UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS
Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE VISUALIZACIÓN PARA LA MONITORIZACIÓN DE VARIABLES AMBIENTALES Y ELÉCTRICAS.

RODRIGO LÓPEZ PUENTE

2020

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS
Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE VISUALIZACIÓN PARA LA MONITORIZACIÓN DE VARIABLES AMBIENTALES Y ELÉCTRICAS.

Autor
RODRIGO LÓPEZ PUENTE

Tutor
ÁLVARO GUTIÉRREZ MARTÍN

2020

Resumen

El propósito de este Trabajo Fin de Grado consiste en el diseño e implementación de un sistema que permite monitorizar variables ambientales y de consumo eléctrico. Para ello se utilizarán los datos obtenidos por diferentes medidores instalados en la red eléctrica de la E.T.S. Ingenieros de Telecomunicación, en sus diferentes edificios.

Para poder gestionar y almacenar todos los datos recibidos, y a su vez desplegar la aplicación a través de la cual podremos visualizar y monitorizar todas las variables, se ha instalado un sistema operativo Debian 10 en un servidor del Laboratorio de Robótica y Control (ROBOLABO) de la ETSIT, con acceso a internet y una IP pública.

Los datos de los diferentes dispositivos son recibidos en el servidor en texto plano, pero para conseguir un correcto funcionamiento y hacer un uso de los mismos, se almacenan en diferentes bases de datos MySQL. Para ello se han diseñado una serie de scripts en R que crean y rellenan las diferentes bases de datos con las mediciones recibidas de los distintos medidores existentes en la escuela.

Es necesario servir los datos almacenados en las bases de datos para poder representarlos de forma gráfica, por ello se ha desarrollado una aplicación de backend con Express, a través de la cual se sirven los datos accediendo a unos endpoints. Por último se ha diseñado una aplicación web con el framework de React, para poder representar de forma eficiente y clara las diferentes variables obtenidas de los dispositivos de lectura.

Palabras clave: Monitorización, bases de datos, Debian, servidor, MySQL, Express, React.

Abstract

The purpose of this End-of-Grade Project consists of the design and implementation of a system that allows monitoring environmental and electricity consumption variables. For this purpose will be used the data obtained by different meters installed in the electrical network of the E.T.S. Ingenieros de Telecomunicación, in its different buildings.

In order to manage and store all the received data, and at the same time deploy the application through which we will be able to visualize and monitor all the variables, an operating system Debian 10 has been installed in a server of the Robotics and Control Laboratory (ROBOLABO) of ETSIT, with access to Internet and a public IP.

The data from the different devices are received in the server in plain text, but to get a correct operation and make use of them, they are stored in different MySQL databases. For this purpose, a series of R scripts have been designed to create and fill in the different databases with the measurements received from the different meters at the school.

It is necessary to serve the data stored in the databases in order to be able to represent them graphically, which is why a backend application has been developed with Express, through which the data is served by accessing some endpoints. Finally, a web application has been designed with the React framework, in order to efficiently and clearly represent the different variables obtained from the reading devices.

Keywords: Monitoring, databases, Debian, server, MySQL, Express, React.

Agradecimientos

Querría agradecer a todas las personas del Robolabo, en especial a mi tutor Álvaro Gutiérrez por su apoyo, y ayuda recibida; también agradecer todas las facilidades que me han otorgado a la hora de llevar a termino este Trabajo Fin de Grado.

A mi familia, especialmente en estos últimos meses, que para nada han sido fáciles de llevar, me ha apoyado en todo momento, dándome ánimos y recordándome que con esfuerzo y constancia conseguiría sacar lo mejor de mi mismo.

Índice general

Resumen v

Abstract vii

Agradecimientos ix

Índice General xi

Índice de Figuras xiii

Índice de Tablas xv

Lista de Acrónimos xvii

1. Introducción, objetivos y estructura 1
- 1.1. Introducción a la monitorización . . . 1
  - 1.1.1. Tipos de monitorización . . . 1
- 1.2. Motivación . . . 3
- 1.3. Objetivos . . . 4
- 1.4. Estructura . . . 4

2. Estado del arte 7
- 2.1. Recepción de datos . . . 7
- 2.2. Tecnologías para el tratamiento de datos . . . 8
  - 2.2.1. R . . . 8
  - 2.2.2. Python . . . 9
  - 2.2.3. SQL . . . 9
- 2.3. Tecnologías para la creación de una API . . . 9
  - 2.3.1. Express.js . . . 11
  - 2.3.2. Spring . . . 11
- 2.4. Tecnologías para la creación de una aplicación web . . . 12
  - 2.4.1. Angular . . . 13
  - 2.4.2. React . . . 13
- 2.5. Herramientas seleccionadas . . . 16

3. Desarrollo e implementación **17**
3.1. Puesta a punto del servidor 17
3.2. Scripts en R para almacenar los datos en BBDD 19
3.3. Desarrollo de una API REST 22
3.4. Desarrollo de la aplicación web 26

4. Validación y pruebas **37**
4.1. Pruebas para la población de las Bases de datos 37
4.2. Pruebas de validación de la API 39
4.3. Pruebas de la renderización de la Aplicación web 42

5. Conclusiones y líneas futuras **45**
5.1. Conclusiones 45
5.2. Líneas futuras 46

Bibliography **47**

A. Aspectos éticos, económicos, sociales y ambientales **49**
A.1. Impacto social 49
A.2. Impacto económico 49
A.3. Impacto ético 50
A.4. Impacto ambiental 50

