y destruir máquinas virtuales además de poder ubicarlas en una infraestructura de red. El problema que tuve con OpenStack fue la propia configuración de entorno de red de la universidad. Y es que había unos determinados puertos que por mucho que los abriese en la máquina virtual el cortafuegos de la Universidad me impedía acceder a ellos. Como veremos más adelante a lo largo de estas páginas esto no tuvo por qué haber sido un problema pero lo descubrí más tarde.

Debido a los problemas anteriormente mencionados con OpenStack me dispuse a buscar las soluciones que me brindaban las distintas “empresas top” del sector.

Las iré evaluando por precio, rendimiento e interfaz. Las he probado todas.

### Amazon Web Services

En precio se encuentra en un rango medio. Creo que de las tres es la que mejor se enfocaba en lo que buscaba. Disponía de buena documentación y la interfaz era compleja pero fácil de usar. La conectividad y el acceso a las máquinas era rápido pero los planes y servidores eran algo limitados.

### Microsoft Azure

El precio es alto, bastante a mi parecer. El manejo y creación de base de datos se complica por el plan de precios que presentan que están almacenados en unos tipos de monederos. No creo que sean malas ideas pero sí que están mal implementadas dificultando el aprendizaje inicial. Los servidores y la configuración de los mismos son más limitados que los de Amazon Web Services.



## Google Cloud

Ofrecen el mejor precio de la competencia. El problema es que la primera vez que entras te encuentras con muchas soluciones que ofrece Google con servicios en la nube. Es una sobreinformación de cosas que realmente no necesitaba para desplegar una simple máquina virtual. Una vez te ubicas que tienes que moverte en el entorno de “Compute Engine” todo se vuelve mucho más fácil. Las máquinas virtuales son fáciles de desplegar y de personalizar y presentan una alta gama de servidores disponibles, al fin y al cabo Google está en todo el mundo.

Mi elección fue Google Cloud. Sin duda para el proyecto iba a ser la mejor opción.



## 3 Fases previas a la construcción de la aplicación

La temporalidad del proyecto no es exacta del todo ya que los puntos 3.1 y 3.2 se hicieron aquellos dos primeros meses de prácticas extracurriculares mientras que los otros son totalmente nuevos.

Para el seguimiento de las fases y tareas se han estado utilizando la herramienta Trello y las nuevas funcionalidades de GitHub que incorpora tableros parecidos a los de Trello.

### 3.1 Reuniones iniciales con los clientes

La reunión inicial fue con Juan Francisco Sanjuan Estrada donde me explicó los distintos objetivos que se tenía pensado para la aplicación y los puntos de vista de los técnicos. La organización, gestión y préstamos del inventario se realizaba mediante la utilización de hojas de cálculo.

Dentro de las hojas de cálculo se hacía un detallado exhaustivo del objeto y también se indicaba su ubicación actual dentro del Edificio Científico Técnico III.

El problema es que esto no estaba tan organizado como parecía en un principio debido a que la gestión no la realizaba únicamente un técnico sino que eran tres. Es decir, había tres filosofías distintas a la hora de ir gestionando una parte dle inventariado del Departamento de Informática. Después de estas aclaraciones Juan me comentó que no tenían un diseño en mente en el Departamento por lo que las propuestas iniciales de diseño iban a ser libres siempre y cuando se satisficieran los requisitos principales de esta como la adición y eliminación del inventario y la concesión de préstamos.

Luego de esta primera reunión acordamos en que podría visualizar los archivos Excel de los técnicos. Estos me compartieron sus ficheros y desde ahí pude empezar a realizar la definición de los diferentes campos que irían ligados a cada tabla en la base de datos. Desde esa primera definición de campos se haría una propuesta inicial.

La propuesta inicial fue esta:

La lógica de la aplicación consistiría en generar un grupoobjetos de un determinado tipo, inventario o fungible, 1 o 0. Dentro de este se contendrían objetos que irían vinculados a una ubicación. También los objetos irían vinculados a una posible configuración.

Esta configuración ¿para qué serviría? Resulta que los técnicos aparte de manejar el inventariado disponible en la universidad también manejan las claves y accesos a esos determinados objetos, como puede ser el armario donde se ubiquen, sus direcciones mac o ip, las bocas de conexión con las regletas y más.

Este objeto tendría la posibilidad de tener varios préstamos, aunque es verdad que sería uno activo por persona, siempre dejando un registro de los anteriores usos que se hayan realizado del mismo. Este préstamo tendría que ir ligado obligatoriamente a un usuario. Contiene un campo de “retiradoPor” en el caso de que un docente o investigador quiera realizar un préstamo para su alumno.

La propuesta inicial fue aceptada por el grupo de los técnicos y por el director del proyecto así que podíamos continuar hacia adelante. Más adelante se realizaron unas modificaciones en los datos que manejarían los objetos por lo que las columnas de la tabla son las definitivas.

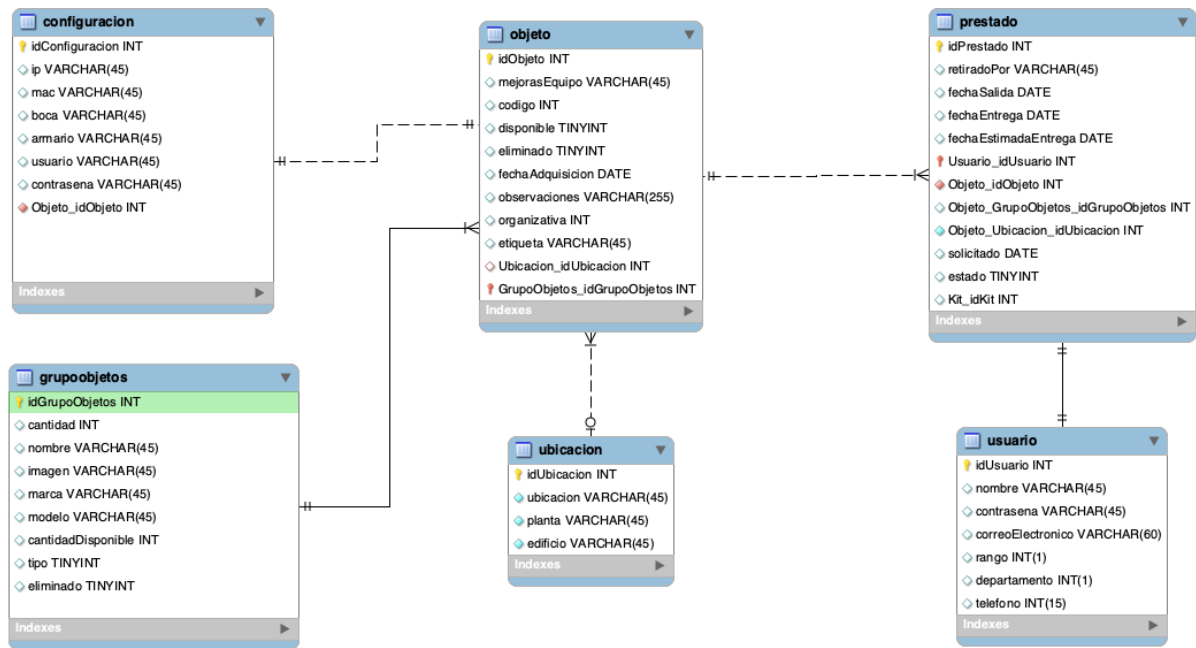


Figure 3.1: Diseño principal de la base de datos

Preparé las propuestas iniciales de diseño mediante la utilización de la herramienta de Adobe XD. El aprendizaje de la herramienta es rápido y fácil de usar, aparte de que ya la había empezado a utilizar unos años atrás. Adobe XD le dió un toque de frescura y profesionalidad a las propuestas de diseño.

Las propuestas iniciales fueron ligadas a un diseño móvil ya que desde el primer momento se buscó que fuera responsive el sitio web. Esto influyó luego a la hora de realizar el diseño ya que se basó en un funcionamiento de “cards”, es decir, componentes web con forma de carta y con fácil adaptabilidad a diseños móviles.

El esquema de navegación de la propuesta inicial nos quedaría tal como podemos ver en la Figura 3.2.

Tenemos dos roles de usuario, un personal docente o investigador y un técnico. Después del análisis hecho en la entrevista y correos mantenidos más tarde con el cliente teníamos el siguiente resultado en la Figura 3.3.

#### 3.1.1 Inicio de sesión y registro

El registro del usuario no conllevaba un registro instantáneo del mismo ya que este tiene que ser dado de alta por un técnico del sistema.

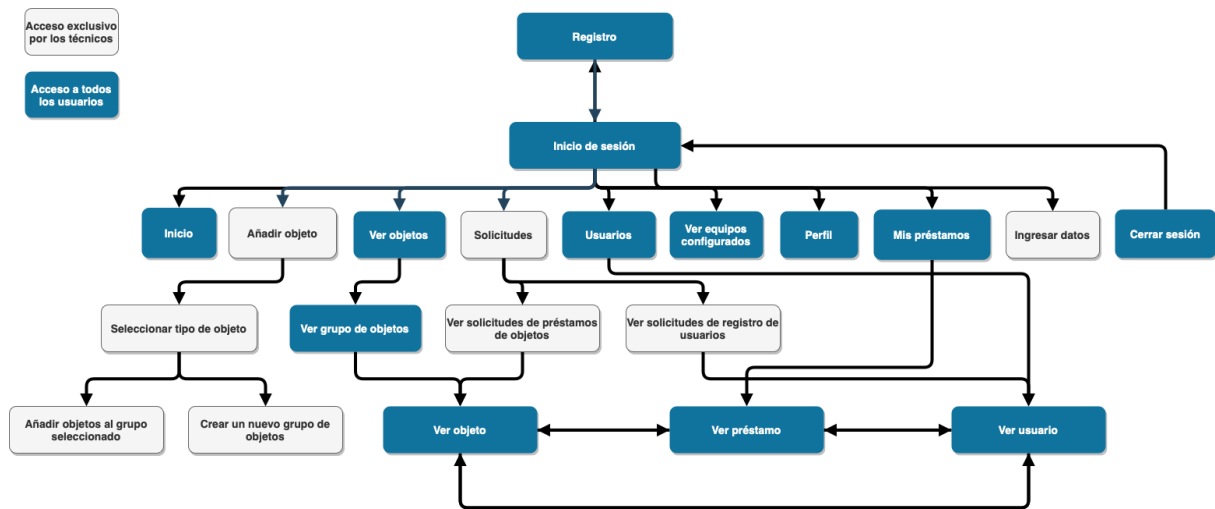


Figure 3.2: Diseño principal del sistema de navegación

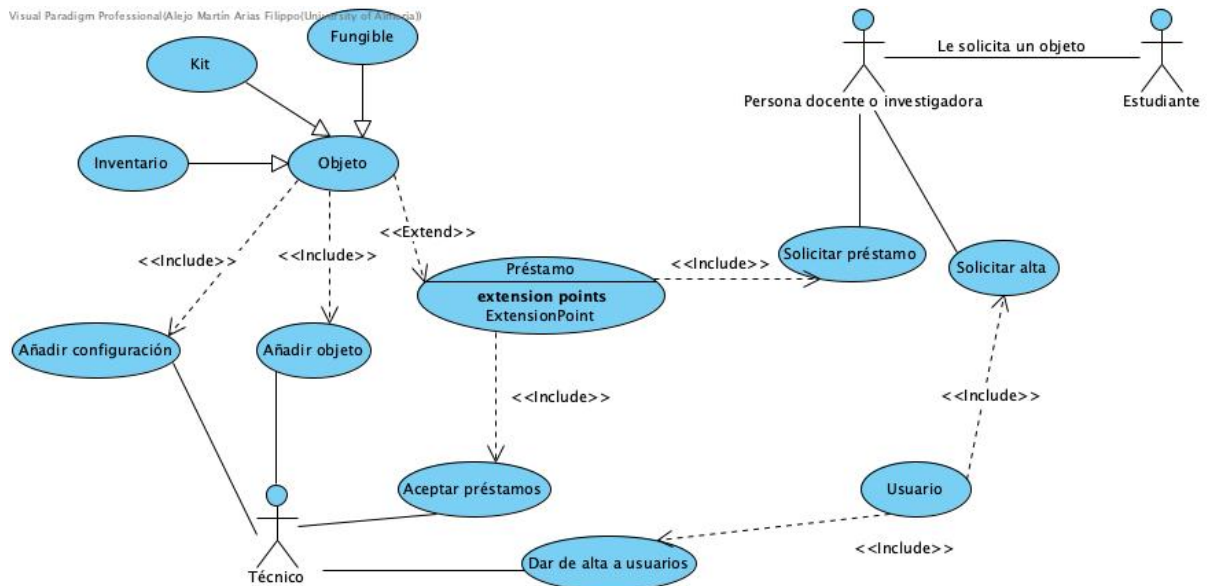


Figure 3.3: Diagrama de casos de uso del funcionamiento principal de la herramienta Inventarium

UAL-Inventarium

Correo electrónico

Contraseña

Iniciar sesión

Solicitar acceso

Solicitar acceso

Nombre completo

Correo electrónico institucional

Contraseña

Repetir contraseña

Departamento

Teléfono

Solicitar acceso

Figure 3.4: Diseño del inicio de sesión y del registro

Dentro del inicio de sesión vemos un campo interesante a considerar y es el de departamento. Una de las solicitudes que hizo el director del proyecto era poder agrupar a los usuarios dentro de departamentos. En este caso refiriéndose a dentro del personal docente e investigador que pertenecen al Departamento de Informática.

