



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica  
Universitat Politècnica de València

Telecomunicación con satélites mediante el  
protocolo IEC 60870-5-104 en una aplicación  
web de Angular

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor:** Adrian Tendaro Lara

**cotutora:** Ana Pont Sanjuan

**cotutor:** Ivan Martín González

2019-2020





## Resumen

---

En este trabajo presentaremos la creación de una aplicación web en *TypeScript*, utilizando el *framework* de Angular con el objetivo de conectarse a una de las Unidades Terminales Remotas (UTR) o más conocidas como *Remote Terminal Unit* (RTU) y administrar el estado actual de la RTU conectada comunicándonos con las RTU mediante el protocolo IEC 60870-5-104 o protocolo 104.

La implementación de este trabajo consistirá en la creación de una aplicación web utilizando el *framework* de Angular 8, donde podremos interactuar con el estado de la RTU de una manera visual mediante la manipulación de imágenes.

**Palabras clave:** RTU, Satélite, Protocolo, Angular, TypeScript

## Abstract

---

In this work we will present the creation of a web application in Typescript, using the Angular framework in order to connect to one of the Remote Terminal Units (UTR) or better known as Remote Terminal Unit (RTU) and manage the current state of the RTU connected communicating with the RTUs through the IEC 60870-5-104 protocol or 104 protocol.

The implementation of this work will consist of creating a web application using the Angular 8 framework, where we can interact with the status of the RTU in a visual way by manipulating images.

**Keywords :** RTU, satellite, protocol, Angular, Typescript



# Tabla de contenidos

---

1.	Introducción .....	12
1.1	Estructura de la memoria.....	12
2.	Objetivos del proyecto .....	13
2.1	Objetivos generales .....	13
2.2	Objetivos Técnicos.....	13
2.3	Objetivos Personales .....	13
3.	Situación actual de la tecnología .....	14
3.1	¿Qué es Angular? .....	14
3.2	Evolución del framework de Angular .....	16
4.	Análisis del problema.....	18
5.	Diseño de la solución .....	19
5.1	Estructura de la aplicación .....	19
5.2	Componentes y servicios de la aplicación.....	20
5.2.1	Componente del Loggin .....	20
5.2.2	Servicio <i>SharedDataService</i> .....	21
5.2.3	Componente Main .....	22
5.2.4	Componente rtutree .....	26
5.2.5	Componente svgDinamizer .....	29
5.2.6	Servicio de codificación .....	32
5.2.7	Servicio IGWS2 .....	38
6.	Pruebas realizadas a la aplicación .....	39
7.	Conclusiones .....	42
8.	Ideas de mejora de la aplicación .....	43



# Índice de ilustraciones

---

ILUSTRACIÓN 1 - CICLO DE LA VIDA	15
ILUSTRACIÓN 2 - VISTA LOGIN	20
ILUSTRACIÓN 3 - COMPONENTE MAIN MARCANDO LOS SUBCOMPONENTES	22
ILUSTRACIÓN 4 - ESTADOS DE LA CONEXIÓN	23
ILUSTRACIÓN 5 - DIAGRAMA DE ESTADOS DE CONEXIÓN	25
ILUSTRACIÓN 6 - TIPO DATARTU	27
ILUSTRACIÓN 7 – RTUTREE	28
ILUSTRACIÓN 8 - SVG DINAMIZADO CON LOS CUATRO ESTADOS	29
ILUSTRACIÓN 9 - MENU CONTEXTUAL DEL SVG	30
ILUSTRACIÓN 10 - FUNCION DE EJEMPLO SUMA	39
ILUSTRACIÓN 11 - RESULTADOS DE LA ÚLTIMA COBERTURA DEL PROYECTO	40
ILUSTRACIÓN 12 - RESULTADOS DE LAS PRUEBAS UNITARIAS DEL PROYECTO	41





# Índice de tablas

---

TABLA 1 - ESTRUCTURA DE UN MENSAJE CODIFICADO .....	32
TABLA 2 -DESCRIPCIÓN DE LOS SIN .....	32
TABLA 3 - DESCRIPCIÓN DE LOS TIPOS COMUNES DE COMANDOS .....	33
TABLA 4 - ESTRUCTURA GENERAL DEL MENSAJE DE EJEMPLO .....	34
TABLA 5 - ESTRUCTURA APCI DEL MENSAJE DE EJEMPLO .....	34
TABLA 6 - ESTRUCTURA ASDU DEL MENSAJE DE EJEMPLO .....	35
TABLA 7 - CODIFICACIÓN BIG ENDIAN .....	37
TABLA 8 - CODIFICACION LITTLE ENDIAN .....	37
TABLA 9- DECODIFICACIÓN DE LE COMO BE .....	37



# 1. Introducción

---

En los últimos tiempos las grandes empresas han ido necesitando un estrecho control de sus recursos. Para ello, la comunicación vía satélite ha supuesto un gran avance y por ello mismo, una empresa ha solicitado una aplicación web que le permita conectarse a las *Remote Terminal Unit* (RTU) que tienen ubicadas por diversas partes del globo mediante un enlace por satélite.

Para ello nuestra aplicación será capaz de seleccionar una RTU de un conjunto de RTU facilitadas por el cliente, e iniciará el proceso de conexión con la RTU a través del satélite que será manejado por nuestra aplicación.

Tras conectarnos la aplicación será capaz de recuperar el estado de la RTU y mostrarlo por la interfaz mediante un *Scalable Vector Graphics* (SVG) que se dinamizara de colores dependiendo del estado de las variables de la RTU, podremos interactuar con la RTU mediante botones y menús desplegables que aparecerán en el SVG.

La comunicación por satélite se realizará mediante mensajes codificados con el protocolo *IEC 60870-5-104* (protocolo 104) por tanto nuestra aplicación será capaz de enviar y recibir mensajes en mediante el protocolo 104, incluyendo la codificación y decodificación del protocolo 104.

Dado que este TFG ha sido realizado en prácticas de empresa se evitará dar en el mismo, datos sobre la identidad del cliente y los usos que el cliente pueda dar a sus activos incluidos las RTU de esta aplicación, por ello mismo todos los ejemplos de uso serán teóricos y no relacionados con la actividad del cliente.

## 1.1 Estructura de la memoria

La memoria seguirá la estructura expuesta a continuación:

- Introducción
- Objetivos del proyecto
- Estado del Arte
- Desarrollo de la solución
- Implantación
- Pruebas
- Conclusiones
- Trabajos futuros
- Anexos

## 2. Objetivos del proyecto

### 2.1 Objetivos generales

El proyecto tiene como objetivo satisfacer la demanda del cliente de disponer un acceso secundario a las RTU que tiene instaladas por diversas partes del globo y que no siempre puede disponer de acceso directo por internet, ya sea por causas propias o externas. Para ello nos ha encargado una aplicación web que sea capaz de comunicarse con un satélite y acceder a través de comunicación vía satélite a las RTU.

### 2.2 Objetivos Técnicos

(añadir una pequeña frase)

- Crear una aplicación web utilizando el lenguaje de programación *TypeScript* con un *framework* de Angular 8.3
- Adaptar el servicio de comunicación del fabricante del satélite en lenguaje *Node* a un servicio compatible con Angular 8.3
- Crear Servicios para nuestra aplicación web que nos permita codificar y decodificar mensajes con protocolo 104.
- Utilizar la herramienta *SourceTree* para manejar un repositorio GitHub donde se alojará el proyecto.
- Control de un SVG para representar los posibles estados de una RTU mediante la dinamización del SVG.
- Realización de prueba utilizando el *framework* de test *Jasmine*, manteniendo una cobertura mínima del 80%

### 2.3 Objetivos Personales

Los objetivos personales del autor son ampliar nuestro conocimiento sobre Angular en concreto sobre la versión 8.3 que es un *framework* que utiliza el *Typescript* como base y será definido con más detalle en el próximo punto. También quiero utilizar este proyecto como reto personal al utilizar tecnología de comunicación mediante satélite para conectarme a equipos remotos.

También es el primer proyecto dentro de la empresa donde estamos realizando prácticas que comienzo y realizo en solitario, la parte de *frontend* que es la parte que ve el usuario y por tanto utilizarlo como una buena forma de aprender a realizar un proyecto desde cero siguiendo los estándares de la empresa ayudándome a adaptarme a mi puesto de trabajo.

### 3. Situación actual de la tecnología

En esta sección explicaremos el estado actual de las tecnologías más relevantes respecto a nuestra aplicación dando especial hincapié al *framework* de Angular que será el principal elemento sobre el que gira la aplicación y en consecuencia el TFG.

#### 3.1 ¿Qué es Angular?

Angular es un *framework* de aplicaciones web de una sola página que utiliza la arquitectura software de modelo-vista-controlador donde el manejador se programa en ficheros de *TypeScript* mientras que la vista o interfaz se programa en ficheros HTML. Angular utiliza etiquetas de *data-binding* bidireccional para relacionar los elementos en el HTML con el controlador que es el fichero *TypeScript* donde se realizan todas las programaciones y así permitir que se sincronicen los valores entre el fichero HTML y el fichero de *TypeScript* permitiendo utilizar variables del fichero *TypeScript* dentro del HTML.

Angular tiene como requisito tener instalada una versión de *node* y con *node* obtenemos el *Node Package Manager* o npm que será utilizado para instalar las librerías requeridas para utilizar la aplicación, así como para ejecutar diversos comandos de manera sencilla.

Así mismo Angular se divide en diversos elementos, entre ellos los más destacados y utilizados en la aplicación son los componentes y los servicios. Estos elementos se unen entre si dentro de la aplicación de Angular mediante los módulos.

Los módulos son ficheros de angular es donde se declara un componente o servicio dentro de angular y se importan todos los elementos necesarios para la utilización del componente o servicio, ya sean otros componentes o servicios de la propia aplicación como si son elementos externos obtenidos mediante librerías. En esta aplicación y TFG se utilizara un module general llamado *app.module.ts* donde se importaran y declararan todos los componentes y servicios.

Los componentes son las unidades funcionales de la aplicación y suelen corresponder cada componente a una pantalla diferente de la aplicación, aunque hay componentes que están dentro de otros componentes (subcomponentes) y manejan un aspecto determinado del componente principal. La única comunicación posible entre componentes, sin utilizar servicios, es mediante los inputs/outputs de los subcomponentes.

Los componentes están formados por tres partes básicas, el controlador que es donde se introduce toda la lógica de la aplicación y se programa en *TypeScript* y el *template* o plantilla que es donde se declaran todos los elementos HTML del componente. Además, un componente suele tener otros dos elementos más. El primero es un fichero CSS específico para la plantilla de este componente y solo se aplicará al componente actual y sus posibles subcomponentes. El segundo elemento es un fichero de testeo donde se escriben diversas pruebas para comprobar que las diversas funciones del controlador funcionen como se espera.

Los servicios en cambio son controladores sin plantillas ni ficheros CSS que a diferencia de los controladores de los componentes pueden ser utilizados por todos los componentes y que suelen implementar funciones y acciones que se suelen repetir en diversos componentes por toda la aplicación, también es una forma efectiva de gestionar las variables que deben ser tratadas por diferentes componentes. Es importante no confundir con una librería de Angular.