B. Presupuesto económico **51**

C. Manual del desarrollador **53**
C.1. Manual de la Aplicación de Express.js 53
C.2. Manual de la Aplicación de React 59
C.2.1. Iniciar aplicación durante su desarrollo 65
C.2.2. Desplegar la aplicación en el servidor 65
C.2.3. Modificar la aplicación una vez desplegada 65

Índice de figuras

1.1. Modelo de monitor IntelliVue MP50 2
1.2. Captura de monitorización de un sistema en Nagios 3

2.1. Dispositivos para la monitorización 7
2.2. Diagrama de la arquitectura cliente-servidor 10
2.3. Diagramas de los diferentes diseños de las aplicaciones web . . . 12
2.4. Diagramas del flujo de la información 14
2.5. Diagramas del funcionamiento del Virtual DOM. 15

3.1. Estructura final de las bases de datos 21
3.2. Contenido de las bases de datos de los edificios A y D 22
3.3. Ámbito de trabajo de la aplicación de *back-end* 23
3.4. Rutas de acceso a la API 25
3.5. Estructuración del proyecto de la aplicación de *back-end* 26
3.6. Ámbito de trabajo de la aplicación de *front-end* 27
3.7. Aplicación básica de *react* 28
3.8. Contenido del archivo *index.js* 29
3.9. Contenido del archivo *app.js* 30
3.10. Mockup de la pagina principal de la aplicación web 31
3.11. Gráfica de la energía consumida al mes en el Edificio A 33
3.12. Página principal de la aplicación web 33

4.1. Información de la base de datos de un medido del Edificio A . . 38
4.2. Respuesta de la API a una petición 39
4.3. Test del *end-point "pro_ energy"* 40
4.4. Test del *end-point "pro_ energy/month"* 41
4.5. Test del *end-point "pro_ power"* 42
4.6. Test del tiempo en representar gráficamente 100.000 puntos . . . 43

C.1. Código del archivo *server.js* 54
C.2. Código del archivo *Routes.js* 54
C.3. Directorio Controllers 55
C.4. Directorio de las conexiones a las bases de datos 55
C.5. Contenido del fichero *dbEdificioA.js* 56
C.6. Contenido del fichero *EdificioA.js* 57

C.7. Contenido del fichero *package.json* 58
C.8. Contenido del directorio de la aplicación de React 60
C.9. Contenido del fichero *App.jsx* 61
C.10. Contenido del fichero *Home.jsx* 61
C.11. Contenido del fichero *Cuadricula.jsx* 62
C.12. Contenido del fichero *Cuadrito.jsx* 62
C.13. Componente *<Cuadricula/>* una vez renderizado 63
C.14. Contenido de la variable datapoints 64
C.15. Contenido del método return del fichero *EdificioA.jsx* 64

Índice de tablas

B.1. Costes de personal. 51
B.2. Costes de recursos materiales. 52
B.3. Costes totales. 52

Lista de Acrónimos

OS: Operating System.

ETSIT: Escuela Técnica Superior de Ingenieros de Telecomunicación.

SPA: Single Page Aplication.

CLI: Command Line Interface.

IDE: Integrated Drive Electronics.

DOM: Document Object Model.

HTML: HyperText Markup Language.

USB: Universal Serial Bus.

BIOS: Basic Input/Output System.

SSH: Secure Shell.

FTP: File Transfer Protocol.

TCP: Transmission Control Protocol.

Capítulo 1

Introducción, objetivos y estructura

1.1. Introducción a la monitorización

La monitorización consiste en la recopilación de diferentes variables de un sistema, y a través de ella poder obtener información relevante sobre el mismo. Por lo general se utiliza para detectar errores o anomalías en un sistema [1]. Existen campos muy diferentes donde se aplica la monitorización, como por ejemplo el control de parámetros médicos, seguimiento del estado en tiempo real de un sistema informático, control y vigilancia de un lugar o la monitorización de actividad dentro de las redes sociales [2, 3].

Este tipo de práctica dentro de los sistemas es muy común hoy en día. Debido a la cantidad de información que ofrece al usuario, muchos de los sistemas de monitorización son capaces de hacer un seguimiento en tiempo real del sistema, funcionalidad muy útil para poder controlar el estado actual del sistema monitorizado. Existen otros sistemas que no buscan la instantaneidad de la monitorización, sino que buscan patrones de fallos del sistema en su historial de datos, para poder anteponerse a problemas o errores futuros cuando estos patrones se repiten.

1.1.1. Tipos de monitorización

La monitorización de sistemas está presente en un gran ámbito de campos muy diferentes. Dentro de cada campo, el sistema a monitorizar y la forma de realizarlo es diferente, pero el fin no cambia: todos pretenden obtener información para conocer el estado del sistema monitorizado y prever diferentes anomalías que se puedan encontrar.

A continuación se exponen diferentes entornos donde se utiliza la monitorización de sistemas:

- **Monitorización en el campo de la medicina**
  Dentro del campo de la medicina existen gran cantidad de variables monitorizadas, dependiendo de los indicadores que se requieran conocer en cada circunstancia. Uno de los mejores ejemplos en este campo es la monitorización de las constantes vitales, crucial en las unidades de cuidados intensivos, donde se controlan de forma exhaustiva y estricta aquellos parámetros que indican el estado del paciente en todo momento [4].