Los valores del departamento en la base de datos pueden ser los siguientes:

- 0 = Departamento de informática (valor por defecto)
- 1 = Ingeniería de sistemas y automática
- 2 = Lenguaje y sistemas informáticos
- 3 = Ciencias de la computación e inteligencia artificial
- 4 = Arquitectura y tecnología de computadores

Otro campo para comprobar es el del correo electrónica institucional teniendo que ser este con la extensión **@inlumine.ual.es** o **@ual.es**.

#### 3.1.2 Barra de navegación

La barra de navegación vertical tuvo una pequeña modificación que era incorporar una pequeña sección de inicio donde aparecieran información pertinente y relevante para el usuario. Esto será explicado más adelante en la parte de desarrollo.



Figure 3.5: Diseño de la barra lateral de búsqueda

Dentro de ella podemos ver cinco apartados que derivan en varios componentes visuales:

- Añadir objeto: un acceso rápido donde poder añadir objetos dentro de grupos de objetos. Sección de menú solo visible para los técnicos.
- Ver objetos: apartado donde se visualizan los grupos de objetos, que contienen tanto el inventariado, fungibles y kits.
- Solicitudes: un menú donde se visualizan las solicitudes tanto de alta de usuarios como de objetos que le puedan llegar a los técnicos.
- Usuarios: componente visual donde cargan los usuarios registrados que hay dentro de la aplicación.
- Ver equipos configurados: aquí cargan únicamente objetos a los que se les haya aplicado una configuración. En caso de que no, no aparecerían.

### 3.1.3 Añadir objeto

Añadir objeto

Inventario

Fungible

Nombre \*

Imagen \*

Cámara

Ubicación \*

Ubicación

Marca

Modelo

Mejoras en el equipo

Cantidad

1

+

-

Configuración

Aplicar

Añadir

Añadir objeto

Inventario

Fungible

Nombre \*

Imagen \*

Cámara

Ubicación \*

Ubicación

Marca

Modelo

Mejoras en el equipo

Cantidad

2

+

-

Código del inventario 1 \*

Código del inventario 2 \*

Configuración

Aplicar

Añadir

Figure 3.6: Diseño de la característica para poder añadir fungibles e inventarios

La única diferencia entre añadir un objeto de tipo inventario y otro fungible es que en el de inventario hay que registrar su código. Siempre que vayamos a añadir uno de estos elementos tiene que ser dentro de un **grupo de objetos** que debemos haber creado anteriormente. Por el resto son los dos idénticos y se le puede añadir el mismo tipo de información. No solamente tenemos fungibles e inventarios sino que también tenemos kits.



Un **kit** podemos considerarlo como otro nivel de agrupación. Ya que un kit contiene más objetos dentro suyo que llamaremos “objetos kit”. Para entender los niveles de agrupación aquí tenéis un esquema explicándolas.

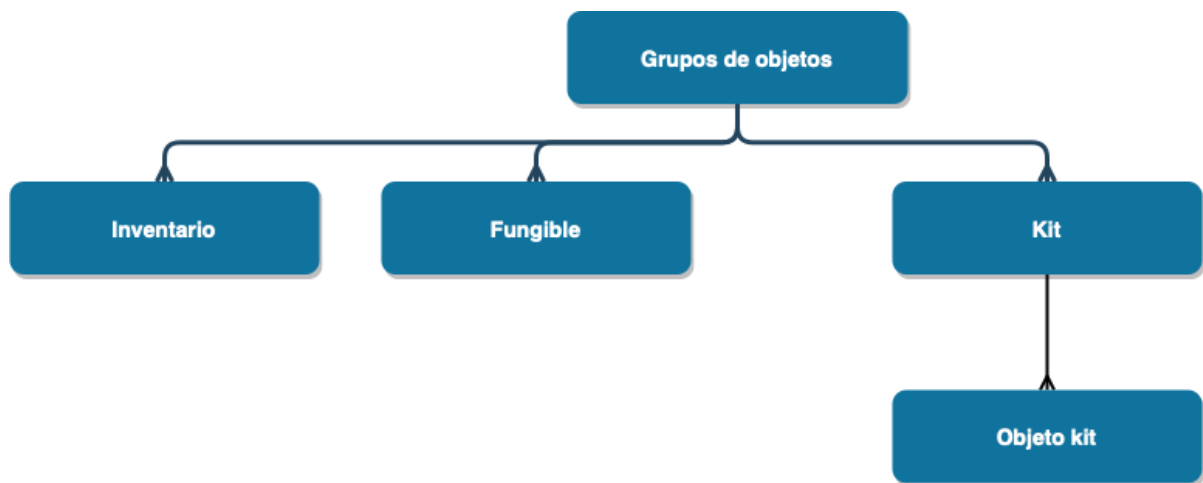


Figure 3.7: Jerarquía de elementos que derivan de grupo de objetos

### 3.1.4 Grupo de objetos y objetos

Dentro de la sección de “ver objetos” de la barra de navegación nos llevará a la visualización de los grupos de objetos de la aplicación. En las cuales habrá también objetos. Estos dos conjuntos se pueden modificar y además se pueden pedir solicitudes para el inventariado, fungibles y kits.

Nombre del objeto

Marca:

Modelo:

Configuración

Disponibles

Prestados

Solo lo verán los profesores si hay unidades disponibles.

Nombre del objeto

Marca:

Modelo:

Mejoras:

Figure 3.8: Grupo de objetos a la izquierda y objetos que contiene a la derecha

### 3.1.5 Usuarios

En este apartado podremos visualizar los usuarios que están registrados en la aplicación. Podemos acceder a cada uno de los perfiles para ver sus respectivos datos y sus solicitudes o historial de préstamos.

Usuarios

Juan Francisco Sanjuan Estrada

José Andrés Moreno Ruiz

20 usuarios máximos por página

1 2 Siguiente

Usuario

Nombre: Juan Francisco Sanjuan Estrada

Correo electrónico: jsanjuan@ual.es

Departamento: Informática

Teléfono: 950 214017

Objetos prestados ▼

Objeto
Objeto

Figure 3.9: Grupo de usuarios a la izquierda y usuario único a la derecha

#### 3.1.6 Préstamos

Los préstamos se pueden encontrar en tres diferentes estados:

- Préstamo en curso: una solicitud de préstamo aprobada por un técnico y que está en vigencia.
- Préstamos solicitado: donde aparecerá una solicitud en espera a ser aprobada por un técnico, en el caso de que el objeto solicitado esté actualmente con un préstamo en curso no se permitirá conceder el préstamo solicitado hasta que finalice el actual.
- Préstamo finalizado: un préstamo en curso que ha sido devuelto, la aprobación de devolución la tiene que conceder un técnico.
- Préstamo vencido: un préstamo en curso que ha sobrepasado su fecha máxima que se había comprometido a devolver. En caso de que quiera solicitar una ampliación tiene que hacerlo nuevamente generando una nueva petición.

Cada solicitud irá con un color representativo dependiendo de cuál sea su estado, una utilidad tanto para el usuario como para el técnico que van a realizar las acciones de ir solicitando y concediendo préstamos.

Nombre completo del destinatario\*

Cantidad

1 ▼

Se podrá seleccionar para que sea indefinida

Fecha estimada de entrega \*

Seleccionar

Cancelar Solicitar

Figure 3.10: Solicitud del préstamo de un objeto

### 3.1.7 Configuraciones

Un objeto puede ir ligado a una información extra o no. Esta información no es algo que vuelva más descriptivo el objeto sino que es una utilidad que se solicitó en el momento del desarrollo de la aplicación. Su objetivo es poder acceder a información a la que suelen recurrir los técnicos de forma bastante frecuente. Como la IP o dirección MAC de un ordenador o las credenciales para poder acceder a él. Gracias a esta distinción podemos hacer que la información de los equipos que disponen de una configuración en particular estén en una sección para ellos solos y facilitarles el acceso a los mismos a los técnicos.

Figure 3.11: Visualización de los campos de configuración a la izquierda y creación de estos a la derecha

## 3.2 Planificación y elaboración del desarrollo del proyecto

Esta sección implica dos apartados de relevante importancia que es de la forma en que la subdividiremos. Por una parte tenemos:

### 3.2.1 Redacción del proyecto

Para la redacción del proyecto como he explicado anteriormente se está utilizando la herramienta LaTeX. Esta serie de archivos viene incorporado al repositorio principal del proyecto dentro de una carpeta llamada documentación.

Nuestra carpeta de documentación a la vez está subdividida en varios apartados:

#### Build

Carpeta que viene por defecto incorporada dentro de la “compilación” del proyecto que realiza LaTeX. Dentro de ella se genera nuestro documento PDF que gracias a Visual Studio Code se va generando cada vez que se guarda un archivo del proyecto que vaya ligado o afecte al main.tex

## Bibliografía

En este apartado añadimos la bibliografía utilizada durante la realización del proyecto.

## Capítulos

Dentro de esta carpeta guardaremos cada uno de los capítulos del documento. En el caso de que un capítulo presente apartados demasiados extensos generamos una carpeta con el nombre del capítulo y metemos las secciones dentro de este. Gracias a hacer esto la modificación de cada zona del documento se hace de una forma mucho más cómoda.

## Diagramas

Aquí almacenamos los diagramas del proyecto. Estos pueden estar en archivos de imágenes aunque también pueden estar generados mediante LaTeX. Debido a la unicidad que presentan los diagramas estos no están separados en diferentes carpetas por capítulos tengamos.

## Imágenes

Como su propio nombre indica almacenamos las imágenes del proyecto. Estas generalmente están divididas por carpetas con el nombre de los capítulos. En el caso de que sea demasiado extensa también se contempla la posibilidad de realizar el almacenaje mediante secciones.

## Include

Aquí añadiremos los archivos que generen inclusiones dentro de nuestro documento. Dependiendo de para qué sean estos añadidos irán con unas determinadas agrupaciones u otras.

## main.tex

Esto no es un directorio pero es un archivo de relevancia. Su contenido es escaso debido a la generalización que presentamos en nuestra disposición de la documentación. Dentro de él nos podemos encontrar las inclusiones al principio del documento. La adición del índice, los capítulos y de la bibliografía al final de la página. Es el archivo que utiliza LaTeX para generar nuestro documento PDF.

### 3.2.2 Elaboración del desarrollo del proyecto

Esta sección consiste en la explicación y descomposición de pasos que vamos a hacer para la elaboración de nuestra aplicación. Una vez hemos hecho la lógica de la aplicación, su dominio y sus casos de uso procederemos a la maquetación de nuestra aplicación:

#### Generación de la base de datos

Generaremos nuestra base de datos, para ello utilizaremos MariaDB para el despliegue de esta.

#### Construcción de la Interfaz de Programación de Aplicaciones (API)

La creación de nuestra API hay que hacerla de forma muy cuidadosa, ya que tiene que incorporar todas nuestras reglas de negocio y cualquier paso adicional que haya que añadir en cada una de las peticiones. Estos pueden consistir en que cuando un objeto es creado hay que sumar una unidad dentro del campo de “objetos” y “objetosDisponibles” de su respectivo grupo de objetos.

## Construcción de la aplicación

Detallaremos cada uno de los pasos más adelante debido a que la construcción de esta depende en pequeña medida al framework utilizado que en este caso es Angular. En todo caso el orden de creación de ficheros sería:

1. Creación de interfaces
2. Creación de servicios
3. Creación de componentes visuales

## 3.3 Preparación del entorno de trabajo

La preparación del entorno de trabajo es una de las secciones que me hacen especial ilusión de este trabajo. Como decía Robert Owen, un reconocido empresario galés, en el siglo XVIII:

Mejorando el entorno se mejora al hombre.