Las librerías de Angular son un conjunto de componentes y servicios que nos permiten importar los componentes a nuestra aplicación de Angular y trabajar con los componentes de la aplicación como si hubiesen sido programados en la propia aplicación web. Se suelen generar librerías generalmente para componentes de uso común como pueden ser graficas o tablas que son usados por componentes de la aplicación principal para mostrar datos. Los controladores de los componentes de las librerías suelen tener inputs y outputs que son la forma de comunicación entre componentes emparentados, es decir un componente padre que tiene uno o más componentes hijos, que en este caso es el componente de la librería.

Una aplicación web es común que contenga varias páginas, en los *frameworks* de Angular la aplicación es de una página, es decir, no hay que realizar ninguna petición http. pero eso no implica que no queramos que nuestra aplicación contenga diferentes rutas de navegación y múltiples páginas que se podrán acceder dependiendo de las circunstancias. Para ello utilizamos un módulo dentro de Angular que es el *Angular/Router* y nos permite definir que componentes se muestran dada una navegación a una determinada ruta.

Angular incorpora características importantes como es el denominado ciclo de la vida de un componente que se inicia cuando empezamos a utilizar un determinado componente de Angular y termina cuando dejamos de usar dicho componente y lo destruimos. Como se puede observar en la ilustración 1 muestra el ciclo completo de vida de un componente.

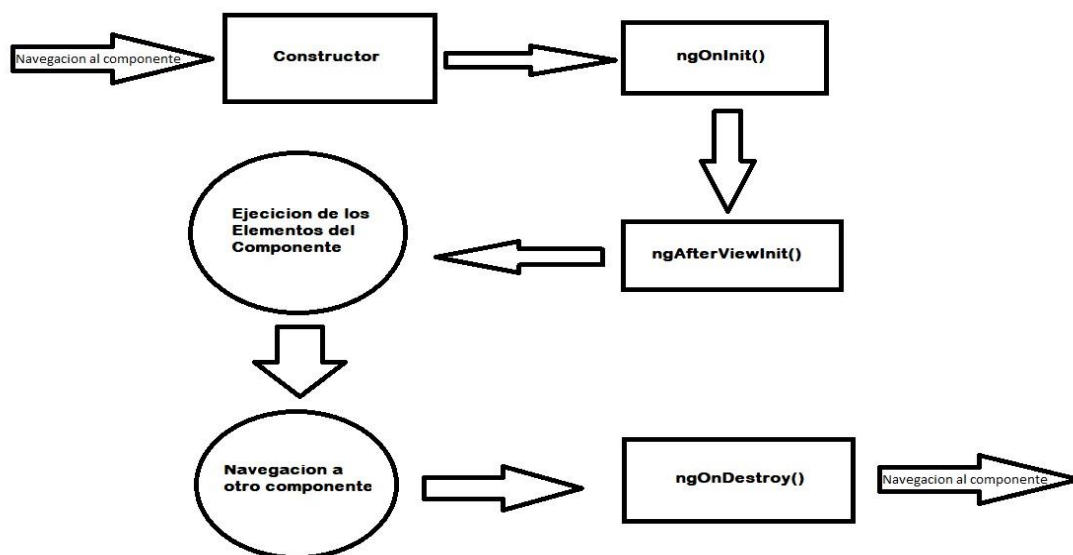


Ilustración 1 - Ciclo de la vida

Cuando un componente se inicia lo primero que se ejecuta es el método *constructor()*, que suele contener todas las declaraciones de los servicios y atributos del componente (variables globales).

Después procede a ejecutar el método *ngOnInit()* que suele contener todas las peticiones iniciales ya sean a los servicios declarados en el constructor como peticiones iniciales de datos al *backend*.

Tras la ejecución del *ngOnInit()* se procede a cargar el *plantilla* del componente y se procede a ejecutar el *ngAfterViewInit()* que se utilizar para cargar en el controlador elementos del DOM como formularios. Al concluir el *ngAfterViewInit()* se procede a ejecutar el cuerpo del código ejecutable del componente.

Y finalmente cuando dejemos de usar el componente, se destruye el componente pero antes de destruirlo ejecuta el *ngOnDestroy()* que se suele utilizar para realizar acciones finales antes de salir del componente como cancelar suscripciones, peticiones de datos, timeouts y intervalos que estén activos dentro del componente que se destruye.

### 3.2 Evolución del framework de Angular

Angular publicó su primera versión fuera de la beta el 14 de junio del 2012, fue un gran avance en las aplicaciones web de una sola página. La versión inicial de Angular o Angular 1 o AngularJS, se está considerando como un *framework* totalmente diferente a Angular que es como de denomina a las versiones de Angular 2 o superiores y una de las grandes diferencias entre la versión de AngularJS y Angular es que la primera versión utilizaba *JavaScript* para el controlador a diferencia de las versiones más actuales que utilizan *TypeScript*. A parte AngularJS no está pensada para aplicaciones móviles siendo una limitación bastante grande para nuestros tiempos puesto que ahora prácticamente todo el mundo tiene móvil y quiere acceder a sitios web desde el dispositivo móvil.

Angular 2 fue la segunda versión de Angular y reelaboro totalmente Angular incluyendo varias novedades destacadas, la primera es que para programar ya no se utilizara *JavaScript*, sino que utilizara la variante de *JavaScript* tipada llamada *TypeScript*. Esto implica un mayor control sobre los valores de las variables ya que al utilizar variables tipadas, es decir indicar que una variable sea de un tipo de dato concreto como (numero, booleano, string, etc) nos permite asegurar un determinado tipo de valor en una variable concreta facilitando las tareas de *testing* (búsqueda de errores de la aplicación) y *debug* (resolución de los fallos de la aplicación). Por último también hay que mencionar que, aunque el tipado es una herramienta fuerte para el programador *TypeScript* también incluye el tipo *any* que en esencia acepta cualquier tipo de valor.

Aunque sigue siendo posible programar en *JavaScript* y es posible utilizar algunas librerías de *JavaScript* y AngularJS, ya que el *TypeScript* se compila y ejecuta como *JavaScript*. La otra novedad destacada es el soporte para las aplicaciones web para móviles permitiendo adaptar la aplicación web para dispositivos móviles u ordenadores dependiendo del dispositivo y resolución detectada.



Las versiones superiores de Angular 2 no contienen cambios tan destacados como la diferencia de AngularJS a Angular 2 y se pueden actualizar los proyectos de Angular 2 a utilizar Angular 4 o superior con relativa facilidad.

La versión de Angular 3 es realmente la que se denomina Angular 4, saltándose Angular 3 por motivos de claridad ya que varios de los módulos de angular, en concreto el *Angular/Router* ya se encontraba en la versión 3 mientras que angular seguía en versión 2. Así para sincronizar las versiones cuando se publicó Angular 3, la versión del módulo del *router* de Angular tendría que haber sido la versión 4. Pero para sincronizar las versiones y evitar posibles confusiones se igualó las versiones del *router* y del *core* de Angular a la misma versión que en este caso es la versión 4 de Angular, siendo denominado Angular 4.

La novedad de Angular 4 consiste en una reducción del 60% aproximadamente del tamaño de los ficheros compilados que se generan cuando haces una *build* del proyecto de Angular, mejorando de gran manera el impacto en el rendimiento de la aplicación, especialmente en servidores con recursos limitados.

En cambio, Angular 5 se centra en mejoras para el *router* que utiliza eventos del “ciclo de la vida” que consiste en un ciclo que comienza cuando el *router* cambia de componente hasta cuando el componente es destruido al navegar fuera de dicho componente. También incluye el componente *httpClient* que sustituye al componente que se utilizaba en las anteriores versiones de Angular de *http*.

Angular 6 por su parte se centra menos en cambios y mejoras del propio *framework* sino en mejorar la cadena de herramientas que dispone angular. En concreto hay que destacar Angular CLI y Angular Material, el primero es una interfaz de líneas de comandos útiles para angular ya que permiten generar nuevos proyectos y componentes así como actualizar los componentes y paquetes de Angular, en cambio Angular Material contiene recursos útiles para mejorar los proyectos de Angular con animaciones, tablas, botones y muchas cosas más que han sido extraídos del core de Angular y son totalmente opcionales, permitiendo que el peso de los proyectos sencillos de angular sean mucho más reducidos.

Angular 7 y Angular 8 por su parte expanden las funcionalidades generales vistas en Angular 6, incluyendo Angular CLI, Angular *core* y Angular Material, entre los que destacan. En Angular 7 el nuevo compilador ‘*Compatibility Compiler*’ (ngcc) o la habilidad del *router* de Angular de manejar rutas incorrectas de navegación. Y en Angular 8 destaca las nuevas mejoras al *router* de Angular, el soporte de los ficheros de estilo de tipo SASS, el soporte a *TypeScript* de 3.2 y las nuevas propiedades de los formularios, permitiendo marcar los elementos como “tocados”.

Aunque este TFG este realizado sobre Angular 8, durante la realización de este salieron 2 nuevas versiones de Angular, Angular 9 tiene como principal novedad un nuevo compilador, el ‘*ivy compiler*’ que reduce el tiempo y el tamaño de los ficheros compilados de alrededor del 30%, compilador que estaba incluido en Angular 8 en fase de pruebas y era opcional, ahora es el compilador de Angular. En cambio, Angular 10 Añade algunas mejoras al módulo de Angular Material y Angular CLI, así como permitir el modo stricto de compilación de *TypeScript* para mejorar el rendimiento del compilado y detección temprana de bugs.

## 4. Análisis del problema

Por toda la geografía española nuestro cliente tiene estaciones, muchas veces en localizaciones y zonas remotas de difícil acceso, que suministran servicios a las poblaciones locales y son operadas a distancia mediante teleoperadores, cuando surge algún problema se conectan mediante conexiones 2G y 3G para intentar solucionarlo de manera rápida y interrumpir el servicio lo mínimo imprescindible.

Pero este sistema no siempre está disponible ya que las estaciones pueden encontrarse fuera de cobertura ya sea por causas medioambientales, problemas en el suministro eléctrico o simplemente zonas de baja cobertura móvil. En estos casos un técnico se tiene que desplazar físicamente a la localización de la estación en fallo y arreglarla para reanudar el servicio a las poblaciones locales lo antes posible. El problema con este sistema es que la central desde la que se desplazan los técnicos se encuentra en Madrid y los desplazamientos a zonas remotas u otras zonas lejanas de la capital pueden llevar horas hasta que el técnico se persone.

Por este motivo la empresa nos ha propuesto realizar una aplicación que sea capaz de conectarse a las estaciones remotas, no mediante conexiones 2G o 3G que son dependientes de la cobertura u otras circunstancias medioambientales, sino mediante comunicación por satélite.

Dado que la comunicación por satélite es costosa, más que mediante el 2G y 3G, pero más fiable este sistema de comunicación se utilizara en forma de sistema de reserva y último recurso antes de enviar un técnico físicamente que dados los inconvenientes que esto conlleva no solo para la empresa sino para todos los usuarios del servicio en las poblaciones locales de la estación.

