IES VALLE DEL JERTE

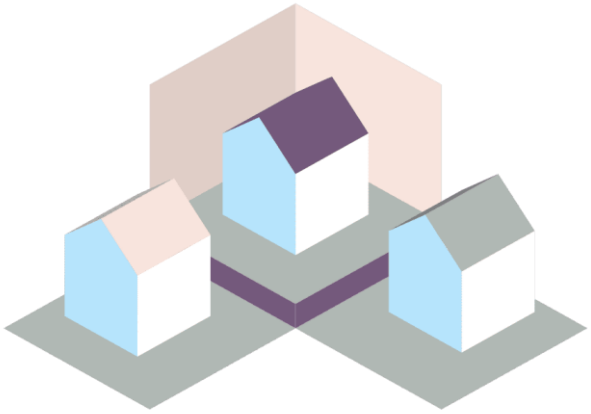
Grado Superior de Desarrollo de Aplicaciones Web

Trabajo Fin de Ciclo

Curso académico 2021/22

PROYECTO DE TELEASISTENCIA: CREACIÓN DE UN CLIENTE ANGULAR PARA DESARROLLAR LAS INTERFACES DE USUARIO

INNOVACIÓN APLICADA



TLA

Servicio de teleasistencia

Autor: Lucía González Martín

DNI: 76137838H

Plasencia, 3 de diciembre 2021

ÍNDICE DE CONTENIDOS.

ÍNDICE DE ILUSTRACIONES.

1.- Descripción del proyecto.

El proyecto que nos ocupa consiste en realizar la interfaz inicial de la aplicación de teleasistencia solicitada por el IES "San Martín" a nuestro centro para que sus alumnos del ciclo formativo de grado medio de *Atención a Personas en Situaciones de Dependencia* (*APSD*) puedan poner en práctica, desde el propio instituto, las tareas de teleoperador realizadas por los distintos centros de atención a personas dependientes.

En lo referente al desarrollo del citado proyecto, se empleará el *framework Angular* con su utilidad de *routing* para crear un cliente web con una navegación intuitiva para el usuario final. La idea fundamental es establecer los estilos, *routing*, *login* y formularios principales de la aplicación para que, en un futuro, esta se siga desarrollando en mayor profundidad.

Por tanto, para implementar este proyecto, se ha intentado aunar los conocimientos adquiridos en los módulos de: *"Diseño de Interfaces Web"* y *"Desarrollo Web en Entorno Cliente"*; efectuando el cliente inicial para la aplicación y un dossier que recoge desde el planteamiento de este cliente hasta los medios empleados y la descripción técnica de todo el desarrollo del mismo.

2.- Justificación del proyecto.

Se ha decidido llevar a cabo este proyecto, consistente en la creación de un cliente web para una aplicación de teleasistencia, dado que supone una

oportunidad de poner en práctica los conocimientos adquiridos en los módulos de *"Diseño de Interfaces Web"* y *"Desarrollo Web en Entorno Cliente"* y, debido también, al interés personal en el desarrollo web *front-end*. Igualmente, se considera interesante abordar un proyecto de estas características, puesto que, al ser de tipo colaborativo, representa una oportunidad para conocer en profundidad la metodología de trabajo aplicada en las empresas a la hora de desarrollar aplicaciones similares.

Por un lado, la elección de la temática del presente trabajo surgió a partir del planteamiento, realizado por el IES "San Martín", consistente en la creación de un *software* informático que permitiese a sus alumnos, del ciclo formativo de *APSD*, poner en práctica y desarrollar, desde el centro educativo, las tareas que realizan los teleoperadores de los centros de atención que prestan servicios de teleasistencia.

En líneas generales, la teleasistencia es un servicio, de carácter autonómico o provincial, dirigido a colectivos de la tercera edad, colectivos con diversidad funcional, personas que sufren violencia de género y colectivos con enfermedades crónicas. Asimismo, debemos tener en cuenta cuales son las principales tareas desarrolladas por estos teleoperadores que, según el profesorado del ciclo de *APSD*, son de dos tipos:

- Responder y auxiliar a los usuarios del servicio, ante posibles situaciones de alerta, a través de la movilización tanto de recursos personales (de un familiar, de un vecino o de un amigo del usuario), como de recursos

comunitarios (ambulancia, bomberos, policía, servicios sociales, …) si fuese necesario.

- Realizar llamadas de agenda, para recordarle a los usuarios ciertos aspectos (citas médicas, tomas de medicamentos, hacer seguimiento, llamadas de compañía, etc.) y, velar así, por la seguridad y cuidado de estos colectivos.

Teniendo en cuenta todo esto, la propuesta realizada por los profesores del IES "San Martín" contempla tres aspectos fundamentales. En primer lugar, buscan que el diseño de la aplicación web cuente con una serie de pestañas, con sus respectivas celdas desplegables, que les permitan gestionar dichas tareas esenciales realizadas en los centros de atención.

En segundo lugar, contemplan la posibilidad y necesidad de que celdas de distintas pestañas estén relacionadas, de modo que, al escribir una llamada de agenda en la pestaña del usuario, esta información se escriba automáticamente en la pestaña de llamadas de agenda en el día que se necesite.

Por último, también precisan de un sistema de acceso a la web mediante usuario y contraseña, a través del cual, poder distinguir entre profesores y alumnos dentro del servicio y gestionar las partes de la web que son accesibles a cada uno de ellos.

Por otro lado, se considera que el contenido del proyecto podría ser relevante, ya que, sentará las bases para que, en próximas convocatorias de trabajos de

fin de ciclo, otros alumnos puedan seguir ampliando y mejorando la interfaz web de la aplicación.

Finalmente, otro punto relevante en la decisión de llevar a cabo este trabajo de fin de ciclo, es que, el cliente web creado será utilizado en un entorno real. A diferencia de otros proyectos de innovación aplicada, de revisión bibliográfica o de creación de empresas, cuyo ciclo de vida se acaba una vez se ha presentado dicho proyecto. Esta aplicación tendrá un recorrido a través de distintos *PFC* para conseguir satisfacer los requisitos de unos usuarios finales, el personal docente del ciclo de grado medio de *APSD* y, finalmente, una vez realizadas las pruebas de funcionamiento, será empleada por estos usuarios e incluso, en un futuro, puede ser utilizada por usuarios de otros institutos con las mismas necesidades.

3.- Objetivos del proyecto.

El principal objetivo perseguido por este proyecto es inicializar un cliente de *Angular*, a través del cual, se desarrollarán las interfaces de usuario de la aplicación de teleasistencia solicitada al IES "Valle del Jerte" por parte del cuerpo docente del ciclo de *APSD*.

Este objetivo general se divide en una serie de objetivos más concretos, como son:

1. Realizar un sitio web, utilizando para ello la herramienta *Angular* con *routing*, que contenga una cabecera común para las diferentes pestañas de la web, una funcionalidad que permita realizar tanto el *login* como el *logout*, una página principal con información sobre el usuario logueado y, una serie de ventanas, por las que el usuario final puede navegar, en las que se encuentran los formularios tanto de creación como de modificación de las entidades con menos dependencias dentro de la aplicación.

2. Elaborar, utilizando los conocimientos adquiridos durante el ciclo, un cliente web *Angular* que contenga unos estilos generales *CSS* consistentes, que esté organizado mediante componentes y servicios y, que emplee formularios como sistema para gestionar la entrada de datos de las entidades.

3. Efectuar una solución en forma de aplicación web, para la propuesta planteada por parte del IES “San Martín”, que sea sencilla y con un funcionamiento intuitivo con el fin de que sus alumnos del ciclo de *APSD* puedan emplearla para adquirir y reforzar conocimientos sobre teleasistencia.

4. Asimismo, de forma simultánea a la construcción del cliente web, se documentará el paso a paso seguido hasta alcanzar el resultado final mediante el *software* de control de versiones *Git* y un repositorio remoto

alojado en *GitHub*. No obstante, también se recogerá en el presente documento una explicación más extensa y detallada de ello, así como una descripción de los medios utilizados, el contexto en el que se desarrolla la aplicación, otras posibles soluciones a esta propuesta, etc.

4.- Metodología de trabajo desarrollada.

Con la finalidad de conseguir los objetivos propuestos, se ha aplicado una metodología basada, por un lado, en la asistencia a una serie de reuniones organizadas por los profesores del IES "Valle del Jerte", encargados de realizar la aplicación de teleasistencia, en las cuales se explicaron las bases de la misma, el planteamiento seguido para su desarrollo, las propuestas de proyecto de fin de ciclo surgidas, el funcionamiento del servidor creado y la manera de desplegarlo, entre otras cuestiones de interés y, por otro lado, en la lectura y posterior consulta de la información proporcionada desde el departamento de *Atención Sociosanitaria* del IES "San Martín" de Talayuela, adjunta en el apartado de documentación del repositorio remoto del proyecto creado por el IES "Valle del Jerte" en *GitHub*.

Tras dichas reuniones iniciales, se procedió a la creación de una cuenta propia en *GitHub* para poder ejecutar un *fork* del proyecto, es decir, hacer una copia del repositorio, de tal manera que las modificaciones derivadas del desarrollo del actual cliente web, elaborado con *Angular*, no afectasen al progreso del repositorio principal o de otros *forks*.

Seguidamente, se clonó el proyecto en local para poder instalarlo y realizar los cambios oportunos. La instalación se llevó a cabo siguiendo los pasos detallados en el repositorio central, raíz del resto de proyectos desarrollados a partir del mismo y, finalmente, se empleó el comando necesario para arrancarlo.

Una vez desempeñados estos requisitos previos, mencionados en los párrafos anteriores, se inicializó el proyecto de *Angular*, con la versión 12.1.1 de dicho *framework*, empleando para ello el IDE *PhpStorm*. Los test de funcionamiento se hicieron mediante el navegador web *Google Chrome* y las peticiones *API Rest* al servidor se probaron a través de *Postman*.

Con el propósito de comprender el funcionamiento del cliente web, su gestión y el progreso seguido durante su producción, se decidió emplear el *software* de control de versiones *Git* y la copia del repositorio remoto alojado en *GitHub* para documentar los cambios y la evolución seguida por medio de *commits* subidos a dicha copia del repositorio y, también, a fin de tener una documentación más exhaustiva, se elaboró la presente memoria.

De acuerdo con esto, la memoria del proyecto se ha estructurado siguiendo los siguientes apartados:

- Una primera parte a modo de preámbulo, en la que se ha mostrado una descripción general del proyecto. Después, se han presentado las motivaciones que han llevado al desarrollo de este trabajo y los objetivos que se persiguen. Finalmente, el apartado ha concluido con la metodología de trabajo desarrollada.

- En la segunda parte, se recoge el contenido propiamente dicho. Este consiste en una descripción técnica y, a su vez, está dividido en una serie de epígrafes que recogen desde una introducción, que sitúa el contexto en el que surgió la idea del proyecto y la base del mismo, hasta los diferentes detalles técnicos del *hardware*, *software*, plataformas u otros elementos empleados, con una explicación suficientemente detallada de su uso complementada con capturas de pantalla. Además, se incluye un epígrafe que contiene un análisis de otras posibles opciones técnicas que se podrían haber utilizado, en el cual se justifica el motivo de haber optado por la solución elegida frente a estas otras opciones.

- Con respecto a la tercera parte, en ella se indican todos los medios, tanto de *hardware* como de *software*, utilizados en el desarrollo e implantación del proyecto y que han sido descritos de forma más extensa en la segunda parte de la memoria.

- En último lugar, se exponen las conclusiones a las que se ha llegado tras desarrollar el proyecto acordado, con la finalidad de determinar el cumplimiento de los objetivos marcados inicialmente.

5.- Descripción técnica.

En este apartado, se recoge toda la información relativa al desarrollo del trabajo realizado, lo cual incluye el escenario planteado para su aplicación, los

antecedentes y precedentes del mismo, el paso a paso de dicho proceso de desarrollo, desde el punto de vista del *hardware*, *software*, plataformas, etc., empleados y, otras soluciones posibles para el despliegue del cliente web creado.

En el epígrafe que sucede, encontramos una breve introducción que sitúa el contexto en el que fue pensado el proyecto que nos ocupa, así como también sus antecedentes y precedentes.

5.1.- Introducción al proyecto de teleasistencia.

Antes de comenzar con la descripción de los detalles técnicos del proyecto, se considera necesario explicar brevemente en qué consiste la teleasistencia para poder construirnos una imagen mental de lo que se pretende conseguir con el conjunto de proyectos de teleasistencia, entre los que se engloba el que trata esta documentación. Todos ellos compondrán la aplicación web final que se aspira a crear.

Podemos definir la teleasistencia domiciliaria como un servicio que permite a las personas mayores y/o personas discapacitadas, entre otros colectivos vulnerables, contar con ayuda de personal preparado para dar respuesta a sus necesidades durante 24 horas al día, todos los días del año. La comunicación entre los usuarios y el personal de los centros de atención a personas dependientes se realiza mediante el uso de una línea de teléfono y una

infraestructura de comunicaciones e informática especifica, ubicadas tanto en el domicilio de estos usuarios como en el centro de atención.

Los teleoperadores del centro de atención cuentan con una serie de recursos humanos y materiales, propios del usuario o existentes en la comunidad, con los que dar solución a las necesidades del paciente.

Así mismo, este servicio se complementa con una agenda de paciente, utilizada para recordarle al mismo la realización de una tarea en un momento determinado, de forma esporádica o con cierta periodicidad. Entre estas tareas se encuentran la toma de medicamentos, la realización de una gestión, las citas médicas programadas, etc.

Este servicio brinda a los usuarios la posibilidad de saber que tienen a su disposición a un grupo de personas cualificadas con las que pueden contactar en caso de emergencia.

Por otro lado, desde el centro de atención se comunican de forma habitual con los usuarios del servicio para llevar un seguimiento permanente, mantener actualizados sus datos e intervenir si las circunstancias lo aconsejan.

La teleasistencia domiciliaria persigue mantener a las personas vulnerables en su medio corriente de vida, evitando así los costes de personal, sociales y económicos que conllevaría este cambio. Además, facilita el contacto de los usuarios con su entorno social y familiar, asegura la movilización de recursos inmediata para solucionar emergencias personales, sociales o médicas que

puedan sufrir los pacientes y contribuye activamente a evitar ingresos innecesarios en centros residenciales.

Una vez que ha quedado claro qué es la teleasistencia, podemos hablar del caso que nos ocupa. Este proyecto se fundamenta en el desarrollo web de un servicio que permita simular las tareas desempeñadas por el personal de los centros de teleasistencia, de modo que, los alumnos del FP de *Atención Sociosanitaria* del IES “San Martín” puedan realizar prácticas en el aula como si de un entorno real se tratase.

Este servicio de teleasistencia se implementará a través de una aplicación web que emulará la realidad del día a día de las alertas recibidas en un centro de atención, contará con una agenda que posibilite la gestión de las llamadas de recordatorio que se deben realizar a los usuarios y, permitirá el almacenamiento de información de los usuarios y los recursos disponibles correspondientes a cada usuario que pueden movilizarse en caso necesario para solucionar una determinada alerta.

Por tanto, el ámbito de implementación de la aplicación que se despliegue será el IES “San Martín”, cuyo departamento de *Atención Sociosanitaria* fueron los principales impulsores de la idea inicial sobre la que se construyeron las bases para la plataforma de prácticas de teleasistencia.

Teniendo en cuenta lo mencionado en el párrafo anterior, el contexto en el cual se ha planteado la aplicación es para ser utilizada como una herramienta que complemente la formación recibida en el ciclo formativo de *APSD*. De esto

podemos deducir que, los clientes finales de la misma serán profesores y alumnos de dicho ciclo, con lo que la aplicación se ha pensado para ser intuitiva y accesible para este tipo de usuarios.

A fin de conocer con mayor profundidad el contexto en el que fue pensado este proyecto, a continuación, se explicarán los antecedentes y precedentes del mismo.

5.1.1.- Antecedentes y precedentes del proyecto de teleasistencia.

Como ya se ha mencionado, el concepto sobre el que se apoya el proyecto general de teleasistencia surgió como una forma de intentar dar respuesta a una necesidad especifica que tenían en el IES "San Martín".

Por ello, desde el departamento de *Atención Sociosanitaria* de este centro, pensaron que la solución al problema podría ser la creación de un *software* informático que cubriese su necesidad de una herramienta con la que formar a sus alumnos en las tareas de teleasistencia de una manera práctica y realista.

En consecuencia, desde este instituto buscaron la forma de desarrollar este *software* y acabaron dando con el IES "Valle del Jerte", donde se imparten una serie de ciclos formativos de la familia de informática y comunicaciones, con el que se pusieron en contacto para trasladarles su propuesta.

Desde el IES "Valle del Jerte" se mostraron abiertos a valorar dicha propuesta y pensaron que podría ser una buena oportunidad para que sus alumnos de los ciclos superiores de *Desarrollo de Aplicaciones Web* (*DAW*), *Desarrollo de*

Aplicaciones Multiplataforma (*DAM*) y *Administración de Sistemas Informáticos en Red* (*ASIR*) desarrollaran este *software* a través de distintos proyectos de fin de ciclo hasta obtener el servicio de teleasistencia solicitado.

Se comunicó a los alumnos del ciclo de DAW esta oferta para conocer si existía interés por su parte y se mantuvo la comunicación con el IES San Martín para tratar de conocer los pormenores de la aplicación web requerida.

Tras varias reuniones telemáticas, en las que se remitió documentación en la cual se exponía en profundidad las necesidades a cubrir y los detalles necesarios con los que debía contar el *software* informático. Con esta información, desde el IES "Valle del Jerte" comenzaron a trabajar para poder realizar un prototipo inicial.

La documentación mencionada se encuentra a disposición en el apartado documentación del repositorio remoto de *GitHub* del proyecto de teleasistencia creado por el IES "Valle del Jerte" para documentar y almacenar una copia de seguridad del proyecto.

Las tareas a abordar para poder poner en marcha dicho prototipo inicial, sobre el que continuar trabajando en un futuro para construir la aplicación final, fueron las siguientes:

1. Definir los roles del sistema: usuario, profesor, alumno, ...
2. Modelar los datos.
3. Crear un *mock-up* del servidor web (*JSON-Server*).

4. Crear un *mock-up* del cliente y pensar en las peticiones que iba a hacer el mismo. Además, probar dichas peticiones con *Postman* sobre el servidor.

5. Crear un cliente mínimamente funcional para mostrar la interfaz que esperaban y funcionando con el servidor de pruebas.

Ahora que se conocen estas tareas, se procederá a concretar cómo se fueron resolviendo cada una de ellas desde un punto de vista técnico.

Para modelar los datos, la aplicación web cuenta con una base de datos cuyo modelo E/R ha sido diseñado con la herramienta *draw.io*.

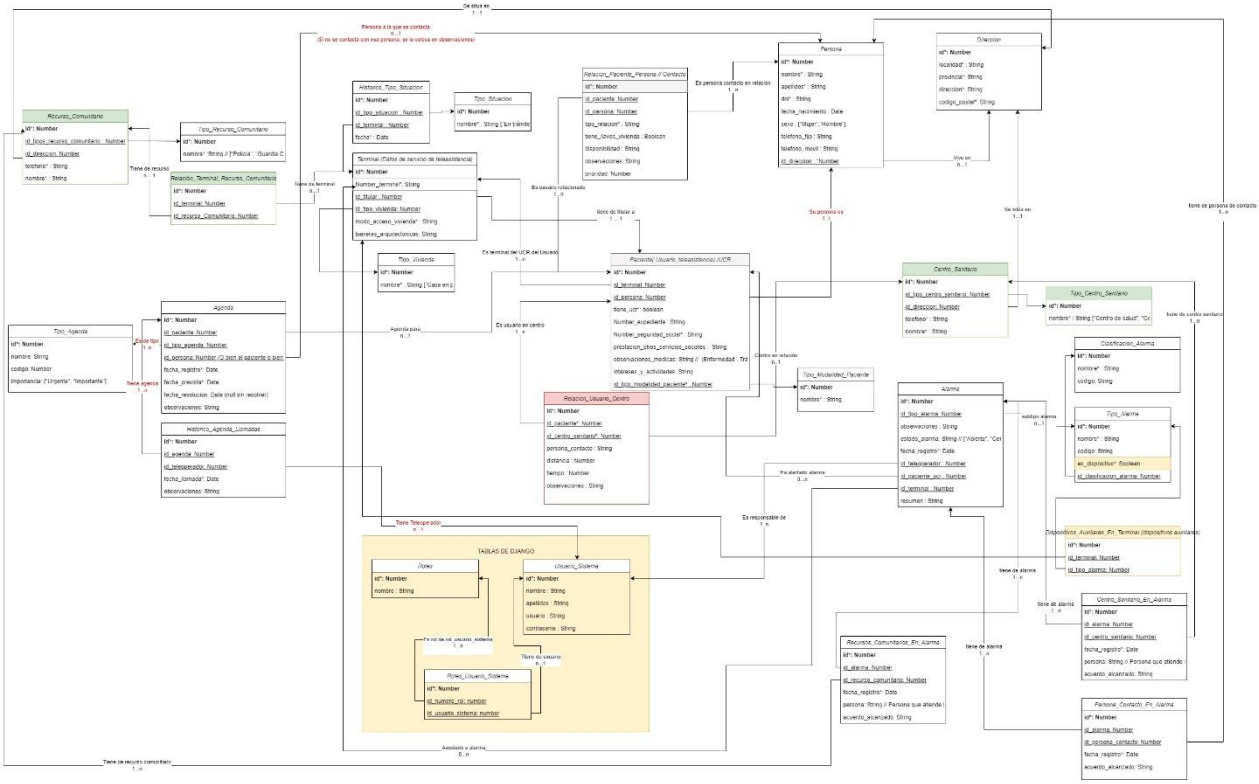


Ilustración 1. Modelo E/R de la base de datos del proyecto de teleasistencia.

Fuente: Documentación del repositorio central del proyecto en GitHub.

Dicha herramienta de diagramación gratuita proporciona una interfaz intuitiva con funcionalidad de arrastrar y soltar, plantillas de diagramas personalizables y una extensa biblioteca de formas que permiten al usuario crear, editar y compartir diagramas dentro de un navegador web. La herramienta en línea funciona con *G Suite/Google Drive* y *Dropbox*, aunque los usuarios también pueden trabajar en los diagramas sin conexión y guardar los mismos localmente usando la aplicación de escritorio de *draw.io* para *macOS*, *Windows* y *Linux*.

El modelo E/R se basa en una serie de mecanismos que permiten definir la realidad a través de un conjunto de representaciones gráficas y lingüísticas. Entre dichos mecanismos destacan los conceptos de entidad, relación y atributo.

Podemos definir una entidad como un objeto que almacena información de una cosa, persona, suceso, etc. del mundo real. Dentro del modelo E/R se representan gráficamente en forma de rectángulos, con su nombre y atributos en su interior. Asimismo, las entidades pueden ser de dos tipos diferentes:

- Entidades débiles. Su existencia depende de otras entidades, no pueden ser identificadas únicamente por sus propios atributos.
- Entidades fuertes. No dependen de ninguna otra y, en caso de ser requerido, pueden prestar alguno de sus atributos, a modo de clave foránea, a una entidad débil para que, esta última, se pueda identificar.

Las distintas entidades pueden estar asociadas, lo cual se denomina relación y cada una de ellas tiene un nombre que describe su función. Las entidades involucradas en una determinada relación, tienen un grado de participación en la

misma. Cuando el número de entidades implicadas en una relación es dos se denomina binaria y, si son tres entidades participantes, entonces es una relación ternaria. Además, existen las relaciones recursivas dónde la propia entidad participa más de una vez en la relación con distintos papeles.

Las relaciones entre entidades se llevan a cabo mediante un atributo que recibe el nombre de clave foránea. El atributo que actúa como clave foránea, se vincula con el identificador o clave primaria de otra entidad de la base de datos. La entidad secundaria contiene la clave foránea y la entidad principal la clave primaria.

El número de entidades con las que está relacionada una entidad dada, se denomina cardinalidad y puede ser de varios grados:

- Uno a uno. Una entidad de A se relaciona únicamente con una entidad en B y viceversa.
- Uno a muchos. Una entidad en A se relaciona con cero o muchas entidades en B. Pero una entidad en B se relaciona con una única entidad en A.
- Muchos a uno. Una entidad en A se relaciona exclusivamente con una entidad en B. Pero una entidad en B se puede relacionar con 0 o muchas entidades en A.
- Muchos a muchos. Una entidad en A se puede relacionar con 0 o muchas entidades en B y viceversa.

Por otro lado, cada entidad tiene unos atributos que denotan las propiedades de dicha entidad y de sus relaciones con otras entidades. El conjunto de valores asociados a un atributo se denomina dominio y define todos los datos posibles que puede tomar dicho atributo.