Figura 1.1: Modelo de monitor IntelliVue MP50

- **Monitorización de sistemas informáticos**
  En el campo de la informática la monitorización de los dispositivos es esencial, y más aún cuando se trata de un gran sistema, con un elevado número de dispositivos conectados a una red. Para ello se utilizan diferentes métodos. Estos son ejecutados desde un ordenador central y recopilan información de todos los ordenadores, servidores o dispositivos conectados a la red. Algunos de los más conocidos son syslog [5] y Nagios [6]. Estos sistemas de monitorización ofrecen gran cantidad de información valiosa, plasmando la información en intuitivos gráficos para obtener conclusiones del estado del sistema de una forma muy sencilla.

Figura 1.2: Captura de monitorización de un sistema en Nagios

- **Monitorización ambiental**
  Consiste en la recolección de mediciones sobre diferentes variables ambientales, como el Dióxido de Nitrógeno (NO_2), Ozono (O_3) o Dióxido de Azufre (SO_2). Existen proyectos en las ciudades más grandes de Europa donde se miden las variables anteriores para poder conocer el Índice de Calidad del Aire [7]. Y según los valores de este índice existen diferentes escenarios de actuación que suelen conllevar restricciones de circulación por el centro de las grandes ciudades.

Las diferentes formas de monitorización anteriormente comentadas, son ejemplos concretos del uso de la monitorización hoy en día. Estos sistemas son útiles debido a que permiten conocer el estado de ciertas variables en tiempo real, para así conocer los efectos sobre los sistemas estudiados. Pudiendo a su vez tomar las decisiones necesarias, para el correcto funcionamiento de los sistemas monitorizados en caso de encontrar anomalías en los datos proporcionados de la monitorización.

1.2. Motivación

Existen multitud de proyectos donde la recolección de datos es clave para conocer el estado en el que se encuentra el sistema. El gran problema existente de estos sistemas, es el procesado de la gran cantidad de datos disponibles. Para que todos los datos disponibles puedan ser útiles y así poder obtener conclusiones de manera ágil, se debe estudiar cuáles son los datos que nos aportan la información más valiosa, y cuál es la forma más apropiada de mostrar los datos. La motivación principal para la realización del presente Trabajo Fin de Grado surge de la necesidad de crear un sistema completo, que estructure de forma correcta los datos de monitorización energética de la ETSIT. Además

deben servirse a una aplicación web donde se representen de forma coherente y visual por medio de gráficas.

1.3. Objetivos

Los objetivos de este Trabajo de Fin de Grado son los siguientes:

- Acondicionar un servidor con el nuevo sistema operativo Debian 10 para la recepción y procesado de los datos energéticos. También en este servidor estará alojada la aplicación web que representará las diversas gráficas con los datos obtenidos.
- Estudiar los datos enviados por los dispositivos instalados en los edificios de la ETSIT.
- Creación de scripts para la lectura y procesado de los datos recibidos, para su posterior inserción en las bases de datos.
- Desarrollo de una aplicación backend para servir los datos almacenados en las bases de datos a la aplicación web.
- Desarrollo y despliegue de una aplicación web, para mostrar gráficamente todos los datos obtenidos.

1.4. Estructura

Para poder lograr los objetivos anteriormente marcados, se ha procedido a realizar las siguientes tareas, desarrolladas en cada uno de los capítulos:

- En el Capítulo 1 se ha introducido el concepto de monitorización y sus diferentes tipos, también se han expuesto la motivación y objetivos de este Trabajo Fin de Grado.
- El Capítulo 2 describe la situación de partida de este TFG, y cómo se reciben los datos energéticos y ambientales de la ETSIT.
- En el Capítulo 3 se analizan los siguientes aspectos:
  - Puesta a punto del servidor, y preparación para su correcto funcionamiento.
  - Mejora de los Scripts para la recolección de los datos recibidos e inserción en las bases de datos.

    - Creación de una aplicación de backend para servir los datos almacenados en las bases de datos en formato JSON.
    - Desarrollo de una aplicación web con React, para la representación mediante gráficas de los datos obtenidos por los diferentes dispositivos.

- En el capítulo 4 se someterá el trabajo realizado a pruebas de validación, para comprobar su calidad.
- En el Capítulo 5 se plantearán las conclusiones y líneas futuras de este Trabajo Fin de Grado.

Capítulo 2

Estado del arte

Este capítulo tiene como objetivo estudiar el formato y el modo en el que se reciben los datos en el servidor, y a su vez estudiar las diferentes tecnologías y herramientas más utilizadas hoy en día para: el tratamiento de grandes cantidades de datos, desarrollo de microservicios y desarrollo de aplicaciones web. En base a los conocimientos obtenidos, se comentará la elección de las herramientas utilizadas para el desarrollo de cada una de las partes. Serán elegidas aquellas que satisfagan los objetivos anteriormente comentados y los requisitos de nuestro sistema.

2.1. Recepción de datos

El sistema de monitorización es un diseño propietario desarrollado a medida para los objetivos de monitorización de sistemas energéticos. Se basa en una comunicación alámbrica mediante modbus sobre bus serie RS-485 a 2 hilos en la que hay un controlador central (TjMon) y diferentes nodos (S-DINs) (ver Figura 2.1) [8].