En nuestro caso en vez de hombre podría ir mi nombre, ya que me ha ahorrado mucho tiempo esta preparación previa, o, el proyecto.

Esta sección irá estructurada en consonancia a la cronología de creación de cada parte del entorno. Empezamos por la semilla de todo: Github.

### 3.3.1 GitHub

Realizaremos la creación de nuestro proyecto en GitHub. Esto se hace de forma muy rápida y fácil.

1. Nos dirigimos a la dirección de `github.com`

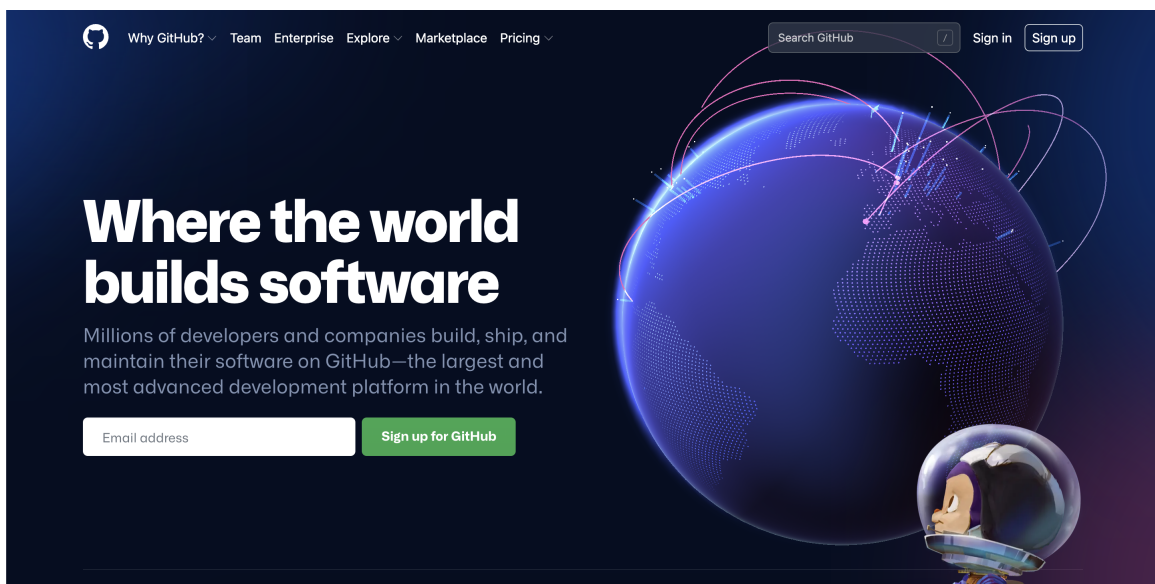
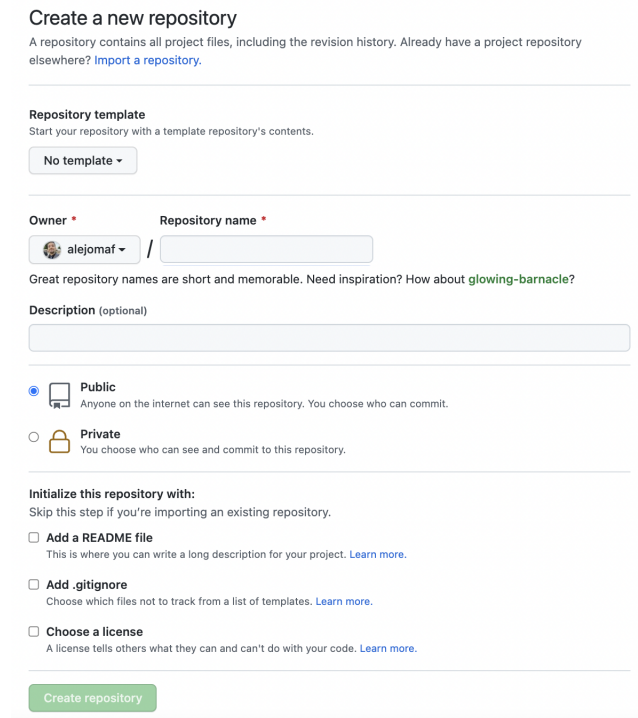


Figure 3.12: Página principal de GitHub

2. En la parte superior derecha nos aparece “Sign up” donde clicaremos y nos permitirá registrarnos dentro de la web.

3. Una vez registrados volvemos a [github.com](https://github.com) y clicamos a la izquierda de “Sign up” que pone “Sign in” donde nos registraremos.
4. Cuando hayamos iniciado sesión nos dirigiremos a la parte superior izquierda y clicaremos en “new” para crear un nuevo repositorio.
5. Rellenamos los campos que nos solicitan para crear un nuevo repositorio. Le damos el nombre de Inventarium y seleccionamos la opción para que el repositorio sea privado. También procedemos a añadirle un readme al archivo para poder describir partes del proyecto dentro de este. Por último pulsamos el botón “Create repository”.



The screenshot shows the GitHub 'Create a new repository' page. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is. Below this, there's a section for 'Repository template' with a 'No template' button. The main form has two input fields: 'Owner' (with a dropdown menu showing 'alejomaf') and 'Repository name' (with a text input field). Below these fields, there's a note about repository names being short and memorable, with a suggestion 'glowing-barnacle'. There's also a 'Description (optional)' text area. Underneath, there are two radio buttons for visibility: 'Public' (selected) and 'Private'. Below this, there's a section 'Initialize this repository with:' with three checkboxes: 'Add a README file', 'Add .gitignore', and 'Choose a license'. At the bottom, there's a green 'Create repository' button.

Figure 3.13: Creando un nuevo repositorio dentro de GitHub

Con estos sencillos pasos ya tendríamos creado nuestro repositorio.

#### 3.3.2 Visual Studio Code y Google Cloud, los mejores amigos

Hoy en día lo nuevo y a lo que nos dirigimos es la nube. Dentro de ella se pretenden que se realicen todos los procesos. La mayoría de herramientas y servicios que se nos ofrece hoy en día cumplen un modelo de caja negra. Es decir, interactuamos con él y recibimos respuestas pero no vemos qué procesos ocurren dentro de aquella cajita.

Por esta razón cada vez se empieza a disponer menos de un software como tal y se empiezan a pasar a servicios en línea. Esto no quiere decir que se dejen de usar programas o aplicaciones móviles. Pero la realidad es que sin internet; la mayoría no funcionaría.

Esto presenta desventajas siendo la principal que se depende constantemente de una conexión en línea que puede parecer que pasa en todos sitios pero en lugares remotos lejos de la ciudad como son los pueblos de montaña el internet no es el mejor compañero por decirlo de una manera. Por suerte este no es mi caso.

Las ventajas son enormes aunque solo nos centraremos en las que para mí implican mayor relevancia.

El tener ya un repositorio generado con nuestro proyecto subido dentro de él ya nos aporta

bastante autonomía ya que podemos acceder a él y descargar nuestros datos desde cualquier lugar. Pero, ¿y configurarlo?

Aquí viene una problemática que no nos resuelve un entorno de repositorios. El tener que configurarlo todo cada vez que queremos trabajar desde un sitio distinto. Esto nos lo resuelven las máquinas virtuales en la nube. El generar un directorio de trabajo donde poder conectar nuestro Visual Studio Code y olvidarnos de preocupaciones.

Lo segundo que me parece más importante es el ahorro de memoria tanto de RAM como de disco y de procesos que se origina al hacer esto. No es lo mismo tener que trabajar con un portátil conectado todo el día a un enchufe que poder ir llevándotelo contigo por el escaso consumo de su batería. Peor si es una torre, solo puedes trabajar desde un único sitio, con la problemática también de que siempre se te puede cortar la luz, y más en una zona de pueblo de sierra como es la mía, suele ocurrir una vez cada dos semanas.

Estas ventajas expuesta son las que personalmente me favorecen a mí, pero ¿y si trabajara con un equipo? El poder tener varios compañeros de un mismo equipo trabajando en el mismo proyecto, tocando los distintos componentes que se están desarrollando dentro de tu aplicación es fantástico.

### Crear una máquina virtual en Google Cloud

Los pasos para la creación de una máquina virtual son sencillos:

1. Los pasos para el registro son fáciles de realizar al ser la herramienta propiedad de Google. Omitiremos este paso.
2. Google Cloud se organiza mediante proyectos. Esto brinda facilidades a la hora de calcular el consumo que vamos a tener por el uso de la computación en la nube. Aparte de poder facilitar la búsqueda por cada proyecto que tengamos activo. Procederemos a crear un nuevo proyecto clicando arriba a la izquierda a la derecha del título de la página y posteriormente clicamos en nuevo proyecto.
3. Luego desplegaremos la barra lateral de la izquierda y nos dirigiremos al apartado de Compute Engine.
4. Clicamos en crear instancia. A la hora de rellenar los parámetros he comprobado que no hay que darle demasiada importancia a la zona o región donde se ubique nuestra máquina. Esta funciona a una velocidad bastante decente. En el apartado de configuración de máquina con 2GB de RAM y 1VCPU tendremos suficiente. El tamaño del disco eligiremos que sea de 20GB. Dentro de la sección Firewall habilitaremos el tráfico HTTP y HTTPS. Y pulsamos en crear.
5. Después de haber creado la máquina le asignaremos una IP estática pública que la usaremos para conectarnos a ella mediante SSH.
6. En el momento de haber creado la máquina esta nos ha dado una clave privada SSH para poder conectarnos a ella. La guardaremos.

### Configurar claves SSH

Teniendo ya la clave privada SSH de nuestra máquina virtual la utilizaremos para realizar la conexión a nuestra máquina. Pero para ello tenemos que configurarla. Accederemos al fichero `usuario/.ssh/config` y añadiremos tres nuevas líneas:

```
Host "Aquí añadimos la dirección IP"
HostName "Aquí ponemos el nombre del host"
User "Aquí ponemos nuestro nombre de usuario"
```

Lo último que nos queda es realizar la conexión mediante SSH a la máquina por Visual Studio Code.

#### Realizar conexión SSH mediante Visual Studio Code

Dentro del apartado de extensiones de Visual Studio Code buscaremos los siguientes plugins para instalar:

- Remote - SSH
- Remote - SSH: Editing Configuration Files

Luego de la instalación de estas dos extensiones procederemos a realizar la conexión. Abajo a la izquierda nos aparecerá la opción de poder conectarnos mediante SSH. Clicamos y luego pulsamos en “Connect to Host” añadiendo la dirección IP estática pública que nos dio Google Cloud.

Ya tenemos la masa de nuestro entorno hecha, solo hace falta darle un poco de forma y calor para poder finalizarlo.

#### 3.3.3 Instalación de las extensiones de Visual Studio Code dentro de nuestra MV

Esta es la lista de componentes con la que trabajaremos en todas las fases del proyecto. Accederemos a las extensiones dentro de Visual Studio Code y procedemos a instalarlas dentro de nuestra máquina virtual.

- Docker
- LaTeX
- LaTeX Workshop

Los complementos de Latex nos ayudarán más tarde con la redacción del Trabajo de Fin de Grado.

En nuestro Visual Studio Code también instalaremos:

- Bootstrap 4 snippets
- HTML snippets
- CSS snippets

#### 3.3.4 Configuración de nuestra máquina virtual

Desde Visual Studio Code accederemos a la terminal de nuestra máquina virtual. Pinchamos en “Ver” en la sección superior y luego en Terminal. Desde ahí controlaremos a la máquina.