Los atributos pueden ser simples, no es posible dividirlos en partes más pequeñas con significado propio, y compuestos por varios componentes, cada uno con un significado.

Además, podemos clasificarlos también en atributos monovaluados, aquellos que tienen solo un valor por cada entidad o relación a la que pertenecen, y multivaluados, con varios valores para cada ocurrencia de la entidad o relación de la que son parte.

Por último, tenemos los atributos derivados cuyo valor se obtiene a partir del valor de uno o varios atributos, los cuales pueden no pertenecer a la misma entidad o relación.

No obstante, cabe destacar que cada entidad tiene un identificador o clave primaria, es decir, un atributo o conjunto de atributos que definen de manera única a la misma. Dicho identificador no puede tener el mismo valor para dos elementos de una misma entidad.

Para entender mejor lo que es una entidad o tabla dentro de una base de datos, en la imagen que encontramos a continuación, podemos ver la representación de la entidad *Recurso_Comunitario* en el modelo E/R diseñado con *draw.io*.

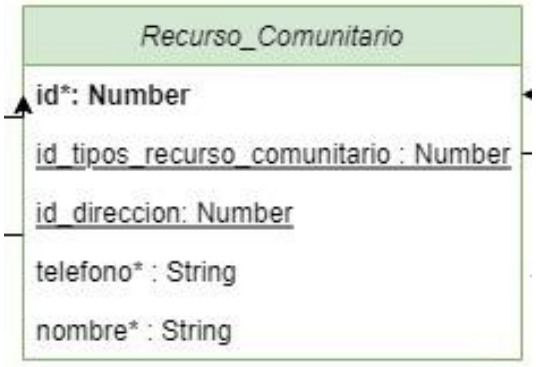


Ilustración 2. Ejemplo entidad Recurso_Comunitario del modelo E/R de la base de datos.

Fuente: Documentación del repositorio central del proyecto en GitHub.

Como se puede observar en la ilustración, la entidad *Recurso_Comunitario* tiene una serie de atributos, entre los que sobresalen el *id*, clave primaria de la entidad, y los atributos *id_tipos_recurso_comunitario* e *id_direccion*, que tienen la función de clave foránea y la relacionan con las entidades Tipo_Recurso_Comunitario y Direccion.

El modelo E/R, explicado en los párrafos anteriores, se ha implementado a través del *framework* de desarrollo web *back-end* de código abierto, *Django*, escrito en lenguaje de programación *Python*.

En *Django* no hace falta trabajar con *SQL*, siempre se trabaja directamente con los modelos. Los modelos son objetos de *Python* que definen la estructura de los datos de una aplicación y proporcionan mecanismos para gestionar (añadir, modificar y borrar) y consultar registros en la base de datos.

Un modelo en *Django* es el equivalente a cada entidad o tabla representada en la base de datos usada para guardar la información de la aplicación de teleasistencia.

La base de datos de datos empleada está almacenada utilizando *SQLite*, una biblioteca de *C*, con soporte para *Python*, que provee una base de datos basada en disco que no requiere un proceso de servidor separado y permite acceder a los datos usando una variación no estándar del lenguaje de consulta *SQL*.

Para programar toda la parte de la base de datos y las consultas sobre la misma realizadas desde el servidor, se ha empleado el entorno de desarrollo integrado *PyCharm*. Este IDE es uno de los más populares a la hora de trabajar con *Python*, ya que, cuenta con un potente interprete en el editor de código que nos ayuda a saber o conocer los posibles errores del código en tiempo real.

Además, cuenta con una versión para las distribuciones *Gnu/Linux*, lo que hace que sea más sencilla su utilización y la creación de programas con este lenguaje de programación. *PyCharm* también admite otros lenguajes de programación como *JavaScript*, *Kotlin* o *CoffeeScript* y otras herramientas como *HTML* o *CSS*.

Una vez montados tanto la base de datos como el servidor, se añadieron datos a la misma empleando el panel de administración que proporciona *Django*.

El siguiente problema a abordar fue la falta de mecanismos de autentificación y de petición de datos, es decir, la falta de una *API Rest*. La solución a esto fue

instalar *Django Rest Framework* y configurarlo con *Oauth* para realizar una autentificación mediante el empleo de *tokens*.

Se pueden encontrar más datos acerca de la resolución de este conflicto en el apartado de *issues* del repositorio remoto del proyecto en *GitHub*, en concreto en el *issue* llamado *Inicializar Django Rest Framework*.

Resueltas las dificultades mencionadas en los dos párrafos anteriores, se procedió a crear el *mock-up* del cliente y pensar en las peticiones que iba a hacer el mismo. A través de *Postman* se creó una batería con las peticiones iniciales realizadas al cliente.

Postman es una herramienta que principalmente nos permite crear peticiones sobre *APIs* de una forma muy sencilla y poder, de esta manera, probar dichas *APIs*. Todo ello basado en una plataforma en la nube.

Los detalles relativos a este proceso se encuentran recogidos en el *issue Moqups de cliente y API Rest* del repositorio del proyecto en *GitHub*.

La ejecución de todas estas tareas sentó las bases de la aplicación de teleasistencia y, a partir de ello, se plantearon dos ideas de proyectos de fin de ciclo para que los alumnos pudiesen contribuir al crecimiento de la misma.

La primera idea de *PFC* consiste en trabajar sobre el *back-end* de la aplicación, realizando el resto de las peticiones en *Postman* y ofreciendo los datos desde la *API Rest* utilizando *Django*. Así como también se sustenta en investigar si existe la posibilidad de automatizar pruebas con *Postman* para comprobar si todas las

peticiones funcionan correctamente y, finalmente, en definir los roles en la *API Rest* y liminar las acciones de los usuarios según su rol.

La otra idea de *PFC* se basa en la creación de un *front-end* básico para la aplicación de teleasistencia. El cliente web se debe realizar empleando el *framework Angular* con su utilidad de *routing* y, además, se deben establecer los estilos, el *login* y los formularios sobre los que se construirá en el futuro la aplicación con más detalles. Esta propuesta de proyecto es en la cual se sustenta la estructura y la descripción técnica de esta memoria.

Ambas propuestas fueron presentadas a los alumnos en una reunión presencial en el centro en la que, también, se puntualizaron los detalles sobre el proyecto y su estado de desarrollo.

Se concedió a dichos alumnos un tiempo para aceptar o rechazar las propuestas y, cuando estos dieron una respuesta afirmativa, se fijó otra reunión presencial para explicarles como instalar y arrancar el proyecto en sus propios equipos y cómo comenzar a trabajar en sus respectivos trabajos finales.

El paso a paso seguido en el transcurso de esta labor es el que se encuentra descrito en el apartado “metodología de trabajo desarrollada” de la memoria y la información sobre cada una de las ideas de proyecto se precisa de manera más extensa en los *issues* del repositorio remoto.

La finalidad de estos dos trabajos de fin de ciclo es poder contar con una primera versión de la aplicación web de teleasistencia sobre la que proseguirán

trabajando otros alumnos en sus futuros *PFC* hasta que la misma esté completamente terminada y lista para implementarse en el entorno final para el que ha sido pensada.

Los precedentes de futuros trabajos que otros alumnos utilizarán para construir sus proyectos, se especifican en el *issue* referente a *Ideas para proyectos de alumnos* del repositorio remoto del proyecto de teleasistencia alojado en *GitHub*.

Teniendo situados el contexto en el que surgió el proyecto, sus antecedes y sus precedentes, en el epígrafe que sucede se describirán los detalles técnicos relativos al desarrollo de la propuesta de *PFC* de inicializar un cliente de *Angular* para desarrollar las interfaces de usuario del proyecto de teleasistencia.

5.2.- Detalles técnicos del desarrollo del proyecto.

En este epígrafe se pretende que el lector comprenda el proceso evolutivo seguido desde la instalación del servidor en el equipo local hasta la obtención de un cliente web funcional que cumpla con los requisitos solicitados.

Con la finalidad de que dicho proceso sea comprendido adecuadamente, se ha decidido dividir este epígrafe en una serie de subapartados que recogen el cometido de cada una de las tareas realizadas para conseguir el resultado final buscado.

Asimismo, se contemplan las dificultades y fallos con las que nos hemos ido encontrando y las soluciones dadas a cada uno de ellos.

Por otro lado, también se indican los medios de tipo *hardware*, *software* y las plataformas utilizadas en las distintas tareas que se han abordado.

En el subapartado que se encuentra a continuación, se explican los requisitos iniciales necesarios para instalar el proyecto en el equipo local antes de comenzar a trabajar en el mismo.

5.2.1.- Pasos para contribuir, instalar y arrancar el proyecto.

Antes de poder crear el proyecto de *Angular* para inicializar el cliente web, fueron necesarios una serie de pasos previos.

El primer paso a realizar fue crearse una cuenta en *GitHub* y crear una copia del repositorio del proyecto de teleasistencia creado por el IES “Valle del Jerte” a nuestro propio repositorio remoto alojado en la cuenta creada.

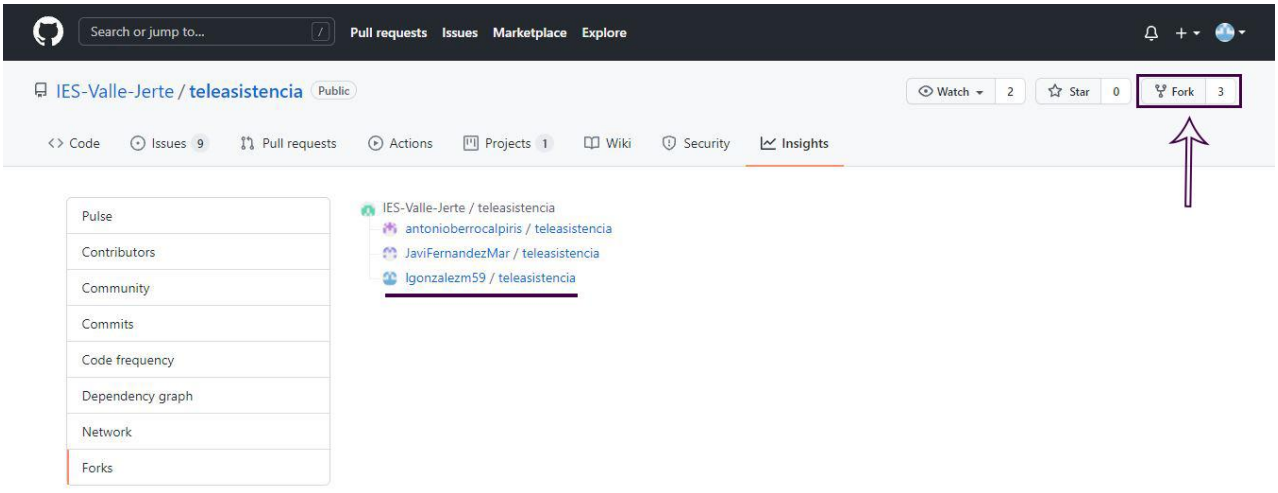


Ilustración 3. Realizar un fork del repositorio de teleasistencia.

Fuente: Captura de pantalla propia.

Seguidamente para clonar el proyecto en local de forma más sencilla, se instaló *Github Desktop* y se trabajó desde su propia interfaz gráfica.

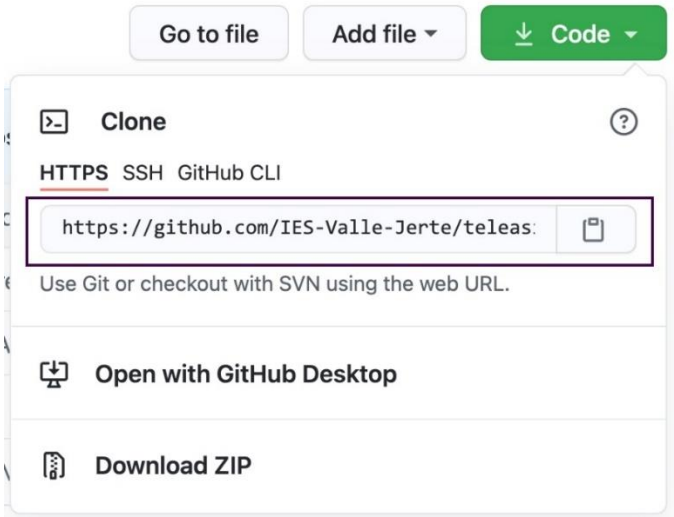


Ilustración 4. Primer paso para clonar el proyecto en nuestro repositorio.

Fuente: Captura de pantalla propia.

Una vez copiado el enlace que se muestra en la imagen anterior, el resto de pasos a seguir para terminar de clonar el proyecto fueron abrir *Github Desktop* y hacer lo que se muestra en las ilustraciones siguientes:

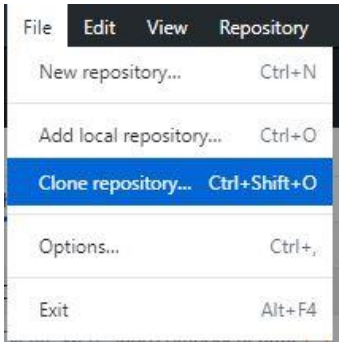


Ilustración 5. Segundo paso para clonar el proyecto en nuestro repositorio.

Fuente: Captura de pantalla propia.

Seleccionamos la opción de clonar el repositorio usando una URL, pegamos el enlace copiado previamente y le dimos clic a clone.

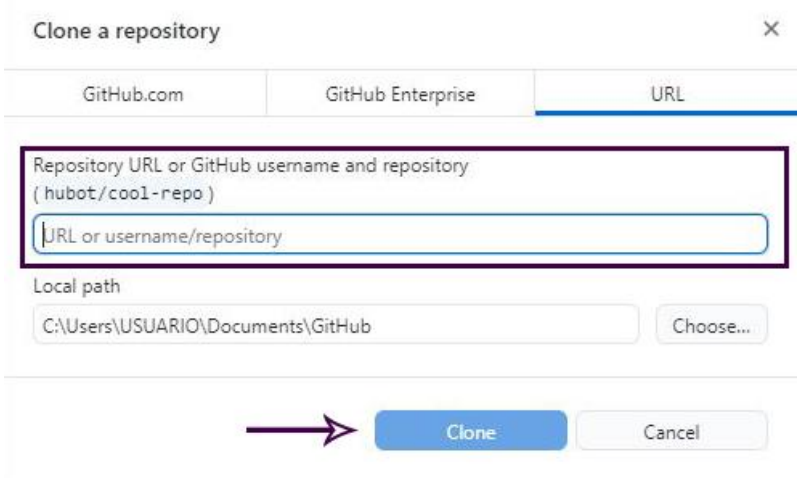


Ilustración 6. Paso final para clonar el proyecto de teleasistencia.

Fuente: Captura de pantalla propia.

El segundo paso a seguir fue realizar la instalación del proyecto en nuestro equipo local. Para ello, comenzamos instalando *Python* y descargando e instalando el entorno de desarrollo *PyCharm*.

Además, accedimos a la ruta dónde se había clonado el proyecto en local, en este caso a la ruta *C:\Users\USUARIO\Documents\GitHub\teleasistencia*, y dentro de la carpeta *Server* abrimos el símbolo del sistema de *Windows* y ejecutamos el comando *virtualenv venviorment* para crear el entorno virtual.

Tras realizar este paso, se nos creó una nueva carpeta dentro de *Server* llamada *venviorment*, accedimos a ella y luego a la carpeta *Scripts*, dentro de esta última carpeta buscamos el archivo *activate* y lo ejecutamos.

Comprobamos que el entorno virtual se había creado correctamente y abrimos el proyecto con *PyCharm* para hacer permanente dicho entorno. Fuimos a *File → Settings... → Project → Python Interpreter* y seleccionamos el intérprete creado.

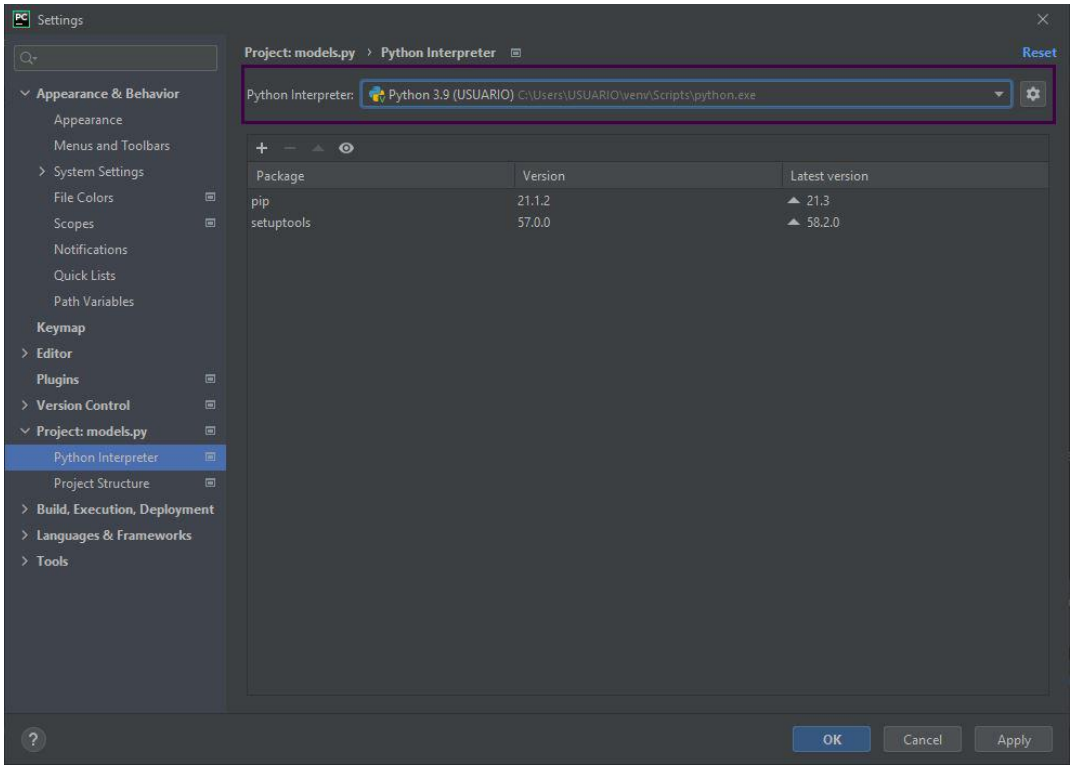


Ilustración 7. Hacer permanente el entorno virtual.

Fuente: Captura de pantalla propia.

Finalmente, abrimos un nuevo símbolo del sistema en la carpeta *Server* del proyecto de teleasistencia y ejecutamos el comando *pip install --upgrade pip*. Después, instalamos los requerimientos usando el comando *pip install -r requerimientos.txt*.

El archivo *requerimientos.txt* contiene las dependencias que se instalan con el comando anterior, necesarias para la ejecución del servidor:

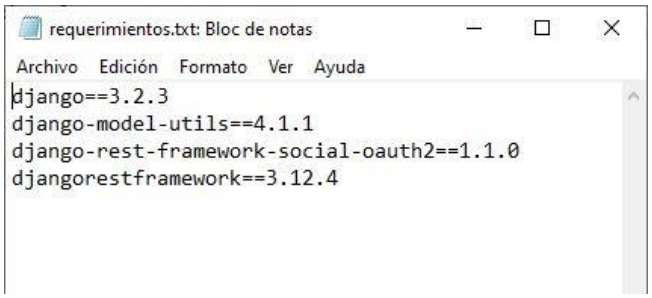


Ilustración 8. Requerimientos del proyecto de teleasistencia.

Fuente: Captura de pantalla propia.

El último paso fue arrancar el servidor del proyecto accediendo a la carpeta *Server\teleasistencia*, abriendo un nuevo símbolo del sistema y ejecutando el comando *python manage.py runserver*.

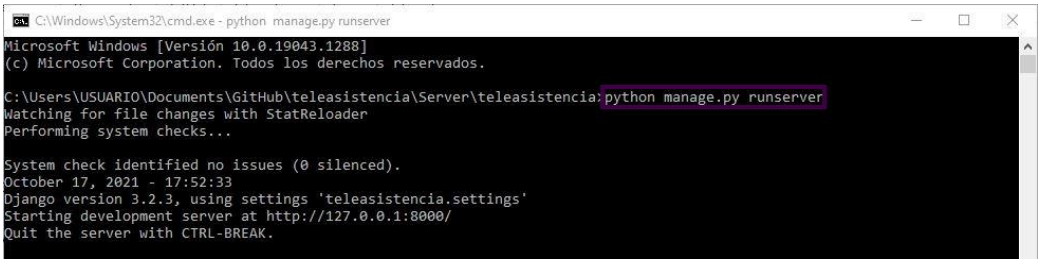


Ilustración 9. Comando para arrancar el proyecto de teleasistencia.

Fuente: Captura de pantalla propia.

Posteriormente a la instalación y el arranque del proyecto de teleasistencia, se pudo proceder con la inicialización del cliente web con *Angular*. El proceso seguido para ello se describe en el subapartado que se muestra a continuación.

5.2.2.- Inicialización del cliente web con *Angular* e instalación de *Git*.

En primer lugar, se instaló el IDE *PhpStorm*, en su última versión en este caso la 2021.1, para facilitar la escritura del código a la hora de crear la interfaz web solicitada.

Se eligió *PhpStorm*, ya que, cuenta con soporte para *Angular* y ayuda para el usuario en cada paso del proceso de desarrollo, desde crear una nueva aplicación *Angular* y trabajar en los componentes hasta depurarla y probarla.

Además, admite el desarrollo, la ejecución y la depuración del código fuente de *TypeScript*, el lenguaje por defecto utilizado por *Angular*.

En segundo lugar, se accedió a la ruta local donde habíamos clonado el proyecto, *C:\Users\USUARIO\Documents\GitHub\teleasistencia*, y se creó una nueva carpeta llamada *Cliente Web* dentro de la cual se alojaría la carpeta del proyecto de *Angular* cuando se inicializase.

Para poder comenzar a trabajar con *Angular*, nos asegurarnos de tener instalado *Node.js* en su versión más actual, la 14.15.1 en el momento en el que se inicializó este proyecto, debido a que vamos a necesitar su gestor de paquetes *npm*. Podemos comprobar si lo tenemos instalado abriendo el símbolo del sistema y ejecutando el comando *node --version*, en el caso de que nos encontremos en un equipo con sistema operativo *Windows*.

Cuando tengamos *Node.js* instalado podremos continuar instalando *Angular CLI* (*Command Line Interface*) usado para crear proyectos de *Angular*, generar código de la aplicación y de las librerías y llevar a cabo muchas de las tareas de desarrollo como pueden ser *testing*, *bundling* y *deployment*.

Empleando el comando *npm install -g @angular/cli* en el símbolo del sistema, instalaremos *Angular CLI* de manera global en nuestro equipo.

Finalmente, nos dirigimos a la ruta de la carpeta *Cliente Web* que habíamos creado para almacenar el proyecto de *Angular*, abrimos el símbolo del sistema dentro de la misma y ejecutamos el comando *ng new proyecto-teleasistencia* para generar dicho proyecto.

El comando *ng new* nos pedirá información sobre las características del proyecto que va a iniciar. En este caso, debemos indicarle que queremos crear el proyecto empleando la funcionalidad de *routing* que incluye *Angular* y, además, le indicamos que el formato de las hojas de estilo será *SCSS*.

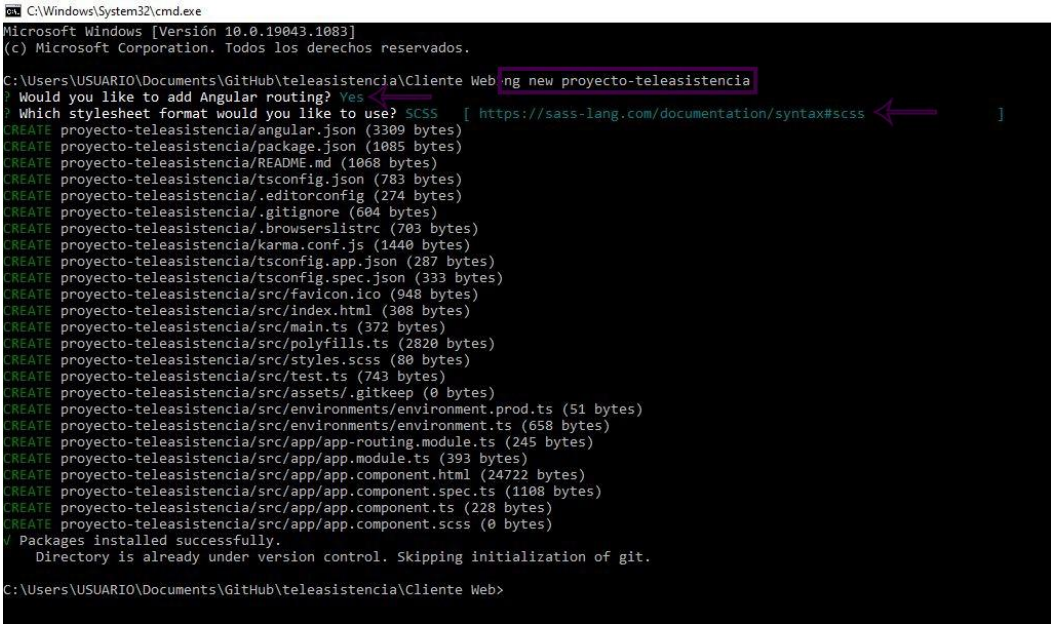


Ilustración 10. Características del cliente de Angular.