Resultando en que, si se puede establecer la comunicación por satélite con la estación en caso de que el sistema primario de comunicaciones no estuviese disponible, la resolución de los problemas de suministro del servicio a las poblaciones locales se realizaría de manera más breve y a menor coste tanto para los consumidores como para nuestro cliente.

## 5. Diseño de la solución

### 5.1 Estructura de la aplicación

Las aplicaciones de Angular aparte de componentes y servicios también incluyen archivos de configuración para el proyecto, siendo especialmente importantes los ficheros *packages.json* y *angular.json* que se encuentran en la raíz del árbol.

El fichero *packages.json* es el fichero donde se encuentra la información básica de la aplicación, siendo esta el nombre y versión de la aplicación, la declaración de los scripts que podrá lanzar NPM cuando estemos trabajando en esta aplicación y las dependencias de la aplicación que tendrán que ser instaladas para poder ejecutar la aplicación.

Las dependencias se dividen en dos categorías, las dependencias generales y las dependencias de desarrollo. Las primeras son las librerías que tanto el desarrollador como el usuario cuando utilice la aplicación deberá tener instaladas para poder utilizar la aplicación. Mientras que las dependencias de desarrollo solo son necesarias a la hora de desarrollar la aplicación, estas dependencias suelen estar más orientadas a librerías de soporte, de pruebas y *debug* de la aplicación que no tiene sentido se utilice en la versión del cliente.

El fichero *angular.json* es el fichero que configura el funcionamiento de Angular y la ejecución de la aplicación, y es donde se indica donde se encuentra el código de la aplicación, que ficheros se tienen que utilizar y que ficheros se tienen que ignorar, a parte también es donde se indica que componente es el componente que Angular tiene que ejecutar como inicial.

Por otra parte, aplicación de Angular suele tener diversos entornos de ejecución que cambian de forma significativa la forma de ejecutar la aplicación al cambiar diversas variables globales. Estos entornos se configuran dentro del fichero *angular.json* como configuraciones.

Actualmente en la aplicación se encuentran dos entornos diferentes, el primer entorno es el de producción que es el entorno de ejecución que utilizara el cliente cuando utilice la aplicación con los debug y mensajes de información totalmente desactivados y el código ofuscado para que no sea visible. Mientras que el segundo entorno es el de *Quality Acceptance* o QA es el entorno que se utiliza para probar la aplicación en un entorno controlado por parte nuestra, este entorno contiene todos los mensajes de información, así como el código totalmente visible y claro.

A parte de los ficheros de configuración el proyecto incluye una carpeta “src” que es donde se encuentra la aplicación propiamente dicha que incluye la configuración de *TypeScript*, las variables globales de los entornos que tendrán un valor u otro dependiendo del entorno de ejecución, los *assets* que incluyen ficheros de apoyo a la aplicación como imágenes, configuraciones o la declaración de tipos y la *app* es donde se encuentran todos los componentes y servicios de la aplicación.

## 5.2 Componentes y servicios de la aplicación

Como hemos mencionado anteriormente una aplicación de Angular consta de componentes y servicios que forman las diferentes páginas que serán mostradas por la aplicación. Las páginas de la aplicación suelen estar formadas por un componente principal como puede ser los componentes de *login* o el *main* y subcomponentes que son utilizados por los componentes principales como puede ser el *svgdinamizer* que forma parte del componente *main*. Los servicios por otra parte son complementos a los componentes de la aplicación siendo utilizados para funciones comunes que uno o varios componentes pueden utilizar de manera repetitiva, como puede ser el servicio de *SharedDataService* que es utilizado por varios componentes de la aplicación.

### 5.2.1 Componente del Login

El componente del *login* es el componente que creamos primero que consta de una imagen con el logotipo del cliente, un par de textos, un formulario con dos campos, Nombre y Contraseña y un botón para confirmar nuestra identidad.

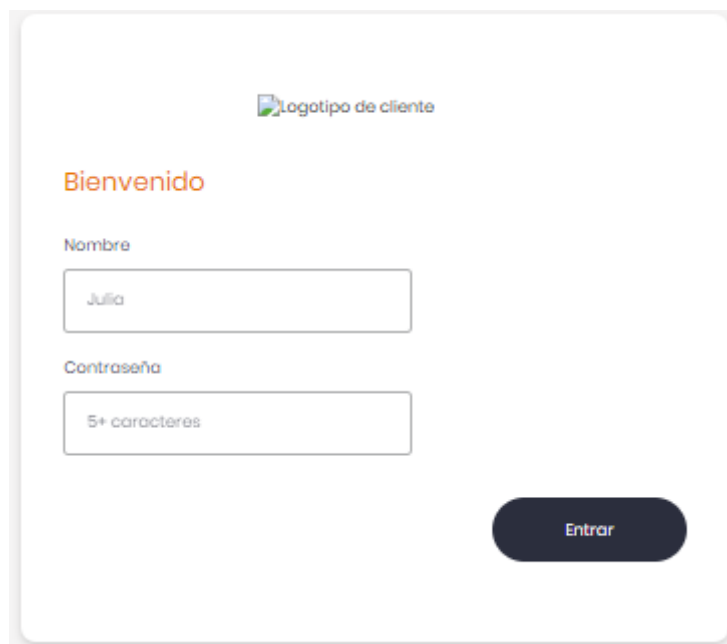
La imagen muestra una interfaz de usuario para el login. En la parte superior hay un icono de cliente con el texto "Logotipo de cliente". Debajo, el texto "Bienvenido" en naranja. Luego, el campo "Nombre" con un input que contiene "Julia". Después, el campo "Contraseña" con un input que contiene "5+ caracteres". En la parte inferior derecha hay un botón redondeado de color azul oscuro con el texto "Entrar" en blanco.

Ilustración 2 - Vista login

Este componente como se puede observar en la *ilustración 2* es muy simple y está hecho a modo de demo de un inicio de sesión ya que este, aunque es funcional es muy inseguro ya que no comprueba los usuarios en una base de datos, sino que tiene grabadas en el propio componente las credenciales de acceso de usuario, siendo un grave problema de seguridad si se llegase a usar de manera profesional.

Siguiendo el ciclo de la vida del componente, lo primero que se realizara al ejecutar el componente será el constructor, este simplemente inicializara el formulario, vaciando y marcando como requeridos por el validador los dos campos del formulario. Después ejecutara el método `onInit()` que se encuentra vacío y el programa quedara a la espera de alguna acción del usuario.

Como se ha mencionado el componente consta de un formulario que consta de dos campos, el usuario y la contraseña, y se comprueba la validez del formulario al pulsar el botón de entrar. Una vez pulsado el botón, la aplicación lanzara un evento `onClick()` que ejecutara la función de *logging()*, esta función comprobará primero que nada que el formulario existe y entonces procederá a recuperar los valores de los campos del formulario y a comprobar que ninguno de ambos valores sea un valor nulo o indefinido y entonces procederá a comprobar si el usuario y la contraseña coinciden con las cadenas de caracteres que tiene indicado en el propio componente. En caso de que coincidan el *login* llamara al primer servicio que haremos mención que es el *sharedDataService*, este servicio es muy simple y será explicado posteriormente, pero en esencia nos permite trasladar valores entre un componente a otro de Angular sin ninguna relación de herencia, es decir, sin utilizar las funciones de *input* y *output* de los componentes de Angular. A parte el *login* también llamara a al *router* de Angular para navegar a la siguiente pantalla que es la pantalla del componente *main* donde está el grueso de la aplicación.

En caso de que el *login* sea incorrecto simplemente llamaremos al *shareDataService* y eliminaremos cualquier valor previo de las credenciales del acceso.

### 5.2.2 Servicio *SharedDataService*

Este servicio es muy simple pero no vamos a entrar en mucho detalle ya que este servicio ha sido extraído y reutilizado de otra aplicación de la empresa en la que no se ha participado.

Este servicio es en esencia una estructura de datos tipo mapa (*map* en inglés) que se estructura de una clave y un valor con dos funciones, una para leer cualquier valor almacenado en el mapa y la otra para asignar el valor o actualizar valores del propio mapa.

Este servicio solo será utilizado para trasladar las credenciales de acceso del componente del *login* al componente *main* que comprobará si en efecto estamos loggeados.



### 5.2.3 Componente Main

Este es el componente principal de la aplicación, nuestra aplicación al no poseer un *backend*, el *frontend* tiene que manejar toda la comunicación desde la RTU a la aplicación a través del satélite y eso se realiza desde el componente *main*.

El componente *main* tiene diversos subcomponentes que realizan funciones específicas del *login* que son los componentes *rtutree* y *svg-dinamicer* que serán explicados en subapartados de este componente.

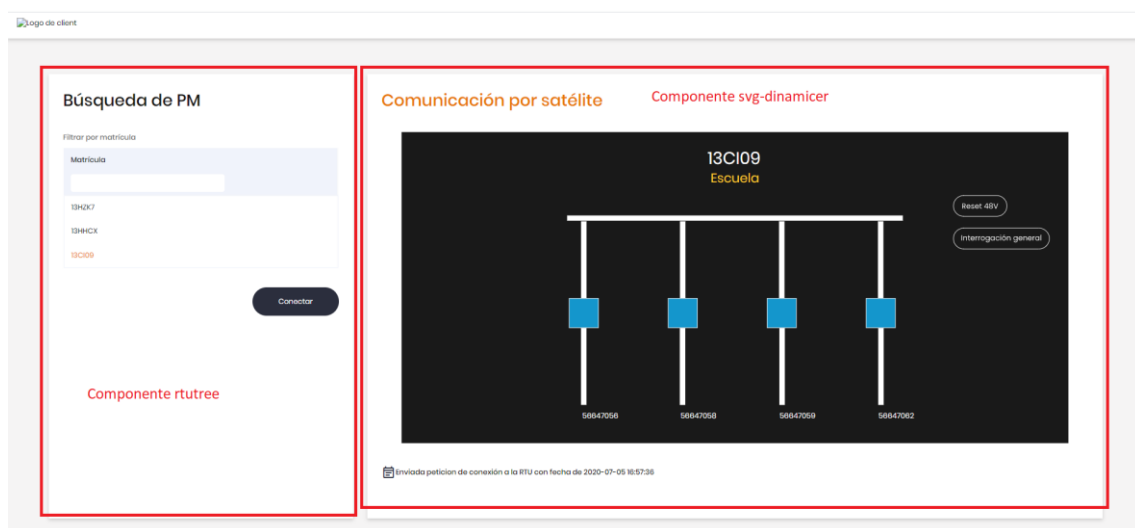


Ilustración 3 - Componente Main marcando los subcomponentes

Como se puede ver en la ilustración 3, el componente main no contiene ningún elemento propio, sino que es un componente que hace de mediador entre los dos subcomponentes que dependen del componente *main*.

El componente main es el que se encarga de realizar la conexión a la RTU, que es seleccionada y recibida del componente *rtutree* a través de un evento output del componente iniciando así en el main un proceso automático de conexión y reconexión en caso de ser necesario a la RTU.

Dado que este es un componente muy complejo que utilizar muchos servicios no vamos a explicar en este apartado el funcionamiento de las funciones de los servicios y se indicaran de manera general el resultado de dicha función.