TJ-MON *SDIN-METER-3PH*

Figura 2.1: Dispositivos para la monitorización

Cada nodo consta de un sistema operativo en tiempo real que obtiene las variables cada 10 milisegundos. Los analizadores de red son *SDIN-METER-3PH* y se encargan de obtener diferentes variables energéticas (tensión, corriente, frecuencia, potencia activa, potencia reactiva, energía activa, energía reactiva, y el factor de potencia) de cada una de las líneas trifásicas. Cada segundo, el nodo incorpora esos registros a sus tablas internas para el acceso desde el exterior a través del protocolo RS-485 RTU-Modbus.

El controlador principal (TjMon) se basa en un sistema operativo Linux, con un microcontrolador con un sistema de alimentación ininterrumpida (SAI) y una tarjeta periférica para conectarse con los nodos S-DINs. Cada minuto, el TjMon adquiere todas las medidas de los nodos y las almacena localmente, junto con la fecha y la hora, en una tarjeta SD. Además, cada 10 minutos el TjMon envía los datos almacenados al servidor de monitorización a través de un cable de red [9], el servidor los almacena en ficheros de texto plano.

2.2. Tecnologías para el tratamiento de datos

Estas herramientas son necesarias para el procesado y modelado de los datos enviados desde los dispositivos. Como se ha mostrado en el punto anterior, los datos se almacenan en archivos de texto plano, generando así un archivo por día. Se desea utilizar esta herramienta para la creación de una serie de scripts que recogerán los datos que se encuentran dentro de los archivos enviados por los dispositivos, organizarlos e insertarlos en sus bases de datos correspondientes. A continuación se exponen las diferentes herramientas más usadas en la actualidad para automatizar estos procesos.

2.2.1. R

R es un entorno de programación formado por un conjunto de herramientas muy flexibles que pueden ampliarse fácilmente mediante paquetes, librerías o definiendo funciones propias [10]. Además es gratuito y de código abierto. Es una herramienta muy poderosa para todo tipo de procesamiento y manipulación de datos, el entorno de trabajo es muy flexible y extensible. Existe una gran cantidad de información sobre todas sus funciones y sobre los más de 6000 paquetes de funciones que existen. R tiene un paquete para casi todas las aplicaciones cuantitativas y estadísticas. Este lenguaje es usado para fines estadísticos y manipulación de grandes cantidades de datos, ya que utiliza operadores de asignación múltiple y estructuras de datos muy variadas y útiles a la hora del manejo de datos.

2.2.2. Python

Python es un lenguaje de programación interpretado. Nace con el objetivo de hacer un lenguaje de programación con una alta legibilidad y compresión de su código [11]. Es un lenguaje de programación muy popular y general, ya que cuenta con una amplia gama de módulos específicos y soporte de una comunidad muy activa. Es verdad que Python es una muy buena opción para el manejo y la manipulación de grandes cantidades de datos, pero el verdadero potencial de este lenguaje se consigue son sus paquetes como Pandas [12] o Tensorflow [13], haciendo de Python una opción sólida para la creación aplicaciones avanzadas de inteligencia artificial.

2.2.3. SQL

SQL es un lenguaje de consulta estructurada, es utilizado para definir, administrar y realizar consultas a las bases de datos estructurales. Es un lenguaje muy eficiente para la consulta, actualización y manipulación de las bases de datos y sus tablas. La sintaxis de SQL es declarativa, lo que facilita la legibilidad de este lenguaje y conocer siempre como va a funcionar la sentencia introducida. SQL es un lenguaje muy útil a la hora de consultar y recuperar datos de una forma ordenada gracias a su almacenamiento y rápida recuperación de los datos almacenados. Algunas implementaciones de este lenguaje son por ejemplo: MySQL, PostgresSQL, SQLite o MariaDB.

2.3. Tecnologías para la creación de una API

API son las siglas en inglés de *"application programming interface"*. Se trata de la interfaz de programación de aplicaciones. Se utilizan para establecer los mecanismos y protocolos para que otros sistemas y aplicaciones puedan consumir y consultar los datos, a través de acceso a *endpoints*. Las APIs se pueden considerar como contratos, con documentación que representa un acuerdo entre las partes. Si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte [14].

Esta funcionalidad es posible gracias a diferentes arquitecturas, la más común y utilizada para este tipo de aplicaciones es la arquitectura cliente-servidor (ver Figura 2.2). Pudiendo ser el cliente una aplicación *front-end*, una pagina web estática, o una simple petición a la interfaz del servicio de datos. En este tipo de estructura es necesario la existencia de una lógica de negocio en

el *back-end*. Esta realiza el procesado de los datos en el servidor, para poder entregar la información requerida por el usuario de la forma más limpia y eficiente posible. El diseño de la arquitectura cliente-servidor es imprescindible ya que no se debe de dejar expuesta en el código fuente a la aplicación *front-end*; o para establecer conexiones a fuentes de datos como, por ejemplo, bases de datos. Una estructura típica que no debemos de perder de vista es la interfaz de usuario. Esta ejecuta una petición de datos o recursos a un servidor, quien consulta las bases de datos correspondientes si fuera necesario.