Fuente: Captura de pantalla propia.

Angular incluye un servidor, así que podemos construir y servir la aplicación en local. Para ello, abrimos la carpeta proyecto-teleasistencia que hemos generado

en *PhpStorm* y lanzamos el servidor empleado el comando *ng serve --open* en el propio terminal del IDE.

Esto nos abrirá directamente en nuestro navegador predeterminado el cliente web que hemos creado en la ruta *http://localhost:4200/*.

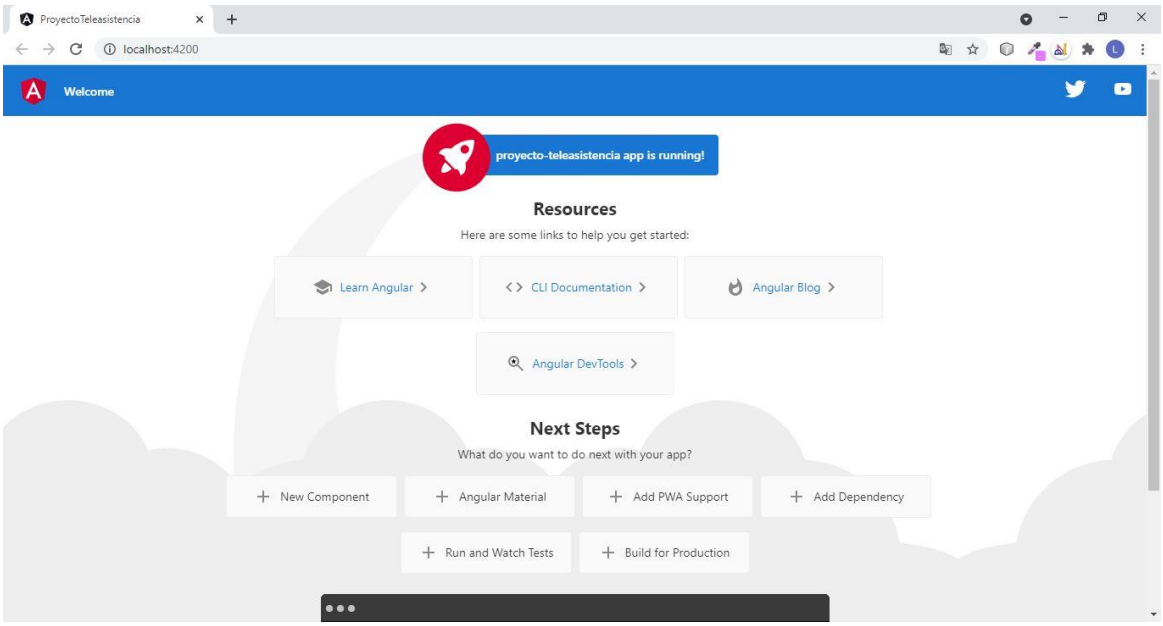


Ilustración 11. Cliente web por defecto generado con Angular.

Fuente: Captura de pantalla propia.

Con estos pasos ya tenemos creado el proyecto de *Angular* sobre el que trabajar para construir el cliente web inicial, pero antes de seguir avanzado en dicha construcción sería interesante instalar el *software* de control de versiones *Git* para poder realizar los distintos *commits* a través de los cuales ir documentando el proceso seguido. También deberíamos realizar las configuraciones necesarias para poder hacer el *push* correspondiente a cada uno de los *commits* al repositorio propio que hemos copiado del original creado por el IES “Valle del Jerte”.

La manera más sencilla de realizar todas estas configuraciones es empleando *PhpStorm*. Vamos a *File* → *Settings…* → *Plugins* e instalamos los plugins para *Git* y *GitHub*.

Ahora si accedemos a la pestaña de *Git* dentro de *PhpStorm* veremos todos los *commits* del proyecto realizados con *Git* y subidos a *GitHub*. Podemos ver estos *commits* debido a que previamente hemos clonado el repositorio remoto en nuestro equipo local.

Accedemos a la pestaña *Commit*, seleccionamos los archivos que queremos incluir en dicho *commit*, añadimos el mensaje del mismo y seleccionamos la opción *Commit and Push*.

Al ser la primera vez que queremos hacer el *push* de un *commit*, se nos solicitarán tanto el usuario como la contraseña de *GitHub*, además de una serie de permisos para acceder a *GitHub* desde *PhpStorm*, los cuales debemos aceptar.

Si accedemos a nuestro repositorio de *GitHub* podremos ver como el *commit* se ha subido y, al haber realizado un *fork* del repositorio principal creado por el IES "Valle del Jerte", nuestros cambios se ven reflejados en el *fork*, no en el original.

Una vez realicemos todas estas configuraciones para poder emplear *Git* y *GitHub*, vamos a poder trabajar creando y subiendo *commits* al repositorio remoto de una forma muy sencilla, ya que, dichas configuraciones solo deben efectuarse una vez.

5.2.3.- Creación de interfaces para definir la estructura de las entidades empleadas por los formularios.

En el apartado anterior se ha explicado cómo se inicializó el proyecto y las configuraciones que fueron necesarias para poder hacer *commits* del paso a paso y subirlos al repositorio remoto, es decir, los dos primeros procesos para poder comenzar a construir el cliente web. Sin embargo, aún quedaba mucho trabajo por delante y la mejor manera de abordarlo fue definiendo el esqueleto general que tendría el proyecto de *Angular*.

Para trabajar de forma más fácil se estructuró el proyecto en interfaces, clases, componentes y servicios.

En este apartado nos centraremos en explicar qué es una interfaz, para que las hemos empleado y cómo se fueron creando las distintas interfaces utilizadas en el proyecto que nos ocupa.

La definición técnica de interfaz contempla a la misma como un mecanismo usado en la programación orientada a objetos para suplir carencias derivadas de la herencia múltiple. En ocasiones necesitamos definir una clase que extienda de varias clases a la vez, algo que en muchos de los lenguajes de programación orientados a objetos no es posible y que las interfaces nos permiten realizar.

Una clase puede extender de otra, heredando propiedades y métodos y declarando que implementa varias interfaces.

La diferencia de las clases extendidas frente a las interfaces, es que, las interfaces no tienen una implementación de sus métodos, por lo que la clase que implementa una interfaz escribirá el código para todos los métodos que contiene.

Por esta razón, se dice que las interfaces son un contrato, que especifica los elementos que una clase debe contener para implementar una interfaz.

Este es un concepto de interfaz genérico, ya que, cada lenguaje de programación tiene sus propias particularidades como es el caso de *TypeScript* que permite definir tanto propiedades como métodos en una interfaz.

Asimismo, **las interfaces** también **nos permiten** definir un nuevo tipo, es decir, podemos **crear elementos** o variables **del tipo de una interfaz**. De esta forma, **dichos elementos deben cumplir con el contrato marcado por la interfaz** y contener todas las propiedades con sus respectivos tipos de datos.

La utilidad de las interfaces descrita en el párrafo anterior **es la razón del empleo de las mismas en este proyecto**. Al trabajar con un cliente web que va a realizar una serie de peticiones contra un servidor que lleva por detrás una base de datos, **resulta interesante crear una interfaz por cada una de las entidades que vamos a crear o modificar mediante los formularios** solicitados en la propuesta de proyecto.

Dichas interfaces definirán la **estructura de** los **datos que espera recibir o devuelve el servidor** ante una petición proveniente desde el cliente.

Las entidades a crear y/o modificar mediante formularios son las siguientes:

- Users
- Tipo_Alarma
- Clasificacion_Alarma
- Tipo_Centro_Sanitario
- Tipo_Modidalidad_Paciente
- Centro_Sanitario
- Tipo_Recurso_Comunitario
- Recurso_Comunitario
- Persona
- Dirección. Esta última entidad no había sido pedida en el planteamiento del *PFC*, pero se consideró apropiado añadir la interfaz, la clase, los componentes y los servicios necesarios para crear los formularios de creación y modificación de la misma, ya que, otras entidades como Centro_Sanitario contenían un atributo del tipo de esta entidad que esperaba recibir como valor un objeto dirección.

Lo primero antes de establecer las interfaces para cada una de las entidades fue crear una carpeta de nombre *interfaces* dentro de la carpeta *src/app* para tener un proyecto de *Angular* organizado siguiendo el planteamiento explicado con anterioridad en este mismo apartado.

Abrimos un terminal dentro de la carpeta *interfaces*, desde el propio *PhpStorm*, y ejecutamos el comando *ng g interface* más el nombre que le queremos dar a la interfaz que vamos a crear.

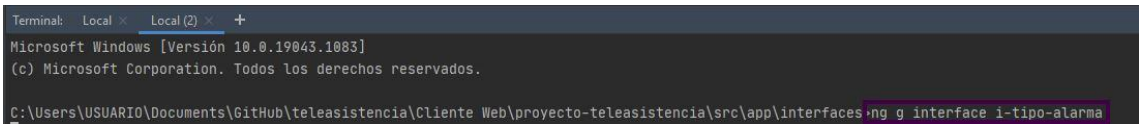


Ilustración 12. Ejemplo comando de creación de una interfaz en Angular.

Fuente: Captura de pantalla propia.

De esta forma *Angular* nos creará la estructura básica de la interfaz dentro de la carpeta que hemos indicado y nosotros solo tendremos que ocuparnos de definir los atributos que queremos que tenga la misma.

Para saber qué atributos definir en cada entidad empleamos la documentación de la *API Rest* realizada con *Postman*, dado que, en la misma podemos encontrar una batería de peticiones iniciales a realizar sobre el servidor desde el cliente web para crear, modificar o eliminar las entidades definidas en la base de datos en las cuales se indican qué datos espera recibir o devuelve el servidor.

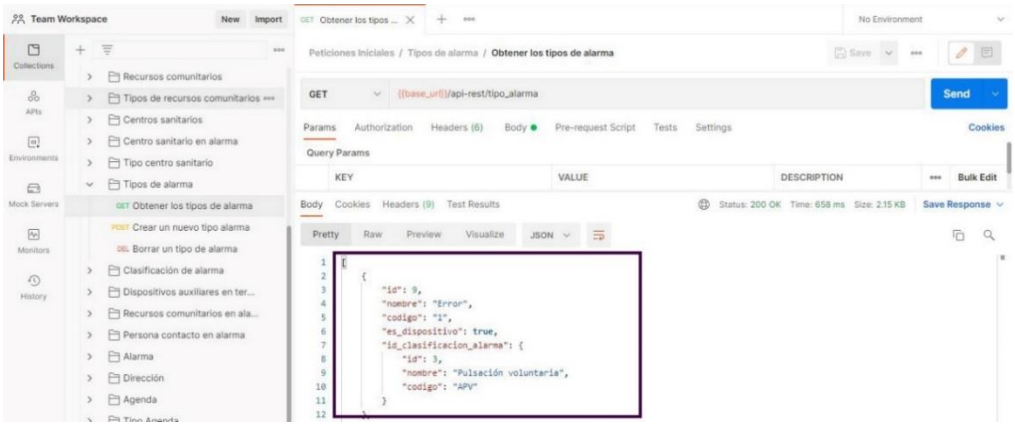


Ilustración 13. Datos devueltos de una petición GET a la entidad Tipo_Alarma, visto desde la documentación en Postman.

Fuente: Captura de pantalla propia.

Podemos acceder a la información sobre las peticiones en *Postman* utilizando el enlace proporcionado en el *issue* del repositorio remoto de *GitHub*, creado por el

IES “Valle del Jerte”, llamado *Moqups de cliente y API Rest* y, solicitando los permisos de acceso correspondientes al grupo de trabajo.

Mirando la documentación de *Postman* referente a las peticiones de tipo *GET* encontramos los datos que devolverá el servidor al consultar las entidades guardadas en la base de datos, estos mismos datos serán los que espera recibir dicho servidor cuando se modifique o se cree una entidad. Por lo tanto, debemos definir los atributos de la interfaz siguiendo esa estructura.

Ilustración 14. Atributos de la interfaz ITipoAlarma.

Fuente: Captura de pantalla propia.

Debemos efectuar este procedimiento hasta tener una interfaz por cada una de las entidades, con los atributos y el tipo de cada uno de ellos que espera el servidor.

5.2.4.- Creación de los componentes y servicios necesarios para realizar los formularios de cada entidad.

Este apartado se centrará en definir que son tanto un componente como un servicio dentro de un proyecto desarrollado con *Angular*, la utilidad de ambos

dentro del cliente web de la aplicación de teleasistencia y la manera en que se deben crear.

En *Angular*, un componente puede entenderse como un controlador de una vista. Hay un componente principal para la vista, y también, puede haber subcomponentes anidados que controlan partes específicas de la página.

Los componentes contienen una serie de metadatos entre los que se encuentran: el *selector*, la *templateUrl* y las *styleUrls*. Los cuales se explican en los siguientes párrafos.

Cada componente se encarga de manipular e interactuar con un *template*. Los *templates*, en este caso, serían las vistas de las que hemos hablado en el párrafo anterior. Un componente contiene código *HTML* que será insertado dentro del selector del componente cuando se cargue. La comunicación entre el componente y el *template* se lleva a cabo mediante *interpolation* y *data-binding*. La *template* que corresponde al componente se asigna mediante la etiqueta *templateUrl*.

Como hemos mencionado, los componentes tienen un atributo de metadatos denominado *selector*. Este selector se coloca en el lugar en el que queremos cargar el *template* en nuestra aplicación. Este selector se puede emplear múltiples veces, instanciando varios componentes. A esto se le llama directiva de un componente.

Asimismo, *Angular* permite asignar una o más hojas de estilos a un componente, utilizando para ello el atributo *styleUrls*. Los estilos serán aplicados únicamente a la *template* asociada a dicho componente.

Ahora que sabemos lo que es un componente y su uso, vamos a centrarnos en su utilidad para precisar la organización de los formularios de creación y modificación de las distintas entidades pedidas.

Se decidió crear cuatro componentes por cada entidad del servidor. El primero de ellos, llamado *lista* más el nombre de la entidad correspondiente, se encargará de manejar la petición que nos devuelve todos los elementos almacenados en una entidad.

El segundo, llamado *item* más el nombre de la entidad, maneja cada uno de los elementos almacenados en la entidad.

El tercer componente, de nombre *detalles* más el nombre de la entidad, administra la petición de modificación de una entidad realizada desde el formulario correspondiente definido en la vista del componente.

El último componente, *nuevo* más el nombre de la entidad, maneja la petición de creación de un nuevo elemento de una entidad realizada desde el formulario correspondiente definido en la vista del componente.

Todos estos componentes para cada entidad los creamos dentro de una nueva carpeta llamada *componentes* dentro de *src/app* ejecutando el comando *ng g component* más el nombre de cada componente.

Una idea para próximos proyectos, sería crear una subcarpeta por cada entidad y dentro de ella, meter los cuatros componentes relativos a la misma para conseguir un mayor orden en la estructura del proyecto.

Con el comando mencionado anteriormente, se genera un nuevo directorio con el nombre del componente y los archivos que se muestran en la imagen.

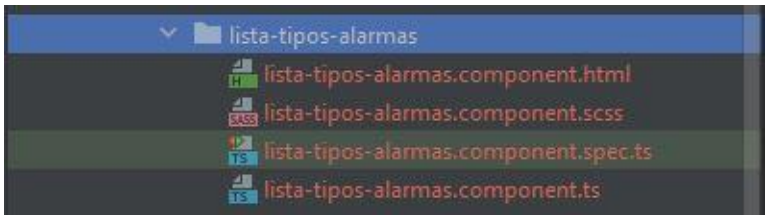


Ilustración 15. Archivos creados dentro del componente lista-tipos-alarmas.

Fuente: Captura de pantalla propia.

Además, automáticamente se habrá actualizado el fichero *app.module.ts* para incluir este nuevo componente en nuestra aplicación.

Antes de modificar el código y la vista de los componentes que hemos creado, debemos crear una serie de servicios.

Un servicio es una clase cuyo propósito es proporcionar datos y lógica entre diferentes componentes. También se utilizan para acceder a datos procedentes, por ejemplo, de un servidor web.

Cuando *Angular* detecta que un componente requiere de un servicio, automáticamente le ofrece a ese componente la clase que solicita. Por tanto, podemos decir que el servicio carga los datos cuando los componentes le preguntan a dicho servicio por la información.

En nuestra aplicación, hemos usado un servicio para guardar los datos de las distintas entidades que obtenemos al realizar las peticiones al servidor. Para ello, creamos una nueva carpeta de nombre *servicios* dentro de la ruta del proyecto *src/app* y ejecutamos el comando *ng g service carga* más el nombre de la entidad. Para poder emplear este servicio, hay que añadirlo manualmente a *src/app/app.module.ts*, importándolo y añadiéndolo a los *providers*.

En una aplicación, la información generalmente se obtiene de un servidor web a través de solicitudes *HTTP*, empleando para ello *AJAX - promises*. Sin embargo, en el *framework Angular*, contamos con un servicio *HttpClient* que nos hace esta tarea. Debemos importar en el fichero *src/app/app.module.ts* el módulo *HttpClientModule*.

Una vez hecho esto, volvemos al servicio de carga que habíamos creado e inyectamos el servicio *HttpClient* dentro de nuestro servicio.

Dentro de este servicio también debemos importar la interfaz generada anteriormente y la clase *Observable*. Un *observable* es una clase que le permite "observar" los cambios que ocurren en una variable. Cada vez que se produzca un cambio sobre lo que almacena un *observable*, se actualizará el valor del mismo.

Además, definimos una variable que almacena la ruta del servidor a la que vamos a realizar dichas peticiones.

Por último, creamos un método por cada una de las peticiones que se van a realizar al servidor, tal y como se muestra en la siguiente captura de pantalla.

Ilustración 16. Métodos del servicio carga-tipo-alarma.

Fuente: Captura de pantalla propia.

Hemos de realizar un servicio de carga por cada entidad solicitada en la propuesta de proyecto siguiendo los pasos que se han explicado anteriormente.

Previamente a aplicar estos servicios a los distintos componentes y a modificar las vistas de estos últimos, vamos a definir las rutas de navegación del cliente web y a llevar a cabo una serie de procedimientos que se detallan en los apartados que encontramos a continuación. No obstante, más adelante retomaremos la explicación de la aplicación de los servicios a los componentes, así como la construcción de dichos componentes y de sus vistas.

5.2.5.- Aplicación del *routing* y definición de las rutas del proyecto de teleasistencia.

El cliente web que estamos construyendo con *Angular* es una *Single Page Application* (*SPA*), es decir, que la única página *HTML* que se descarga del

servidor es el *index.html* y toda la aplicación es manejada desde este documento. Esto es posible gracias a que el *router* de *Angular* nos permite navegar en la aplicación.

Ya tenemos todos los componentes creados, ahora debemos emplear el módulo de *routing* de *Angular* en nuestro módulo principal dentro de la ruta del proyecto *src/app/app.module.ts*. Como inicializamos nuestro proyecto en *Angular* con *routing*, ya tenemos importado el módulo *RouterModule* que necesitamos.

Para definir las rutas de la aplicación, abrimos el archivo *app-routing.module.ts* que se encuentra en la ruta *src/app*. Dentro de este archivo encontramos el objeto *RouterModule* que llama al método *forRoot()*, en el cual asociaremos las *URL* que la aplicación tendrá y el componente que tiene que ser cargado en cada caso. Nuestras rutas serán:

- */inicio* → Debe cargar el componente *home*.
- */login* → Debe cargar el componente con la pantalla del *login*.
- */nombre de la entidad* → Debe cargar el componente lista-nombre de la entidad.
- */nombre de la entidad/modificar/:id* → Debe cargar el componente detalles-nombre de la entidad que muestra información detallada de un solo elemento de la entidad. Esta ruta tendrá un parámetro, el id del elemento de la entidad.
- */nombre de la entidad/nuevo* → Debe cargar el componente nuevo-nombre de la entidad.

- Por defecto, cuando una ruta no es correcta, redirigiremos al componente home.
- Si no ponemos ninguna ruta, la raíz también redirige a home.

Para que nuestro *routing* funcione, comprobamos que en la vista o *template* principal, que se encuentra en la ruta *src/app/AppComponent.html* del proyecto, tenemos incluido el elemento <*router-outlet*></*router-outlet*>.

Por otro lado, para que el título de nuestra página no sea siempre el mismo y se modifique cada vez que naveguemos a una ruta diferente, debemos emplear un servicio denominado *Title*. Si no empleamos este servicio, el título de la página será siempre el mismo, ya que, este se encuentra dentro de index.html, no dentro del componente <*app-root*></*app-root*>, que es en el que estamos aplicando *routing*.

Lo que tenemos que hacer es cargar el servicio *Title* en el archivo que se encuentra en la ruta *src/app/app.module.ts*, importándolo y añadiéndolo a *providers*.

En cada componente del *router* en el que queramos cambiar el título debemos cargar ese servicio y modificar el título.

Finalmente, antes de continuar avanzando en el proyecto debemos generar el componente *home*. Nos vamos a la ruta *src/app/componentes* y abrimos un terminal en el que ejecutamos el comando *ng g component home*.

Este comando nos genera una nueva carpeta llamada *home* dentro de *componentes* que contiene una serie de archivos, entramos al archivo

home.component.ts, cargamos el servicio *Tittle* y modificamos el título de la página, como se muestra en la siguiente imagen.

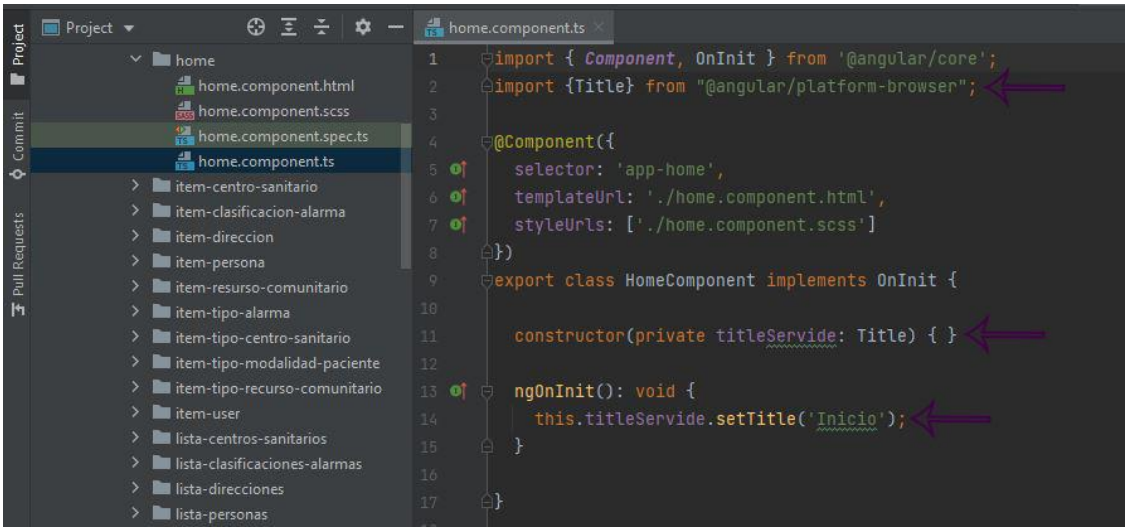


Ilustración 17. Ejemplo de carga del servicio Tittle y modificación del título de la página del componente home.

Fuente: Captura de pantalla propia.

Por ahora no realizaremos ninguna modificación en la vista del componente home, la dejaremos como viene por defecto al crear el componente y más adelante añadiremos el *HTML* y los estilos correspondientes a la misma.

En este apartado hemos visto cómo aplicar *routing* al proyecto de *Angular* y cómo definir las rutas, en el próximo apartado de la memoria hablaremos sobre las guardas y su importancia a la hora de desarrollar un cliente web de *Angular* con *routing*, además, retomaremos lo que quedó pendiente de realizar para aplicar los servicios a cada componente y veremos otros aspectos relativos a la codificación de dichos componentes.