#### Actualización del sistema

Actualizaremos el sistema para evitar posibles fallos en un futuro:

```
sudo apt-get update  
sudo apt-get upgrade
```

#### Instalación de Node JS

```
sudo apt install nodejs
```



### Instalación de Angular 12

```
sudo npm install npm@latest -g
sudo npm install -g @angular/cli
```

### Instalación de Docker y Docker Compose

\\Instalacion de Docker

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal"
sudo apt install docker-ce
```

\\Instalacion de Docker Compose

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.26.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

### Configuración de credenciales de Git e instalación del repositorio

//Configuración de credenciales

```
git config --global user.name "tu nombre de usuario"
git config --global user.email "tu correo electrónico"
git config --global user.password "tu contraseña"
```

//El repositorio lo ubicaremos dentro de la carpeta raíz del usuario

```
git clone https://github.com/alejomaf/Inventarium.git
```



## 4 Construcción de la aplicación

Dentro de este capítulo explicaré los distintos procesos por los que ha pasado la construcción de la aplicación. Todos tienen un punto común y es Docker Compose.

### 4.1 Introducción a Docker y Docker Compose

Al finalizar cada apartado se explicará cómo se realizó su adición dentro de nuestro entorno que despliega Docker Compose. Pero, ¿cómo funciona Docker Compose? O mejor dicho ¿para qué vamos a necesitar docker?

#### 4.1.1 ¿Para qué sirve Docker?

Gracias a Docker en vez de máquinas virtuales podemos utilizar contenedores. Los contenedores presentan diferentes ventajas:

- El entorno de pruebas donde ejecutaremos los distintos componentes de la aplicación son idénticos al de un servidor.
- Obtenemos mayor modularidad. Gracias a esto tenemos un entorno ideal donde poder trabajar con microservicios.
- Podemos ejecutar nuestra aplicación en un entorno que sabemos que en caso de fallo lo volvemos a levantar.

Docker es un avance para la informática. Nos ayuda a ahorrar recursos en nuestro ordenador y poder dedicar el tiempo invertido en la configuración de entornos de servidor a otras actividades.

#### 4.1.2 Docker Compose: el SimCity de los entornos

¿Por qué SimCity? Al igual que en el famoso juego SimCity levantábamos ciudades en cuestión de segundos gracias a Docker Compose levantamos entornos en el mismo tiempo.

Docker Compose nos ayuda a definir un entorno de contenedores con la posibilidad de poder conectarlos entre ellos. También nos ofrece la posibilidad de generar volúmenes de almacenamiento donde ir almacenando la información generada dentro de esos volúmenes. Lo hace todo menos la comida, ojalá.

Docker Compose funciona sobre un archivo llamado `docker-compose.yml` dentro del cual iremos añadiendo los componentes que querramos configurar. Todo esto lo iremos viendo a lo largo de las siguientes secciones.

## 4.2 Generación y puesta en marcha de la base de datos

Ha llegado el momento de ponernos manos a la masa y dejarnos de preparativos o presentación de herramientas. Vamos a levantar nuestra base de datos.

El diagrama final de la base de datos quedaría así: Generamos el script “.sql” para poder generar nuestra base de datos.

Esta configuración la basaremos únicamente en la sección de código que añadiremos a nuestro archivo `docker-compose.yml` para poder hacerlo.

Añadimos la versión que utilizaremos de docker-compose:

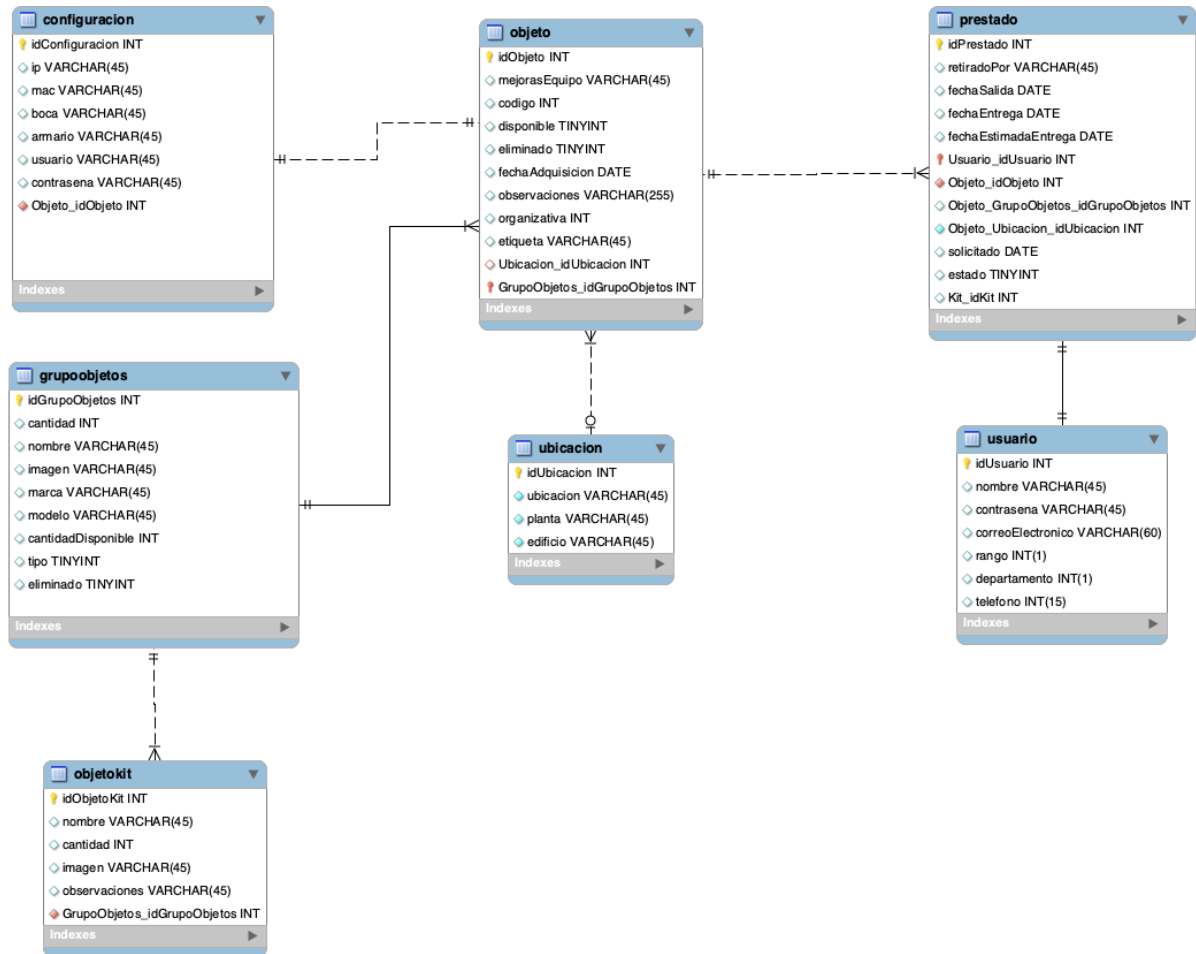


Figure 4.1: Diseño final de la base de datos

```
version: "3.9"
```

Creamos el apartado “services”:

```
services:
```

Y añadimos nuestro primer contenedor, el de la base de datos:

```
db:
  container_name: inventarium_sql
  image: mariadb:10.7.1-focal
  ports:
    - "3306:3306"
  volumes:
    - ./dump:/docker-entrypoint-initdb.d
    - persistent:/var/lib/mysql
  networks:
    - default
  environment:
    MYSQL_DATABASE: ualinventory
    MYSQL_USER: ualinventory
    MYSQL_PASSWORD: contraseñasecreta
    MYSQL_ROOT_PASSWORD: contraseñasecreta
```

- **db:** Este es el nombre que le hemos puesto como referencia para docker-compose.
- **container\_name:** Es el nombre que tendrá el contenedor que vayamos a desplegar.
- **image:** La imagen utilizada, es decir, la “máquina virtual” que vamos a levantar. En este caso es una máquina virtual de MariaDB.
- **ports:** Los puertos que utilizará la máquina, a la izquierda son los que le llegan a la máquina y a la derecha los que salen.
- **volumes:** Esta sección es muy importante ya que aquí podremos importar archivos en el momento del despliegue de nuestro entorno. En este caso estamos importando la carpeta “dump” donde dentro está el archivo de generación de la base de datos. También le enlazamos el volumen “persistent” que creamos más adelante. Esto es para que no perdamos datos cuando paremos el contenedor. Ya que los datos de la base de datos de esta se almacenará en un sitio externo.
- **networks:** Aquí podemos añadir las redes que manejará nuestro contenedor. En este caso es la default.
- **environment:** Dentro de environments podremos definir variables que se necesiten en el momento del despliegue de nuestro contenedor. En este caso definimos el nombre de la base de datos, el usuario junto a su contraseña y la contraseña del usuario root.

```
phpmyadmin:
  container_name: inventarium_php_adm
  image: phpmyadmin:5.1
  links:
    - db:db
  ports:
    - 8000:80
```

```
environment:
  MYSQL_USER: user
  MYSQL_PASSWORD: user
  MYSQL_ROOT_PASSWORD: user
networks:
  - default
```

Este despliegue es para phpmyadmin, nuestro gestor de base de datos. Como añadido al punto anterior tenemos el apartado **links** que nos ayuda a crear una vinculación de un contenedor con otro. En este caso le estamos pasando la información de nuestra base de datos para que trabaje sobre ella.

```
volumes:
  persistent: {}
```

Dentro de la sección **volumes** podemos generar volúmenes que se van a utilizar dentro de nuestros contenedores. En este caso hemos creado nuestro volumen “persistent” para poder almacenar la información que se almacene dentro de nuestra base de datos.

Para levantar nuestra base de datos ejecutaríamos dentro del directorio donde hemos creado nuestro docker-compose.yml el siguiente comando:

```
sudo docker compose up
```

Ya tendríamos nuestra base de datos en funcionamiento.

### ¿Cómo comprobamos los resultados?

Visual Studio Code realiza una **tunelización de los puertos** a partir de la conexión SSH. Es decir, podemos ir habilitando redireccionamientos de puertos para poder ir viendo los resultados en nuestra máquina. Esto resulta de gran ayuda ya que no hay que realizar una configuración de la máquina previa de apertura de puertos para poder ir trabajando sobre ellas. Ya que el funcionamiento del sistema será interno con poder disponer de los puertos HTTP Y HTTPS abiertos no necesitamos más.

## 4.3 Diseño de la arquitectura de la aplicación

Antes de empezar con el desarrollo de la API y del sitio web he de explicar la arquitectura que se ha utilizado para el desarrollo de estos elementos.

### 4.3.1 ¿Qué es una arquitectura de software?

Según el IEEE la arquitectura es la organización fundamental de un sistema compuesto por sus componentes, las relaciones que tienen unos con otros y con el entorno y los principios que guían su diseño y evolución.

Entendiendo sistema como un conjunto de componentes que se organizan para cumplir una determinada función o conjunto de funciones.

Un sistema existe para cumplir una o más misiones en su entorno.

El entorno o contexto determina la configuración y circunstancias en el desarrollo, las operaciones, la política y demás aspectos que puedan influenciar a un sistema.

La arquitectura de software es de vital importancia en el desarrollo de un sistema ya que determina su estructura conllevando un aspecto directo sobre la capacidad de este para satisfacer los requisitos y evolución de un proyecto.

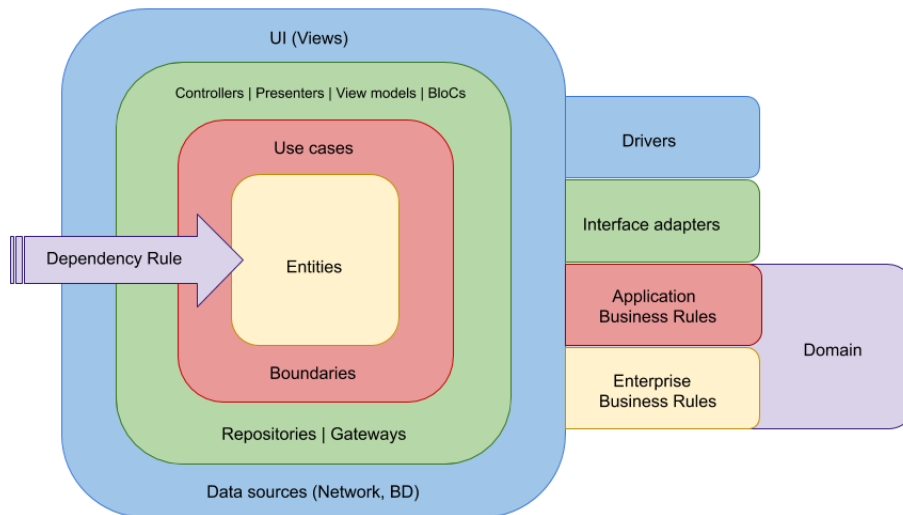


Figure 4.2: Clean Architecture

### ¿Por qué es importante definir una arquitectura software?

Definir una arquitectura de software presenta varias ventajas. Entre ellas tenemos las siguientes:

- **Independencia de los frameworks:** Una arquitectura puede ser definida en múltiples frameworks y lenguajes pues no dependen de ellos.
- **Independencia de las reglas de negocio:** Las reglas de negocio no se verán alteradas por el cambio de arquitectura software.

#### 4.3.2 Clean Architecture o Arquitectura Limpia

Es un conjunto de principios cuya finalidad principal es ocultar los detalles de implementación a la lógica de dominio de la aplicación.

Gracias a esto podemos mantener a la lógica de la aplicación aislada de forma que sea más mantenible a lo largo del tiempo.

Para explicar qué es la arquitectura limpia utilizaremos el gráfico 4.2.

Podemos ver que la estructura del diagrama está formada por capas. Las capas representan distintos conjuntos que componen el sistema.

Podemos ver una flecha en la que aparece escrito “Dependency Rule”, en español, regla de dependencia. Significa el sentido que tomarán los distintos componentes de la aplicación. En este caso de afuera hacia dentro. Los componentes más externos dependen de los más internos y las entidades o el dominio de la aplicación depende de ella misma.

En el momento de la separación de funcionalidades tenemos que tener en cuenta que nuestro sistema se divide en tres partes. La web, la API y la base de datos.

#### Entidades

Las entidades representan la pieza básica de nuestra aplicación. Estos serían las tablas del diagrama de nuestra base de datos. Cada una de las tablas conforma un tipo de datos. Pueden tener dependencias entre ellos pero nunca van a tener dependencias externas.

Por eso es lo primero que se define ya que si hay que cambiarla habría que cambiar los elementos que dependieran de ella.

### Casos de uso

Los casos de uso están definidos en nuestra API, es la que gestiona todos los procesos internos que podemos realizar con los datos, desde la solicitud de préstamos hasta dar de alta a un usuario.

### Controladores, presentadores y vistas de los modelos

Este conjunto está definido por las herramientas que nos ayudan a solicitar y adaptar la información más interna de la aplicación. También en esta parte tenemos las interfaces las cuales nos ayudan a adaptar los datos que nos llegan de los casos de usos para poder interpretarlos.

### Vistas y fuentes externas

Aquí tenemos los frameworks, los adaptadores de red, el servidor de la base de datos y las vistas que es la forma en que estructuramos y diseñamos la información para que se muestre al usuario.

## 4.4 Creación y puesta en marcha de la Interfaz de Programación de Aplicaciones (API)

Las APIs son una de esas pequeñas implementaciones que han cambiado enormemente los desarrollos en la informática, hoy en día si tu aplicación no dispone de una API implica varios aspectos:

- Tu aplicación no presenta una separación cliente servidor. Esto quiere decir que su modularidad es baja, en caso de querer realizar cambios sobre la aplicación estos van a ser complicados de acometer. En otro caso de que tu aplicación se quiera expandir a una nueva tecnología esta tendrá que volver a implementar otro modelo de comunicación con la base de datos.
- La expansión de tu aplicación a otras tecnologías conllevará un esfuerzo mayor que si tuviera una API.
- Tu servidor requiere más servicios y por tanto recursos para funcionar.
- La lógica de negocio de la aplicación ha sido integrada en conjunto con la web presentando una alto nivel de acoplamiento y dificultando la modificación de la misma.

Una API nos ofrece una capa de abstracción para nuestra aplicación. Nuestro famoso modelo de caja negra del que he hablado con anterioridad. Esto permite a distintas tecnologías orientadas a desarrollo web, móvil y de programas de ordenador a tener un punto en común entre todas. A que cada una no necesite personalizar sus comunicaciones con la base de datos y pueda usar un intermediario. Así es el funcionamiento de las APIs.

Para implementar nuestra API hemos utilizado Node junto a Express. Gracias a Node podemos levantar un servidor web que no necesite de demasiados recursos para su funcionamiento. Esto se haría con las siguientes líneas de código:

```
app.listen(port, () => {  
  console.log('Inventarium API listening at http://api:3000')  
});
```

Otra de las partes que nos aporta Node es su gestor de paquetes, Node Package Manager (NPM), gracias a esto la implementación de funcionalidades más complejas se pueden realizar en un tramo de de tiempo muy pequeño.

Para inicializar un proyecto de NPM utilizamos el siguiente comando dentro de un directorio:



```
npm init
```

Express es uno de los frameworks principales que presenta Node. Es un marco de desarrollo minimalista para Node que nos permite estructurar aplicaciones, crear enrutamientos y un muchos más aspectos relacionados con lo que sería un entorno web.

Para instalar Express en un proyecto NPM y guardarlo en la lista de dependencias escribimos el siguiente comando:

```
npm install express --save
```

#### 4.4.1 Creación del sistema de directorios

Un apartado a tratar en este capítulo es la gestión de directorios y como los distintos ficheros que hay en su ubicación interactúan entre ellos.

Ya habiendo creado nuestro proyecto Node y habiéndole añadido Express esta sería la estructura de directorios que usaríamos:

- **images:** Dentro de este directorio almacenamos las imágenes que se van subiendo a nuestro sitio web. Estas imágenes van ligadas al tipo de dato “grupo de objeto”.
- **node\_modules:** La carpeta de node\_modules es donde se gestionan todos los paquetes de nuestro NPM.
- **routes:** Dentro de la carpeta routers crearemos todo el enrutamiento de nuestra aplicación. En ella se ubica un archivo por cada tabla que se nos presenta en la base de datos.
- **services:** El objetivo de la carpeta services es el de poder realizar todos los tipos de consultas que requiera la aplicación.
  - *dockerignore:* La funcionalidad de este fichero es el de poder ignorar un directorio o fichero en el momento de la creación de una imagen de un contenedor docker. En este caso el único directorio que estamos ignorando es node\_modules. ¿Por qué? Porque en el momento de la generación de una imagen docker suele dar problemas el importar directamente las dependencias que se iban a utilizar a mano. Es mejor que desde el propio sistema sea el gestor de paquetes quien, después de leer el package-lock.json, sea quien instale las dependencias nuevamente.
  - *env:* Aquí ubicamos las variables con las que trabajaremos en nuestro entorno. Es una forma fácil y sencilla de poder generalizar secciones del código. El contenido del archivo es el siguiente:

```
DB_HOST='localhost',
DB_USER='user',
DB_PASSWORD='secretpassword',
DB_NAME='ualinventarium',
```

- *config.js:* Dentro del fichero de configuración añadimos los parámetros que va a coger nuestra variable db para conectarse a la base de datos.

```
const config = {
  db: {
    host: env.DB_HOST,
    user: env.DB_USER,
    password: env.DB_PASSWORD,
    database: env.DB_NAME'
  },
```

```
listPerPage: env.LIST_PER_PAGE || 10,
};
```

#### Utilización del archivo .env

Podemos ver como nuestro archivo que genera la configuración para la base de datos solicita la información dentro de nuestro fichero `env`. El último atributo que intenta solicitar es “LIST\_PER\_PAGE” en el caso que no lo pueda obtener, que será lo que va a ocurrir, pondrá como valor predeterminado 10.

- *Dockerfile*: Este archivo funciona para la generación de una imagen de un contenedor docker.
- *helper.js*: Un pequeño fichero que nos brinda un par de funciones a la hora de manejar consultas con la base de datos.
- *index.js*: El archivo principal de nuestra API, dentro de él inicializaremos todo.
- *package-lock.json*: El fichero principal de NPM que nos ayuda a gestionar todas las dependencias con los paquetes que tenemos instalados en nuestro proyecto.
- *package.json*: El paquete json es el corazón de cualquier proyecto de Node. Registra metadatos importantes sobre un proyecto que necesarios para la publicación de la aplicación, y también define atributos funcionales de un proyecto que npm usa para instalar dependencias, ejecutar scripts e identificar el punto de entrada a nuestro paquete.

#### 4.4.2 Funcionamiento de index.js

Dentro de `index.js` gestionaremos todas las llamadas a los diferentes componentes de nuestra API. Desde añadir el ruteo hasta gestionar la utilización de diferentes librerías que hayamos instalado dentro de nuestro proyecto.

Una variable importante a recalcar es la de Express. Que la definimos así:

```
const app = express();
```

Gracias a nuestra variable “app” enlazamos toda la configuración principal del ruteo que va a tener nuestra API.

Por ejemplo, de nuestra ruta `/api/grupoobjetos` queremos que el usuario pueda hacer interacciones con la tabla de grupo de objetos. Para gestionarlo enlazamos ese punto a nuestro archivo de ruteo que explicaré en el siguiente apartado:

```
//La variable grupoobjetos es nuestra referenciación del archivo de ruteo
app.use('/api/grupoobjetos', grupoobjetos);
```

Realizamos otras gestiones en nuestro componente de Express entre las que se encuentran:

- Aumentar el tamaño de las peticiones que lleguen para la carga de imágenes.
- Configurar un endpoint en la raíz de nuestro servidor que tenga como finalidad comprobar que la API funciona correctamente.

Por último configuramos el puerto de entrada del servidor:

```
app.listen(port, () => {
  console.log('Inventarium API listening at http://api:3000')
});
```

Esto nos permite que nuestra aplicación se encuentre “escuchando” cada petición de entrada que le envíe la página web.

### 4.4.3 Enrutamiento de la aplicación

El objetivo del directorio de “routes” es la correcta gestión de todas las peticiones que necesitamos que se gestionen para el correcto funcionamiento de nuestra página web.

Cuando creamos una herramienta para la gestión de unos recursos de una base de datos lo más normal es que necesitamos realizar el repertorio de herramientas llamadas CRUD, create, read, update and, delete, es decir: crear, leer, actualizar y eliminar.

Al inicio de nuestro archivo de enrutamiento llamamos a la funcionalidad de Router que incorpora Express:

```
const router = express.Router();
```

#### Métodos de petición HTTP

Los métodos de petición HTTP es la definición de un conjunto de elementos que tiene como objetivo realizar diferentes acciones para la gestión de un recurso determinado.

Estos métodos serán los que nos ayuden en nuestra implementación del repertorio de herramientas CRUD. Los que utilizaremos serán:

#### GET (leer)

El método GET es el encargado de solicitar un recurso en específico. Estas peticiones, por norma general, deben tener como único objetivo la recopilación de datos.

Por ejemplo, en el archivo de grupoobjetos implementamos el siguiente método:

```
/* GET group_of_objects. */
router.get('/', async function (req, res, next) {
  try {
    res.json(await group_of_objects.getMultiple(req, req.query.page));
  } catch (err) {
    console.error('Error while getting group_of_objects ', err.message);
    next(err);
  }
});
```

Dentro de nuestra llamada al método “get” de “router” realizamos un “try catch” en el que llamamos a nuestro método “getMultiple” de la clase que hemos importado con anterioridad de group\_of\_objects con el siguiente método:

```
const group_of_objects = require('../services/grupo_objetos');
```

El método “getMultiple” nos devuelve una lista con los diferentes grupos de objetos que hay en el servidor. Ahondaremos más en ese método en la siguiente sección.

Dentro del “catch” hacemos que la API devuelva un mensaje de error como respuesta en caso de fallo.

#### Personalización de peticiones

Recordemos que el endpoint que gestionaba “grupoobjetos” era `/api/grupoobjetos` por lo que al estar definiendo dentro del `router.get` el parámetro `/` queremos que nuestro API ofrezca esa petición desde la raíz del endpoint. Por ejemplo, si quisiéramos personalizar más la petición podríamos modificar este apartado y en vez de usar `/` usamos `/fungibles` para poder acceder a esa consulta GET habría que llamar a `/api/grupoobjetos/fungibles`.

No hace falta que le pasemos ningún parámetro a nuestra petición GET pero lo hemos dejado en caso de realizar alguna implementación extra.

### POST (escribir)

El método POST lo utilizaremos para el envío de una entidad a un determinado recurso. La API hará uso de este método para la adición de elementos a la base de datos.

### PUT (actualizar)

Este método se encarga del reemplazo de una determinada entidad. Lo utilizaremos para la actualización de las filas de nuestra base de datos.

### DELETE (eliminar)

Como su propio nombre indica en inglés el método DELETE lo utilizaremos para borrar un recurso en específico.

Hemos añadido solamente un ejemplo dentro del método GET porque el resto de llamadas se realiza de forma muy parecida. En el POST y PUT se pasarían objetos como parámetros y en el PUT especificaríamos una ID para la actualización de nuestro objeto. En DELETE solamente especificaríamos la ID al igual que dentro del PUT. Por ejemplo, si queremos realizar una llamada eliminando el grupo de objetos con id 4 llamaríamos al método DELETE de la API con la siguiente dirección */api/grupoobjetos/4*.

#### 4.4.4 Consultas con la base de datos

Dentro de nuestras carpeta “services” tenemos las consultas con nuestra base de datos. En ella definiríamos todas las posibles interacciones que querría realizar nuestro usuario con la base de datos.

Las funciones coincidentes en todos los ficheros que se encuentran ubicados en este directorio son las siguientes:

- **getMultiple:** dentro de ella llamamos al método SELECT de MariaDB.
- **create:** llamamos al método INSERT INTO.
- **update:** llamamos al método UPDATE.
- **remove:** llamamos al método DELETE.

Para explicar la estructura de estas funciones cogeré como ejemplo la “getMultiple”:

```
const offset = helper.getOffset(page, config.listPerPage);
const rows = await db.query(
  'SELECT idGrupoObjetos, cantidad, nombre, imagen, marca,
  modelo, cantidadDisponible, tipo, eliminado
  FROM grupoobjetos WHERE eliminado = 0 LIMIT ?,?',
  [offset, config.listPerPage]
);
const data = helper.emptyOrRows(rows);
const meta = { page };

return {
  data,
  meta
}
```

Como podremos comprobar llamamos al método `offset` que nos ayudará a realizar la paginación de nuestro sitio web.

Después de la consulta llamamos a nuestro método de la clase definida anteriormente “helper” llamado “`emptyOrRows`”. Este método nos ayuda a evitar cualquier problemática que nos pueda causar la API si la base de datos nos devuelve el array vacío.

Por último en el “`return`” de la función devolvemos un JSON que tiene como primer elemento la respuesta de la petición y como segundo la página.

#### 4.4.5 Configuración del archivo Dockerfile

Un archivo Dockerfile sirve para la generación de la imagen de un contenedor. En este caso la imagen la utilizaremos posteriormente para nuestro Docker Compose.

**FALTA POR HACER, FINALIZAR API Y PREPARAR EL MONTAJE Y LUEGO DECIDIR**

#### 4.4.6 Gestión de archivos con Express

Para poder realizar la subida de imágenes a nuestra base de datos haremos uso de la librería “`formidable`”.

Esta se encontrará importada en nuestro fichero “`index.js`”. De la siguiente forma:

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

Procedemos luego a incorporarla a nuestro Express de la siguiente forma:

```
app.use(formidable());
```

Gracias a esto ya podremos analizar el campo de archivos de las peticiones que nos lleguen. Veremos el ejemplo de subida de imagen dentro de grupo de objetos.

#### Descomposición de la petición

Dentro de la petición que nos llega a la API cogemos el campo de archivos que va vinculada a ella gracias la utilización de *formidable*.

```
files = req.files
```

#### Análisis del archivo

Luego podemos analizar nuestro archivo para que cumpla los siguientes requisitos:

- Que sea únicamente un solo archivo.
- El método “`isFileValid`” comprueba que el archivo tenga la extensión *jpg*, *jpeg*, *png*. Es decir, que sea una imagen. He seleccionado únicamente esta extensión de archivos porque luego si les cambio la extensión a *jpg* se pueden visualizar sin problemas.

```
// Comprobamos que el archivo sea uno o más de uno
if (!files.length) {
  //En el caso de que sea solo un archivo
  const file = files.image;
  // Comprobamos que el archivo sea válido
  const isValid = isFileValid(file);
  // Creamos un nombre en base al momento actual en que se
  // está subiendo el archivo
```

```
const fileName = time + ".jpg";

if (!isValid) {
  // Si el archivo no es válido lanzamos un error
  return res.status(400).json({
    status: "Fail",
    message: "The file type is not a valid type",
  });
}
```

### Subida de la imagen

Gracias a esto podemos realizar la subida de la imagen sin problemas. Ubicamos nuestro directorio de subida dentro de la carpeta “images”. Como he indicado anteriormente.

```
const uploadFolder = path.join(__dirname, "..", "images",
  "group_of_objects");
try {
  // Cambiamos el nombre del archivo en el directorio
  fs.renameSync(file.path, path.join(uploadFolder, fileName));
} catch (error) {
  console.log(error);
}

} else return;
```

### Enviamos la consulta a la base de datos

Mandamos una solicitud a la base de datos para finalizar con la creación del grupo de objeto. En caso que diera error la subida este proceso se pararía y devolvería error. Como podremos comprobar, al llamar al método de creación de grupo de objetos le pasamos como parámetros el campo *fields*, donde va el cuerpo de nuestra petición.

```
//Consulta post en la base de datos
res.json(await group_of_objects.create(req.fields, time));

} catch (err) {
  console.error('Error while creating group of objects', err.message);
  next(err);
}
});
```

Ya tendríamos nuestra imagen subida con nuestra columna dentro de la base de datos.

## 4.5 Creación de UAL Inventarium

Este es la sección donde se juntan todos los conocimientos y herramientas que hemos ido exponiendo a lo largo de este documentos y los cohesionamos para crear UAL Inventarium.

UAL Inventarium o Inventarium es una herramienta web diseñada para la gestión del inventario del Departamento de Informática de la Universidad de Almería.

La página web está hecha en Angular 12. Angular es una plataforma de desarrollo compuesta por un framework y librerías. Angular nos brinda todas las herramientas necesarias para la creación de un sitio web.

### 4.5.1 Estructura de un proyecto Angular

Para poder desplegar un entorno donde trabajar en Angular primero tenemos que instalarlo en nuestro Node Package Manager (NPM) con el siguiente comando:

```
npm i -g @angular/cli
```

Al hacer esto se nos desplegará nuestro entorno de desarrollo para Angular. El directorio **node\_modules** es donde se almacena el Framework de Angular, el CLI y los distintos componentes que vayamos instalando con el NPM.

#### ¿Qué diferencias hay entre Angular CLI y Angular Framework?

Angular CLI es la Command Line Interface la cual permite poder crear proyectos Angular, añadir componentes, servicios o directivas desde una línea de comandos. Angular CLI se encarga de la gestión de las distintas posibilidades que nos puede ofrecer el framework de Angular.

Los ficheros que se nos despliegan sobre el directorio raíz al crear un proyecto Angular son:

- **.editorconfig**: Un archivo de configuración para editores de código.
- **README.MD**: Archivo de texto que procesa GitHub en sus repositorios. El contenido inicial del fichero en el momento de la creación del proyecto trata sobre documentación acerca del Framework.
- **angular.json**: Esta es la configuración predeterminada que nos aporta el CLI de Angular para poder construir la aplicación, generar el servicio y testear los diferentes componentes.
- **package.json**: Este fichero se encarga de manejar las dependencias de NPM, precisamente las que están habilitadas dentro del espacio de trabajo.
- **package-lock.json**: Nos aporta información del versionado de los distintos paquetes que están en node\_modules.
- **tsconfig.json**: Esta es la configuración básica de TypeScript para el proyecto.
- **proxy.conf.json**: Este es el fichero que nos va a ayudar a poder consumir nuestra API. Reenvía las peticiones que llegan a nuestra aplicación al puerto 3000 que es donde se encuentra ubicada.

En la misma carpeta raíz tenemos un directorio llamado *src*, su descomposición es la siguiente:

- **app**: Directorio que contiene todos los distintos componentes de los que está compuesta la aplicación.
- **assets**: Contiene imágenes y otros recursos para ser copiados en el momento que se construya la aplicación.
- **environments**: Gracias a este fichero podemos configurar una opción en particular de construcción de la aplicación.
- **favicon.ico**: El ícono que sale en la parte superior de la pestaña de la página web.
- **index.html**: La página principal que tiene cualquier web. El CLI se dedica a añadir automáticamente todo el JavaScript y el CSS cuando construye la aplicación. No es un fichero que se use.

- **main.ts**: Este fichero es el punto de entrada principal de la aplicación. Compila la aplicación y arranca el módulo raíz de la aplicación (AppModule) para que se ejecute en el navegador.
- **polyfills.ts**: Provee de adaptaciones para distintos navegadores.
- **styles.css**: Es un archivo de configuración global de estilos para todos los componentes de la aplicación.
- **test.ts**: El punto de entrada principal para los test que se realicen en la aplicación.

El contenido del directorio *app* en el momento de la creación del proyecto es el siguiente:

- **app.component.ts**: Define la lógica para la aplicación raíz.
- **app.component.html**: Define el diseño HTML asociado con el elemento raíz.
- **app.component.css**: Define el elemento de diseño para el elemento raíz.
- **app.component.spec.ts**: Define el conjunto de pruebas asociado con el elemento raíz.
- **app.module.ts**: Define el módulo raíz, este fichero le comunica a Angular cómo se tiene que realizar el ensamblaje de la aplicación. Inicialmente está declarado dentro de él el propio módulo raíz, pero a medida que vayamos añadiendo elementos a nuestra aplicación irá incluyendo más módulos.

Dentro de la carpeta *src* hay tres directorios más:

- *components*
- *interfaces*
- *services*

### Componente

Los componentes son las estructuras principales de construcción que tenemos en Angular. Para poder generarlos utilizamos el siguiente comando:

```
ng g c direccion_y_nombre_del_componente
```

Cuando lo ejecutemos generará un nuevo directorio con el nombre del componente. Dentro de él se habrán creado cuatro ficheros diferentes:

- **componente.html**: Aquí irá ubicado el diseño html que tendrá el componente.
- **componente.css**: Este documento de estilos se aplicará únicamente al componente.
- **componente.ts**: En nuestro fichero TypeScript tenemos la lógica del componente y cualquier tipo de procesamiento de datos que haya que realizar.
- **componente.spec.ts**: Este será nuestro fichero de pruebas unitarias para el componente. Durante el desarrollo del proyecto no lo he utilizado ya que el testing del proyecto lo he hecho de otra forma.



## Servicio

Los servicios sirven para aislar más el modelo de vista controlador que presentan los componentes. Estos serán los encargados de comunicarse con nuestra API.

Para generar un servicio utilizamos el siguiente comando:

```
ng g s directorio_y_nombre_del_servicio
```

Se generarán dos ficheros TypeScript, uno para el conjunto de pruebas y otro para el servicio.

## Interfaz

La interfaz sirve para definir los elementos con los que vamos a trabajar. Estos elementos corresponden a los que tenemos en nuestra base de datos.

Para poder generar una interfaz escribimos lo siguiente en la terminal dentro de nuestro proyecto:

```
ng g i directorio_y_nombre_de_la_interfaz
```

Un ejemplo de interfaz sería por ejemplo la de un objeto del tipo *Configuración*:

```
export interface Configuracion {  
  idConfiguracion: number,  
  ip: string,  
  mac: string,  
  boca: string,  
  armario: string,  
  usuario: string,  
  contrasena: string,  
  Objeto_idObjeto?: number  
}
```

Estos elementos también nos ayudarán a procesar las respuestas que nos llegarán desde la API y poder manipularlos en nuestros componentes sin problemas.

Ya sabemos qué tipos de componentes conforman un proyecto Angular. Ahora metemos de lleno las manos en la masa.

### 4.5.2 Disposición y elementos que hay en Inventarium

Empezaremos hablando de los servicios y las interfaces y cómo se presenta su contenido en el caso de los primeros.

## Interfaces

Tenemos una interfaz por cada elemento que se encuentra en la base de datos.

## Services

Los servicios pueden ser los que hayan dado más tipos de problemas a lo largo del desarrollo. Son los encargados de generar las consultas HTTP de las que hemos hablado en la sección anterior en el desarrollo de la API.

Tenemos tantos servicios como tablas en la base de datos quitando el de *user* que sirve para iniciar sesión dentro de la aplicación.

Para mostrar la estructura de uno de los servicios de la aplicación mostraré como ejemplo el fichero *group-of-objects.service.ts*. Al principio del documento tendríamos las importaciones de código y más adelante declaramos lo siguiente:

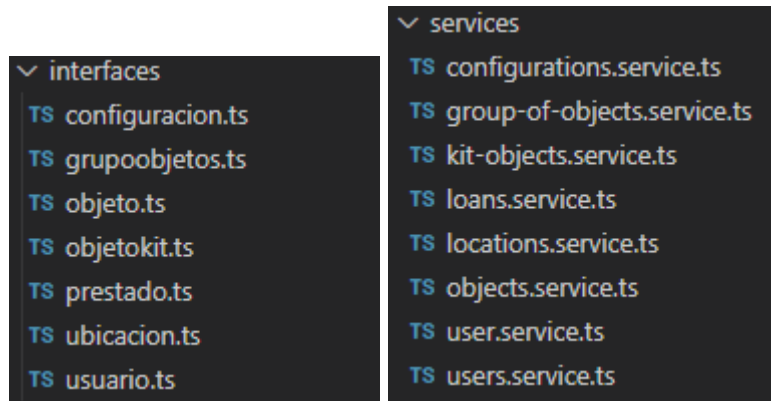


Figure 4.3: Interfaces y servicios de UAL Inventarium

```
@Injectable({
  providedIn: 'root'
})
```

El `@Injectable` junto al `provideIn: 'root'` sirve para que el servicio pueda utilizarse en todo el proyecto Angular.

Avanzando un poco más por el código nos encontramos con la declaración de nuestra variable `_url` a la cual le indicaremos la dirección a la que tiene que mandar la solicitud.