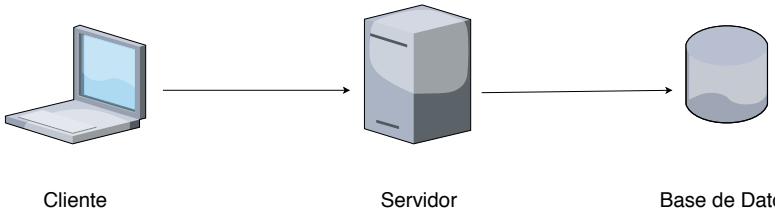

Figura 2.2: Diagrama de la arquitectura cliente-servidor

Conviene recordar que la estructura mostrada en la Figura 2.2 no es la única, en este tipo de arquitecturas, es la base de muchas otras. Existen arquitecturas basadas en este modelo, el cliente realiza una petición de ciertos recursos a un servicio principal de *back-end*. En este caso, este servicio de *back-end* no es capaz de satisfacer la petición del cliente, por lo que llama a un segundo servicio de *back-end*. Es este segundo quien se encarga de acceder a la base de datos correspondiente, y satisfacer la petición de recursos realizada por el cliente. Este es el principio de lo que hoy se denomina microservicios, consiste en un desarrollo de aplicaciones de forma vertical, a diferencia del desarrollo horizontal más utilizado en los últimos años. Los microservicios se caracterizan por no tener un único servicio funcionando, sino por la división de los mismos para que estos sean más sencillos, ocupen menos memoria y consuman menos recursos del servidor donde se alojan [15].

Existen diferentes interfaces de comunicación entre el cliente y el servidor, la más utilizada a lo largo de los años han sido las APIs. Quizás hoy en día son las API REST las que abarcan la mayor parte del mercado. Estas APIs aprovechan el protocolo HTTP para realizar las peticiones y respuestas entre el cliente y el servidor. Las aplicaciones que ofrecen esta arquitectura REST, se les denomina aplicaciones RESTful. A continuación se explican con detalle diferentes herramientas utilizadas para la creación de una interfaz de programación de aplicaciones.

2.3.1. Express.js

Express es un framework flexible de *Node.js* que permite la creación y desarrollo de una aplicación de servidor en una estructura cliente-servidor. *Express* es una opción muy adecuada para el desarrollo de un servidor cuando se quiera crear y exponer APIs para comunicarse como cliente con la aplicación de servidor. Estas aplicaciones desarrolladas con *Express* tienen una estructura muy sencilla, simplemente se debe elegir en qué puerto del servidor va a estar disponible la interfaz de la aplicación. A continuación se agregan la diferentes rutas de acceso a la API, también conocidas como endpoints, y se asigna el método que se quiera utilizar : GET, POST, PUT, DELETE. Una vez asignado las diferentes rutas de acceso a la interfaz, se devuelve lo que se haya acordado cuando se accede a cada una de las diferentes rutas.

Estas rutas pueden devolver desde un texto para saber que el servicio funciona y se encuentra devolviendo las respuestas al cliente que se lo solicite, hasta un JSON con los objetos obtenidos después de realizar una consulta a la base de datos que corresponda en cada caso.

2.3.2. Spring

Spring es el frameworks más popular del lenguaje Java. Este framework simplifica al máximo el código que se ha de escribir para la creación de una aplicación API REST [16]. *Spring* Framework es quien se encarga de gestionar la petición del cliente, analiza el tipo de datos y decide sobre qué clase de Java tiene que trabajar para poder procesar esos datos. Así, cuando un endpoint se le asocia una URL, desde el cliente se realiza la petición al endpoint y es *Spring* Framework quien acude a la llamada, la recoge, y decide qué clase puede procesar esa información. Posteriormente ejecuta esta clase, procesa la información, y lo envía de vuelta al cliente. Desde un punto de vista del desarrollo, esto supone una gran ayuda, ya que reduce los tiempos, y el desarrollador solo se debe fijar a qué método se llama en cada uno de los endpoints. Existen diversos módulos dentro de este framework que realizan numerosos servicios, algunos de los más conocidos y usados son los siguientes:

- **Acceso a datos:** Trabaja usando el Java Database Connectivity (JDBC).
- **Modelo Vista Controlador:** Provee de herramientas para la personalización de servicios web y servicios REST.
- **Testing:** Soporte para la creación de clases de pruebas.

2.4. Tecnologías para la creación de una aplicación web

En la actualidad existen diversidad de frameworks y librerías que facilitan la creación y el desarrollo de aplicaciones web. Para ello se deben de tener en cuenta los diferentes modelos de arquitectura que existen a la hora de la creación de una aplicación web. Hoy en día existen dos enfoques muy diferentes: por un lado existen las aplicaciones web tradicionales que realizan la mayor parte de su lógica en el lado del servidor. Por otro lado, existen las aplicaciones de página única (SPA, de sus siglas en inglés), que por lo contrario ejecutan la mayor parte de su lógica de parte del cliente. No se trata de dos arquitecturas incompatibles, ya que se puede alojar una SPA dentro de una arquitectura tradicional más grande. A este tipo de arquitectura se le denomina arquitectura híbrida. A continuación, en la Figura 2.3 se puede observar las diferencias en las dos arquitecturas anteriormente comentadas.