Para empezar a explicar vamos a seguir el funcionamiento del componente siguiendo el orden del ciclo de vida empezando por la función del constructor donde declaramos todos los servicios que van a ser usados en este componente que son los siguientes.

Servicios utilizados en este componente:

- *Router*
- *SharedDataService*
- *Igws2Service*
- *Codification Service*
- *NGXLogger*

Después de declarar los servicios el componente ejecuta su método *ngOnInit()* que es donde utilizamos el *SharedDataService* para comprobar que al navegar a este componente, ya sea mediante navegación directa del navegador o una navegación errónea por parte del login, estemos autorizados mediante las credenciales de acceso, en cuyo caso el componente *main* se mantiene a la espera de eventos por parte del usuario mediante el componente *rtutree*. En caso de no estar autorizados el componente navega automáticamente a la pantalla del login y ejecutando el método *ngOnDestroy()* que explicaremos más adelante cuando el resto del componente y elementos importantes de este estén explicados.

Este componente contiene una máquina de estados para controlar la situación de la comunicación con la RTU para ello utilizamos una variable global de tipo enumerale, tal como se muestra en la ilustración 4.

```
enum ConnectionState {  
    DISCONNECT = 0,  
    AWAITCONNECT,  
    CONNECT,  
    AWAITSDT,  
    STARTDT,  
    IG  
}
```

Ilustración 4 - Estados de la conexión

La idea del componente *main* es estar realizando peticiones de recuperación de mensajes continuamente mediante un intervalo de quince segundos y dependiendo del Estado de conexión en que se encuentre se realizará una acción u otra.

Para ello el primer estado que utilizaremos es el estado desconectado y antes de empezar a realizar las recuperaciones de datos, comprobaremos primero que tenemos una conexión con el propio satélite. Para ello solicitaremos al satélite la versión del hardware que está utilizando y introduciremos las credenciales de conexión al satélite que nos permitirán comunicarnos con el satélite ya con los permisos necesarios. Después comprobaremos que la configuración el satélite sea correcto y en caso de que no sea la configuración correcta la cambiaremos antes de empezar el proceso de conexión.

Una vez que hemos comprobado qué tenemos comunicación con el satélite y que la configuración de conexión del satélite sea correcta entonces y solo entonces comenzaremos el proceso de conexión.

Lo primero que realizaremos una vez empezado el proceso de conexión es actualizar la fecha a partir de la cual empezaremos a recuperar mensajes de la RTU. Después realizaremos una petición de datos especial donde intentaremos recuperar no los mensajes actuales, sino los mensajes con una antigüedad de cinco minutos e intentaremos a partir de los mensajes recuperados saber en qué estado de conexión nos encontramos.

Dependiendo del Estado de conexión en el que encontremos en la recuperación de mensajes anterior nos saltaremos unos pasos u otros ahorrándonos trabajo y permitiendo trabajar desde el punto en el que se detecta que está la conexión. Para el proceso de este ensayo supondremos que empezamos desde el estado desconectado que no hemos recibido ningún mensaje relevante que nos cambie el estado de conexión en el que nos encontramos.

Para ello el primer paso es realizar la conexión con la RTU Desde el satélite enviando el comando de conexión. el comando conexión se generará en el servicio de codificación Y se explicará su composición en dicho componente. Una vez enviado el comando de conexión la máquina de estados de las conexiones pasará al estado de *Awaitconnect*. En dicho estado el componente lo que realizará será una recuperación de datos esperando la confirmación de conexión a la RTU mientras espera en este estado la confirmación de conexión a la RTU se activará un *time out* que contará el número de recuperación de mensajes realizados mientras se espera la confirmación de conexión.

En el caso de *time out* salte realizaremos una desconexión y volveremos a intentar desde el principio una conexión. Pero en el caso de que recibamos una confirmación de conexión Entonces nuestra máquina de Estados pasará al estado de *Connect*. Entonces una vez estando conectados realizaremos la siguiente parte de la conexión que es la solicitud de transferencia de datos entre la RTU y la aplicación. Para ello se enviará un mensaje denominado *startDT* (*Start Data Transfer*) y donde solicitaremos a la RTU el inicio de la transferencia de datos y una vez enviado este mensaje cambiaremos el estado de conexión al estado de *Awaitstartdt*, que es un estado similar al de *Awaitconnection* y realizará una función parecida esperando la recepción de la confirmación del *startDT*, con las mismas condiciones que en el estado anterior activando un *time out* de diez recuperaciones de mensajes Antes de volvernos a desconectar en caso de no recibir la confirmación.

Una vez recibida la confirmación del *startDT* pasaremos al estado *Startdt* qué es un estado de enlace qué enviara una interrogación general a la RTU entonces cambiaremos otra vez el estado de la conexión Para llegar al último estado de conexión que es el de IG. En el estado IG esperaremos la recepción de mensajes ya sean mensajes de cambio de valores mensajes de confirmación y mensajes de resultados de las interrogaciones generales. En este estado somos libres de explorar y utilizar cualquier comando a nuestro alcance ya que tenemos la certeza de estar conectados a la RTU y tener activada la transferencia de datos entre la RTU y la aplicación.



La mayoría de las acciones que se pueden realizar mediante eventos que provienen del componente *svg-dinamicer* entre los que se incluyen el cambio de estados de las variables, el *reset de 48V* y el envío interrogaciones generales manuales por parte del usuario.

De las tres acciones mencionadas anteriormente el *reset de 48V* reinicia totalmente la RTU por lo tanto se desconecta la RTU y se tiene que volver a empezar la conexión desde cero, esta es una característica solicitada por el cliente porque muchas veces las RTU se bloquea y al reiniciar se solucionan bastantes problemas.

Las peticiones de recuperación de mensajes se realizan de manera periódica esto significa que nosotros utilizamos un intervalo que cada quince segundos realiza una petición teniendo esto en cuenta ya podemos explicar el método *ngOnDestroy()* que teníamos pendiente de explicar anteriormente. Ahora podemos ver el método *ngOnDestroy()* tiene que eliminar cualquier time out e intervalo que quede activo antes de destruir el componente.

Así mismo cuando nos desconectamos ya sea que recibamos una confirmación de desconexión u realicemos nosotros la desconexión También tendremos que eliminar cualquier intervalo activo, así como cualquier time out.

También realizaremos una desconexión cuando seleccionemos otra RTU a conectarnos ya que podríamos recuperar mensajes de la RTU anterior y dar lugar a confusión por tanto cada vez que demos al botón de conexión realizaremos una desconexión Intentaremos conectarnos a la nueva RTU esto también es un método eficaz si queremos reiniciar la conexión Ya que comenzaremos desde cero habiendo cerrado la conexión anteriormente existente. En la ilustración 5 se puede observar el proceso explicado anteriormente.

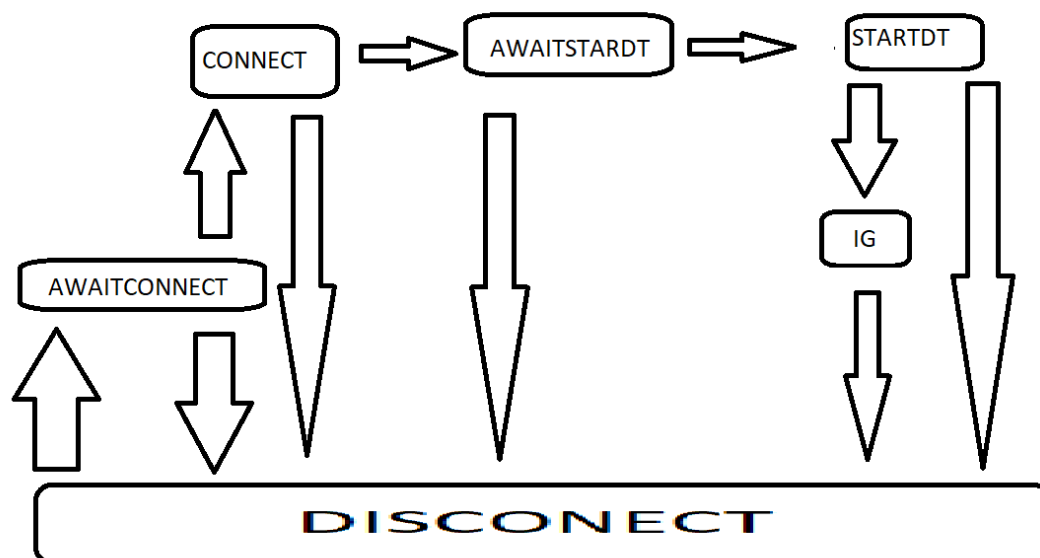


Ilustración 5 - Diagrama de estados de conexión

Cómo se puede observar en la ilustración 5 empezando desde el estado desconectado pasamos al estado de esperar la conexión una vez que enviamos a la petición de conexión a la RTU. Entonces una vez recibida la confirmación de conexión, pasamos al estado de conectado enviamos la petición de inicio de transferencia de datos y esperamos la confirmación de que se inicia la transferencia de datos. En ese momento pasamos al estado de transferencia de datos que es lo que realizamos es una petición de interrogación general este estado simplemente es para realizar esta petición y a todos los demás efectos sería como si ya estuviéramos en el siguiente estado de IG que se consigue al recibir los primeros datos independientemente de que sea de la IG u de otro tipo de valor. Aparte cómo se puede observar desde cualquier estado se puede ir al estado desconectado eso es debido a que se puede hacer el *reset de 48 V* desde nuestra aplicación o podemos recibir un mensaje de desconexión por parte de la RTU en cuyo caso siempre independientemente del estado en el que nos encontremos excluyendo el caso de desconectado pasaremos al propio estado desconectado.

Para averiguar en el estado que nos encontramos de conexión y cuando pasar al siguiente tenemos que decodificar los mensajes recuperados mediante el servicio de codificación. Este servicio también contiene la codificación de los mensajes para enviar las peticiones de conexión e inicio de transferencia de datos.

Durante la recuperación de mensajes van a llegar de muchos tipos y muchos mensajes que no están relacionados con la aplicación ya que son mensajes y notificaciones internas de la RTU que no nos interesa en el contexto de nuestra aplicación y por ello lo que hacemos es durante la recuperación de los mensajes es filtrarlos. Por ello cuando expliquemos el servicio de codificación también explicare cómo están compuestos los mensajes y cómo se dejó estructuran los mensajes ya que es una parte importante de este trabajo fin de grado.

#### 5.2.4 Componente *rtutree*

En este componente utilizamos y configuramos una librería externa de pago que utilizamos para darnos soportes en diversos proyectos, esta librería se llama *kendo for Angular* e incluye un componente llamado *kendo-treelist*.

Este componente recibe una lista de objetos y lo utilizará como datos, entonces una vez suministrado los objetos, podemos indicarle al componente que propiedades del objeto tiene que mostrar en la lista y muy importante también que propiedad o propiedades tiene que utilizar para filtrar los elementos mostrados.

Para ello vamos a explicar que propiedades debe tener el objeto de tipo *dataRTU* tal como está definido este tipo de dato dentro del código y el objeto de *dataRTU* que tenemos actualmente indicando para que sirve cada propiedad.