5.2.6.- Creación de las guardas de tipo *resolve* y aplicación de los servicios a cada componente.

Las guardas o *guards* en *Angular* son interfaces que permiten proteger las rutas e indican al *router* si se permitirá la navegación a una ruta o no.

En este caso utilizaremos la guarda de tipo *ResolveGuard*, aunque existen otros cuatro tipos más y son los siguientes: *CanActivate*, *CanActivateChild*, *CanDeactivate* y *CanLoad*.

La utilidad de las guardas de tipo *resolve* en este proyecto reside en que estas permiten prevenir la redirección a páginas que no pueden cargar ningún elemento de una determina entidad.

Por ejemplo, en lugar de cargar un elemento de una entidad desde *detalles-nombre de la entidad*, lo cargaremos desde una guarda de este tipo para asegurarnos que al llegar a *detalles-nombre de la entidad* ya tenemos el elemento cargado.

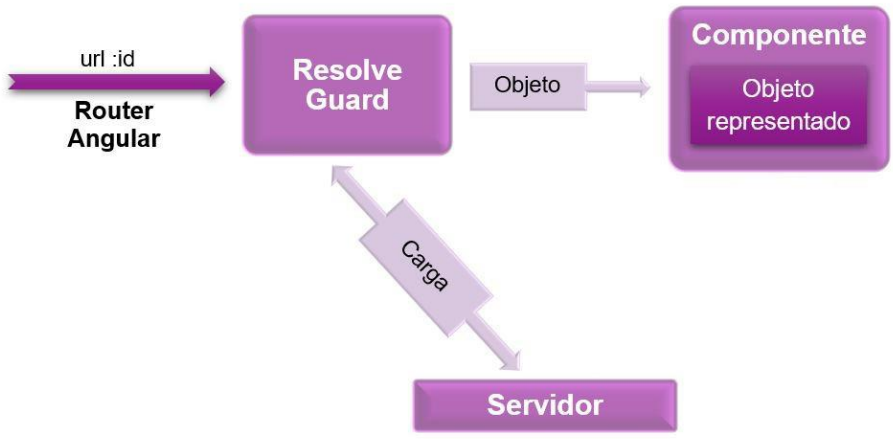


Ilustración 18. Esquema funcionamiento ResolveGuard en Angular.

Fuente: Elaboración propia.

La misma lógica que se ha explicado en el párrafo anterior se aplicará para *lista-nombre de la entidad*, de forma que se precarguen todos los elementos almacenados en una entidad antes de acceder a su ruta asociada.

En ambos casos, si las guardas no pueden cargar ningún elemento de la entidad, podremos prevenir la redirección a estas rutas, llevando al usuario directamente a */inicio*.

Este tipo de guardas puede devolver *datos primitivos*, una *promesa* o un *observable* al que el *router* de Angular se suscribe de forma automática.

Además, se implementan como un servicio por lo que para crearlas ejecutamos los comandos *ng g service detalles-nombre de la entidad-resolve* y *ng g service lista-nombre de la entidad-resolve* en un terminal que debemos abrir dentro de la carpeta *servicios*.

Abrimos el archivo *.ts* del servicio *detalles-nombre de la entidad-resolve* que acabamos de crear y lo configuramos como aparece en la imagen.

En este caso, vamos a centrarnos en explicar cómo se crearía la guarda de tipo *resolve* para precargar los detalles de un elemento de la entidad *Tipo_Alarma*, ya que, la codificación de este tipo de guardas es similar para realizar la carga de los de detalles de cualquier entidad antes de acceder a la ruta correspondiente.

El primer paso es implementar la interfaz *Resolve* que permite a la clase del servicio que hemos creado convertirse en un proveedor de datos, en concreto para el ejemplo que hemos escogido, del tipo *ITipoAlarma*.

Dicha clase que ahora provee datos, la cual se utiliza en conjunto con el *Router* de *Angular*, permite resolver los datos de cada elemento de la entidad *Tipo_Alarma* durante la navegación. Para ello, es necesario definir el método *resolve()*, que es invocado cuando se inicia la navegación, haciendo que el *Router* espere a que se resuelvan los datos antes de redirigir a la ruta */tipos_alarmas/modificar/:id* o, en caso de no poder resolverse, redirige al usuario a la ruta */inicio*.

detalles-tipo-alarma-resolve.service.ts

```
import {Injectable} from '@angular/core';
import {ActivatedRouteSnapshot, Resolve, Router, RouterStateSnapshot} from "@angular/router";
import {ITipoAlarma} from "../interfaces/i-tipo-alarma";
import {CargaTipoAlarmaService} from "./carga-tipo-alarma.service";
import {Observable, of} from "rxjs";
import {catchError} from "rxjs/operators";

@Injectable({
  providedIn: 'root'
})
export class DetallesTipoAlarmaResolveService implements Resolve<ITipoAlarma> {

  constructor(private cargaTiposAlarmas: CargaTipoAlarmaService, private router: Router) {
  }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<ITipoAlarma> {
    return this.cargaTiposAlarmas.getTipoAlarma(route.params['id']).pipe(
      catchError( selector: error => {
        this.router.navigate( commands: ['/inicio']);
        return of( args: null);
      })
    );
  }
}
```

Ilustración 19. Guarda de tipo resolve para precargar los detalles de un elemento de la entidad Tipo_Alarma.

Fuente: Captura de pantalla propia.

A continuación, abrimos el archivo *lista-nombre de la entidad-resolve.ts*, seguimos con el ejemplo anterior para la entidad *Tipo_Alarma*, en el cuál debemos añadir lo siguiente:

```typescript
import {Injectable} from '@angular/core';
import {ActivatedRouteSnapshot, Resolve, Router, RouterStateSnapshot} from "@angular/router";
import {ITipoAlarma} from "../interfaces/i-tipo-alarma";
import {CargaTipoAlarmaService} from "./carga-tipo-alarma.service";
import {Observable, of} from "rxjs";
import {catchError} from "rxjs/operators";

@Injectable({
  providedIn: 'root'
})
export class ListaTiposAlarmasResolveService implements Resolve<ITipoAlarma> {

  constructor(private cargaTiposAlarmas: CargaTipoAlarmaService, private router: Router) {
  }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<ITipoAlarma> {
    return this.cargaTiposAlarmas.getTiposAlarmas().pipe(
      catchError( selector: error => {
        this.router.navigate( commands: ['/inicio']);
        return of( args: null);
      })
    );
  }
}
```

Ilustración 20. Guarda de tipo resolve para precargar la lista de elementos de la entidad Tipo_Alarma.

Fuente: Captura de pantalla propia.

Debemos crear estas dos guardas de tipo *resolve* por cada una de las entidades y configurarlas como se ha explicado en las imágenes mostradas anteriormente. Para poder emplear las guardas, abrimos el archivo de configuración de las rutas que se encuentra en *src/app/ app-routing.module.ts* y en las rutas */nombre de la entidad* y */nombre de la entidad/modificar/:id*, aparte de indicar el *path* y el *component*, le indicamos el *resolve*. En la imagen se puede ver cómo se realizaría está configuración para la ruta */tipo_alarmas*, pero debemos hacer lo mismo con cada una de las rutas que hayamos definido.

```typescript
{
  path: 'tipos_alarmas',
  component: ListaTiposAlarmasComponent,
  resolve: {
    tipos_alarmas: ListaTiposAlarmasResolveService
  }
},
```

Ilustración 21. Configuración de la ruta tipos-alarmas.

Fuente: Captura de pantalla propia.

Una vez tenemos creados todos los servicios, guardas y componentes y, además, hemos realizado las configuraciones de las rutas podemos comenzar a configurar el archivo *.ts* de cada uno de los componentes.

Vamos a empezar viendo la forma de configurar los componentes *lista-nombre de la entidad* e *item-nombre de la entidad*. Claro está, prosiguiendo con el ejemplo de la entidad *Tipo_Alarma*, aunque para el resto de entidades la manera de realizar este proceso es la misma.

Como ya había dicho en el apartado 5.2.4., el componente *lista-nombre de la entidad* se encargará de manejar la petición que nos devuelve todos los elementos almacenados en una entidad. Dicho componente tiene como componentes hijos o componentes anidados a uno o varios componentes *item-nombre de la entidad*.

Un hijo o componente anidado es un componente cuyo *template* representa un fragmento de la vista total actual.

Para configurar el archivo *.ts* del componente *lista-tipos-alarmas*, debemos crear un array del tipo de la interfaz *ITipoAlarma*, que se encarga de definir la estructura de datos de la entidad. Dicho array almacenará todos los elementos devueltos de la petición *GET* correspondiente que hemos realizado a través de la guarda de tipo *resolve*.

Además, debemos importar el servicio *Title* para cambiar el título de la página y la clase *ActivatedRoute* para poder traernos los elementos guardados en la entidad obtenidos usando la guarda de *resolve* creada anteriormente.

La configuración final quedaría tal y como podemos ver en la captura de pantalla.

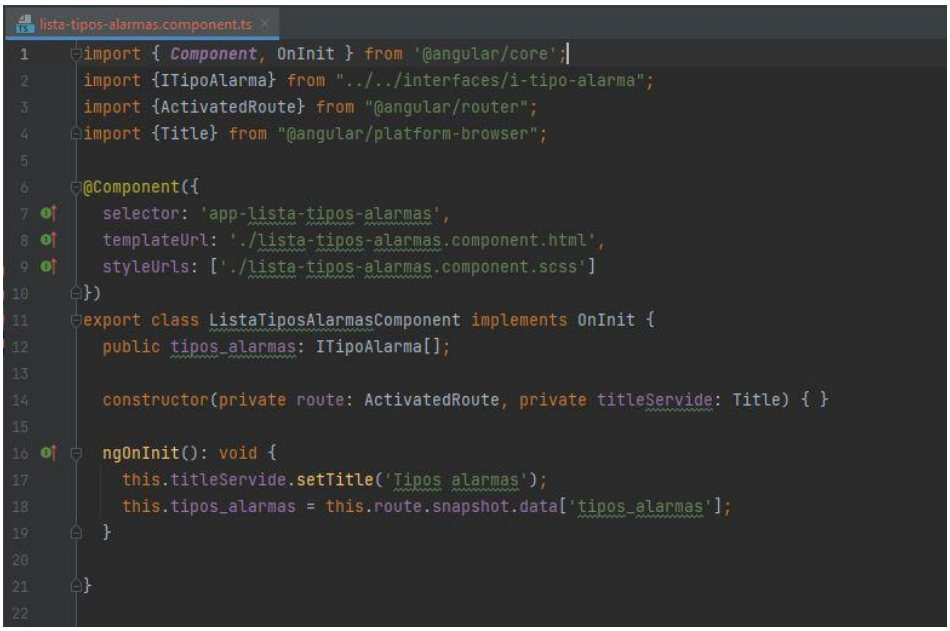


Ilustración 22. Configuración controlador del componente lista-tipos-alarmas.

Fuente: Captura de pantalla propia.

Ahora configuramos el controlador del componente hijo *item-tipo-alarma*. Lo primero será modificar el selector del mismo para que no sea un elemento, sino que el selector se lleve a cabo a través de un atributo, así evitaremos problemas con el *HTML* de la vista del padre.

Seguidamente, debemos codificar la comunicación entre un elemento padre y su hijo, de forma que, el hijo pueda mostrar la información que se va desglosando desde el componente padre. Esto se indica en el controlador empleando *@input* para marcar que la propiedad correspondiente se inicializa desde un atributo proveniente del padre.

Podemos ver como quedaría finalmente la codificación del controlador *item-tipo-alarma.ts* en la siguiente ilustración.

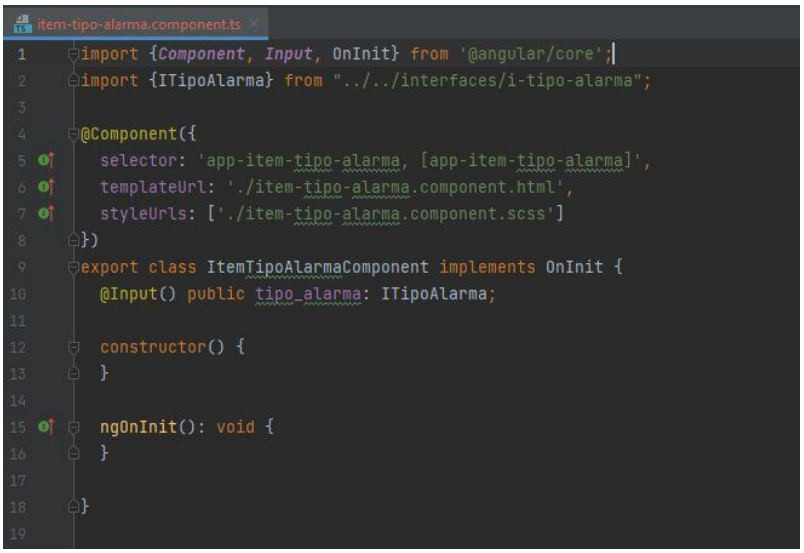


Ilustración 23. Configuración controlador del componente item-tipo-alarma.

Fuente: Captura de pantalla propia.

En la vista del padre vamos a crear una tabla que dentro del *<tbody></tbody>* muestre un *<tr></tr>* por cada *item-tipo-alarma*. También tenemos que pasarle los datos desde la vista del padre, al controlador del hijo incluyendo los atributos en el elemento vista del padre que representa dicho controlador.

De esta forma la vista del componente *lista-tipos-alarmas* quedaría así:

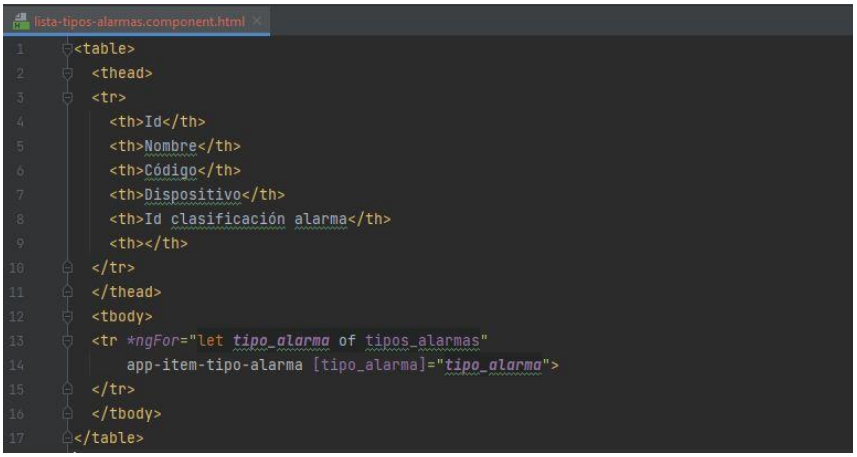


Ilustración 24. Vista del componente lista-tipos-alarmas.

Fuente: Captura de pantalla propia.

En la vista del componente hijo creamos un *<td></td>* por cada uno de los datos que queremos mostrar del elemento de la entidad en la fila correspondiente. Además, en el último *<td></td>* se ha incluido un *<button></button>* con un atributo *routerLink*. Este atributo provee un link que va a funcionar sobre la misma página, es el relativo a *href*, pero para que funcione por *routing* en lugar de redirección. En este caso le indicamos la ruta */tipos-alarmas/modificar* y el id del elemento para que al hacer clic sobre el botón nos lleve al componente *detalles-tipo-alarma* que nos va a permitir modificar los datos de dicho elemento.

La vista del componente *item-tipo-alarma* quedaría tal y como podemos ver en la imagen que se muestra a continuación:

item-tipo-alarma.component.html

```
<td>{{tipo_alarma.id}}</td>
<td>{{ tipo_alarma.nombre }}</td>
<td>{{ tipo_alarma.codigo }}</td>
<td>{{ tipo_alarma.es_dispositivo }}</td>
<td>{{ tipo_alarma.id_clasificacion_alarma.nombre }} {{ tipo_alarma.id_clasificacion_alarma.codigo }}</td>
<td><button type="submit" [routerLink]="['/tipos_alarmas/modificar', tipo_alarma.id]">Modificar datos</button></td>
```

Ilustración 25. Vista del componente item-tipo-alarma.

Fuente: Captura de pantalla propia.

Acabamos de explicar cómo se realizaría la codificación tanto del *controlador* como del *template* de los componentes *lista-tipos-alarmas* e *item-tipo-alarma*, ahora vamos a proceder a detallar esa misma codificación, ahora faltaría por configurar el componente *detalles-tipo-alarma*.

No obstante, en este caso vamos a centrarnos en la manera de realizar estas configuraciones para entidades en las que todos sus atributos son de tipos simples, es decir, entidades que ninguna de sus propiedades sea del tipo de otra entidad. Más adelante, en un apartado creado específicamente para ello, se

detallará la forma de codificar el componente *detalles-nombre de la entidad* para dichas entidades debido a que la manera de realizar esta codificación es ligeramente diferente.

Por tanto, vamos a continuar explicando tales configuraciones ahora para la entidad *Clasificacion_Alarma*. Abrimos el controlador del componente *detalles-clasificacion-alarma* y creamos dos variables, una para almacenar el elemento que vamos a modificar y otra para almacenar el id de dicho elemento que obtenemos a través de un parámetro de la ruta de navegación.

Importamos el servicio *Title*, la clase *ActivatedRoute*, el servicio de carga y la clase *Router*.

Dentro del método *ngOnInit()* inicializamos las dos variables creadas empleando el atributo de la clase *ActivatedRoute* que hemos creado en el constructor, el cual nos permite obtener parámetros desde una ruta. También modificamos el título de la página dentro de este método para que se muestre el que nosotros queremos.

Ahora creamos un método que nos va a permitir modificar los datos de la clasificación de alarma mediante un formulario. Para ello, empleamos el servicio correspondiente que hemos inyectado en el constructor, para este ejemplo en concreto se denomina *cargaClasificacionesAlarmas*, y llamamos al método para modificar una clasificación de alarma que habíamos creado dentro de dicho servicio. A este método del servicio le pasamos la clasificación de alarma modificada por el formulario y para que se ejecuten los cambios debemos suscribirnos al observable del servicio.

Si todo funciona de manera correcta mostramos por consola un mensaje y redirigimos al usuario a la ruta */clasificaciones_alarmas* donde se muestra una tabla con todos los elementos almacenados en dicha entidad. Para realizar esto último usamos el atributo de la clase *Router* que hemos creado en el constructor. Si ocurre algún error, mostramos el mismo por consola.

La configuración del controlador debe quedar igual que en la siguiente imagen:

detalles-clasificacion-alarma.component.ts

```
export class DetallesClasificacionAlarmaComponent implements OnInit {
  public clasificacion_alarma: IClasificacionAlarma;
  public idClasificacionAlarma: number;

  constructor(private route: ActivatedRoute, private titleServide: Title, private cargaClasificacionesAlarmas: CargaClasificacionAlarmaService,
              private router: Router) {
  }

  ngOnInit(): void {
    this.idClasificacionAlarma = this.route.snapshot.params['id'];
    this.clasificacion_alarma = this.route.snapshot.data['clasificacion_alarma'];
    this.titleServide.setTitle('Modificar clasificación alarma ' + this.idClasificacionAlarma);
  }

  modificarClasificacionAlarma(): void {
    this.cargaClasificacionesAlarmas.modificarClasificacionAlarma(this.clasificacion_alarma).subscribe(
      next: e => {
        console.log('Modificada la clasificación alarma ' + e.id);
        this.router.navigate( commands: ['/clasificaciones_alarmas']);
      },
      error: error => {
        console.log(error);
      }
    );
  }
}
```

Ilustración 26. Configuración controlador del componente detalles-clasificacion-alarma.

Fuente: Captura de pantalla propia.

Dentro de la vista de este componente por ahora simplemente vamos a crear un formulario que nos permita modificar los atributos la clasificación de alarma seleccionada.

El primer paso será importar en el fichero *src/app/app.module.ts* el módulo *FormsModule*, ya que, es necesario para poder utilizar la directiva *ngModel* y para realizar las validaciones de los formularios pedidas en la propuesta de proyecto, pero esto lo haremos más adelante.

ngModel es una directiva que vincula *input*, *select* y *textarea*, permitiendo almacenar el valor de usuario requerido en una variable y pudiendo usar esa variable siempre que necesitemos ese valor. También se utiliza durante las validaciones en un formulario.

Aparte de crear los inputs para cada atributo de la clasificación de alarma que queremos modificar, creamos dos botones: uno para volver a la página del componente donde se listan todos los elementos almacenados en la entidad, empleamos *routerLink* para llevar esto a cabo, y el otro botón, que nos va a permitir enviar el formulario y ejecutar un evento al hacer clic para modificar la clasificación de alarma que deseamos llamando al método que habíamos creado en el controlador.

El *template* quedaría por tanto tal y como se puede apreciar en la captura de pantalla:

detalles-clasificacion-alarma.component.html

```
<div>
  <h1>Modificar datos clasificación alarma {{idClasificacionAlarma}}</h1>
  <form>

    <div>
      <label for="nombre">Nombre</label>
      <input type="text" name="nombre" id="nombre" [(ngModel)]="clasificacion_alarma.nombre">
    </div>

    <div>
      <label for="codigo">Código</label>
      <input type="text"  name="codigo" id="codigo" [(ngModel)]="clasificacion_alarma.codigo">
    </div>

    <button type="button" class="btn" [routerLink]="['/clasificaciones_alarmas']">Volver</button>
    <button type="submit" class="btn" (click)="modificarClasificacionAlarma()">Guardar</button>

  </form>
</div>
```

Ilustración 27. Vista del componente detalles-clasificacion-alarma.

Fuente: Captura de pantalla propia.

Lo expuesto hasta el momento, empleando ejemplos concretos de ciertas entidades, sobre las guardas de tipo *resolve*, la aplicación de los servicios a un componente y la configuración, tanto de la vista como del controlador de dicho componente, se realizaría siguiendo los mismos pasos para el resto de entidades, a excepción de las que presentan atributos del tipo de otra entidad.

Nos falta por explicar la configuración del componente *nuevo-nombre de la entidad* para las entidades en las que todos sus atributos son de tipos simples, pero esto lo haremos en el siguiente apartado, ya que, surgieron algunos fallos y hubo que realizar ciertos procedimientos adicionales.

Por otro lado, al igual que para el componente *detalles-nombre de las entidades* con atributos que referencian a otra entidad, la codificación del componente *nuevo-nombre de la entidad* para estas entidades se explicará en un apartado específico más adelante debido a que la codificación presenta ciertos cambios.

5.2.7.- Creación de clases para instanciar las interfaces.

Como hemos mencionado anteriormente, en este apartado se describe la configuración tanto del controlador como de la vista del componente *nuevo-nombre de la entidad* de las entidades con atributos de tipos simples. Usando de guía el componente *nueva-clasificacion-alarma* para, proseguir así, con el ejemplo que veníamos usando en el apartado anterior para abordar la configuración del componente *detalles-clasificacion-alarma*.

Comenzaremos explicando la forma de codificar el controlador y el motivo por el cual fue necesario crear una clase que implementase la interfaz que define la estructura de cada entidad.

El primer paso para codificar el controlador fue crear una variable para almacenar los datos de la nueva clasificación de alarma del tipo de la interfaz *IClasificacionAlarma* de la entidad *Clasificacion_Alarma* a la que queremos añadir la misma.

Además, importamos el servicio *Title*, la clase *ActivatedRoute*, el servicio de carga y la clase *Router*.

Dentro del método *ngOnInit()* modificamos el título de la página para que se muestre el que nosotros queremos. Además, dentro de este mismo método **inicializamos la variable que habíamos definido para crear una nueva interfaz** de la misma.

Al realizar este paso, el **intérprete de *PhpStorm* nos marcaba un fallo** y la solución que nos indicaba era inicializar la variable creando una nueva clase que implementaba la interfaz.

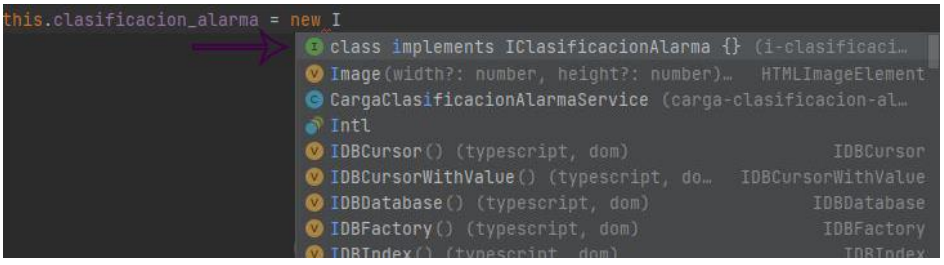


Ilustración 28. Fallo al inicializar la variable clasificacion_alarma como una nueva interfaz IClasificacionAlarma.