```
_url = "api/grupoobjetos"
```

Como vemos esa es la ruta a la que accederá a nuestra API. La dirección es así porque se ha habilitado un proxy que redirige las peticiones. De tal proxy se hablará más adelante.

En el constructor de nuestra aplicación inicializaremos la siguiente variable:

```
constructor(private http: HttpClient) {}
```

`HttpClient` es el servicio encargado de mandar las solicitudes HTTP a la API.

Ahora ya podemos empezar a crear las funciones que utilizaremos a lo largo del desarrollo:

```
getGroupOfObjects() {
  return this.http.get(this._url);
}
```

En este caso la petición que se manda es del tipo `get` si queremos crear un grupo de objetos sería así (con una petición `post`):

```
addGroupOfObject(objectGroup: FormData) {
  return this.http.post(this._url, objectGroup);
}
```

Donde como parámetro pasamos un campo del tipo formulario.

Para el resto de peticiones `put` y `delete` solamente le añadimos un campo numérico que sea la id del objeto que se va a modificar `/:id` y en el caso de `put` también le pasamos otro tipo de objeto formulario.

Con esto ya tendríamos definido nuestro servicio y listo para funcionar.

### 4.5.3 Creación de componentes

Ya sabemos cómo crear un componente pero no he explicado un poco de forma más detallada su funcionamiento.

Como ejemplo hablaremos de nuestro componente `groups-of-objects` que es donde se visualizan todos los grupos de objetos.

## Preparar nuestro apartado TypeScript

Declaramos las variables que necesitaremos, en este caso solo es una:

```
group_of_objects: GrupoObjetos[] = [];
```

Aprovechamos también para inicializar nuestro array de grupo de objetos. Como se puede ver al declarar la variable hemos llamado a su interfaz para que su utilización sea mucho más cómoda. Para poder cargar los grupos de objetos que hay definidos en Inventarium tendremos que importar su servicio:

```
constructor(private group_of_objects_service: GroupOfObjectsService)
```

Y al querer que cuando accedamos a la página esta cargue los respectivos grupos de objetos, dentro del constructor tendremos que llamar al método para que lo haga:

```
this.group_of_objects_service.getGroupOfObjects().subscribe(
  (data : any) => {
    this.group_of_objects = res.data;
  },err => console.log('Error', err));
```

El método *getGroupOfObjects()* es el método que hemos definido anteriormente, el *.subscribe* es para poder realizar la consulta. Dentro de él podremos decidir cómo manipular los elementos que nos devuelva la petición.

Este elemento es un objeto del *json* pero Angular lo interpreta perfectamente. Objeto que puede tener uno de estos dos atributos, *data* que es un array de grupo de objetos, significando que la consulta no ha tenido errores y *err* que nos devuelve una cadena de caracteres en las que sale el tipo de error que ha ocurrido.

Con esto ya tendríamos nuestro objeto cargado, pero ahora tenemos que mostrarlo.

## Preparar nuestro archivo HTML

Nuestro fichero HTML será el siguiente:

```
<div class="row d-flex-inline justify-content-center">
  <!--Contenido-->
  <div *ngFor="let go of group_of_objects" style="width: fit-content;">
    <app-group-of-object>
      
      <h5 nombre [routerLink]="['/group-of-object',
        go.idGrupoObjetos]" style="cursor:pointer"
        class="card-title text-dark text-center">
        {{go.nombre}}
      </h5>
      <span cantidad>{{go.cantidad}}</span>
      <span marca>{{go.marca}}</span>
      <span modelo>{{go.modelo}}</span>
      <span cantidadDisponible>{{go.cantidadDisponible}}</span>
      <span *ngIf="go.tipo==0" tipo>Inventario</span>
      <span *ngIf="go.tipo==1" tipo>Fungible</span>
      <span *ngIf="go.tipo==2" tipo>Kit</span>
    </app-group-of-object>
  </div>
</div>
```

El componente es bastante sencillo. Primero definimos un contenedor donde iremos insertando tantos componentes nos haya devuelto la petición mandada anteriormente.

Ahora definimos otro contenedor donde iteraremos sobre nuestro array de grupo de objetos asignandolo a una variable auxiliar *go*. Esto lo haremos utilizando *\*ngFor*.

Luego de iterar llamaremos a nuestro componente hijo que será el que iremos generando por cada grupo de objeto que cargue. Este componente lo veremos ahora después.

Dentro del componente hijo definiremos los componentes HTML que le queramos pasar. Para poder definir esto basta con añadirle un nombre que después referenciaremos en el otro componente.

Podemos ver al final del código que definimos un atributo dentro de nuestro componente HTML *span* llamado *\*ngIf*. Este atributo es un condicional, en caso de que sea *true* se cargará el componente. Si es *false* no lo hará.

Ahora definiremos nuestro componente hijo que es al que estamos llamando:

```
<div class="card border rounded p-3 m-2"
  style="width: 22rem;background-color:#FDF7FF;">
  <div style="width: 100%; height: 230px;">
    <ng-content select="[imagen]"></ng-content>
  </div>
  <div style="width: 100%;" class="card-body list-group-item-dark border">
    <ng-content select="[nombre]"></ng-content>
  </div>
  <ul class="list-group list-group-flush">
    <li class="list-group-item bg-light">
      <b>Marca</b>
      <a><ng-content select="[marca]"></ng-content></a>
    </li>
    <li class="list-group-item bg-light">
      <b>Modelo</b>
      <a><ng-content select="[modelo]"></ng-content></a>
    </li>
    <li class="list-group-item bg-light">
      <b>Cantidad</b>
      <a><ng-content select="[cantidad]"></ng-content></a>
    </li>
    <li class="list-group-item bg-light">
      <b>Cantidad disponible</b>
      <a><ng-content select="[cantidadDisponible]"></ng-content></a>
    </li>
  </ul>
  <li class="list-group-item list-group-item-dark
  font-weight-bold text-center">
    <ng-content select="[tipo]"></ng-content>
  </li>
  <ng-content select="[botones]"></ng-content>
</div>
```

Este es el modelado que tiene nuestro objeto. El componente HTML llamado *ng-content* será el encargado de tomar los objetos que le está pasando el componente padre. Estos los referencia con el atributo *select* y le indica el nombre del tipo de componente que quiere coger.

Para poder ir revisando cómo quedan nuestros componentes utilizaremos una funcionalidad que nos incorpora Angular. Dentro de nuestro archivo *package.json* NPM nos define una serie de scripts que podemos utilizar. Uno de ellos es el siguiente:

```
"start": "ng serve"
```

Para poder ejecutar este script ejecutaríamos el siguiente comando:

```
npm run start
```

Que sería lo mismo que utilizar:

```
ng serve
```

Esto nos despliega un servidor de la aplicación de Angular en el puerto 4200. Una de las ventajas que nos ofrece esto es una compilación continua del proyecto. Es decir, a medida que vayamos realizando la construcción de la aplicación, la generación de componentes y la creación de rutas la web se irá actualizando en cada guardado y será de gran utilidad poder ir viendo estos cambios al momento.

#### 4.5.4 Definir el archivo de rutas

Definir el archivo de rutas de la aplicación sirve para saber qué direcciones y que comportamientos tendrá esta en su uso.

Las rutas de la aplicación son las siguientes:

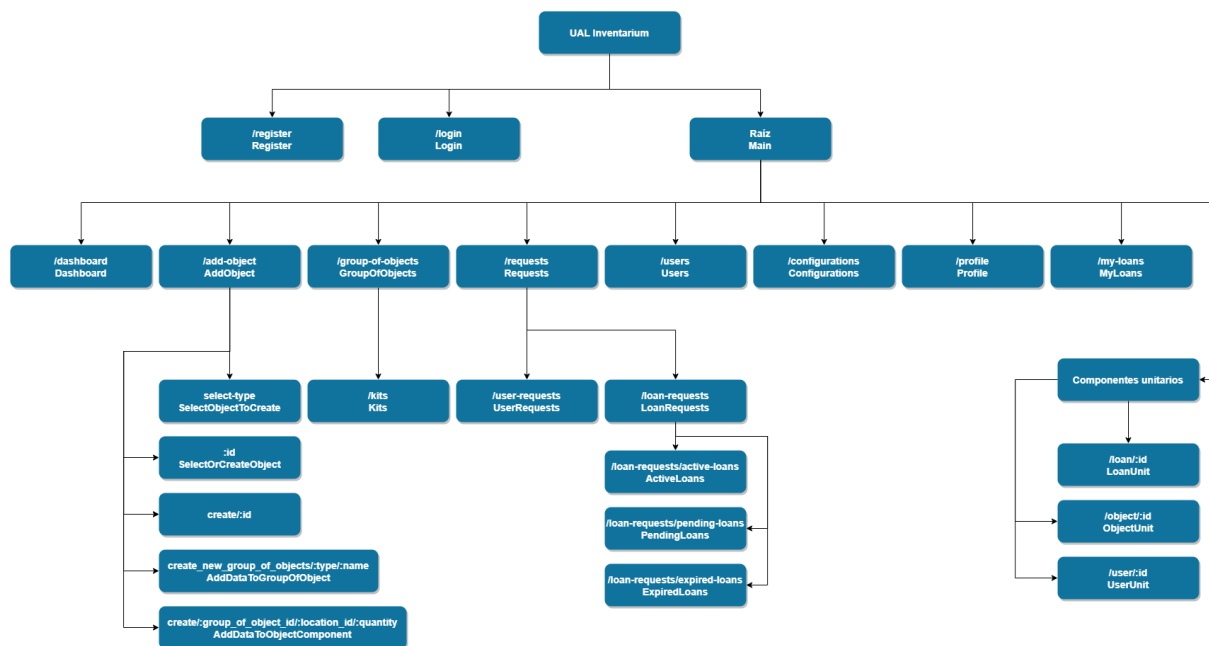


Figure 4.4: Rutas disponibles en la aplicación

Las rutas de la aplicación tienen una correlación directa con el manejo de los componentes. Estas rutas se definen dentro del fichero *app-routing.module.ts* que está dentro de *app*. Este archivo importa todos los componentes que vayamos a utilizar en nuestras rutas. El comienzo del archivo consiste en definir una constante que llamaremos *routes* y en la que irá toda la lógica de nuestra aplicación:

```
const routes: Routes = [
{
  path: '', component: MainComponent,
  children: [
    { path: 'dashboard', component: DashboardComponent },
    {
```

```
path: 'add-object', component: AddObjectComponent,
children: [
  { path: 'select-type', component: SelectObjectToCreateComponent },
  .
  .
  .
```

El archivo es bastante más extenso pero con este pequeño trozo podremos ver y explicar los diferentes componentes que contiene.

Para definir una ruta de forma básica la estructura será la siguiente *path: nombre\_de\_la\_ruta, component: nombre\_del\_componente* con esto al acceder a la ruta ya se nos mostraría dicho contenido.

Lo interesante que nos aporta Angular es la posibilidad de añadir hijos a estas rutas. Lo haremos llamado al atributo *children: [array\_de\_rutas]*.

Esto es muy importante ya que el uso de hijos en las ruta hace que los componentes de rutas superiores no se descarguen. Se explicará mejor con el siguiente ejemplo utilizando nuestro *MainComponent.html*:

```
<app-vertical-navbar></app-vertical-navbar>
<div class="page-content p-5" id="content">
  <!-- Toggle button -->
  <button id="sidebarCollapse" (click)="sidebarCollapse()" type="button"
    class="btn btn-light bg-white rounded-pill shadow-sm px-4 mb-4">
    <i class="fa fa-bars mr-2"></i>
    <small class="text-uppercase font-weight-bold">Toggle</small>
  </button>
  <router-outlet></router-outlet>
</div>
```

Dentro del HTML nos encontramos con un componente llamativo *router-outlet*. Este está relacionado directamente con el routing que tengamos en nuestra aplicación ya que es desde ahí donde se cargarán nuestros hijos. La barra de navegación, por ejemplo, no desaparecería al acceder a */dashboard*.

Para realizar el importado de rutas dentro de Angular llamaremos a *@NgModule* al final del documento para que las exporte a nuestro componente principal.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
```

#### 4.5.5 Definir el proxy en nuestra aplicación

La finalidad de un proxy inverso es reenviar las solicitudes que realiza la aplicación a la API, esta por políticas de CORS no puede reenviar una solicitud que entra al puerto 80 al puerto 3000 del mismo sitio. Tampoco podemos hacer que la API funcione sobre el puerto 80 ya que este puerto está siendo utilizado por nuestra aplicación web. Por ello tenemos que hacer que este proceso se haga de forma interna en nuestro servidor y para ello utilizaremos un proxy inverso. Un proxy inverso se encarga de reenviar las solicitudes que llegan a unas determinadas rutas del sitio web a otro puerto de nuestro entorno pero que no está alojado en nuestro sitio. En este caso el proxy sería el encargado de que todas las solicitudes que entrasen por el puerto 80 de las direcciones definidas en la sección anterior sean redirigidas a nuestra API que trabaja sobre el puerto 3000.