Los objetos de tipo *DataRTU* contienen nueve propiedades de las cuales dos son opcionales, estas propiedades se pueden observar en la ilustración 6 y será explicado a continuación la utilidad de cada propiedad.

```
export interface DataRTU {  
  id: string;  
  name: string;  
  ip: string;  
  port: string;  
  plate: string;  
  asdu: number;  
  interruptors: Array<Interruptor>;  
  nInterruptos?: number;  
  image?: string;  
}
```

Ilustración 6 - Tipo *DataRTU*

La primera propiedad es el *id* de RTU, esto sirve para identificar la RTU objetivo donde tiene que conectarse el satélite si es necesario para realizar cualquier envío de mensajes.

La segunda propiedad el *name*, es el nombre de la RTU y el nombre que aparecerá en las etiquetas de dentro de los SVG.

La propiedad *code* es el identificador de la instalación donde se encuentra la RTU.

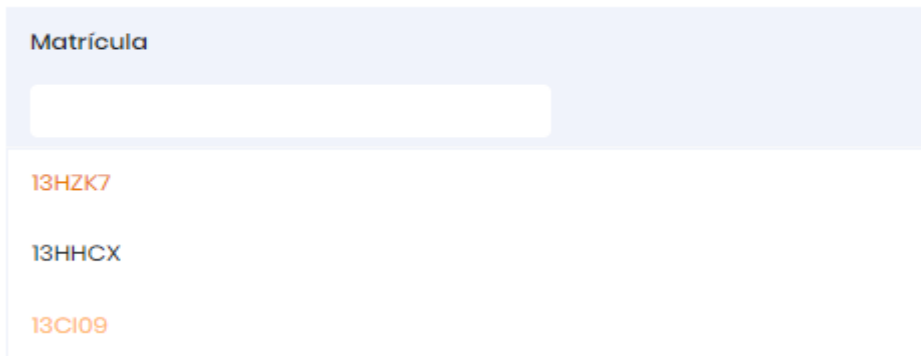
La propiedad *plate* o matrícula es el identificador utilizado por el cliente para denominar internamente las RTU que dispone. Las propiedades IP y puerto indican donde tiene que realizar la conexión IP para conectar con la RTU el satélite.

La propiedad de *interruptors* es un array que contiene la información de cada interruptor relevante para una estación determinada, en la versión actual de la aplicación nosotros estamos considerando un número máximo de interruptores de diez, aunque la aplicación, podría representar un número determinado de interruptores mayor que diez, si se utiliza la propiedad *image* para asignarle una imagen SVG con el número deseado de interruptores, en caso de no ser utilizada esta propiedad opcional, solo se representara y trabajara con los 10 primeros interruptores. Por último, también está el caso, en el cual, nosotros deseemos mostrar en el SVG más interruptores de los que actualmente tenemos en el array de interruptores, para ello utilizaremos la propiedad *nInterruptores* que fijara el número de interruptores a tratar al número indicado. En este caso los interruptores que se muestren por esta causa se mostraran en estado desconocido y no se podrá enviar o recibir comandos.

La interfaz de *Interruptors* lo explicaremos y comentaremos con más detalle cuando explique el dinamizado del SVG ya que es en ese componente donde se utilizan todas las propiedades de los interruptores y parte de las propiedades de la interfaz *DataRTU* que se acaba de explicar.

El componente *rtutree* está programado para trabajar con el componente de la librería de Kendo (*treelist*), este componente muestra una lista de elementos que se pueden filtrar y seleccionar. Actualmente está configurado para mostrar todas las RTUs mostrando únicamente su matrícula, y solo filtra por esa misma propiedad a petición del cliente. En una versión previa el componente mostraba y filtraba tanto por el nombre de la estación como de la matrícula, mostrando ambos atributos. Ha continuación en la ilustración 7 se puede ver la visualización del componente *RTUtree*.

#### Filtrar por matrícula



The screenshot shows a web interface for filtering RTUs by their license plate (matrícula). At the top, there is a light blue header with the label 'Matrícula'. Below this is a white search input field. Underneath the input field is a list of three RTU IDs: '13HZK7' (highlighted in orange), '13HHCX' (in black), and '13CI09' (highlighted in orange). The entire list is enclosed in a light blue border.

Ilustración 7 – *RTUtree*

Una vez representadas todas las RTUs disponibles dentro del *treeview* nos queda la lógica de selección y envío de la RTU seleccionada al componente *main*. Esta lógica se realiza mediante los inputs y outputs del componente y del *treeview*.

Cuando el usuario selecciona un elemento de la lista el componente de kendo lanza un evento (Output) de tipo selección que nos devuelve el objeto seleccionado, no solo la matrícula sino el objeto completo con todas sus propiedades, que se guarda dentro del componente *RTUtree* sin enviarlo al *main*. En ese momento podemos seleccionar cualquier otra RTU sin realizar ninguna operación de conexión y recuperación de datos sobre la RTU.

Para diferenciar la RTU seleccionada se ha implementado un cambio de estilo CSS que responde a tres situaciones de las RTUs de las listas, el estado normal que se pintará en negro, el estado de RTU seleccionada que se pintará en naranja oscuro y el estado de *hover* o ratón por encima que será de un color naranja más claro, tal como se puede apreciar en la *ilustración 7*.

Para terminar con el componente hay que mencionar que este contiene un botón con el texto “Conectar” que al pulsar sobre este botón el componente emitirá un evento que contendrá la RTU seleccionada y será recibido y tratado por el componente *main*.

Tras esto el componente *main*, intentará conectar a la RTU seleccionada, aunque hay que mencionar que cada vez que se pulse el botón de conectar el componente *main* recibirá de nuevo el evento e intentará conectar desde el principio a la RTU recibida, haya cambiado o no la RTU emitida por el output del componente, en esencia una forma de reiniciar la conexión.

### 5.2.5 Componente svgDinamizer

Para comenzar a explicar en este componente lo primero es definir la palabra dinamizar del nombre del componente para no dar lugar a confusiones. La definición de dinamizar un SVG (*Scalable Vector Graphics*) y a lo que nos referimos al menos en este contexto, es la modificación de la imagen SVG para concordar con los datos y estado de la aplicación de manera visible en el SVG. Una vez explicado lo anterior se puede deducir que este componente será el encargado de a partir de los datos recibidos cambiar el estado del SVG mostrado en pantalla. Pero con las actualizaciones y mejoras constantes, este componente ha pasado de ser un simple dinamizador a ser de facto el controlador del SVG, que controla otras funciones relacionadas todas con el SVG. De las acciones añadidas al componente se encuentran la gestión de un menú contextual que se abre sobre cualquier interruptor mostrado en el SVG a eventos de botón sobre los dibujos en el SVG de botones, todo ello enviando eventos al *main* component.

En nuestra aplicación nosotros tenemos diez imágenes SVG para poder representar en la aplicación hasta diez interruptores diferentes visibles en todo momento y nuestro componente será capaz de analizar los datos decodificados recibidos del satélite y seleccionar la imagen más adecuada a la cantidad de interruptores que tiene cada RTU.

Una vez seleccionado la imagen SVG adecuada para la cantidad de interruptores que queremos mostrar el siguiente paso es dinamizar el SVG seleccionado a partir de los datos que vayamos recibiendo del satélite. Los datos del satélite recibidos normalmente no serán todos relevantes y habrá bastantes variables y estados que no nos interese representar y para ello tenemos la plantilla de la RTU que hemos seleccionado en el componente *'rtutree'*.

Utilizando estas plantillas de las RTUs que hemos recibido, el componente *svgDinamizer* es capaz de asignar todas las variables relevantes del SVG y seleccionar las variables de estado de la RTU que queremos mostrar para cada interruptor.



Ilustración 8 - SVG dinamizado con los cuatro estados

Los interruptores tienen cuatro estados distintos como hemos visto en la ilustración 8 dependiendo del estado de variable, que tenga asignada cada interruptor. Estos estados son: Estado desconocido, Estado de Interruptor Abierto, Estado de Interruptor Cerrado y Estado de fallo. Dados estos estados cuando inicializamos y dinamizamos por primera vez el SVG antes de recibir ningún dato el componente asigna todos los interruptores a estado desconocido y comienza a pintar las etiquetas de nombres, valores y otros datos del SVG que vienen definidos por la plantilla de la RTU y no varían con los datos recibidos.

Como se puede observar en la imagen el SVG contiene dos botones que se manejan con este componente, el primero es un “reset 48V” que en esencia reinicia el equipo remoto y el botón de interrogación General lo que hace es indicarle al componente main, que tiene que enviar una interrogación general. Estas acciones provocan que el componente svgDinamizer envíe mediante outputs la información de que se requiere realizar una acción u otra.

También otra cosa muy interesante es el menú contextual que se abre al hacer clic derecho sobre cualquier interruptor, sea el estado que sea, para poder lanzar un output indicando que se tiene que enviar un comando de apertura o cierre del interruptor como se puede ver en la ilustración 9



*Ilustración 9 - Menu Contextual del SVG*

Por último, hay que mencionar que no se verá reflejado ningún cambio en el SVG por acciones del usuario hasta que se reciba la confirmación por parte de la RTU, por ejemplo, al abrir un interruptor no se verá cambio alguno hasta que se reciba un mensaje con el nuevo valor de la variable de estado.

Para terminar de explicar el componente vamos a seguir paso a paso el funcionamiento de este componente desde que se recibe una plantilla de la RTU hasta que se recibe y pinta un cambio de estado de variable.

Para empezar el proceso desde el principio tenemos que recalcar que el componente tiene dos inputs de entradas de datos. El primer input es el que nombra como “data” y es donde recibimos los datos de la plantilla de la RTU y nos permite inicializar el svg de cero. El segundo es el input denominado “dataIG” y cuyo significado viene a ser data de la Interrogación General (IG), y es por qui por donde recibimos los cambios de variables para dinamizar los colores y otros valores correctamente.

El comienzo de las acciones relevantes del componente empieza cuando se recibe por el input de data una plantilla de RTU, esto sucede porque estamos iniciando una nueva conexión a una RTU, eso activa la inicialización del componente, que empieza a iterar sobre la plantilla para obtener los datos que mostrara el SVG, como son el nombre y matricula de la estación, el número de interruptores o en número de variables de información que se quiere mostrar junto a cada interruptor o si tiene alguna imagen alternativa.

Una vez obtenida la información, el componente mediante una petición http obtenemos de nuestro local, el fichero SVG que coincide con el número de interruptores que se quiere mostrar o en caso de imagen alternativa, intenta buscar primero la imagen alternativa, que en caso de no encontrarse realizara un *fallback* a la imagen por defecto dependiendo del número de interruptores.

Una vez obtenida la imagen SVG correspondiente el componente continúa inicializando el SVG realizando una dinamización inicial, es decir, completa todos los campos del SVG con los datos obtenidos de la plantilla y pone todos los interruptores de la imagen al estado '0' o desconocido que es el color azul, indicando que no tenemos ningún estado conocido para ese interruptor.