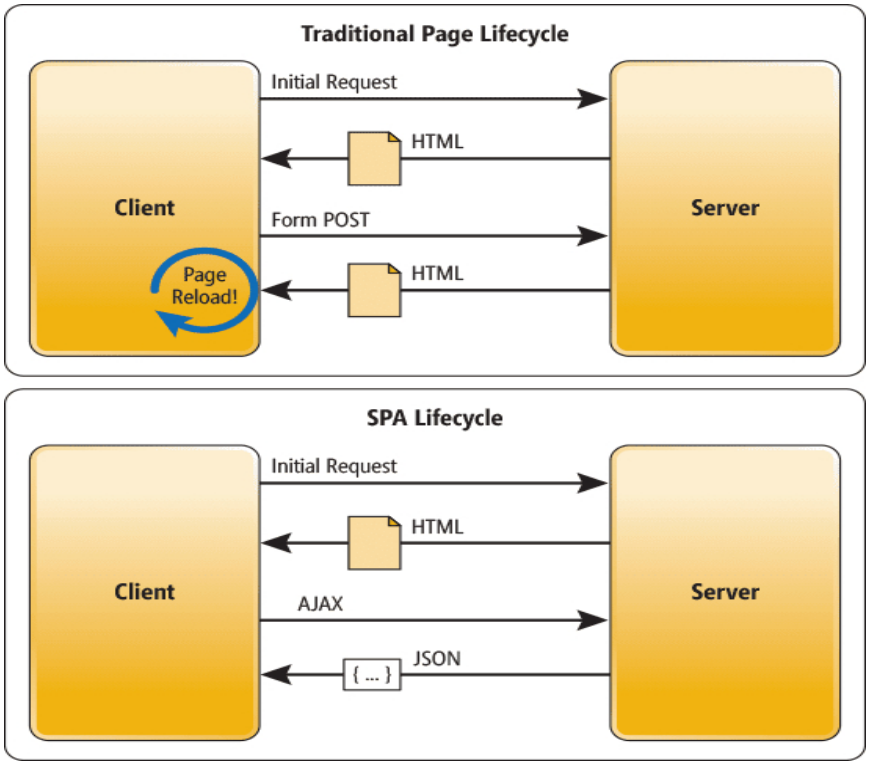

Figura 2.3: Diagramas de los diferentes diseños de las aplicaciones web

Una de la mayores ventajas que existe en el uso de las SPAs sucede por la desagregación de la actualización de las vistas de las peticiones al servidor. Después la carga inicial, la ejecución de la aplicación web depende exclusivamente del navegador. Éste, según las interacciones del usuario, hace peticiones asíncronas al servidor y vuelve a renderizar las partes o componentes

que hayan sido modificados, dejando el resto de la web como antes.

Otra de las ventajas de las SPAs es la reducción de datos transmitidos entre el cliente y el servidor. Esto implica una reducción y mejor aprovechamiento del ancho de banda disponible por el cliente. Se consigue obtener una mejor latencia al depender en menor medida del servidor, porque toda la lógica interna de la aplicación web, se encuentra en el código JavaScript, y se realiza directamente en el navegador del cliente. A continuación se describen las dos herramientas más utilizadas en el mercado actualmente:

2.4.1. Angular

Angular es un framework de código abierto para el desarrollo de aplicaciones web, desarollado en TypeScript, se trata de una evolución de JavaScript para convertirlo en un lenguaje con fuerte tipado de datos. Este framework es mantenido por Google, y en su mayoría es utilizado para la creación de aplicaciones de una única pagina (SPA). Para programar en Angular pueden utilizarse diversos lenguajes como ES5, ES6, Dart o TypeScript [17]. En Angular se programa por componentes. Esto supone una gran ventaja a la hora de hablar del rendimiento de este framework, porque al tratarse de una SPA reduce el tiempo y costes en memoria, a la hora de cargar los diferentes componentes de la aplicación. Permite la utilización de plantillas, se pueden obtener sugerencias de código y detección de errores en la mayoría de IDEs. También incorpora una interfaz en la linea de comandos (CLI) para gestionar diferentes herramientas de Angular, como por ejemplo, empezar a la creación de la aplicación rápidamente, creación de nuevos componentes y la realización de test.

2.4.2. React

React es una librería de JavaScript, fue creada por Facebook, quien la sigue manteniendo en la actualidad. Se trata de una librería open source, aunque no lo era en sus inicios. Fue desarrollada para construir interfaces de usuario de forma rápida y eficiente de cara al desarrollador y de forma óptima de cara al cliente y servidor. Durante los últimos años se ha producido un cambio en la forma de programar, tendiendo siempre a una programación reactiva y dando pie a la asincronía de los procesos dentro de la aplicación. *React* surge como librería funcional construida a partir de componentes, facilitando así la creación de SPAs.

React permite la composición de la aplicación a través de pequeños componentes independientes entre sí. Estos componentes se renderizarán de

nuevo y de forma independientes unos de otros, cuando las propiedades o el estado se haya visto modificado, debido a la interacción del usuario con la aplicación.

Dentro de una aplicación desarrollada con *react*, los diferentes componentes se organizan jerárquicamente: los componentes padres son los que envían información (*props*) a los hijos, pero no al revés. Esto se debe a que la unidireccionalidad de la comunicación, conocida como *One-Way Data Binding*, a su vez cada componente de los que forman nuestra aplicación tiene asociado un estado, el cual almacena el valor de las variables internas de cada componente. Bien cuando el estado del componente cambia, o llega nueva información a través de las *props*, el componente automáticamente se vuelve a renderizar para poder mostrar la información actualizada.