Fuente: Captura de pantalla propia.

Este fallo se debía a que **no se pueden crear objetos nuevos** del tipo **de una interfaz sin instanciar antes una clase**.

Antes de continuar codificando el controlador del componente *nueva-clasificacion-alarma*, **se decidió crear una nueva carpeta llamada *clases*** en la ruta *src/app* **y dentro de ella crear una clase para implementar cada una de las interfaces que definen la estructura de datos de las entidades** solicitadas en la propuesta de *PFC*.

La decisión de realizar esto se tomó porque quedaba mejor estructurado el proyecto implementando cada una de las interfaces en clases en una carpeta creada para ello que, haciéndolo directamente en el método *ngOnInit()* del controlador del componente *nuevo-nombre de la entidad*.

El comando para crear una nueva clase en *Angular* es *ng g class* más el nombre de la clase. Ejecutamos el mismo en un terminal dentro de la carpeta *clases* que hemos creado y le indicamos a la clase que implemente la interfaz correspondiente de la forma que se muestra en la imagen:

clasificacion-alarma.ts

```
1  import {IClasificacionAlarma} from "../interfaces/i-clasificacion-alarma";
2
3  export class ClasificacionAlarma implements IClasificacionAlarma{
4    codigo: string;
5    id: number;
6    nombre: string;
7  }
```

Ilustración 29. Clase ClasificacionAlarma que implementa la interfaz IClasificacionAlarma.

Fuente: Captura de pantalla propia.

Solventado este pequeño fallo, podemos volver al controlador del componente *nueva-clasificacion-alarma* y continuar con la programación del mismo, importando la clase que hemos creado y cambiando en el constructor la implementación que nos realizaba *PhpStorm* por simplemente inicializar la variable creada como un nuevo objeto de dicha clase.

Finalmente, tendremos un método que nos va a permitir crear una nueva clasificación de alarma con los datos que se envíen mediante un formulario en *HTML*. Para ello, emplearemos el método correspondiente que habíamos creado en el servicio *cargaClasificacionesAlarmas* y le pasaremos los datos recogidos a partir del formulario.

Si todo funciona de manera correcta mostramos por consola un mensaje y redirigimos al usuario a la ruta */clasificaciones_alarmas* donde se muestra una tabla con todos los elementos almacenados en dicha entidad. Para realizar esto último usamos el atributo de la clase *Router* que hemos creado en el constructor.

Si ocurre algún error, mostramos el mismo por consola.

Los demás controladores de los componentes *nuevo-nombre de la entidad*, para las entidades en las que todos sus atributos son de tipos simples, también se configurarían de la misma forma.

En la imagen, se muestra el ejemplo para el caso en concreto del controlador del componente *nueva-clasificacion-alarma*.

```typescript
export class NuevaClasificacionAlarmaComponent implements OnInit {
  public clasificacion_alarma: IClasificacionAlarma;

  constructor(private route: ActivatedRoute, private titleServide: Title, private cargaClasificacionesAlarmas: CargaClasificacionAlarmaService,
              private router: Router) {
  }

  ngOnInit(): void {
    this.titleServide.setTitle('Crear nueva clasificación alarma');
    this.clasificacion_alarma = new ClasificacionAlarma();
  }

  nuevaClasificacionAlarma(): void {
    this.cargaClasificacionesAlarmas.nuevaClasificacionAlarma(this.clasificacion_alarma).subscribe(
      next: e => {
        console.log('Clasificación alarma creada');
        this.router.navigate( commands: ['/clasificaciones_alarmas']);
      },
      error: error => {
        console.log(error);
      }
    );
  }
}
```

Ilustración 30. Configuración controlador del componente nueva-clasificacion-alarma.

Fuente: Captura de pantalla propia.

En la vista de este componente creamos un formulario que nos permita recoger los datos del nuevo elemento a almacenar en la entidad correspondiente, de la misma forma que hicimos en el componente *detalles-clasificacion-alarma*. La vista quedaría de la siguiente forma:

```html
<div>
  <h1>Crear nueva clasificación alarma</h1>
  <form>

    <div>
      <label for="nombre">Nombre</label>
      <input type="text" name="nombre" id="nombre" [(ngModel)]="clasificacion_alarma.nombre">
    </div>

    <div>
      <label for="codigo">Código</label>
      <input type="text" name="codigo" id="codigo" [(ngModel)]="clasificacion_alarma.codigo">
    </div>

    <button type="submit" class="btn" [routerLink]="['/clasificaciones_alarmas']">Volver</button>
    <button type="submit" class="btn" (click)="nuevaClasificacionAlarma()">Crear</button>
  </form>
</div>
```

Ilustración 31. Vista del componente nueva-clasificacion-alarma.

Fuente: Captura de pantalla propia.

5.2.8.- Resolviendo los problemas con los formularios de creación de las entidades que a su vez contienen otra entidad.

En este apartado, se detallará la manera de realizar los formularios para añadir nuevos elementos a entidades que contienen atributos del tipo de otra entidad. Para ello, retomaremos la entidad *Tipo_Alarma* y explicaremos la manera de configurar tanto la vista como el controlador para el componente *nuevo-tipo-alarma*.

Sin embargo, debemos tener en cuenta que, aunque hablemos de un ejemplo específico para facilitar la compresión de dichas configuraciones, los pasos que se van a ir detallado a lo largo de este apartado, sirven como modelo para configurar el componente *nuevo-nombre de la entidad* para las entidades que no solo contienen atributos simples.

Asimismo, al final de dicho apartado, se detallarán una serie de errores encontrados en la parte del servidor y que han impedido que los formularios de creación para todas las entidades de este tipo funcionen adecuadamente.

Comenzamos abriendo *Postman* y comprobando los datos que espera recibir la entidad *Tipo_Alarma* al realizar un *POST*. Como se observa en la imagen, dicha entidad espera recibir para su atributo *id_clasificacion_alarma*, que es del tipo *IClasificacionAlarma*, el valor del id de una clasificación de alarma que ya haya sido creada con anterioridad.

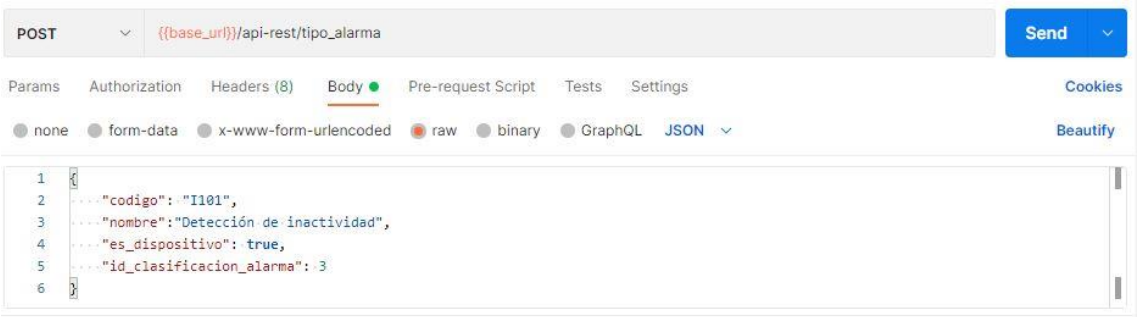


Ilustración 32. Datos que recibe la entidad Tipo_Alarma al realizar un POST.

Fuente: Captura de pantalla propia.

El primer paso para configurar el controlador del componente *nuevo-tipo-alarma*, es abrir el archivo del mismo y crear una variable para almacenar los datos del nuevo tipo de alarma, del tipo de la interfaz *ITipoAlarma*, y otra variable que almacenará un array del tipo de la interfaz *IClasificacionAlarma*, para poder, posteriormente en la vista, utilizar un *select* en el que cada *option* represente cada una de las clasificaciones de alarmas que contendrá dicho array.

Además, importamos el servicio *Title*, la clase *ActivatedRoute*, el servicio de carga y la clase *Router*.

Seguidamente, en el método *ngOnInit()* modificamos el título de la página para que se muestre el que nosotros queremos, inicializamos la variable que almacenará el nuevo tipo de alarma como un nuevo objeto de la clase TipoAlarma, cuyo motivo de creación se explicó en el apartado 5.2.7 del presente documento y, finalmente, dentro de dicho método debemos inicializar el array con las clasificaciones de alarmas almacenadas en el sistema.

Para realizar este último paso, debemos ir al archivo dónde se crearon las rutas del proyecto de *Angular* y añadir a la ruta *tipos_alarmas/nuevo* el elemento *resolve* para poder precargar las clasificaciones de alarmas utilizando la guarda de tipo *resolve* que habíamos creado para esto.

```
{
  path: 'tipos_alarmas/nuevo',
  component: NuevoTipoAlarmaComponent,
  canActivate: [LoginGuard],
  resolve: {
    clasificaciones_alarmas: ListaClasificacionesAlarmasResolveService
  }
},
```

Ilustración 33.Configuración de la ruta tipos-alarmas/nuevo.

Fuente: Captura de pantalla propia.

Una vez realizada esta configuración previa, ya podemos inicializar el array de clasificaciones de alarmas usando su guarda. Además, inicializamos la variable *es_dispositivo* a *true* por defecto para que, siempre esté ese valor seleccionado en la vista del formulario.

Por último, creamos un método que nos permita crear un nuevo tipo de alarma con los datos recibidos del formulario presente en la vista, cuya codificación procederemos a explicar en este mismo apartado. Para realizar esto, empleamos la variable que hemos creado en el constructor del tipo del servicio de carga, *cargaTiposAlarmas*, y llamamos al método para añadir un nuevo tipo de alarma creado dentro de este servicio.

A este método del servicio le pasamos el tipo de alarma que queremos crear y para que se ejecute debemos suscribirnos al observable devuelto por el mismo.

Si todo funciona de manera correcta mostramos por consola un mensaje y redirigimos al usuario a la ruta */tipos_alarmas* donde se muestra una tabla con todos los elementos almacenados en dicha entidad. Para realizar esto último usamos el atributo de la clase *Router* que hemos creado en el constructor.
Si ocurre algún error, mostramos el mismo por consola.

El controlador del componente *nuevo-tipo-alarma* quedaría de la siguiente forma:

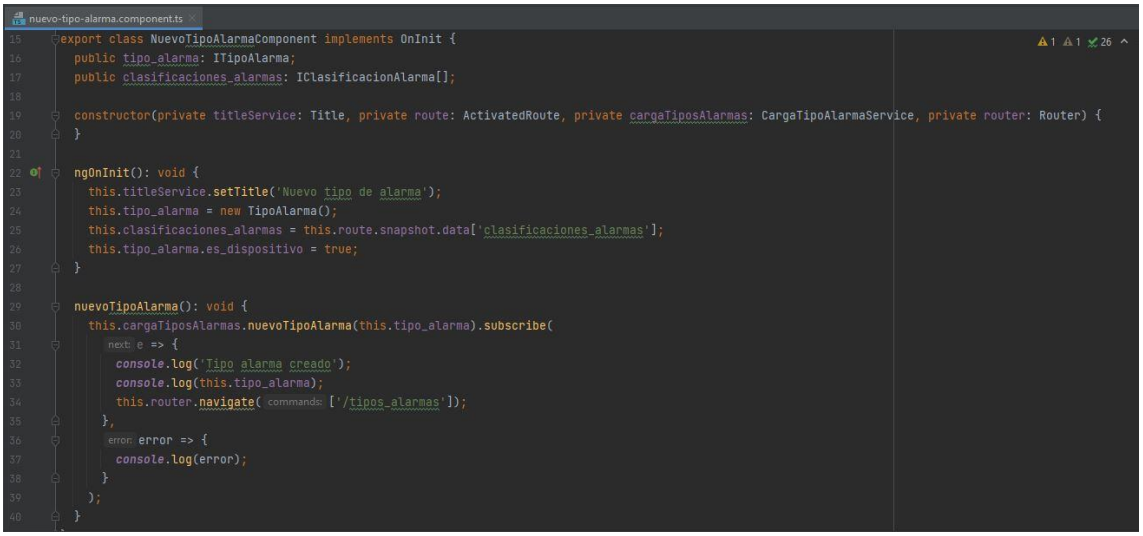


Ilustración 34. Configuración controlador del componente nuevo-tipo-alarma.

Fuente: Captura de pantalla propia.

Respecto a la vista de este componente, añadimos un formulario que nos permita recoger los datos del nuevo elemento, teniendo en cuenta que, para recoger el valor de la clasificación de alarma en *HTML*, debemos realizar un *select* en el que cada *option* se corresponde con las clasificaciones de alarmas que existen en el sistema e indicamos usando *[(ngModel)]* la propiedad a la que queremos añadir los datos. Para este ejemplo en concreto la propiedad sería *tipo_alarma.id_clasificacion_alarma* y el *[value]* del *option* indicaría el id de la

clasificación de alarma. De esta forma, al seleccionar la clasificación de alarma que queremos asociar al nuevo tipo de alarma, se le asigna el id de la misma a la propiedad *id_clasificacion_alarma*.

La vista de este componente quedaría tal que así:

```
tipo-alarma.component.html
<main>
<div>
  <h1 class="component-title">Crear nuevo tipo alarma</h1>
  <form>
    <div>
      <label for="nombre">Nombre</label>
      <input type="text" name="nombre" id="nombre" [(ngModel)]="tipo_alarma.nombre">
    </div>
    <div>
      <label for="codigo">Código</label>
      <input type="text" name="codigo" id="codigo" [(ngModel)]="tipo_alarma.codigo">
    </div>
    <div>
      <span>Es dispositivo </span>
      <div>
        <input type="radio" id="es_dispositivo_true"
               name="es_dispositivo_true" [value]="true" [(ngModel)]="tipo_alarma.es_dispositivo">
        <label for="es_dispositivo_true">Si</label>
      </div>
      <div>
        <input type="radio" id="es_dispositivo_false"
               name="es_dispositivo_false" [value]="false" [(ngModel)]="tipo_alarma.es_dispositivo">
        <label for="es_dispositivo_false">No</label>
      </div>
    </div>
    <div>
      <label for="clasificacion_alarma">Clasificación alarma</label>
      <select id="clasificacion_alarma" name="clasificacion_alarma"
              [(ngModel)]="tipo_alarma.id_clasificacion_alarma">
        <option
                *ngFor="let clasificacion_alarma of clasificaciones_alarmas"
                [value]="clasificacion_alarma.id">{{ clasificacion_alarma.nombre }} {{ clasificacion_alarma.codigo }}</option>
      </select>
    </div>
    <button type="submit"  (click)="nuevoTipoAlarma()">
      Guardar
    </button>
    <button type="button" [routerLink]="['/tipos_alarmas']">
      Volver
    </button>
  </form>
</div>
</main>
```

Ilustración 35. Vista del componente nuevo-tipo-alarma.

Fuente: Captura de pantalla propia.

Antes de dar paso al siguiente apartado de la memoria, en el que se explicará la forma de configurar el componente para modificar entidades cuyos atributos pueden ser del tipo de otra entidad, es de importancia señalar algunos fallos

encontrados al efectuar la codificación del componente *nuevo-nombre de la entidad* para las entidades *Centro_Sanitario*, *Recurso_Comunitario* y *Persona*.

Tanto en la entidad *Centro_Sanitario* como en la entidad *Recurso_Comunitario*, el nombre del atributo que guarda la dirección no coincide, en el *GET* se llama *id_direccion* y en el *POST* se llama *direccion*. Por lo tanto, no se crean nuevos elementos de este tipo de entidades, lo que quizás se deba a que el nombre del atributo de la dirección no coincide y sería algo a modificar en el lado del servidor de la aplicación de teleasistencia.

En las capturas de pantalla, se muestran los datos devueltos por un *GET* a la entidad *Recurso_Comunitario* y los que espera recibir el *POST* al crear un nuevo *Recurso_Comunitario*, todo ello revisado desde la batería de peticiones al servidor creada en *Postman*.

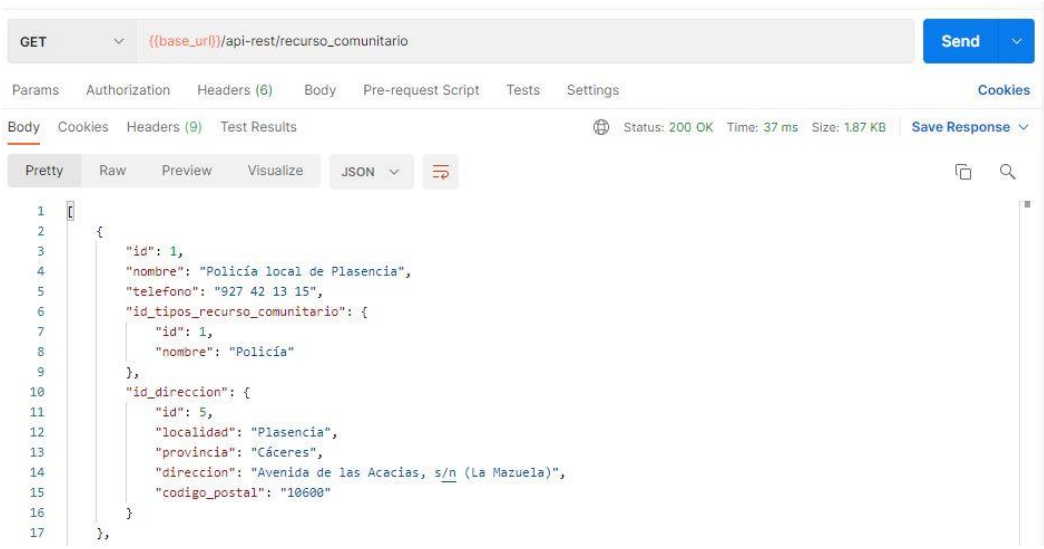


Ilustración 36. Datos que recibe la entidad Recurso_Comunitario al realizar un GET.

Fuente: Captura de pantalla propia.

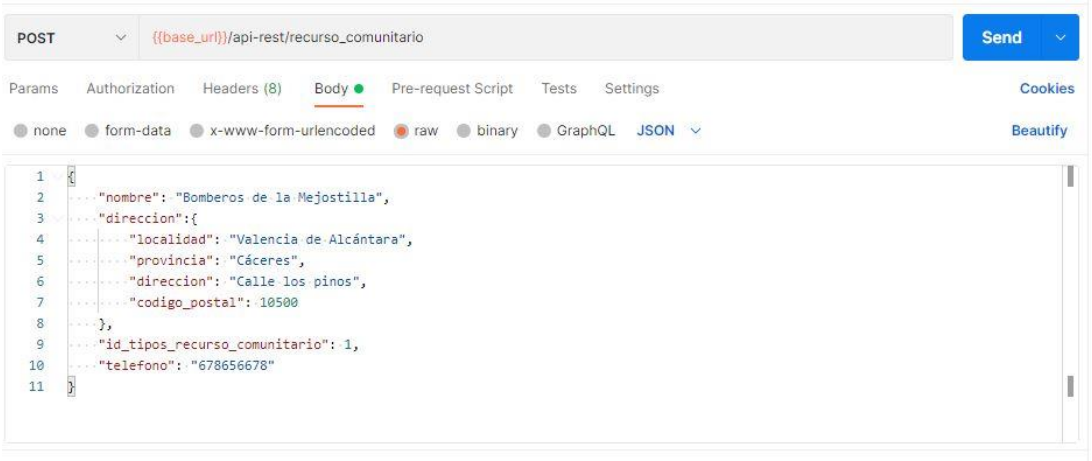


Ilustración 37. Datos que recibe la entidad Recurso_Comunitario al realizar un POST.

Fuente: Captura de pantalla propia.

Esto mismo ocurre con la entidad *Centro_Sanitario* y, con la entidad *Persona*, al crear una persona añadiendo la dirección y no utilizando un *select* con las direcciones ya creadas.

Realizando pruebas desde el *Network* del inspector de código de *Google Chrome*, al tratar de crear un nuevo *Recurso_Comunitario* vemos como nos marca que hay un error con el campo que almacena la dirección.

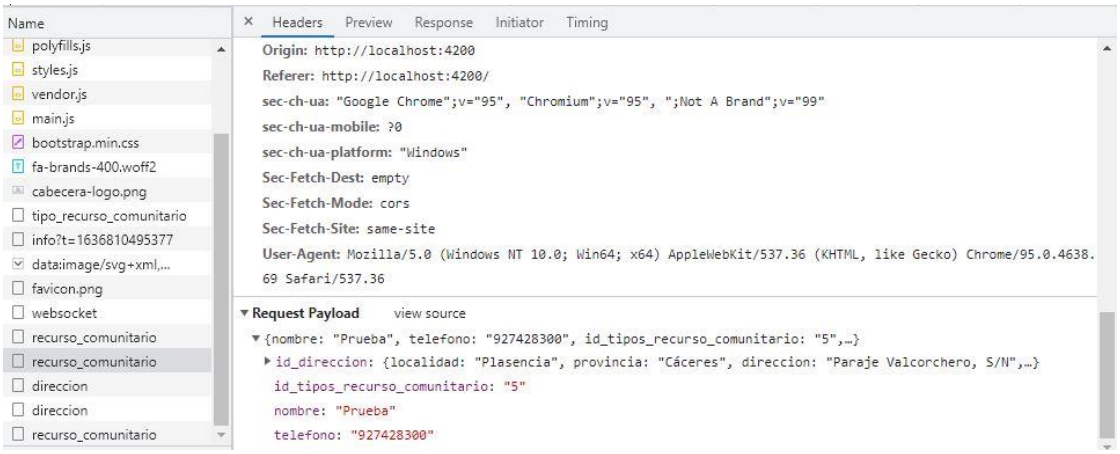


Ilustración 38. Prueba creación de un nuevo recurso comunitario desde el formulario creado.

Fuente: Captura de pantalla propia.

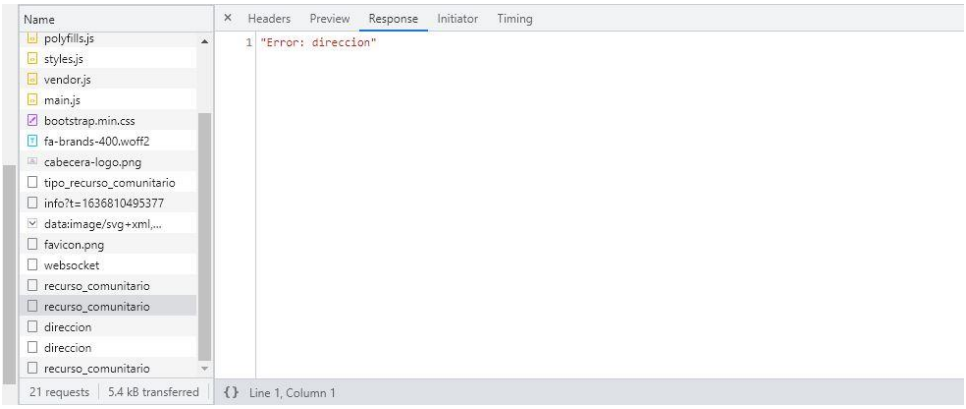


Ilustración 39. Respuesta obtenida de la prueba realizada al tratar de añadir un nuevo recurso comunitario.

Fuente: Captura de pantalla propia.

5.2.9.- Resolviendo los problemas con los formularios de modificación de las entidades que a su vez contienen otra entidad.

Anteriormente hemos explicado la forma de configurar el componente *nuevo-nombre de la entidad*, utilizando como ejemplo el componente *nuevo-tipo-alarma*, para entidades que contienen atributos del tipo de otra entidad. Por tanto, solo falta por detallar la codificación del componente *detalles-nombre de la entidad* para las entidades con estas características.

Usaremos como ejemplo el componente *detalles-tipo-alarma*, pero es similar para todos los componentes que modifican entidades que a su vez contienen otra entidad.

Abrimos *Postman* y consultamos los datos que espera recibir la entidad *Tipo_Alarma* al realizar un *PUT*. Como vemos en la imagen, los datos que espera son los mismos que esperaba al realizar un *POST*, es decir, para su atributo

id_clasificacion_alarma, que es del tipo *IClasificacionAlarma*, espera el valor del id de una clasificación de alarma que ya haya sido creada con anterioridad.

Ilustración 40. Datos que recibe la entidad Tipo_Alarma al realizar un PUT.

Fuente: Captura de pantalla propia.

Como ya teníamos creado el componente *detalles-tipo-alarma*, abrimos el controlador del mismo y creamos tres variables, una para almacenar el tipo de alarma que vamos a modificar, otra para almacenar el id de dicho tipo de alarma que obtenemos a través de un parámetro de la ruta de navegación y, por último, otra variable que almacenará un array del tipo de la interfaz *IClasificacionAlarma*, para poder, posteriormente en la vista, utilizar un *select* en el que cada *option* represente cada una de las clasificaciones de alarmas que contendrá dicho array, apareciendo seleccionada la que ya tenemos almacenada y permitiendo modificarla por otra si lo deseamos.