Mientras se realizaba el proceso anterior, el componente *main*, estaba intentando conectarse a la RTU y solicitar los estados de los interruptores de la RTU. Cuando la aplicación obtenga mensaje nuevo, sobre el estado de las variables, el componente *main* lo envía mediante el input 'dataRTU' a este componente. Estos datos pueden o no contener información sobre variables que nos interesen, para ello, iteramos sobre cada variable del mensaje, comprobando sus direcciones para comprobar si coinciden o no con alguna de las direcciones que el dinamizador está controlando en el SVG. En caso de que alguna dirección de las variables del mensaje coincida con las que estamos dinamizando actualizamos su valor en el SVG y en caso contrario ignoramos su valor y no realizamos ninguna acción.

Por este motivo cuando nosotros enviamos un comando de abertura o cierre mediante el menú contextual, esperamos la respuesta del servidor, antes de actualizar los cambios, porque si actualizamos los cambios al pulsar en el menú contextual la información mostrada no sería real y podría dar lugar a errores por parte del operador.

### 5.2.6 Servicio de codificación

Este servicio es el encargado de codificar y decodificar los mensajes que se van a utilizar en la aplicación mediante la aplicación del protocolo IEC60870-5-104.

Para que podamos explicar de manera clara y concisa como funciona este componente lo primero que vamos a hacer es explicar la estructura y el funcionamiento de los mensajes que vamos a tratar. Para ello vamos a trabajar en un ejemplo sencillo de un mensaje de inicio de transferencia de datos.

Tomando como ejemplo el siguiente mensaje

Mensaje de *StartDT* = [128, 1, 12, 11, 3, 104, 4, 7, 0, 0, 0, 102, 102, 102, 102]

El significado de los números tal cual sería el mostrado en la Tabla 1:

SIN	MIN	TAMAÑO MENSAJE	TAMAÑO COMANDO	COMANDO	CODIGO 104	CRC
128	1	12	11	3	104, 4, 7, 0, 0, 0	102, 102, 102, 102

Tabla 1 - Estructura de un mensaje codificado

Donde el código 104 es lo que tenemos que tratar, en este caso codificándolo para que de este resultado ya que es un mensaje que envía la aplicación para iniciar la transferencia de datos entre la aplicación y la RTU. Esto lo explicara más detalladamente cuando explique el proceso de codificado y decodificación de los 104.

El SIN o *System Info Number* indica cuales de los servicios de los que dispone el programa interfaz del satélite ha generado el mensaje. En nuestro caso nuestra aplicación siempre enviara o tratara mensajes de SIN 128, salvo una excepción y es cuando iniciamos la conexión por primera vez que comprobaremos la configuración de la conexión del satélite con nuestra aplicación y en caso de ser incorrecta la configuración, la corrección de esta.

Por tanto, los valores con los que vamos a trabajar son los que se muestran en la Tabla 2:

SIN	DESCRIPCIÓN
128	Es un valor reservado para las aplicaciones de usuarios que trabajen con el satélite
16	Son para mensajes relacionados con la administración y control del satélite, como pueden ser servicios de identificación, de medidas o de peticiones
26	Proporciona acceso al CLI ( <i>Command-line Interface</i> ) del programa del satélite

Tabla 2 - descripción de los SIN

El siguiente número es el MIN y en nuestra aplicación en todos los mensajes relevantes para esta sin excepción el MIN siempre tendrá el valor 1 y por tanto siendo transparente.



Después tenemos el tamaño del mensaje, este número es muy importante y si no es correcto la decodificación fallara, porque este número nos indica que, a partir de ese valor, el array del mensaje tiene exactamente ese tamaño, es decir, si contamos la cantidad de elementos del vector que compone el mensaje encontraremos que concuerda con este valor.

El siguiente valor es igual de importante que el anterior, ya que nos indica el tamaño del comando, y aplicando el mismo procedimiento que para el tamaño del mensaje, este nos indica cuantos valores del vector de mensaje tenemos que traducir conjuntamente como un comando, ya que un mensaje, puede y suele tener unos cuantos comandos juntos. En este caso como el valor es uno menos que el tamaño del mensaje nos indica que este mensaje contiene un solo comando.

El valor del comando viene a ser un parámetro, que nos indica de qué tipo de mensaje estamos tratando, actualmente utilizamos cuatro comandos diferentes, que van a ser descritos en la Tabla 3.

COMANDO	DESCRIPCION
1	Nos indica que estamos tratando con mensajes relacionados con la conexión
2	Nos indica que estamos tratando con mensajes relacionados con la desconexión
3	Nos indica que estamos tratando con mensajes relacionados con el inicio de transferencia de datos
83	Nos indica que estamos tratando con mensajes relacionados con las variables de estado de la RTU

Tabla 3 - Descripción de los tipos comunes de comandos

Si seguimos la información de la Tabla 3 podemos ver que al ser el valor del comando el número tres, podemos ver que se trata de un mensaje relacionado con la transferencia de datos, en este caso como menciono antes, es una petición de inicio de transferencia de datos.

Y por último tenemos el propio código 104 que es donde está la información más relevante y describiremos con más profundidad más adelante, aquí solo voy a explicar un poco por encima la estructura del código 104, ya que el ejemplo actual no es el mejor para explicar en profundidad el código 104.

El código 104 se compone de dos apartados, la primera es el APCI (*Associated Parameters of Control and Information*) o parámetros de control que es lo único que contiene el código 104 en el ejemplo del *Start Data Transfer* que es obligatorio en todos los códigos 104. Y la otra parte no presente es el ASDU (*Associated Statistics and Data Units*) o parámetros de datos, que no están presentes en el ejemplo, ya que la petición de *Start Data Transfer* no los requiere y causaría un error que cerraría la conexión

Por último, está el CRC esto siempre está presente en todos los mensajes y tiene un tamaño fijo de cuatro valores, pero para nosotros, este parámetro es totalmente transparente y al decodificarlo lo único que hacemos con él es eliminarlo.

Una vez explicado la estructura general del mensaje, nos vamos a centrar en explicar la estructura de la codificación 104 y como vamos a codificarlo según los parámetros que nos muestre la estructura. Para ello primero vamos a explicar un mensaje a decodificar 104 bastante sencillo y simple y continuaremos avanzando a partir de ese ejemplo en casos más complicados.

También iremos explicando cómo se decodifican los mensajes según el tipo de valor, por ejemplo, nosotros no podemos decodificar un valor de coma flotante (números reales) de 32 bits de la misma forma que un entero de 32 bits.

El primer ejemplo que vamos a tratar el caso más simple de mensaje, donde solo tiene un valor que decodificar siendo este valor de tipo entero de 32 bits. Pero antes vamos a profundizar en la estructura ASPI y ASDU mencionada anteriormente.

El mensaje de ejemplo es el siguiente:

Mensaje de ejemplo = [104, 15, 12, 0, 2, 0, 1, 130, 20, 0, 4, 0, 100, 0, 0, 0, 0]

APCI	ASDU
104 15 12 0 2 0	1 130 20 0 4 0 100 0 0 0 0

Tabla 4 - Estructura general del mensaje de ejemplo

Ahora que hemos dividido el código en las estructuras APCI y ASDU correspondiente vamos a ir explicando estas estructuras con más detalle comenzando por la APCI, ya que es la que menos valor y menos interviene en el proceso de decodificación.

La estructura APCI es fija en el contexto de esta aplicación y siempre tiene los tres mismos elementos en todos los mensajes. Tenemos el bit de inicio de comando que siempre se encuentra al principio del código con el valor 104. Después nos encontramos con el bit que nos indica el tamaño del código 104 incluyendo también la parte de la estructura ASDU y por último nos encontramos con 4 bits de control, estos bits de control en nuestra aplicación son transparentes al igual que el CRC y no realizamos ninguna acción con ellos a parte de eliminarlos.

Por tanto, la estructura del APCI quedaría según indica la Tabla 5.

Inicio de comando	Tamaño	Control
104	15	12 0 2 0

Tabla 5 - Estructura APCI del mensaje de ejemplo

A continuación, vamos a explicar la parte más interesante del código 104, la parte donde más hemos trabajado y la más complicada, que es donde se encuentran los datos del mensaje propiamente que sería el ASDU.

La estructura del ASDU es variable en tamaño y contenido ya que depende del tipo de valor y de la cantidad de valores que contenga, por eso mismo hemos seleccionado este código 104 en concreto ya que nos va a permitir ver uno de los problemas con la que nos hemos encontrado

durante el desarrollo de la aplicación, también procederé a explicar un poco cual es la solución que hemos encontrado y cuál es el problema que actualmente tiene ese tipo de mensajes.

Lo primero que haremos es explicar la estructura del ASDU y que significa cada parte de esta. Se indica también la decodificación del mensaje para que se pueda ir relacionando con la Tabla 6.

Mensaje Decodificado = {causeTx: 20, dir: 4, typeID: 'M\_SP\_NA\_1', value:[{dir:100, val:0}, {dir: 101, val:0}]}

<i>typeID</i>	Numero Variables	<i>CauseTx</i>	Dirección RTU (dir) [2 bits]	dirección de la variable	Valor de la variable
1	130	20	0 4	100	0

Tabla 6 - Estructura ASDU del mensaje de ejemplo

Lo primero que notaremos en la tabla es que faltan valores, en concreto los tres ceros que se encontraban al final del código y eso es porque los códigos 104 también tienen incluidos un CRC, que es preciso localizar y eliminar a la hora de decodificar los valores. En este caso el CRC solo contiene tres bits, los tres ceros del final. Pero dependiendo del tipo de dato puede variar y incluso añadir datos opcionales como fechas o localizaciones, aunque eso en nuestra aplicación no necesitamos esos datos y no los tratamos.

El primer bit que tratamos es el de *typeID* no hemos traducido los nombres de la tabla para que se puedan localizar fácilmente las propiedades en el mensaje decodificado y se pueda ver la relevancia, el *typeID* podemos decir que es el tipo de la variable y en nuestro caso es de tipo *M\_SP\_NA\_1* que viene a significar que es un numero natural de 32 bits, es decir sin signo ni datos adicionales.

Los otros tipos de datos (*TypeID*) que nuestra aplicación trata de forma activa son los tipos *M\_ME\_NC\_1* y *M\_ME\_NA\_1*, el primer tipo representa a números reales de 32 bits que pueden ser positivos o negativos y suelen ser utilizados para las variables de datos de nuestra aplicación. El segundo tipo de dato representa los números enteros con signo de 16 bits, implicando que este tipo tiene un rango mas limitado de valores que los otros dos tipos.

Hay más tipos y cuando terminemos de explicar la estructura ASDU de los códigos 104 pondremos ejemplos de los tipos de valores que más vamos a utilizar en la aplicación, pero por el momento este es el más simple y sencillo de explicar.

La siguiente parte de la estructura ASDU es clave y una parte donde ha dado problemas al decodificar y es donde se indica el número de variables que contiene el código 104, y aquí es donde debemos llamar la atención respecto al mensaje decodificado y ver que contiene 2 variables, una variable en la dirección 100 con valor 0 y otra variable con dirección 101 y valor 0. Como podemos observar no coincide con el número que nos indica el mensaje de 130 y esto es un caso específico de este mensaje.