Este proxy inverso lo definiríamos en el fichero *proxy.conf.json* de la siguiente forma:

```
"/api/configuracion*": {  
  "target": "http://localhost:3000",  
  "changeOrigin": true  
}
```

Y para poder utilizarlo durante el desarrollo de la aplicación modificaremos el comando que utilizamos para inicializar el entorno de pruebas por el siguiente:

```
"start": "ng serve -o --proxy-config proxy.conf.json --host 0.0.0.0"
```

`/api/configuracion*` es para indicar que todas las solicitudes que llegen a la web con esa dirección se reenvíen a `"target": "http://localhost:3000"` y que se cambie el origen de la solicitud `"changeOrigin": true`.

#### ¿Qué son las políticas de CORS?

CORS (Cross-Origin Resource Sharing) es un mecanismo o política de seguridad que permite controlar las peticiones HTTP asíncronas que se pueden realizar desde un navegador a un servidor con un dominio diferente de la página cargada originalmente. Este tipo de peticiones se llaman peticiones de origen cruzado (cross-origin). Estas peticiones no están permitidas por ley porque suelen ser utilizadas para la piratería informática.

#### 4.5.6 Compilar nuestro sitio Web

Para compilar nuestro sitio web ejecutaremos el siguiente comando dentro de la raíz de nuestro proyecto:

```
ng build
```

Esto nos generará un directorio *dist* con otro directorio dentro con el nombre de nuestro proyecto. En este caso *Inventarium*. Este contendrá todos los ficheros para poder añadirlo directamente a un servidor web.





## 5 Deploy de nuestra Web

Al terminar la construcción de nuestra página lo que haremos será preparar los archivos para añadirlos a nuestro servidor web.

En un principio, es más, en la anterior frases que escribí pensé que el tiempo dedicado a este apartado sería de poco más de cinco horas, ya que no era mucho trabajo. Me equivocaba.

Un objetivo de la preparación del deploy del sitio es que estuviera todo en un archivo docker-compose. Esto en vistas a una mejor configuración y a poder asegurar el entorno antes de llevarlo a producción.

Por lo que nuestro fichero docker-compose se tendría que encargar de levantar los siguientes elementos:

- La base de datos de MariaDB
- El servidor Node que maneja nuestra API
- El servicio de PHPMyAdmin para manejar la base de datos ante cualquier problemática
- El servidor Web con el Deploy de Angular que irá con Nginx

Más adelante en un futuro añadiremos otro bloque más que será el gestor de copias de seguridad de la base de datos.

Del apartado de configuración de la base de datos con PHPMyAdmin ya hemos hablado por lo que pasaremos directamente a los dos últimos puntos. Las joyas de la corona.

### 5.1 Levantar el servidor para nuestra API

Para poder levantar el servidor con nuestra API necesitábamos sí o sí realizar una configuración de una imagen docker Node para poder inicializar el Node Package Manager.

También quería ahorrar intermediarios por lo que si podía generar la imagen Docker desde Docker Compose sería mucho mejor.

Un fichero Dockerfile sirve para generar una imagen docker (no un contenedor) que podremos ir manipulando a nuestro antojo para que tenga la configuración que necesitamos o nos provea de los servicios que necesitamos.

La estructura de un archivo Dockerfile es la siguiente:

```
FROM node:16.4
```

Primero definimos la imagen desde la que vamos a partir. Este es un proceso obligatorio. Generaremos nuestra imagen para la API desde la versión de node 16.4 que es con la que se ha llevado a cabo el desarrollo.

```
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
```

Nuestro segundo paso será definir nuestra dirección del directorio de trabajo. Esta dirección `/app` será donde se ubicará el directorio raíz de nuestra API.

Luego copiaremos todos los archivos que empiecen por *package* que en este caso son dos: *package.json* y *package-lock.json*. Dentro de estos ficheros se encuentra toda el versionado y la configuración de los plugins que vamos a utilizar en nuestro sistema.

Luego de copiar la configuración del versionado iniciamos nuestro Node Package Manager. Este, en base a los paquetes que hemos declarado anteriormente, será el encargado de generar todos los directorios que necesita Node para funcionar.

Después de haber instalado NPM el siguiente paso es copiar todo el directorio raíz de nuestra API dentro de la imagen que estamos creando, eso lo hacemos con `"COPY . ."`.

### ¿Por qué no copiamos directamente nuestro directorio raíz con los plugins y demás?

Esta pregunta se me ocurrió hace unos dos años, ya que me parecía una tontería volver a descargar exactamente los mismos paquetes para tenerlos en nuestra Aplicación. Frente a todos los sitios web que no realizaban ese copiado entero me dispuse a hacerlo yo. Lo que ocurre cuando haces una copia exacta es que tu sistema ha adaptado las librerías y alguna configuración extras a tu entorno. Por lo que luego al contenerizarlo da error. Lo bueno que nos permite Docker es que esté donde esté el sistema que vayamos a utilizar, mientras esté en un contenedor, esté será el mismo para todos los usuarios.

```
EXPOSE 3000
```

```
CMD ["npm", "run", "dev"]
```

Tenemos ya nuestra imagen casi creada, lo siguiente sería abrir el puerto 3000 para que el usuario pueda mandar solicitudes a la API y ejecutar el comando `npm run dev` para que esta empiece a funcionar.

Junto a este Dockerfile también tenemos que generar un *.dockerignore* que al igual que *.gitignore* sirve para que en el momento de la creación de nuestra imagen que hacemos con el Dockerfile nuestro sistema Docker no coja ni los ficheros ni directorios que se encuentren declarados dentro del documento.

El archivo contiene los siguientes campos de texto:

```
node_modules
Dockerfile
.git
```

Ignoramos el directorio *node\_modules* que es donde se descargan nuestros paquetes. El *Dockerfile* y el *.git* es para que la imagen no tenga contenido extra que no vaya a usar.

Con todo esto ya definido pasaríamos a añadir el apartado de nuestra API en el Docker Compose.

```
api:
  build:
    context: ../API
    dockerfile: ../API/Dockerfile
```

Dentro de nuestra sección *build* definiremos el sitio donde Docker Compose generará la imagen que vayamos a utilizar para el contenedor. En el apartado *context* definimos el directorio que será la raíz de la generación y en el apartado *dockerfile* como su propio nombre indica seleccionamos el archivo Dockerfile que utilizaremos.

```
image: inventarium_api:1.0
restart: always
```

Aquí definimos el nombre de la imagen que estamos creando y *restart: always* nos servirá para que en el momento en el que el contenedor se pare este se vuelva a iniciar.

### ¿Cuándo se para un contenedor?

Un contenedor puede dejar de funcionar por varias razones pero, la principal, es porque se ha quedado sin tareas que realizar. En nuestro caso si ocurre eso significaría que nuestra API ha dado un error y se ha parado. Como no queremos que esta deje de funcionar hacemos que se vuelva a ejecutar.

```
container_name: api
ports:
  - "3000:3000"
volumes:
  - ../API:/app
networks:
  - default
```

Por último definimos el nombre del contenedor, que redirija el puerto 3000 a el puerto 3000 del sistema, que realice un copiado de archivos del contenido de la carpeta API dentro de la raíz del sistema en caso de que haya nuevos archivos para subir y por último que esté funcionando sobre la red con el nombre *default*.

## 5.2 Levantar nuestro servidor Web

La primera vez que levanté un servidor Web para el desarrollo de un proyecto final de la asignatura Desarrollo Rápido de Aplicaciones este lo hice con NPM. Es decir, no compile el proyecto de Angular.

Esta vez no iba a ser tan fácil, aquella vez tampoco lo fue, y el haber tenido que preparar todo para que funcionara desde un entorno Docker dificultó bastante la realización de este apartado. Han surgido dos problemas principales al intentar realizar esto:

1. Configurar las rutas de la aplicación desde el servidor web.
2. Configurar un proxy reverso desde el servidor web.

El servidor web que se utilizó fue **Nginx**. La configuración inicial fue bastante rápida. Quedando el fichero *docker-compose.yml* así en un principio:

```
client:
  image: nginx:latest
  ports:
    - 80:80
  volumes:
    - ../inventarium/dist/inventarium:/usr/share/nginx/html
```

Importamos la última imagen de nginx, redirigimos el puerto 80 al puerto 80 nuestro y por último importamos el contenido del sitio web dentro de el directorio */usr/share/nginx/html* de nginx.

Hasta este punto parecía que estaba todo bien pero al acceder al sitio web me di cuenta de dos cosas: que no llegaban las solicitudes a la API y que tampoco podía acceder al archivo de rutas más allá que la raíz de Angular. Es decir, no podía acceder a la ruta */group-of-objects* por ejemplo.

Con *ng build* no se había importado el proxy que habíamos creado en nuestro sitio web y tampoco

funcionaba la configuración de rutas, aunque, si navegaba dentro de la aplicación sí lo hacía. Esto se debía a la configuración de rutas que estaban configuradas en nginx. Rutas que se modificaban desde `/etc/nginx/conf.d/default.conf`.

Por lo tanto procedí a crear un archivo llamado `default.conf` y añadí una línea más en la sección de `volumes` del fichero del Compose. Esta línea era:

```
- ./web/default.conf:/etc/nginx/conf.d/default.conf
```

Teníamos que configurar las rutas en `default.conf`.

### 5.2.1 Configuración del ruteo de la web

El objetivo de configurar las rutas de la web es que todas las direcciones apuntásen al mismo archivo `index.html`. Para eso añadiríamos dentro del fichero las siguientes líneas:

```
location / {
    root /usr/share/nginx/html;
    try_files $uri $uri/ /index.html;
}
```

Esta configuración define las rutas entrantes a la dirección raíz `“/”`, en el caso de que sean allí apuntará a nuestro directorio `html` que hemos indicado anteriormente y si no es el caso lo hará sobre `try_files $uri $uri/ /index.html;`. Sobre `index.html`. Es decir, lo mismo que si apuntara a la raíz del servidor.

### 5.2.2 Configuración del proxy inverso de la web

Este apartado se volvería el más complicado pero por unas razones que explicaré en el apartado de resolución de problemas.

Como expliqué en la sección 4.5 para poder conseguir que la aplicación se comunicara con la API esta tenía que disponer de un proxy que redirigiese unas determinadas rutas dentro del puerto 80 a otras del puerto 3000.

Para poder hacer esto en Nginx añadiremos las siguientes líneas de código en el archivo anterior:

```
location /api/users/login {
    proxy_pass http://api:3000;
    proxy_pass_request_headers on;
}
```

Sigue casi la misma estructura que en el apartado anterior siendo `proxy_pass` la dirección donde reenviará las peticiones. El dominio donde las reenvía se llama `api`, esto es debido a que dentro del entorno que genera Docker Compose podemos llamar a las máquinas creadas en base a su referenciación.

La siguiente línea sirve para poder reenviar el encabezado de nuestras solicitudes para que en la ampliación del proyecto podamos generar un sistema de tokens con el objetivo de querer aumentar la seguridad.

Con esto ya tendríamos definido nuestro servidor web donde se localizaría la aplicación.

## **6 Pruebas durante el desarrollo y corrección de errores**

Hablar de Postman y de Selenium y las herramientas de google chrome



## **7 Conclusiones y posibles mejoras**





# References

- Biot, M. A. (1962). Mechanics of deformation and acoustic propagation in porous media. *Journal of applied physics*, 33(4), 1482–1498.
- Heinz, M., Carsten, & Hoffmann, J. (2014, March). *The listings package, march 2014*. <http://texdoc.net/texmf-dist/doc/latex/listings/listings.pdf>. Retrieved 12/12/2014, from <http://texdoc.net/texmf-dist/doc/latex/listings/listings.pdf>