A continuación, importamos el servicio *Title*, la clase *ActivatedRoute*, el servicio de carga y la clase *Router*, definiendo un atributo para cada uno de estos elementos en el constructor.

En el método *ngOnInit()* inicializamos las variables *tipo_alarma* e *idTipoAlarma* creadas empleando el atributo de la clase *ActivatedRoute*, el cual nos permite obtener parámetros desde una ruta. También modificamos el título de la página

dentro de este método para que se muestre el que nosotros queremos e inicializamos el array con las clasificaciones de alarmas almacenadas en el sistema.

Para realizar este paso final, debemos ir al archivo dónde se crearon las rutas del proyecto de Angular y añadir a la ruta *tipos_alarmas/modificar/:id*, dentro del elemento *resolve*, la guarda que precarga las clasificaciones de alarmas, a parte de la que ya teníamos para precargar los detalles de un tipo de alarma.

```
{
  path: 'tipos_alarmas/modificar/:id',
  component: DetallesTipoAlarmaComponent,
  canActivate: [LoginGuard],
  resolve: {
    tipo_alarma: DetallesTipoAlarmaResolveService,
    clasificaciones_alarmas: ListaClasificacionesAlarmasResolveService
  }
},
```

Ilustración 41. Configuración de la ruta tipos_alarmas/modificar/:id.

Fuente: Captura de pantalla propia.

El siguiente paso, es crear un método llamado *optionSelected()* que recibe el índice de la clasificación de alarma que ya tenemos almacenada y añade al *option* correspondiente a la misma el atributo *selected* para que aparezca dicha *option* por defecto marcada en el *select* al mostrarse en el cliente web.

Finalmente, creamos un método para modificar los datos del tipo de alarma que queremos cambiar por los que se envíen mediante un formulario.

Para ello, empleamos la variable que hemos creado en el constructor del tipo del servicio de carga, para este ejemplo *cargaTiposAlarmas*, y llamamos al método para modificar un tipo de alarma creado dentro de este servicio.

A dicho método del servicio le pasamos el tipo de clasificación de alarma a modificar y para ejecutar los cambios nos suscribirnos al observable devuelto por el mismo.

Si todo va bien, mostramos por consola un mensaje y redirigimos al usuario a la ruta */tipos_alarmas* donde se muestra una tabla con todos los elementos almacenados en dicha entidad. Para realizar esto último usamos el atributo de la clase *Router* que hemos creado en el constructor. Por el contrario, si ocurre algún error, mostramos el mismo por consola.

La configuración del controlador debe quedar igual que en la siguiente imagen:

```
detalles-tipo-alarma.component.ts
export class DetallesTipoAlarmaComponent implements OnInit {
  public tipo_alarma: ITipoAlarma;
  public idTipoAlarma: number;
  public clasificaciones_alarmas: IClasificacionAlarma[];

  constructor(private titleService: Title, private route: ActivatedRoute, private cargaTiposAlarmas: CargaTipoAlarmaService, private router: Router) {
  }

  ngOnInit(): void {
    this.tipo_alarma = this.route.snapshot.data['tipo_alarma'];
    this.idTipoAlarma = this.route.snapshot.params['id'];
    this.clasificaciones_alarmas = this.route.snapshot.data['clasificaciones_alarmas'];
    this.titleService.setTitle('Modificar tipo alarma ' + this.idTipoAlarma);
  }

  optionSelected(i: number): void {
    document.getElementsByClassName( classNames: 'clasificacion_alarma_option')[i].setAttribute( qualifiedName: 'selected', value: '');
  }

  modificarTipoAlarma(): void {
    this.cargaTiposAlarmas.modificarTipoAlarma(this.tipo_alarma).subscribe(
      next: e => {
        console.log('Tipo alarma ' + e.id + ' modificado');
        console.log(this.tipo_alarma);
        this.router.navigate( commands: ['/tipos_alarmas']);
      },
      error: error => {
        console.log(error);
      }
    );
  }
}
```

Ilustración 42. Configuración controlador del componente detalles-tipo-alarma.

Fuente: Captura de pantalla propia.

Ahora vamos a proceder a configurar la vista de dicho componente, añadiendo un formulario que nos permitirá recoger los datos del tipo de alarma que queremos modificar. Para recoger el valor del atributo clasificación de alarma en *HTML*, beberemos realizar un *select* en el que cada *option* se corresponda con

las clasificaciones de alarmas existentes. De esta forma, al seleccionar la clasificación de alarma que queremos asociar para rectificar el tipo de alarma, se le asignará a la propiedad *id_clasificacion_alarma* el valor elegido.

Finalmente, empleando la interpolación, por cada *option* comprobamos si el *id* del atributo que almacena el *id_clasificacion_alarma* del tipo alarma a modificar es igual al *id* de la clasificación de alarma correspondiente y, en caso afirmativo, llamamos al método *optionSelected()* pasándole el índice del *option* que cumple la condición. En caso contrario, añadimos unas comillas vacías.

La vista de este componente quedaría tal como se muestra en la captura de pantalla realizada de la misma. Como podemos ver su código *HTML* es prácticamente el mismo que el de la vista del componente *nuevo-tipo-alarma*.

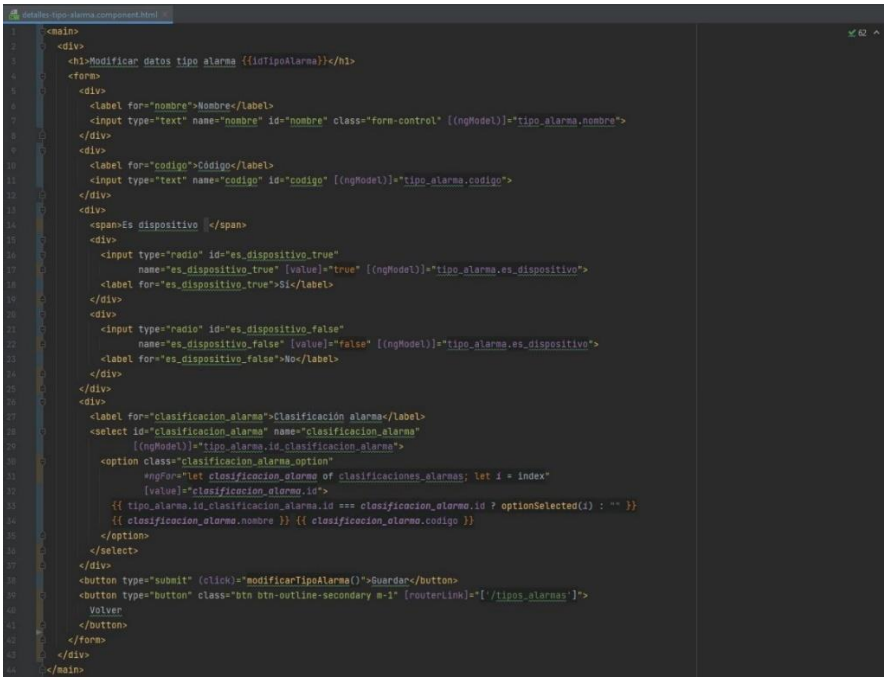


Ilustración 43. Vista del componente detalles-tipo-alarma.

Fuente: Captura de pantalla propia.

Antes de continuar con el siguiente apartado de este documento, merece la pena destacar algunos fallos encontrados a la hora de modificar las entidades que a su vez contienen otra entidad. Las entidades que tienen tales fallos son *Tipo_Alarma*, *Centro_Sanitario* y *Recurso_Comunitario*.

Estos problemas son dos: no aparece como *selected* el atributo del tipo de la otra entidad, pero si revisamos con el inspector de *Google Chrome* el código si que aparece como *selected* el *option* que ya tenemos almacenado y, el otro problema, es que al modificar atributos del tipo de otra entidad pasándole el *id* del mismo no se cambia el valor almacenado.

Realizando pruebas desde el *Network* del inspector de código de *Google Chrome*, al tratar de modificar un *Recurso_Comunitario* observamos que, aunque se envía el *id* del nuevo tipo de recurso comunitario, la respuesta del servidor nos devuelve un objeto con el tipo de recurso comunitario almacenado al crear el recurso comunitario en primera instancia.

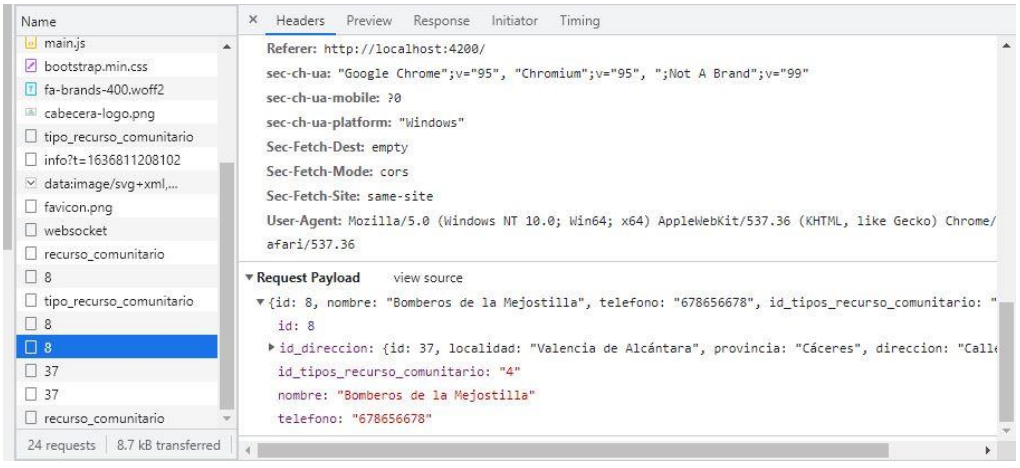


Ilustración 44. Prueba modificación de un recurso comunitario desde el formulario creado.

Fuente: Captura de pantalla propia.

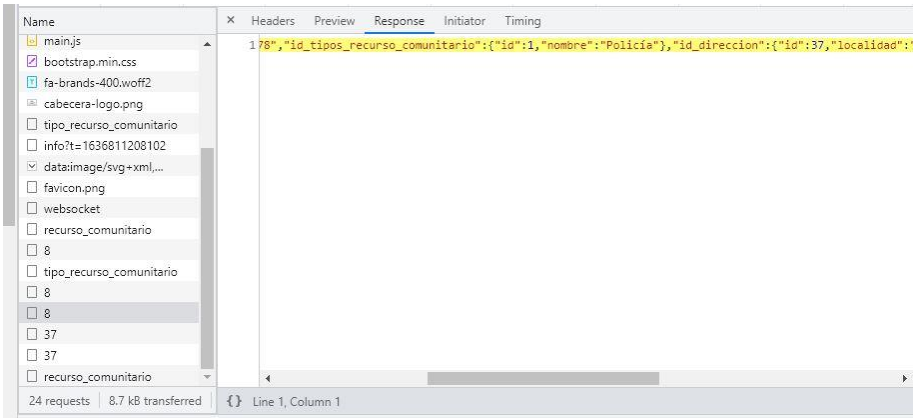


Ilustración 45. Respuesta obtenida de la prueba realizada al tratar de modificar un recurso comunitario.

Fuente: Captura de pantalla propia.

5.2.10.- Añadiendo validaciones a los formularios.

Una vez creados todos los formularios, se procedió a establecer la forma de validar los mismos. Las opciones para ello eran emplear *Reactive forms* o, bien, *Template-driven forms*.

Se revisó la documentación proporcionada a través del *issue* relativo a la propuesta de proyecto de fin de ciclo del repositorio central de *GitHub* sobre validación de formularios en *Angular*.

Asimismo, se buscó más información en internet para valorar cuál de las opciones de validación era más óptimo aplicar.

A continuación, se muestra una tabla comparativa de ambos sistemas de validación obtenida de la entrada *Template-driven forms de Angular* de la página web Java desde 0.

En ella podemos ver las diferencias entre uno y otro sistema desde diferentes aspectos como, por ejemplo, para qué tipo de formularios son más apropiados, la forma de aplicar las validaciones, el módulo de *Angular* a importar para poder emplearlos, la complejidad de la configuración, etc.

Teniendo todo esto en mente, se decidió finalmente aplicar la validación de formularios usando *Template-driven forms* debido a que los formularios ya habían sido planteados utilizando la directiva *[(ngModel)]* y, también, a la mayor facilidad que presentaban este tipo de formularios al ser configurados y a la hora de añadir avisos sobre las validaciones no cumplidas.

	Template-driven Forms	Reactive Forms
Idea principal	Fáciles de usar ya que están basados en plantillas	Más flexibles pero requieren más practica ya que están basados en modelos para proporcionarnos un mayor control.
Magníficos para	Implementar formularios sobre escenarios simples pero complicados para formularios complejos.	Manejar escenarios complejos.
Form validation	Basado en directivas añadidas en el código HTML (template) por lo que el código en el TypeScript del componente será mínimo	Basado en funciones por lo que tendrá más código en la clase de TypeScript del componente y menos en el template HTML.
Data binding	Utilizan el concepto de Two way data binding (datos bidireccionales) usando la sintaxis de [(NgModel)]	No se realiza ningún enlace de datos.
Testing	Pruebas unitarias más complicadas	Pruebas unitarias más sencillas
Módulo a importar	FormsModule	ReactiveFormsModule
Diseño Directivas Controles	Template - HTML	Componente - TypeScript
	ngForm	FormBuilder, formGroup, FormGroup
	ngSubmit	ngSubmit
	ngModel	formGroupName, formControlName, FormControl
Validaciones	•Directiva *ngIf •Directivas (MinLengthValidator, MaxLengthValidator, etc.) •Atributos HTML + CSS	• Validators en el componente (min, max, required, maxLength, etc.). • CSS
Configuración	•Menos explícita, creada en la clase por directivas.	• Más explícita, creada en la clase del componente TS
Data model	•Desestructurado	• Estructurado
Mutability	•Immutable	• Mutable

Ilustración 46. Comparativa entre Template-driven forms y Reactive forms.

Fuente: Template-driven forms de Angular (Java desde 0).

Comenzamos comprobando que tenemos importado el módulo *FormsModule* en el apartado de *imports* del archivo *app.module.ts*, de no ser así, debemos realizar la importación del mismo, ya que, es necesario para poder configurar las validaciones de los formularios con el método seleccionado para ello.

La comunicación entre los valores de la clase o interfaz de cada entidad y el formulario de nuestro *template HTML* se realizará mediante *ngModel*. De la manera que se observa en el esquema siguiente, obtenido también del artículo ya mencionado de la web Java desde 0.

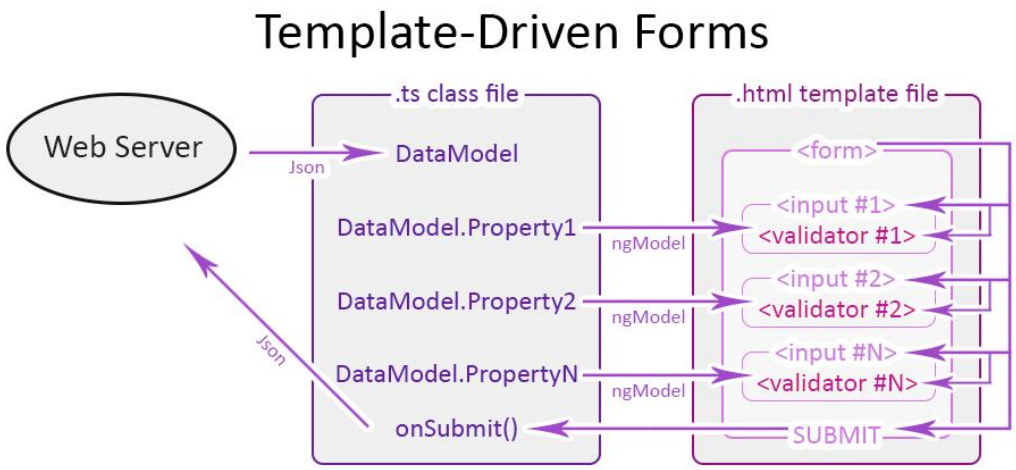


Ilustración 47. Esquema funcionamiento Template-driven forms de Angular.

Fuente: Template-driven forms de Angular (Java desde 0).

La mejor manera de explicar cómo realizar la configuración de las validaciones de un formulario utilizando este sistema, es cogiendo el formulario de creación o modificación de alguna entidad como referente. En este caso, explicaremos dichas configuraciones a través del formulario de modificación de la entidad

Clasificacion_Alarma, creado dentro de la vista del componente *detalles-clasificacion-alarma.*

Debemos tener en conocimiento que, la forma de realizar las validaciones es muy similar para todos los formularios creados, por tanto, los pasos que se van a detallar en este apartado de la memoria se pueden extrapolar para realizar la validación del resto.

Abrimos el archivo del *template* del componente *detalles-clasificacion-alarma* y declaramos una variable que referencie a la directiva *ngForm* que, se creó y adjuntó automáticamente a la etiqueta *<form></form>*, al importar el módulo *FormsModule*.

Actualizamos la etiqueta del formulario y le añadimos una referencia a la directiva *ngForm*, como podemos ver en la imagen:

Ilustración 48. Añadiendo la directiva ngForm al formulario de modificación de una clasificación de alarma.

Fuente: Captura de pantalla propia.

Después, asociamos cada *input* con la directiva *ngModel*, creando una variable para ello, pero **el nombre de dicha variable debe coincidir con el del atributo *name* del *input* correspondiente para que las validaciones funcionen bien**.

La directiva *ngModel* en un *input* verifica el estado de dicho *input*. De esta forma, sabemos si el usuario tocó el control, si el valor cambió o si el valor se volvió

inválido. *Angular* establece clases *CSS* especiales en el elemento de control para reflejar el estado, como se muestra en la siguiente tabla.

Estado	Clase si es cierto	Clase si es falso
Se ha visitado el control.	`ng-touched`	`ng-untouched`
El valor del control ha cambiado.	`ng-dirty`	`ng-pristine`
El valor del control es válido.	`ng-valid`	`ng-invalid`

Ilustración 49. Estados input directiva ngModel de Angular.

Fuente: Template-driven forms de Angular (Java desde 0).

Por otro lado, también debemos añadir a cada *input* las validaciones que deseemos aplicar como son el *minlength*, el *maxlength*, un *pattern* y/o la propiedad *required*.

El siguiente paso, para realizar las comprobaciones y ver si se cumplen las validaciones indicadas en cada campo, es aplicar distintos **ngIf* que evalúen el estado de los campos de nuestro formulario.

Finalmente, podemos deshabilitar el envío del formulario si este no cumple con todas las validaciones requeridas.

La configuración de las validaciones del formulario del componente *detalles-clasificacion-alarma* quedaría como se aprecia en la imagen.

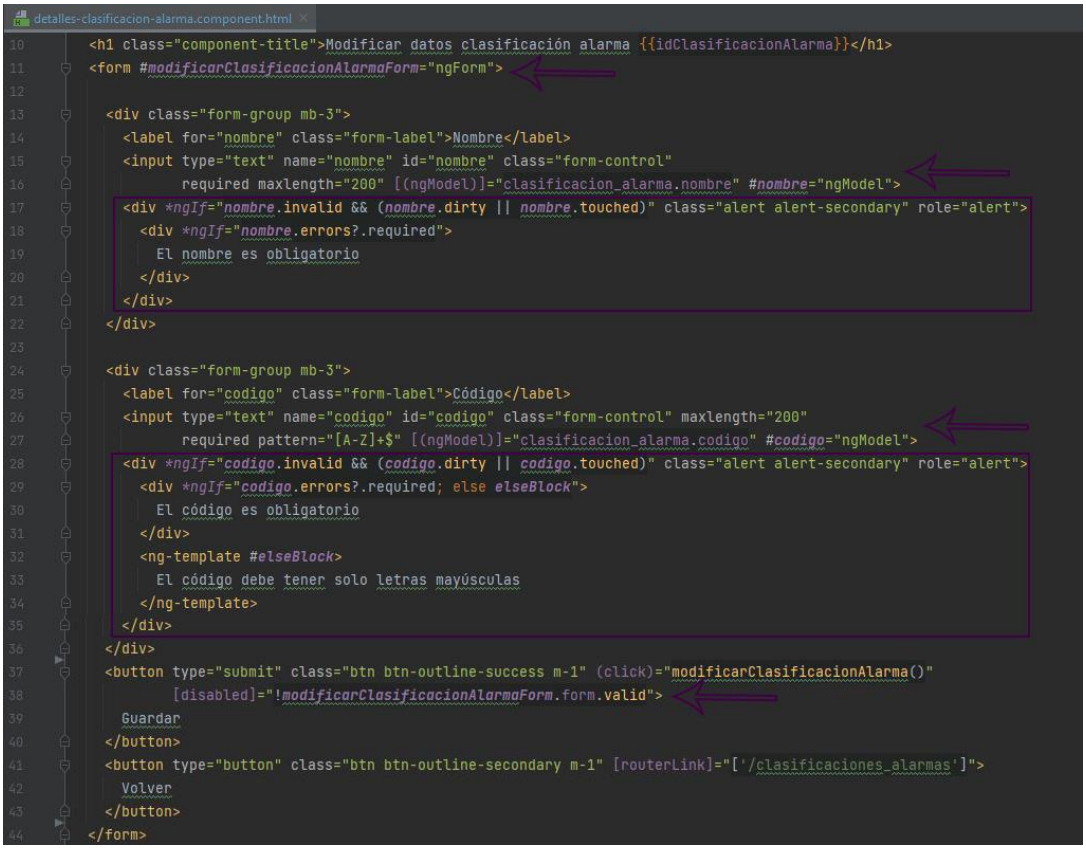


Ilustración 50. Aplicando validaciones usando Template-driven forms al formulario de modificación de la entidad Clasificacion_Alarma.

Fuente: Captura de pantalla propia.

Como se puede apreciar en la imagen las alertas con los errores cometidos al rellenar un campo, algunos botones y campos del formulario contienen clases con estilos de *Bootstrap*, versión 5.1.1, pero la aplicación de estilos al cliente web se explicará en el próximo apartado de la memoria.

No obstante, antes de terminar el apartado actual es de importancia mencionar que las validaciones de ciertos campos se han realizado aplicando la lógica, como es el caso de la validación de un email, un código postal, un nombre propio, entre otros campos, pero hay otros para los cuales se han aplicado validaciones

que, deberán revisarse en futuros proyectos de fin de ciclo, debido a que representan códigos de ciertos tipos de entidades y, a fecha de realización del presente trabajo, se desconocían los requisitos que debían cumplir.

5.2.11.- Creación del *header*, el *footer* y aplicación de estilos a las vistas de cada componente.

Antes de comenzar a explicar cómo se crearon el *header*, el *footer* y se aplicaron los estilos al cliente web de *Angular*, empezaremos hablando de la creación del logo para la aplicación y de cómo se decidió el nombre para la misma.

El tipo de logotipo elegido es un imagotipo o, lo que es lo mismo, la unión de una representación gráfica o símbolo, que representa el concepto de la aplicación de teleasistencia, junto con una parte tipográfica correspondiente al nombre y el eslogan de dicha aplicación.

La elección del nombre se realizó empleando la técnica de los acrónimos. Un acrónimo es una clase de sigla cuya pronunciación se realiza del mismo modo que una palabra. Para el caso que nos compete, de la palabra teleasistencia resultó el acrónimo *TLA*.

La tonalidad del imagotipo se realizó con tonos pastel dado que este tipo de tonalidades transmiten sensación de calma y bienestar.

El diseño del mismo se realizó utilizando la herramienta online *Hatchful*, la cual permite realizar diseños de logotipos de manera gratuita y sin conflictos de

propiedad intelectual o derechos de autor. A continuación, se muestra una imagen del mismo.

Ilustración 51. Logo del cliente web desarrollado.

Fuente: Elaboración propia a partir de la herramienta Hatchful.

En lo que se refiere al diseño visual de la aplicación, este se realizó empleando *Bootstrap* en su versión 5.1.1, para lo cual se tuvo que importar el *CDN* correspondiente de *Bootstrap* en el fichero */src/styles.scss* y, también, hubo que añadir el *CDN* correspondiente al *JavaScript* del mismo al final del elemento *<body></body>* de la vista del fichero */src/index.html*. Dentro del elemento *<head></head>* de este fichero también se añadieron el *CDN* de las fuentes empleadas de *Google Fonts* y el de *Font Awesome* para poder emplear los iconos proporcionados por esta última herramienta.

Un *CDN* (*Content Delivery Network* o *Red de Distribución de Contenidos*) permite almacenar en el caché del navegador contenido para, de esa forma, mejorar el rendimiento un sitio web utilizando los recursos proporcionados a través de este contenido.