Este mensaje es especial porque el satélite cuando nos envía la información si hay dos variables consecutivas en el mismo mensaje con el mismo valor el satélite nos añade un bit adicional indicando que son valores consecutivos con el mismo valor y eso al traducirlo del hexadecimal que envía el satélite al decimal significa que en vez de aparecer como dos bits independientes le suma al número de variables el valor de 128 quedando en este caso el valor 130.

Para solucionar este problema he tenido que modificar la decodificación para que detecte los valores mayores de 128 en el número de variables y en ese caso realizar la resta y trabajar a partir del resultado, esto nos lleva al problema que tenemos en la aplicación actual y es el caso de que un comando 104 realmente contenga más de 128 variables. Ya que el proceso de decodificación realizaría la decodificación de los primeros  $n-128$  valores siendo  $n$  mayor que 128. En este caso el proceso de decodificación fallaría y no sería capaz de devolvernos ningún resultado y provocando la pérdida de información.

En el resto de los casos este valor siempre ha sido exactamente la cantidad de variables que encontramos el código 104 del mensaje.

Después del Numero de variables tenemos el *CauseTx* o la causa del mensaje, que básicamente nos indica que ha causado la recepción de este mensaje, que en este caso es la causa 20 que significa que se ha recibido como parte de una respuesta a una petición por parte de nuestra aplicación de una interrogación general.

Ha continuación tenemos 2 bits que nos están indicando la dirección de la RTU donde se ha originado este mensaje en este caso tenemos la dirección [0 4] que se decodifica en 4, en nuestra aplicación no tratamos de ninguna forma este valor a parte de decodificarlo.

Por último, tenemos las variables que contiene el sistema, esta parte no es fija, sino que dependiendo de la cantidad de variables y del tipo encontraremos una cantidad de bits diferente indicando otras posibles propiedades de la variable (como fechas). En nuestro caso tenemos puesto un ejemplo donde tenemos 2 variables consecutivas con el mismo valor, eso lo podemos ver al observar el que el número de variables es mayor que 128 como ya se explicó anteriormente.

De forma general los diferentes estados de las variables serán suministrados de en forma de dos campos de bits variables, según el tipo de dato con el que estemos tratando, en este ejemplo y el empleado en esta aplicación consistirá en 2 bits, uno para la dirección y otro para el valor.

Estos valores los recibimos dentro del mensaje codificados en forma de bits de *Little Endian*, es decir están ordenados del bit menos significativo al bit más significativo. Esto es un problema para entender fácilmente la decodificación ya que la mayoría de las operaciones que realizamos en la vida cotidiana utilizamos la codificación *Big Endian* o lo que es lo mismo cuando escribimos un numero ordenamos el numero del bit mas significativo al menos significativo.

A continuación, vamos a ilustrar con el ejemplo del número once como sería la codificación *Big Endian* y *Little Endian*. Primero vamos a codificar el numero 10 en binario utilizando el *Big Endian*, ya que es la forma mas natural de ver los números y así poderlo comparar posteriormente con la codificación de *Little Endian*.

Como podemos observar el número once en binario *Big Endian* es 1011 que se puede traducir a la suma de las siguientes potencias de 2, tal como muestra la tabla 7.

Decimal	$2^3$	$2^2$	$2^1$	$2^0$
11	1	0	1	1

Tabla 7 - Codificación *Big Endian*

Ahora vamos a comparar como seria la codificación del número once en *Little Endian* (Tabla 8) respecto a la codificación *Big Endian*.

Decimal	$2^0$	$2^1$	$2^2$	$2^3$
11	1	1	0	1

Tabla 8 - Codificación *Little Endian*

Como se puede observar la principal diferencia está en la ordenación de las potencias de la tabla. En la tabla 7 teníamos el bit mas significativo o la potencia mas grande en la primera columna de la izquierda. Mientras que en la tabla 8, el primer bit es el de la potencia menor o el bit menos significativo. Por tanto, el número once en binario *Little Endian* seria 1101.

Esto es importante que lo tengamos claro puesto que si intentamos decodificar un binario en *Little Endian* como si fuese un *Big Endian* nos devolverá un numero completamente erróneo. Tomando el ejemplo del binario del once en *Little Endian* y lo decodificamos como *Big Endian* obtendremos el resultado mostrado en la tabla 9.

Binario LE	$2^3$	$2^2$	$2^1$	$2^0$	Resultado
1101	1	1	0	1	13

Tabla 9- decodificación de LE como BE

Como podemos ver en la tabla 9 si intentamos decodificar un *Little Endian* como *Big Endian* nos devuelve como resultado en decimal trece, que son la suma de la tabla ( $8+4+1 = 13$ ), y por tanto un número erróneo.

Explicado esto último ya podemos ser capaces de leer y decodificar un mensaje recibido de la RTU sin mayor problema.

### 5.2.7 Servicio IGWS2

El nombre de este servicio es el nombre de una librería interna que utilizaba *JavaScript* para manejar las peticiones y recepción de mensajes de la RTU. Aunque ha mantenido el nombre y la librería se ha basado en los contenidos de *JavaScript*, pero se ha realizado bastantes modificaciones para hacerlo compatible con el *framework* de Angular.

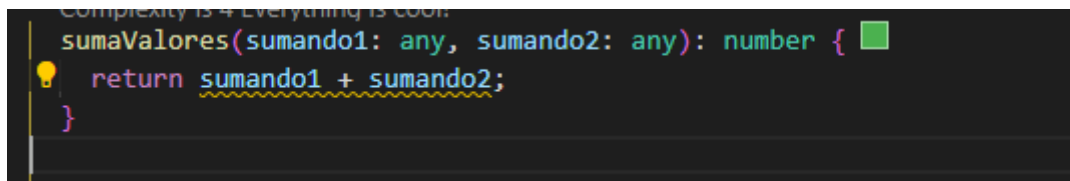
Este servicio es básicamente la aplicación del servicio de Angular *HttpClient* que contiene las peticiones básicas HTTP. En nuestra aplicación solo utilizamos las peticiones GET para obtener los mensajes de la RTU que son procesados en el componente main y las peticiones POST que son las que empleamos para enviar mensajes, ya sea los mensajes de conexión de main, como las peticiones de reinicio de 48V.

En resumen, este servicio contiene la implementación y utilización del servicio de Angular *HttpClient* y simplemente es la puerta de entrada y salida de los mensajes de nuestra aplicación con la RTU.

## 6. Pruebas realizadas a la aplicación

Nuestra empresa exige un cierto nivel de calidad en las aplicaciones, estos criterios de calidad tienen que superarse antes de poder entregarse algo a nuestros clientes y para ello realizamos en su mayoría pruebas unitarias.

Las pruebas unitarias consisten en ejecutar pequeños fragmentos de código, usualmente funciones, encargados de comprobar que dado un determinado estado de entrada a una función siempre resulte en el mismo resultado. Esto es mas visible si consideramos una función simple como puede ser una función que recibe dos parámetros numéricos y devuelva el resultado de la suma de ambos números.



```
sumaValores(sumando1: any, sumando2: any): number {  
  return sumando1 + sumando2;  
}
```

Ilustración 10 - Función de ejemplo Suma

Como se puede observar en la función de ejemplo, podemos llamar a la función de la siguiente forma, `sumaValores(2,3)` y esperar que nos devuelva el numero 5. Lo que hemos hecho en la frase anterior sería una prueba unitaria.

Pero eso no bastaría para comprobar todas las casuísticas, por ejemplo, si utilizáramos la siguiente llamada de función, `sumaValores(true, 3)` nos devolvería como resultado 4.

Esto es un fallo de la función ya que nosotros solo queremos sumar números y el booleano `true` no es un número, pero cuando se emplea como operador en operaciones numéricas este se traduce al valor numérico 1. Pero es una prueba válida realizar la llamada anterior y esperar que falle la función.

Por tanto, gracias a las dos pruebas unitarias anteriores de la función hemos podido comprobar que la función esta mal implementada, ya que permite que los tipos de los parámetros sean de cualquier tipo. Entonces procederíamos a cambiar la función para que ambos parámetros solo acepten como sumando variables de tipo numérico.

Según todo lo que hemos explicado anteriormente en este apartado una prueba unitaria consiste, en resumen, en que nosotros escogemos una función de un componente, le pasamos valores y comprobamos si el resultado concuerda con lo esperado.

Ahora bien, las pruebas unitarias no es el único requisito de calidad que nuestra empresa exige, sino que también, nos exige al menos un ochenta por ciento de cobertura. La cobertura es la cantidad de líneas y condiciones, como instrucciones *if*, que están cubiertos y comprobados por una prueba unitaria. En la ilustración 11 podemos observar el ultimo resultado la cobertura de nuestro proyecto.

Summary	
<b>Generated on:</b>	7/3/2020 - 11:19:10 AM
<b>Parser:</b>	CoberturaParser
<b>Assemblies:</b>	14
<b>Classes:</b>	20
<b>Files:</b>	20
<b>Covered lines:</b>	832
<b>Uncovered lines:</b>	166
<b>Coverable lines:</b>	998
<b>Total lines:</b>	3282
<b>Line coverage:</b>	83.3% (832 of 998)
<b>Covered branches:</b>	171
<b>Total branches:</b>	241
<b>Branch coverage:</b>	70.9% (171 of 241)

Ilustración 11 - Resultados de la última cobertura del proyecto

En la ilustración 11 se puede observar como tenemos dos porcentajes de cobertura diferentes, la de *line coverage* (cobertura de líneas) y los resultados de *branch coverage* (cobertura de ramas) que es en esencia la cantidad de instrucciones condicionales que se cubren mediante las pruebas, ya que una condición divide la ejecución del código en varias ramas de ejecución según la condición.

Si visualizamos las condiciones mencionadas, podemos observar que la cobertura de línea supera el ochenta por ciento, pero que la cobertura de rama solo llega al setenta por ciento. Este resultado ha superado la prueba de cobertura del ochenta por cien porque no tenemos en cuenta los porcentajes de cobertura individuales (de línea y de rama), sino que utilizamos la siguiente formula de cálculo.



$$\text{Cobertura} = (\text{covered lines} + \text{covered branches}) / (\text{coverable lines} + \text{total branches})$$

Por tanto, la cobertura real es igual a  $(832 + 171) / (998 + 241) = 80.9\%$

Esto significa que como mínimo debemos tener una prueba unitaria por función en el caso que se compruebe en una única prueba todas las ramas posibles, y por tanto, los resultados de dicha función o, podemos realizar una prueba unitaria por cada rama o condición de una función.