Este tipo de comunicación permite tener siempre al estado de la aplicación como única fuente de verdad (*single source of truth*), asegurando así el correcto funcionamiento de la arquitectura jerárquica propuesta por *react* [18]. El flujo de la información es descendente, del componente padre al los componentes hijos. En cambio, la propagación de eventos tiene un flujo ascendente, de los componentes hijos al componente padre. Para ello las funciones manejadoras de estos eventos son pasadas como parámetros a los componentes hijos, obteniendo así la estructura jerárquica en perfecto funcionamiento con un único sentido de la información.

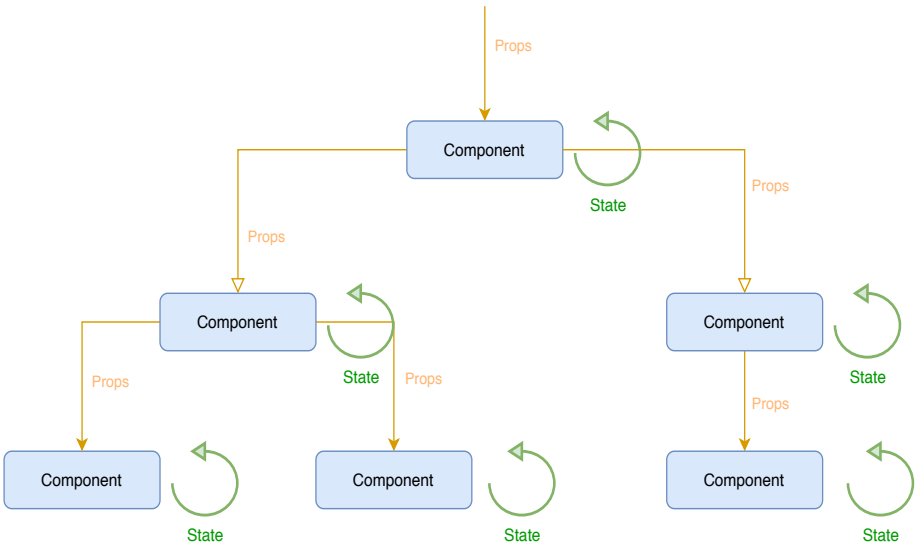

Figura 2.4: Diagramas del flujo de la información

Una de las mayores desventajas y complicaciones que existen a la hora de desarrollar aplicaciones web, es lidiar con el Modelo de Objetos del Documento (DOM, de sus siglas en inglés), este proporciona una representación estructurada la web, y permite la modificación de su contenido. Hace de puente entre la representación gráfica y la lógica de la web [19]. Y es su manipulación la que hace que esta tarea no sea cosa sencilla. *React* propone una solución para ello: un DOM Virtual (Virtual DOM). El DOM virtual es el encargado de guardar en memoria el estado actual de la página web para después sincronizarlo con el DOM real. Esto se lleva acabo a través de una librería como *ReactDOM* [20].

Esto supone que cada cambio que se realice en un componente de la aplicación web, genera una modificación del DOM en un nuevo DOM virtual. La librería virtual DOM es la encargada de comparar las diferencias y realizar los cambios oportunos en el DOM real, como se puede observar en la Figura 2.5 [21]. Una vez actualizados estos cambios en el DOM del navegador, se llama a cada uno de los componentes para que se pueden renderizar de nuevo con el actual estado o con las nuevas propiedades.

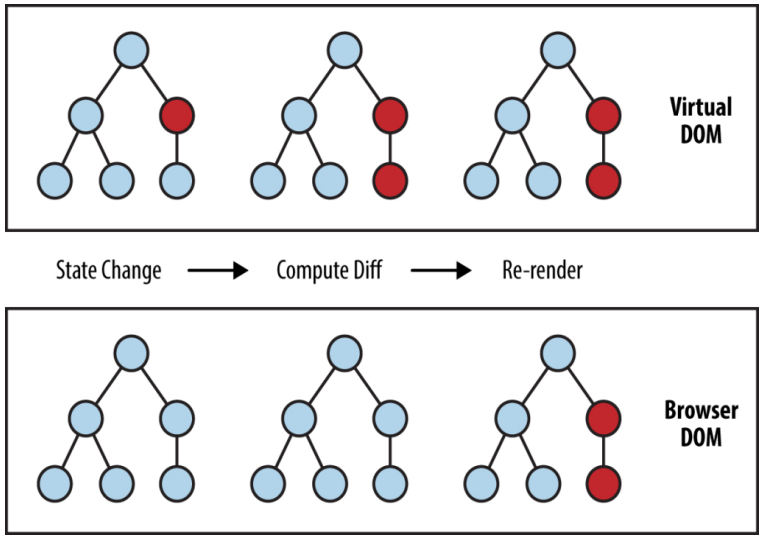

Figura 2.5: Diagramas del funcionamiento del Virtual DOM.

React recomienda el uso de una extensión de JavaScript llamada JSX, utilizada para describir la interfaz de usuario. Defienden que la lógica de renderizado debe estar intrínsecamente unida a la lógica de la interfaz de usuario. De esta forma hablamos de una librería que fusiona la interfaz de usuario con la lógica de la SPA, de tal forma que acerca el diseño (HTML) a JavaScript, y no al revés, como hacen por ejemplo el framework anteriormente comentado Angular, en el punto 2.4.1.