Asimismo, se añadieron ciertos estilos con *SCSS* a los componentes y formularios de *Bootstrap* añadidos a la aplicación para personalizarla a nuestro gusto.

A la hora de crear tanto el *header* como el *footer* se estableció un componente para cada uno de ellos y dentro de la vista de los mismos se aplicó el *HTML* y *Bootstrap* deseado. Para añadirlos a cada pestaña o vista de la aplicación, solo hubo que incluir el selector de dichos componentes dentro del componente de la vista del *app-root* que se encarga de ir cargando la vista de cada uno de los componentes de la aplicación web.

app.component.html

```
1  <app-header></app-header>
2  <router-outlet></router-outlet>
3  <app-footer></app-footer>
```

Ilustración 52. Añadiendo el header y el footer a todos los componentes del cliente web.

Fuente: Captura de pantalla propia.

Para cada componente creado se añadieron clases de *Bootstrap* que permiten establecer unos estilos predefinidos y, finalmente, dentro del archivo */src/styles.scss* se personalizaron dichos estilos. Además, se añadieron *media*

queries dentro de este mismo archivo para que la aplicación fuese *responsive*, es decir, adaptable al tamaño de pantalla de diferentes dispositivos.

A continuación, encontramos una serie de hipervínculos con enlaces a diferentes vídeos en los cuales se pueden ver los estilos aplicados sobre la web y su diseño *responsive*.

Vídeo explicativo diseño de la aplicación de teleasistencia

Vídeo explicativo diseño responsive de la aplicación de teleasistencia

5.2.12.- Solventado los problemas con el *login* y aplicando una alternativa funcional.

A la hora de querer plantear lo solicitado en la propuesta de proyecto para el *login*, es decir, gestionar las peticiones para poder loguearse o desloguearse en la aplicación, además de guardar el *token* generado tras realizar dicho *login* por el sistema de autentificación, basado en *OAuth 2.0* de *Google*, surgieron una serie de complicaciones.

OAuth 2.0 de *Google* es un mecanismo de autenticación y autorización que nos permite usar las *APIs* de *Google*, como *Gmail*, para loguearnos en un sitio web.

El principal problema era que, al acceder a la ruta correspondiente del servidor para realizar el *login*, es decir, a la ruta *http://localhost:8000/login/* y pulsar el botón Login con Google, nos aparecía el siguiente error en pantalla y no se podía realizar dicho *login* a la aplicación.

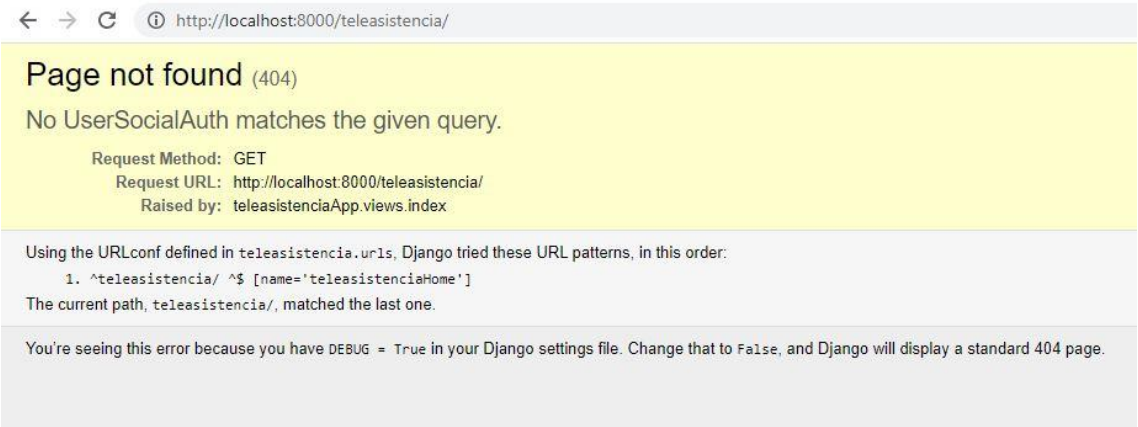


Ilustración 53. Error devuelto por el servidor al tratar de realizar el login a la aplicación de teleasistencia.

Fuente: Captura de pantalla propia.

Por tanto, no había manera de poder realizar las peticiones de *login/logout*, ni de guardar el *token* de acceso devuelto.

Dentro del sistema de *OAuth 2.0* de *Google,* un token es un identificador único que se le asigna a cada usuario al acceder a la aplicación y lo identifica dentro de la misma.

Para no dejar este punto de la propuesta sin realizar, se optó por crear un componente con una pantalla para loguearse en la aplicación e implementar un servicio de *login* con tres métodos. Uno para hacer *login*, otro para el *logout* y, por último, un método para comprobar si el usuario está logueado.

La función de estos métodos consiste en emplear *localStorage* para definir un atributo llamado *token*, el cual debería guardar el *token* devuelto de la petición de *login*, pero como dicha petición no se estaba realizando adecuadamente por parte del servidor, se definió el valor del atributo *token* como usuario.

El objeto *Storage* (*API* de almacenamiento web), nos permite almacenar datos de forma local en el navegador y sin tener que conectarnos a la base de datos. Una de sus propiedades es precisamente *localStorage*, mediante la cual podemos definir un nuevo *item*, obtenerlo y eliminarlo.

Creamos el servicio *login* dentro de la carpeta de servicios del proyecto empleando el comando *ng g service login* y añadimos los métodos mencionados anteriormente, codificándolos tal y como se observa en la captura de pantalla.

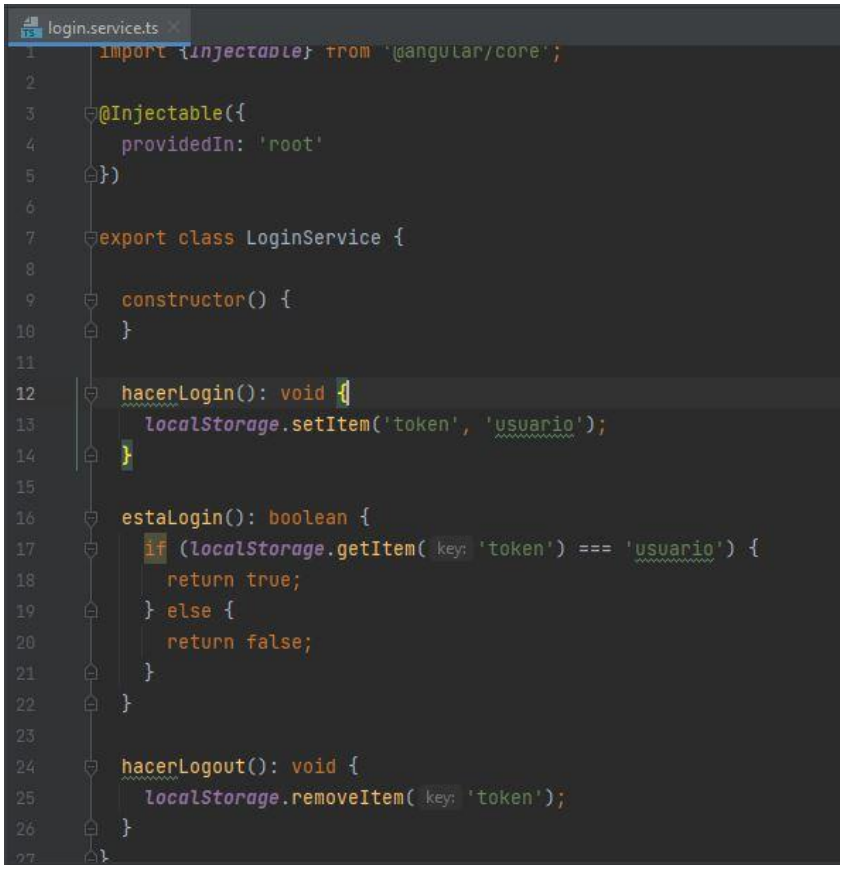


Ilustración 54. Servicio creado para gestionar el login.

Fuente: Captura de pantalla propia.

Por otro lado, creamos una nueva interfaz y una nueva clase para guardar los campos que recogerá el formulario de *login*, el nombre de usuario y la contraseña.

A continuación, abrimos el controlador del componente *pantalla-login* y definimos dos variables *login* del tipo *ILogin* y *estaLogin* de tipo boolean. En el contructor creamos una variable para el servicio *Title*, otra para el servicio de *login* y otra para la clase *Router*.

Dentro del método *ngOnInit()* inicializamos la variable *login* como un nuevo objeto de la clase *Login* y añadimos el título de la pestaña del navegador.

Para poder inicializar la propiedad *estaLogin* para que recoja el valor booleano devuelto por el método *estaLogin()* del servicio de *login* creado, debemos implementar el método ***DoCheck*** en el controlador del componente.

Este método se ejecuta cada vez que se produce algún cambio en el componente o en la aplicación de Angular, ya que, al tratar de inicializar dicha variable en el método *OnInit*, el valor de la misma no se actualizaba de forma inmediata, sino que era necesario recargar la página.

Finalmente, creamos dos métodos uno para hacer *login* y otro para hacer *logout* dentro de los cuales simplemente utilizamos el servicio de *login* para llamar a los métodos que hemos creado para gestionar ambas tareas. Al realizar un *login*, redirigimos a la ruta /inicio, en cambio, al hacer *logout* redirigimos a la ruta */login*.

El controlador del componente *pantalla-login* quedaría como se observa en la imagen:

```typescript
// pantalla-login.component.ts
export class PantallaLoginComponent implements OnInit, DoCheck {
  public login: ILogin;
  public estaLogin: boolean;

  constructor(private titleService: Title, private loginService: LoginService, private router: Router) { }

  ngOnInit(): void {
    this.titleService.setTitle('Login');
    this.login = new Login();
  }

  ngDoCheck(): void {
    this.estaLogin = this.loginService.estaLogin();
  }

  hacerLogin(): void {
    this.loginService.hacerLogin();
    console.log(this.login);
    this.router.navigate( commands: ['/inicio']);
  }

  hacerLogout(): void {
    this.loginService.hacerLogout();
    this.router.navigate( commands: ['/login']);
  }
}
```

Ilustración 55. Configuración controlador del componente pantalla-login.

Fuente: Captura de pantalla propia.

Dentro de la vista de este componente, creamos un formulario para loguarse utilizando nombre de usuario y contraseña, con un botón que al hacer clic sobre él llama al método *hacerLogin()* del controlador. Dicho formulario se muestra cuando la variable *estaLogin* sea igual a *false*.

Además, dentro de la vista, también creamos un cuadro de diálogo que se muestra si la variable *estaLogin* es *true* y nos permite continuar con la sesión ya iniciada en la aplicación o desloguearnos para iniciar sesión con otro usuario.

Este cuadro de diálogo tiene dos botones, el botón cancelar que nos redirige de nuevo a la página de inicio y el botón *logout*, el cual llama al método *hacerLogout()* del controlador.

pantalla-login.component.html

```
<div *ngIf="!estaLogin" class="container h-100">
  <div class="h-100 justify-content-center align-items-center">
    <form class="col-md-12 col-lg-9 col-sm-12" #formLogin="ngForm">
      <div class="formulario shadow bg-white">
        <div class="row">
          <div class="col-md-6 col-lg-6 d-flex justify-content-center align-items-center">
            <div class="formularioLeft text-center">

              <h1>Iniciar sesión</h1>
              <img class="mb-4" src="../../../assets/login-user.png" alt="Logo usuario" width="120"
                   height="120">
              <div class="form-group position-relative mb-4">
                <input type="text" class="form-control border-0 rounded-0 shadow-none" name="username"
                       id="username" placeholder="Nombre de usuario" [(ngModel)]="login.username" #username="ngModel"
                       required minlength="4" maxlength="20" pattern="^[a-zA-Z0-9](_(?!(\.|_))|\.(?!(_|\.))|[a-zA-Z0-9]){2,18}[a-zA-Z0-9]$">
                <i class="fas fa-user"></i>
              </div>
              <div *ngIf="username.invalid && (username.dirty || username.touched)"
                   class="alert alert-secondary col-md-12 col-lg-12 col-sm-12" role="alert">
                <div *ngIf="username.errors?.required">
                  El nombre de usuario es obligatorio
                </div>
                <div *ngIf="username.errors?.minlength">
                  El nombre de usuario debe tener al menos 4 caracteres
                </div>
                <div *ngIf="username.errors?.pattern">
                  Introduzca un nombre de usuario válido
                </div>
              </div>
              <div class="form-group position-relative mb-4">
                <input type="password" class="form-control border-0 rounded-0 shadow-none"
                       name="password" id="password" placeholder="Contraseña" [(ngModel)]="login.password"
                       #password="ngModel" required minlength="8" maxlength="16"
                       pattern="^(?=\w*\d)(?=\w*[A-Z])(?=\w*[a-z])\S{8,16}$">
                <i class="fas fa-key"></i>
              </div>
              <div *ngIf="password.invalid && (password.dirty || password.touched)"
                   class="alert alert-secondary" role="alert">
                <div *ngIf="password.errors?.required; else elseBlock">
                  La contraseña es obligatoria
                </div>
                <ng-template #elseBlock>
                  La contraseña debe tener al menos un digito, al menos una minúscula y al menos una mayúscula
                </ng-template>
                <div *ngIf="password.errors?.minlength" class="pt-1">
                  La contraseña debe tener entre 8 y 16 caracteres
                </div>
                </div>
              </div>
              <button type="button" class="btn btn-success btn-block shadow border-0 py-2 text-uppercase"
                      [disabled]="!formLogin.form.valid" (click)="hacerLogin()">
                Login
              </button>
            </div>
          </div>
          <div class="col-md-6 col-lg-6">
            <div
              class="formularioRight position-relative d-flex justify-content-center flex-column align-items-center text-center
              p-5 text-white">
              <h2 class="position-relative px-4 pb-3 mb-4">Bienvenido</h2>
              <p>Por favor, introduzca sus datos para poder acceder al sistema</p>
            </div>
          </div>
        </div>
      </div>

    </form>
  </div>
</div>
```

Ilustración 56. Formulario de login de la vista del componente pantalla-login.

Fuente: Captura de pantalla propia.

```
<div *ngIf="estaLogin" class="modal modal-alert position-static d-block py-5" tabindex="-1" role="dialog"
     id="modalChoice">
  <div class="modal-dialog" role="document">
    <div class="modal-content rounded-4 shadow">
      <div class="modal-body p-4 text-center">
        <h5 class="mb-0">Confirmar</h5>
        <p class="mb-0 mt-3">Actualmente ha iniciado sesión como Lucía González Martín, necesita salir antes de volver a
          entrar con un usuario diferente.</p>
      </div>
      <div class="modal-footer flex-nowrap p-0">
        <button type="button" class="btn btn-lg btn-link fs-6 text-decoration-none col-6 m-0 p-1 rounded-0 border-right"
                (click)="hacerLogout()"><strong>Logout</strong></button>
        <button type="button" class="btn btn-lg btn-link fs-6 text-decoration-none col-6 m-0 p-1 rounded-0"
                data-bs-dismiss="modal" [routerLink]="['/inicio']">Cancelar
        </button>
      </div>
    </div>
  </div>
</div>
```

Ilustración 57. Cuadro de diálogo de la vista del componente pantalla-login.

Fuente: Captura de pantalla propia.

Seguidamente, definimos la ruta que cargará este componente en el archivo *app-routing.module.ts*.

Para restringir el acceso a personas no logueadas, creamos dentro de la carpeta de servicios una nueva guarda del tipo *CanActivate*, empleando para ello el comando *ng g guard login*.

En el apartado 5.2.6. del presente documento ya se explicó que eran las guardas de *Angular*. Sin embargo, ahora vamos a centrarnos en el caso concreto de las guardas *CanActivate* cuya función consiste en permitir el acceso de un usuario a determinadas páginas.

Para configurar la misma, debemos crear en el constructor dos variables, una del tipo del servicio de *login* y otra del tipo del servicio de *Router* de *Angular*.

Dentro del método *canActivate* que devuelve un booleano, comprobamos utilizando el método *estaLogin()* del servicio de *login*, si la persona que intenta acceder a las rutas de consulta, modificación y creación de los elementos de las

distintas entidades está logueada en el sistema. En caso de no estarlo, redirigimos a la ruta */login* y retornamos false, impidiendo el acceso al resto de rutas de la aplicación. Si está logueada retornamos true y, en este caso, si podría acceder a las distintas pestañas del cliente web.

Ilustración 58. Configuración de la guarda de tipo canActivate del login.

Fuente: Captura de pantalla propia.

Para aplicar dicha guarda a las rutas que queremos proteger, debemos acceder al archivo *app-routing.module.ts*, en el que hemos definido las distintas rutas de nuestra aplicación, importar la guarda creada y añadir un campo a las rutas que deseemos llamado canActivate con el guard.

Ilustración 59. Aplicación de la guarda canActivate a la ruta tipos_alarmas.

Fuente: Captura de pantalla propia.

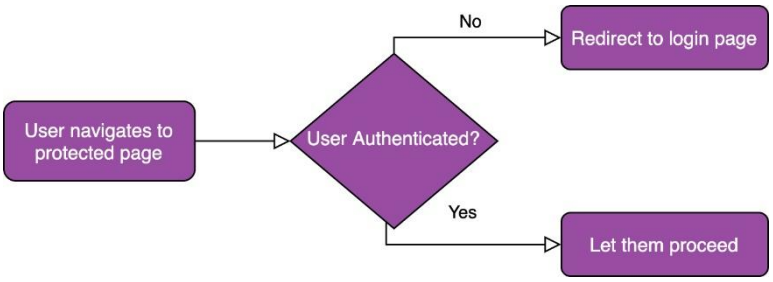


Ilustración 60. Esquema funcionamiento guarda canActivate de Angular.

Fuente: Angular Authentication: Securing Routes with Route Guards (Jacob Neterer).

Finalmente, para controlar el *login*/*logout* desde la cabecera de la web, hubo que crear un nuevo componente con un botón para gestionar el *login* si el usuario aún no ha iniciado sesión y un * * que muestra el nombre del usuario logueado, una imagen de perfil y un enlace para hacer *logout*. Este componente hubo que insertarlo dentro del componente de la cabecera de la web.

La configuración de la vista del componente se muestra en la imagen que vemos a continuación:

botones-login.component.html

```
<button *ngIf="!estaLogin" type="button" class="btn btn-outline-secondary me-2" [routerLink]="['/login']">Login</button>


<div *ngIf="estaLogin" class="dropdown text-end">
  <a href="#" class="d-block link-dark text-decoration-none dropdown-toggle" id="dropdownUser1" data-bs-toggle="dropdown" aria-expanded="false">
    <p class="user">Lucia González Martin</p>
    <img src="../../../assets/user.png" alt="mdo" width="40" height="40" class="rounded-circle">
  </a>
  <ul class="dropdown-menu text-small" aria-labelledby="dropdownUser1">
    <li><a class="dropdown-item" (click)="hacerLogout()">Logout</a></li>
  </ul>
</div>
```

Ilustración 61. Vista del componente botones-login.

Fuente: Captura de pantalla propia.

Dentro del controlador de dicho componente, llamado *botones-login*, se creó una variable booleana de nombre está *estaLogin* y se inicializó dentro del método *DoCheck*. Asimismo, se creó un método para hacer *logout* igual que en el controlador del componente *pantalla-login*.

La configuración del controlador quedaría de la siguiente forma:

```typescript
@Component({
  selector: 'app-botones-login',
  templateUrl: './botones-login.component.html',
  styleUrls: ['./botones-login.component.scss']
})

export class BotonesLoginComponent implements OnInit, DoCheck {
  public estaLogin: boolean;

  constructor(private loginService: LoginService, private router: Router) {
  }

  ngOnInit(): void {
  }

  ngDoCheck(): void {
    this.estaLogin = this.loginService.estaLogin();
  }

  hacerLogout(): void {
    this.loginService.hacerLogout();
    this.router.navigate( commands: ['/inicio']);
  }
}
```

Ilustración 62. Configuración controlador del componente botones-login.

Fuente: Captura de pantalla propia.

Para que quede totalmente claro el funcionamiento del *login* de nuestro cliente, seguidamente encontramos un enlace a un vídeo dónde se puede ver el funcionamiento del mismo.

Vídeo explicativo login de la aplicación de teleasistencia

Actualmente se puede acceder introduciendo cualquier nombre de usuario y contraseña, ya que, las peticiones al servidor no se están realizando, pero en un futuro habrá que prevenir que los distintos usuarios tengan el mismo nombre de usuario para evitar problemas de acceso y comprobar si existe en el sistema un usuario con el nombre de usuario y la contraseña introducidos para realizar el *login*.

5.2.13.- Problemas con el intercambio de recursos de origen cruzado (*CORS*) y otros fallos encontrados.

El intercambio de recursos de origen cruzado (*CORS*) es un mecanismo que utiliza encabezados *HTTP* adicionales para permitir que un agente de usuario, es decir, un programa de ordenador que representa a una persona como puede ser un navegador web, acceda a recursos específicos desde un servidor, en un origen (dominio) diferente al que pertenece. Dicho agente de usuario realiza una solicitud *HTTP* de origen cruzado cuando solicita un recurso de un dominio, protocolo o puerto diferente al del documento en el que se creó.

En el caso que nos ocupa, la solicitud de origen cruzado se genera al intentar acceder desde el código *TypeScript* del *front-end* de la aplicación web de *Angular*, ubicada en el dominio *http://localhost:4200*, a un recurso del servidor que se localiza en el dominio *http://localhost:8000/api-rest/tipo_alarma*. La respuesta que obtenemos es del tipo *XMLHttpRequest*, avisándonos de que el acceso al recurso del servidor ha sido bloqueado por la política de *CORS*, ya que, no hay un encabezado *'Access-Control-Allow-Origin'* presente en el recurso solicitado o, lo que es lo mismo, en el servidor no se han configurado de forma correctas las cabeceras *HTTP* para evitar estos problemas de *CORS*.

XMLHttpRequest es un objeto *JavaScript* diseñado por *Microsoft* y adoptado por *Mozilla*, *Apple* y *Google*. Actualmente es un estándar *W3C*. Su función es proporcionar una forma sencilla de obtener información de una *URL* sin tener que volver a cargar toda la página.

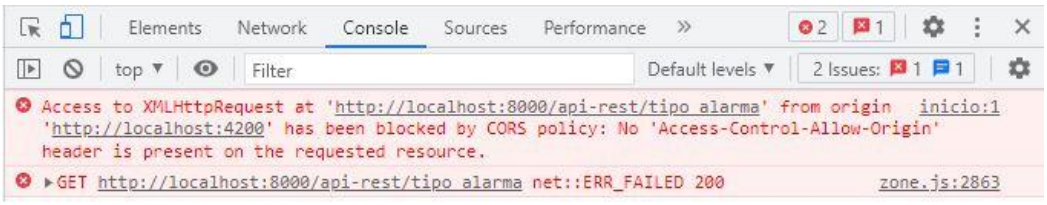


Ilustración 63. Error de CORS al tratar de hacer una petición desde el cliente web al servidor de teleasistencia.

Fuente: Captura de pantalla propia.

Como desde cliente web era imposible solucionar dicho problema, la forma que se encontró de poder realizar las peticiones fue instalar en el navegador web *Google Chrome* un *plugin* para deshabilitar las *CORS*.

El plugin instalado fue *Moesif Origin & CORS Changer* que permite enviar solicitudes entre dominios, anular el origen de la solicitud y los encabezados *CORS*. Además, puede anular el encabezado *Request Origin* y también tener *Access-Control-Allow-Origin* establecido en *.

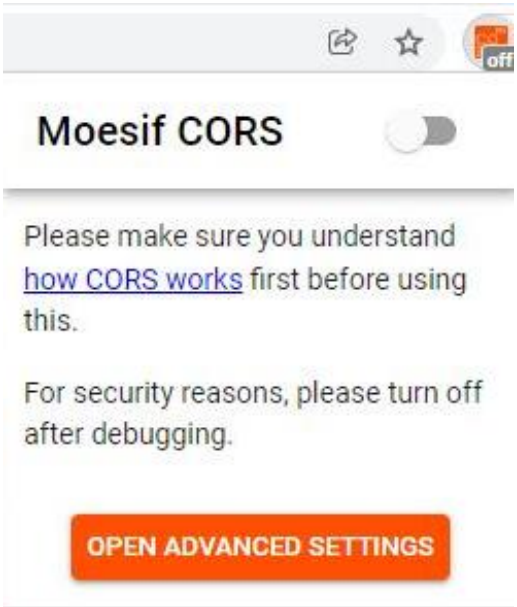


Ilustración 64. Moesif Origin & CORS Changer plugin.