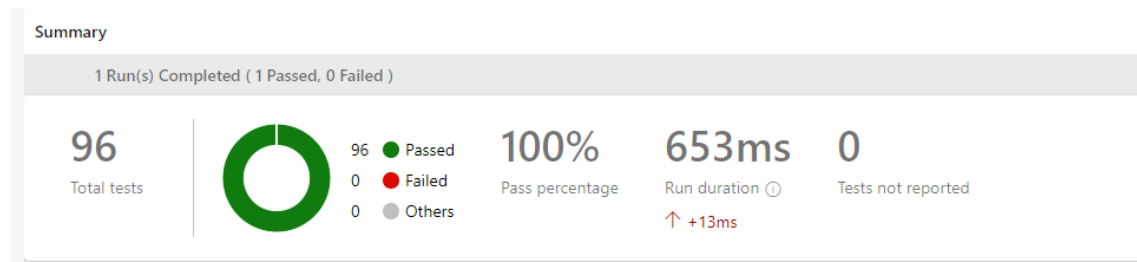


Ilustración 12 - Resultados de las pruebas unitarias del proyecto

Como se puede observar en la ilustración 12, nuestra aplicación ha implementado un total de noventa y seis pruebas unitarias en total y apenas llega para cubrir un ochenta por ciento de la cobertura de la aplicación. Realizar noventa y seis pruebas unitarias no es nada sencillo y consume bastante tiempo. De hecho, realizando un análisis a posteriori de los tiempos hemos podido observar que más de un tercio del tiempo de desarrollo del proyecto, lo hemos dedicado a realizar pruebas. Y, además, otra parte significativa del tiempo a corregir los fallos detectados durante la realización de las pruebas.

Para realizar estas pruebas utilizamos los *frameworks* de *Jasmine* y *karma* conjuntamente. Estos *frameworks* nos permiten ejecutar componentes de manera independiente del flujo de la aplicación y controlar las variables de entorno para realizar las pruebas. Esto significa que, si por ejemplon estamos probando la desconexión del componente *main*, no tenemos que realizar todo el proceso de conexión, es más, no necesitamos una conexión con la RTU. En todo caso, nosotros ni siquiera comprobamos que se produzca la desconexión de la RTU, recibiendo el mensaje de confirmación de desconexión de la RTU. Simplemente comprobamos que, si llamamos a nuestra función de desconexión, llamemos al servicio que se encarga de mandar el mensaje de desconexión, sin comprobar que el mensaje sea correcto ni que el servicio funcione correctamente. Nuestro enfoque siempre está orientado ha comprobar que el componente que estamos probando sea correcto y, el resto de los servicios, componentes o librerías externas a nuestro componente suponemos que funcionan perfectamente.

Para concluir este apartado es interesante mencionar que también se ha realizado una prueba de campo con nuestro cliente, aunque el autor del proyecto y desarrollador no ha estado implicado en este proceso, sí puede decir que ha sido un éxito completo.

## 7. Conclusiones

Durante este proyecto de fin de grado hemos podido adquirir muchas experiencias, en especial las diferencias entre un proyecto académico y un proyecto comercial para una importante empresa. En un proyecto comercial como es el sujeto de nuestro trabajo las exigencias de calidad son mucho mas altas que en uno académico, se realizan muchísimas mas pruebas como son las pruebas unitarias y la cobertura de líneas mínima.

También hay que destacar que bastantes de los conocimientos aplicados en este proyecto fin de grado, los hemos adquirido en varias asignaturas dentro de este grado académico. En especial me gustaría destacar las asignaturas de EDA (Estructuras de Datos y Algoritmos) y ISW (ingeniería del Software) que nos han enseñado las bases de conocimientos sobre los algoritmos empleados y como planificar y estructurar adecuadamente nuestro proyecto.

Pero también ha supuesto que nosotros podamos entrar en un campo fascinante como es la telecomunicación de satélites. El poder aprender la codificación 104, que es un estándar para la telecomunicación de satélites.

A parte hemos conseguido utilizando la codificación 104, crear una aplicación web en Angular 7 y realizar la actualización a Angular 8 en mitad del proyecto y aun con estas dificultades hemos sido capaces de mostrar la información de una RTU de una instalación remota mediante la telecomunicación con satélites y operar con las RTU de manera remota. No de manera arcaica mediante largos comandos de consola, sino de una forma mas visual y intuitiva mediante la manipulación de las imágenes SVG, la utilización de tablas o el uso de menús contextuales.

Y por último nos gustaría destacar algo aún mas importante que todo lo anterior, y es el aprender a trabajar dentro de un grupo interprofesional donde nosotros realizábamos las labores de programación.

## 8. Ideas de mejora de la aplicación

En este apartado se comentan algunas ideas de mejora que han ido surgiendo durante el desarrollo de nuestra aplicación, algunos cambios serán menores relacionados con la mejora de experiencia del usuario mientras que otras sugerencias serían tan grandes como cambios de la estructura de toda la aplicación.

Uno de los problemas de la aplicación es que no sigue la estructura clásica de *frontend* y *backend*, ya que en nuestra aplicación toda la carga y procesamiento de los datos recibidos del satélite se está realizando en un servicio dentro del *frontend* y esto es la definición básica de un *backend*.

Esto supone que, si se recibe muchos mensajes o mensajes muy largos y se tienen que procesar, este procesado de los datos va a recaer en los recursos del ordenador del usuario que este cargando la aplicación. Esto puede suponer que el ordenador del cliente se pueda colapsar al utilizar nuestra aplicación puesto que estamos consumiendo muchos de los recursos del ordenador del usuario.

Para solucionar este problema y mejorar la aplicación se ha sugerido la posibilidad de realizar un *backend* donde se trate todo el proceso de comunicación con la RTU, incluyendo el manejo del estado de la conexión, el propio proceso de conexión que actualmente se realiza en el *main.component.ts*, la codificación y decodificación de los mensajes que se envíen tanto del *frontend* a la RTU como de la RTU al *frontend*.

Realizar esta propuesta supondrá que el ordenador del usuario no tenga utilizar sus recursos para realizar los cálculos de la codificación y decodificación, sino que sea la maquina donde está instalada la aplicación web. Pero, aunque las mejoras sean grandes también hay que tener en cuenta el coste de realizar esta idea de mejora, ya que supone realizar un *backend* prácticamente de cero y modificar el *frontend* de forma que prácticamente se tenga que realizar una reforma del *frontend* total de modo que en la práctica es posible que la realización de un *frontend* de cero sea la mejor solución.

Si vamos a hacer una reforma de la aplicación integral para aplicar los cambios sugeridos en los párrafos anteriores también tendríamos que aplicar otras mejoras importantes tanto para la calidad del código como para la eficiencia en la ejecución de nuestra aplicación.

Para ello en la reforma de la aplicación, en concreto del *frontend*, proponemos empezar a utilizar Angular 9 o Angular 10 desde el principio en nuestro proyecto, evitando tener que hacer la actualización como nos ocurrió durante la realización de este proyecto que tuvimos que actualizar de Angular 7 a Angular 8 porque algunas librerías no suportaban Angular 7.

También utilizar una versión superior de Angular como puede ser Angular 9 supone que algunas de las características de Angular 8 sean bastante mejoradas y con un mayor soporte.

A parte de lo anterior otra mejora ya no a nivel de usuario sino a nivel de programador que se puede aplicar a nuestra aplicación es mejorar la estructura y organización del código. Ya que actualmente tenemos muy pocos componentes, pero con un código muy largo y complejo. Esto nos supone que para realizar pruebas, correcciones o otros cambios y mejoras pueda dificultarse bastante, especialmente si el programador no es quien ha creado la aplicación.

Para ello proponemos utilizar una estructura similar a la que tienen los proyectos nuevos de la empresa donde tenemos una estructura clara y simple donde podemos saber el lugar de cada componente y servicio según su utilización. A continuación, describiremos con más detalle esta estructura.

La nueva estructura de los proyectos se ha planteado en tres grandes grupos de elementos. El primero de los grupos es el que denominamos *core* o núcleo y componen todos aquellos servicios, componentes o utilidades necesarias para el correcto funcionamiento del proyecto. En esta agrupación podremos encontrar componentes como sería el *login* o registro de una aplicación o servicios importantes como pueden ser el servicio de peticiones HTTP o un servicio de manejo de las credenciales de la aplicación. El núcleo está ya programado con lo básico, manejo de credenciales, servicios de peticiones HTTP para que, al empezar un nuevo proyecto, esta carpeta se realice el mínimo número de cambios posibles por parte del programador de turno.

El segundo grupo importante de la nueva estructura de proyecto sería los *modules* o módulos, estos no hay que confundirlos con los *module* de Angular, sino que en este contexto hacen referencia a un conjunto de elementos de la aplicación que son independientes de otros módulos de la aplicación y representan una funcionalidad concreta, en nuestro caso solo tendríamos un módulo que sería el *main.component*.

Por último, pero no menos importante serían los componentes *shared* o compartidos, los elementos y servicios dentro de esta categoría son complementos que se pueden utilizar en varios módulos. Un claro ejemplo de esto es el componente *rtutree.component* este componente básicamente recibe unos datos y lo muestra en forma de árbol. Este componente podría ser reutilizado en otros módulos en caso de haberlos.

Esta nueva estructura de datos está diseñada para aplicarse con una nueva metodología de programación en Angular, la denominada *Lazy loading*, que en español se traduciría como carga difusa. Esta nueva metodología difiere totalmente de la que hemos empleado en nuestro proyecto ya que la filosofía es justamente la contraria.

En el proyecto actual cuando ejecutamos la aplicación, cargamos todos los componentes y servicios de inmediato, es decir carga todos los módulos de la aplicación, aumentando el tiempo inicial de carga de la aplicación y el consumo de memoria.

La metodología *Lazy Loading* en cambio solo cargaría los módulos conforme fuesen necesarios y en un principio solo cargaría el módulo del *login* y los módulos que se puedan acceder desde ese componente.

Como solo tenemos un módulo que sería el *main.component* este cambio de estructura con su nueva metodología no parece reportar ningún beneficio, y es cierto. Estos cambios no reportan apenas beneficios a la aplicación actual, pero si se desea ampliar las funcionalidades de dentro del proyecto, entonces sí que supondrá una mejora importante.

# APENDICE

---

## 1. Glosario de términos

**Frontend:** Es la parte de la aplicación que contiene los contenidos y funcionalidades que se visualizan en el ordenador del usuario

**Backend:** Es la parte de la aplicación que corresponde a la gestión de datos de la aplicación localizado en el servidor de la aplicación

**RTU:** *Remote Terminal Unit*, hacen referencia a los ordenadores que controlan de sistemas físicos y se acceden de forma remota.

**SVG:** (*Scalable Vector Graphics*) un formato de imagen definido mediante vectores escalables, que se puede manipular muy fácilmente para cambiar sus propiedades

**Framework:** Es el entorno de trabajo que incluye las prácticas y sistemas de soporte estandarizados para la programación.

**Typescript:** Lenguaje de programación tipado basado en el lenguaje de programación *JavaScript*.

**Angular:** Es un *framework* basado en *TypeScript* y facilita la programación.

**Time out:** Un *time out* en programación es cuando se activa una ejecución de código retrasada por una cuenta a tras normalmente indicada en milisegundos

**Componente:** hace referencia a una unidad de trabajo dentro del *framework* de Angular que se compone de un fichero de código *TypeScript* donde se hace referencia a una plantilla de código HTML y estilos CSS. (estos dos elementos pueden ir en ficheros separados referenciados por el fichero de *Typescript*)

**Módulo:** Es donde se declaran todos los componentes y servicios de Angular para poder ser cargados y utilizados, también es responsable de importar las dependencias de los componentes.

**Servicio:** Es un conjunto de funciones que no tienen representación visual (plantilla HTML) y sirve de soporte para los componentes, ejecutando peticiones de los componentes.