Fuente: Captura de pantalla propia.

Instalamos dicho *plugin* y volvemos a probar a realizar una petición al servidor, esta vez ya si obtenemos la respuesta deseada y podemos realizar las distintas peticiones *HTTP* deseadas desde nuestro cliente.

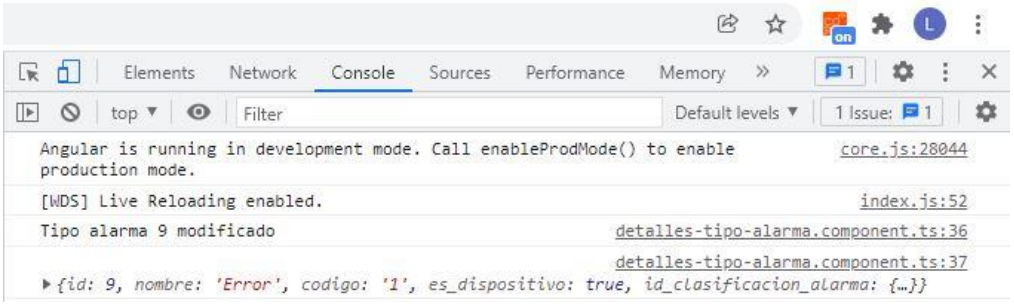


Ilustración 65. Resultado de realizar una petición de modificación de un elemento de la entidad Tipo_Alarma después de instalar el plugin para deshabilitar las CORS.

Fuente: Captura de pantalla propia.

Antes de dar paso al último apartado de la descripción técnica, queremos comentar que nos dimos cuenta realizando las pruebas finales de funcionamiento, de que, para la entidad *Users* a la hora de realizar una petición para obtener dicho recurso del servidor o para añadir un nuevo elemento al mismo, los atributos definidos en *Postman* para tales peticiones son distintos en el *GET* y en el *POST*.

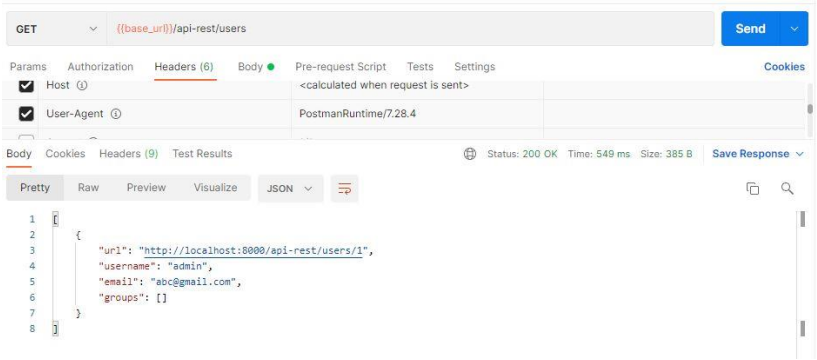


Ilustración 66. Datos que recibe la entidad Users al realizar un GET.

Fuente: Captura de pantalla propia.

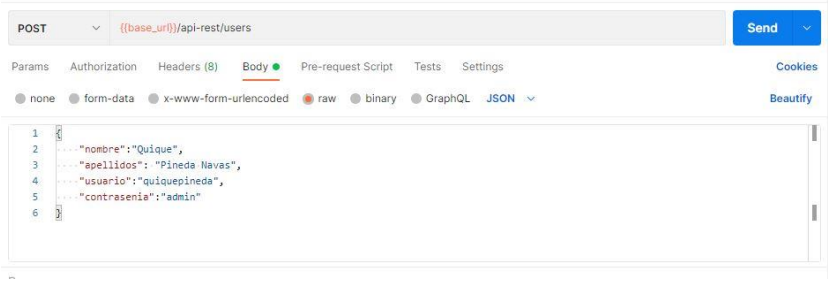


Ilustración 67. Datos que recibe la entidad Users al realizar un POST.

Fuente: Captura de pantalla propia.

5.3.- Otras posibles soluciones técnicas para la creación del cliente web.

Este apartado se centrará en justificar la decisión de elegir *Angular* como *framework* para desarrollar el cliente web solicitado, comparando el mismo con *Vue* y *React* que, también, habrían sido opciones válidas para diseñar la interfaz de usuario de la aplicación de teleasistencia.

Hablaremos de la utilidad de cada uno de estas herramientas de desarrollo web, veremos las ventajas y desventajas de cada una de ellas desde el punto de vista técnico, de sencillez a la hora de programar el código, etc.

En el caso de *Angular*, es una herramienta de desarrollo de *JavaScript* creada por *Google*. El objetivo de *Angular* es facilitar el desarrollo de aplicaciones web *SPA*, así como proporcionarnos herramientas para trabajar con los componentes del sitio web de una manera mejor y más sencilla.

Otro objetivo de *Angular* es la separación completa de *front-end* y *back-end* en una aplicación web.

Una aplicación web *SPA* (*Single Page App*) creada con *Angular* se basa en una única página web en la que se navega por las secciones y páginas de la aplicación, además de cargar datos, de forma dinámica, casi instantánea, realizando llamadas al servidor asíncrono (*back-end* con *API Rest*), sin necesidad de actualizar la página en ningún momento.

En la tabla que se puede ver a continuación, vemos algunas de las ventajas y desventajas de *Angular*.

Ventajas de Angular	Inconvenientes de Angular
Escalable a equipos de desarrollo grandes.	No es tan flexible como otros frameworks y librerías.
Gracias a su estructura, los errores difícilmente pueden ocasionar que la aplicación falle.	Aprender TypeScript es un requisito obligatorio.
Tiene un el mayor surtido de funciones *"out of box"* ya que es un *framework*.	Se necesita aprender nuevos conceptos (e.g. Modulo, Directiva, Componente, Servicio, Inyección de dependencias)
MVVM (Model-View-View-Mode) permite a muchos desarrolladores trabajar en la misma sección de código y usar el mismo *dataset*.	Angular usa una manipulación directa del DOM, mientras React o Fue utilizan un DOM virtualidad.
Extensa documentación acerca de cualquier versión de Angular.	Curva de aprendizaje superior a Vue y React.
Posee una gran comunidad que proporciona a Angular soporte y constantes actualizaciones.	Librería más pesada de todas (+500KB)

Ilustración 68. Tabla ventajas e inconvenientes de Angular.

Fuente: Estudio de la actualización de un portal bancario SPA para la gestión de clientes (Luis Iván Rosales Carrera).

Respecto a *Vue*, es *framework* de *JavaScript* de código abierto que nos permite crear interfaces de usuario con mucha facilidad, también del tipo *SPA*. La curva de aprendizaje para este es relativamente baja, aunque se debe estar muy familiarizado con *JavaScript* y saber cómo manejar *callbacks*, promesas y objetos, entre otros temas.

Una de las características más importantes de *Vue* es trabajar con componentes. En pocas palabras, un componente de *Vue* es un elemento basado en código reutilizable, dentro del cual podemos encontrar etiquetas *HTML*, estilos *CSS* y código *JavaScript*.

Esta característica bajo mi criterio personal, hace que la separación de la vista, los estilos y el controlador dentro de un componente de *Vue* no sea tan clara como en el caso de los componentes de *Angular* que almacenan cada uno de estos elementos en archivos diferentes dentro de la carpeta del componente correspondiente.

En la imagen vemos las ventajas y desventajas de *Vue*:

Ventajas de VueJS	Inconvenientes de VueJS
VueJS tiene una fácil curva de aprendizaje	Exceso de flexibilidad y puede tener problemas al integrarse en grandes proyectos puesto que todavía no hay experiencia útilizandose en grandes entornos de trabajo
No es necesario utilizar librerías de terceros ya que este potente framework viene cargado de funcionalidades.	Ya que el mayor desarrollo de VueJS comenzó en china, muchos elemento y descripciones todavía no están completamente traducidas por lo que aún tiene una complejidad parcial en algunas etapas del desarrollo.
Es una buena solución ya que sus pequeños componentes se integran de forma fácil a la infraestructura.	Vue tiene la proporción más pequeña en cuanto a ofertas de trabajo según la plataforma StackOverFlow
Posee un tamaño inferior a Angular o ReactJS y aun así continua manteniendo su velocidad y flexibilidad que permite alcanzar un rendimiento mucho mejor en comparación con otros framework.	De momento la comunidad no es tan grande y el potencial de su comunidad reside en China

Ilustración 69. Tabla ventajas e inconvenientes de Vue.

Fuente: Estudio de la actualización de un portal bancario SPA para la gestión de clientes (Luis Iván Rosales Carrera).

Centrándonos en *React*, es una biblioteca de *JavaScript* desarrollada por *Facebook* y diseñada para ayudarnos a crear una *SPA*, cuyo propósito específico es intentar facilitar la tarea de desarrollo de la interfaz de usuario.

Este *framework* utiliza un paradigma conocido como programación orientada a componentes. Estos componentes se representan como clases heredadas de la clase de *Component* cuyo único requisito específico es definir un método render que define el contenido del mismo.

Estos componentes se definen mediante una sintaxis especial llamada *JSX* que permite escribir etiquetas *HTML* en *JavaScript* para mejorar la expresividad del código.

Seguidamente, encontramos una tabla con ciertas ventajas e inconvenientes de *React*:

Ventajas de ReactJS	Inconvenientes de ReactJS
La sintaxis es mas fácil de aprender que en Angular	React es no ionizado, lo que significa que los desarrolladores a veces tienen demasiadas opciones
Simplifica el trabajo con el DOM, al utilizar un DOM virtual, permite optimizar el número de cambios antes de actualizar y renderizar el DOM real, y con esto permite la gestión rápida de los elementos del árbol DOM	Se necesita librerías de terceros para el almacenamiento de datos en React.
Es una librería Open Source actualizada por un gran conjunto de desarrolladores.	No hay demasiadas funciones ‘*out of box*’ por lo que tienes que crear tus propias librerías
Los desarrolladores pueden actualizar y volver a versiones antiguas fácilmente y sin poner demasiado en riesgo su código.	Unicamente posee una capa de vista por lo que para realizar llamadas asíncronas necesita librerías externas

Ilustración 70. Tabla ventajas e inconvenientes de React.

Fuente: Estudio de la actualización de un portal bancario SPA para la gestión de clientes (Luis Iván Rosales Carrera).

Todos estos *frameworks* mencionados anteriormente son de tipo *open source*, por tanto, desde el punto de vista económico no hay diferencias entre ellos. Asimismo, desde el punto de vista de la sencillez para el usuario final de la aplicación todos ellos dan como resultado aplicaciones *SPA* con un

funcionamiento bastante intuitivo según la forma en la que el programador haya estructurado la aplicación.

La decisión final que llevó a realizar la aplicación web del proyecto de teleasistencia con *Angular* fueron los conocimientos previos que se tenían sobre él, adquiridos en el módulo *"Desarrollo Web en Entorno Cliente"*, y también, la facilidad para organizar el código por componentes y servicios, separando las vistas *HTML,* de los estilos y de los controladores en cada componente.

6.- Medios utilizados.

Con la finalidad de conseguir los objetivos propuestos, se han empleado diferentes medios técnicos.

Por un lado, el *hardware* empleado para arrancar el servidor, en desarrollo por parte del equipo docente del IES "Valle del Jerte", y para crear el cliente web ha sido un ordenador portátil, modelo *HP Pavilion Notebook* con procesador Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz 2.59 GHz, sistema operativo Windows 10 Home de 64 bits y 12,0 GB de RAM. Asimismo, se ha necesitado conexión a una red *Wi-Fi*, tanto para descargar los recursos de *software* necesarios como para poner en marcha el servidor y poder hacer pruebas en el cliente.

Por otro lado, en cuanto a los programas, herramientas y tecnologías utilizadas para desarrollar el proyecto, podemos dividirlas en dos categorías:

- Tecnologías empleadas del lado del servidor. En primer lugar, fue necesario instalar *GitHub Desktop* con el fin de clonar la aplicación en nuestro equipo local.

  Para poder lanzar el servidor debimos instalar *Python*, un lenguaje de programación interpretado, multiparadigma y multiplataforma usado, principalmente, en *Big Data*, *AI* (Inteligencia Artificial), *Data Science*, *frameworks* de pruebas y desarrollo web.

  Además, ha sido necesario instalar el entorno de desarrollo *PyCharm* y el *framework Django*.

  Para comprobar que las peticiones al servidor funcionan correctamente se han realizado pruebas con *Postman*.

- Tecnologías empleadas del lado del cliente. A la hora de construir la interfaz web de la aplicación de teleasistencia, se ha usado el IDE *PhpStorm* para facilitar la escritura del código. Así como también se ha utilizado el *framework Angular* con su utilidad de *routing* por su sencillez a la hora de desplegar sitios web navegables y bien estructurados.

  Respecto al lenguaje de programación empleado para codificar el cliente, se ha empleado *TypeScript* que es lenguaje por defecto que utiliza *Angular*.

  Finalmente, para realizar el diseño de la web se usó *HTML*, *Bootstrap*, *CSS*, *Font Awesome* y *Google Fonts*. Cabe mencionar que el cliente web se ha planteado para correr en *Google Chrome* y que todas las pruebas de funcionamiento se han realizado en este navegador.

Todo el proceso de puesta en marcha de la aplicación está documentado mediante *Git* y subido a un repositorio remoto en *GitHub* para facilitar la tarea de seguir creando la aplicación a futuros alumnos que continúen con este proyecto como *PFC*.

El logotipo del proyecto se diseñó a través de la herramienta *Hatchful* y el póster publicitario con el *software Canva*.

Las imágenes de la web se obtuvieron de *Pexels*, una plataforma que provee fotografías y material de archivo libre de derechos de autor. Respecto a los iconos empleados para el diseño de la misma fueron recuperados de *Flaticon*, otra plataforma con una base de datos de iconos gratuitos editables.

En cuanto a la memoria y la presentación, se han realizado utilizando *Microsoft Word* y *Microsoft PowerPoint*, respectivamente.

Estos han sido los principales medios con los que hemos trabajado a la hora de hacer este proyecto, entre otras herramientas, plataformas o elementos que han sido de utilidad en la obtención de la documentación necesaria para elaborar el proyecto, su maquetación y la presentación del mismo.

7.- Conclusiones finales.

Atendiendo al objetivo general fijado al comienzo del trabajo que era inicializar un cliente de *Angular* que nos permitiese desarrollar una primera interfaz de usuario para la aplicación de teleasistencia, solicitada por parte del equipo

académico del ciclo de *APSD* del IES “San Martín”, la cual, una vez terminada a través de proyectos de futuros alumnos, deberá permitir poner en práctica y desarrollar las tareas que realizan los teleoperadores de los centros de atención, y tras haber desarrollado de forma detallada cada uno de los objetivos específicos y puntos de los que consta la propuesta de proyecto, podemos constatar que, aunque al principio se pensase que iba a ser complicado cumplir con todos los requisitos pedidos, al final se ha logrado crear un cliente web funcional, intuitivo y con un diseño general bastante atractivo para los usuarios.

En relación al primer objetivo específico, que sí recordamos consistía en realizar una página web utilizando la herramienta *Angular* con *routing* que tuviese una cabecera común para las distintas ventanas, una funcionalidad para realizar el *login* y el *logout*, una página principal con información del usuario logueado y una serie de pestañas que permitiesen acceder a los formularios de modificación y creación para las entidades indicadas, hemos visto que no ha sido posible cumplir con todos los aspectos del mismo como se había planteado en un primer momento.

La cabecera sí fue posible realizarla de manera adecuada, incluyendo en ella el logotipo de la aplicación, un componente con los botones para hacer *login* o *logout* y una barra de navegación que permite al usuario, una vez logueado en el sistema, acceder a las distintas rutas para consultar, modificar y crear nuevos elementos de las distintas entidades.

Sin embargo, en cuando al sistema de autentificación en la aplicación, hubo que realizar una serie de cambios, los cuales han sido explicados en el apartado correspondiente a la resolución de los problemas con el login de la presente memoria. Además, se diseñó una vista para el componente encargado de la gestión de acceso a la aplicación consistente en un formulario con los campos nombre de usuario y contraseña, aunque su funcionalidad no se ha terminado de aplicar. Aunque se ha dejado todo preparado para la rápida integración en futuros proyectos.

En la página principal hubo que concretar la información a mostrar en la misma, ya que, las peticiones para loguarse o desloguearse no estaban funcionando y no se podía incluir la información del usuario en esta.

Por otro lado, se encontraron una serie de fallos en el servidor que impidieron que los formularios de todas las entidades funcionasen, de lo cual se informó al profesorado encargado de la implementación del mismo, aunque en nuestro cliente web están dichos formularios correctamente codificados, tanto la vista de los mismos, con las validaciones de cada campo, como el controlador de cada componente empleado para gestionarlos.

Respecto al tema de las validaciones que acabamos de mencionar, resulta de interés dejar constancia de que, deberían acordarse los criterios que debe cumplir cada atributo de las distintas entidades, para así validar los campos de cada formulario de forma más fidedigna a los datos que finalmente serán recogidos, dado que, con ciertos campos tales como el código de las

clasificaciones de alarmas, no se sabía con exactitud que valores era posible almacenar para dicho atributo en la base de datos.

Centrándonos en el segundo objetivo propuesto, que consistía en organizar el cliente web en componentes y servicios, así como también, establecer unos estilos generales consistentes a lo largo de toda la aplicación, podemos decir que este se ha realizado de forma satisfactoria, logrando abordar todos los aspectos relativos a él.

Teniendo en cuenta el tercero de los objetivos específicos, referido a aplicar una solución web sencilla e intuitiva para los usuarios finales de la misma, tanto los profesores como los alumnos del ciclo de *APSD*, podemos concluir que el enfoque que se le ha dado a la aplicación hasta el momento hace que su uso resulte sencillo y sienta las bases del rumbo a seguir en cuanto a usabilidad de la aplicación web.

Finalmente, haciendo referencia al cuarto objetivo específico propuesto, consistente en documentar el paso a paso del desarrollo del cliente web de *Angular*, se han ido subido a *GitHub* los *commits* con los cambios y correcciones aplicados a lo largo del proyecto, usando para ello el *software* de control de versiones *Git*.

No obstante, se ha realizado una explicación más extensa y detallada del proceso seguido en el presente documento, el cual ofrece una visión global suficientemente clara, tanto de los antecedentes de dicho proyecto, como del

trabajo desarrollado y el enfoque posterior que seguirá la aplicación de teleasistencia.

8.- Evaluación personal.

Para finalizar este Trabajo de Fin de Ciclo quiero aportar mi opinión personal de manera global con el fin de reflejar tanto la aplicación de los conocimientos adquiridos a lo largo de la formación recibida en estos dos años como las dificultades encontradas durante su realización.

El cliente de *Angular* que ha sido elaborado incorpora navegación a través de diferentes ventanas empleando *routing*, formularios para modificar/crear elementos de las distintas entidades y funcionalidad para realizar el *login* en la aplicación. Se han empleado fundamentalmente los conocimientos del módulo *"Desarrollo Web en Entorno Cliente"*. En dicho módulo he aprendido cómo se debe realizar un proyecto con *Angular*, desde su inicialización, la configuración de las rutas, la creación de interfaces, clases, componentes y servicios y la codificación de todos estos elementos.

Por otro lado, el aprendizaje adquirido en el módulo *"Diseño de Interfaces Web"* me ha permitido realizar las vistas de los distintos componentes que forman la aplicación de teleasistencia aplicando estilos a las mismas utilizando *Bootstrap* y *SCSS* para darle un toque más personal al diseño de la web. Asimismo, se ha

realizado un diseño *responsive* para que sea adaptable a la mayoría de dispositivos.

A la hora de desarrollar el trabajo, los mayores obstáculos que se han encontrado son: la falta de tiempo debido a tener que realizar el proyecto durante el período de *FCT*; los fallos derivados de la parte del servidor que han impedido que algunos formularios funcionen correctamente; el desconocimiento de la forma de trabajo al ser un proyecto colaborativo entre varios centros formativos, ciclos, alumnos y profesores; y la dificultad de tener que desarrollar el primer prototipo de cliente web.

9.- Bibliografía.

1. Álvarez, M. Á. (28 de Octubre de 2017). *Definición de interfaces TypeScript*. (Desarrollo Web) Recuperado el 18 de Octubre de 2021, de https://desarrolloweb.com/articulos/definicion-interfaces-typescript.html
2. Canva, Inc. (1 de Enero de 2012). *Canva*. Recuperado el 5 de Septiembre de 2021, de https://www.canva.com
3. Canva, Inc. (2014). *Pexels*. (Pexels) Recuperado el 6 de Noviembre de 2021, de https://www.pexels.com
4. Coding Potions. (23 de Agosto de 2018). *Angular - Seguridad, protegiendo vistas con guards*. (Coding Potions) Recuperado el 15 de Noviembre de 2021, de https://codingpotions.com/angular-seguridad
5. Flaticon. (2013). *Iconos Vectoriales y Stickers gratis - Miles de recursos para descargar*. (Flaticon) Recuperado el 6 de Noviembre de 2021, de https://www.flaticon.es
6. Fonticons, Inc. (s.f.). *Font Awesome*. (Fonticons) Recuperado el 12 de Noviembre de 2021, de https://fontawesome.com
7. García Sánchez, B. (s.f.). Bases de datos. Tema 3 - Modelo entidad/relación. AIU. Recuperado el 21 de Noviembre de 2021, de https://cursos.aiu.edu/base%20de%20datos%20SOG/Sesi%C3%B3n%205.pdf

8. González Martín, L. (19 de Junio de 2021). *GitHub - lgonzalezm59/teleasistencia*. (GitHub) Recuperado el 28 de Noviembre de 2021, de https://github.com/lgonzalezm59/teleasistencia
9. Google Inc. (s.f.). *Validating form input*. Recuperado el 22 de Septiembre de 2021, de Angular.io: https://angular.io/guide/form-validation
10. Google, LLC. (s.f.). *Google Fonts*. (Google) Recuperado el 12 de Noviembre de 2021, de https://fonts.google.com
11. IES Valle del Jerte. (15 de Abril de 2021). *GitHub - IES-Valle-Jerte/teleasistencia*. (GitHub) Recuperado el 23 de Junio de 2021, de https://github.com/IES-Valle-Jerte/teleasistencia
12. JetBrains. (2 de Junio de 2021). *PyCharm: el IDE de Python para desarrolladores profesionales*. (JetBrains) Recuperado el 23 de Junio de 2021, de https://www.jetbrains.com/es-es/pycharm
13. MDN contributors. (27 de Noviembre de 2021). *Control de acceso HTTP (CORS) - HTTP | MDN*. (MDN Web Docs) Recuperado el 28 de Noviembre de 2021, de https://developer.mozilla.org/es/docs/Web/HTTP/CORS
14. MDN contributors. (16 de Octubre de 2021). *Introducción a Django* . (MDN Web Docs) Recuperado el 17 de Octubre de 2021, de https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Introduction

15. Moesif. (12 de Diciembre de 2020). *Moesif Origin & CORS Changer*. (Moesif) Recuperado el 26 de Septiembre de 2021, de https://chrome.google.com/webstore/detail/moesif-origin-cors-change/digfbfaphojjndkpccljibejjbppifbc?ucbcb=1

16. Python Software Foundation. (17 de Octubre de 2021). *DB-API 2.0 interfaz para bases de datos SQLite*. (Python.org) Recuperado el 17 de Octubre de 2021, de https://docs.python.org/es/3.10/library/sqlite3.html

17. Redondo García, J. (s.f.). *UT7: Angular.* Recuperado el 20 de Septiembre de 2021, de https://moodle.educarex.es/ccff_iesvjp

18. Robles, V. (s.f.). *Master en Frameworks JavaScript: Aprende Angular, React, Vue*. (Udemy) Recuperado el 3 de Octubre de 2021, de https://www.udemy.com/course/master-en-frameworks-javascript-aprende-angular-react-vue-js

19. Rosales Carrera, L. I. (s.f.). *Estudio de la actualización de un portal bancario SPA para la gestión de clientes.* Escola politècnica Superior (Universitat de Lleida). Recuperado el 28 de Noviembre de 2021, de https://repositori.udl.cat/bitstream/handle/10459.1/68101/lrosalesc.pdf

20. Shopify.com. (2006). *Crear logo gratis - Shopify Hatchful*. (Hatchful) Recuperado el 1 de Septiembre de 2021, de https://hatchful.shopify.com

21. *Template-driven forms de Angular*. (19 de Diciembre de 2020). (Java desde 0) Recuperado el 22 de Septiembre de 2021, de https://javadesde0.com/template-driven-forms-de-angular

22. Thornton Otto, M. J. (s.f.). *Bootstrap*. (Bootstrap) Recuperado el 10 de Noviembre de 2021, de https://getbootstrap.com