2.5. Herramientas seleccionadas

Las herramientas elegidas para la realización del desarrollo software del presente TFG, han sido las siguientes:

- **VS Code:** se trata de un editor de código desarrollado por Microsoft. Es un software *open source*, y multiplataforma, disponible para: Windows , Linux y macOS. Incluye soporte para depuración de código, es compatible con gran cantidad de lenguajes de programación, Git se encuentra integrado como el software de control de versiones. Incluye una enorme lista de extensiones, los cuales se pueden ir agregando para recibir una mejor experiencia a la hora de desarrollar cualquier proyecto software [22]. Dado la cantidad de lenguajes y librerías que se van a utilizar para desarrollar el proyecto, se ha optado por el uso de este editor de texto, porque incorpora soporte para múltiples herramientas y para diferentes lenguajes.

- **RStudio:** es un entorno de desarrollo para el lenguaje de programación R, esta compuesto por un editor de texto, una consola, y ofrece la oportunidad de depuración del código y permite la navegación sencilla por el espacio de trabajo y un útil gestor de los valores de las variables y datos [23]. Este entorno de desarrollador ha sido seleccionado dado que se instala en el servidor y puede ser manejado desde cualquier otro ordenador.

- **MySQL Workbench:** es una herramienta visual para la creación y gestión de las bases de datos. Incorpora una pequeña venta para la ejecución de consultas sobre la tabla seleccionada [24]. Se ha decidido la utilización de la misma, ya que es la única que ofrece una visualización completa de las bases de datos, y permite conectarse a cualquier base de datos. Permitiendo sesiones de MySQL sobre SSH.

- **PostMan:** se trata de un cliente HTTP, que permite gestionar las peticiones a las APIs, pudiendo comprobar si la interfaz de la aplicación se encuentra activa y verificar que la respuesta concuerda con la petición realizada [25]. Esta herramienta ha sido seleccionada debido a la sencillez con la que se puede realizar diversas peticiones a las APIs, y visualizar de forma intuitiva las respuestas de las mismas.

- **GitHub:** se trata de una plataforma donde alojar los proyectos que han sido gestionados con control de versiones Git. Los proyectos se pueden alojar de forma pública o privada, siempre que se disponga de una cuenta de pago, o cuenta educativa [26]. Se ha elegido esta plataforma para albergar los repositorios de forma remota, porque entre otras razones permite la creación de repositorios privados con una cuenta educativa.

Capítulo 3

Desarrollo e implementación

A lo largo de este capítulo se va a detallar como se ha desarrollado el trabajo necesario para la creación del sistema de monitorización de variables ambientales y eléctricas de la ETSIT. Para ello este capítulo tendrá cuatro secciones diferenciadas, cada una de ellas tratará sobre cada uno de los grandes bloques que se han llevado a cabo para conseguir un sistema de monitorización completo, útil y fiable.

3.1. Puesta a punto del servidor

Como ordenador principal y donde se llevarán acabo todas las operaciones relacionadas con la obtención y almacenamiento de los datos, el servicio de la API, para poder responder a las peticiones hechas al *back-end* y por último se usará como servidor web para alojar la aplicación web donde se podrá visualizar en gráficas las variables monitorizadas.

Se ha utilizado para ello uno de los servidores disponible dentro del Laboratorio de Robótica y Control (ROBOLABO) de la ETSIT. Se decidió instalar el sistema operativo Debian 10, debido a su ligereza, fiabilidad y gran administración del hardware donde es instalado. También se tuvo en cuenta la cantidad de software disponible para agregar a este servidor y poder así obtener una maquina muy potente, por la gran cantidad de paquetes estables, y *drivers* disponibles, por su uso ligero de la memoria, y un servidor de los más seguro gracias al OS instalado.

Lo primero fue obtener una imagen de instalación del OS Debian 10, disponible en la pagina oficial [27]. Durante la instalación se configurará entre otras cosas el uso horario, las particiones de disco a crear, y la selección de programas a instalar que se deseen, en este caso solo se marcaría la opción de instalar el servicio para el uso de ssh, para permitir las conexiones remotas a dicho servidor.

SSH es un protocolo para el acceso remoto a un servidor u otro ordenador, a través de un canal seguro en el que toda la información viaja cifrada. Permite la creación de sesiones FTP cifradas para la transferencia de datos, gestionar el acceso con claves RSA, a este servicio se le asigna el puerto TCP 22. Una de las configuraciones más recomendadas para el uso del SSH en el servidor consiste en, no poder acceder a la maquina con el usuario *root* a través del protocolo SHH. Se trata de una forma de proteger la integridad del servidor, evitando exponer la contraseña de *root* del servidor.

En el punto en el que nos encontramos disponemos de un servidor en el cual se ha terminado de realizar una instalación del sistema operativo, de forma limpia y eficaz con el USB donde se almacenaba la imagen del sistema, a través de la cual se ha podido arrancar el servidor, y comenzar con la instalación. A partir de ahora no es necesario estar y manipular dicho servidor localmente, ya que salvo excepciones o mantenimiento, se accederá al mismo de una manera mucho más cómoda, se hará a través de SSH desde cualquier máquina, y desde cualquier lugar.

Para poder llevar a cabo las diferentes actividades que se van a realizar en este servidor fue necesaria la instalación de diversos paquetes disponibles para este sistema operativo, gracias a su gran gestor de paquetes, la mayoría de software necesario se han podido instalar a través de él. Fue necesaria la instalación de R, para poder ejecutar los scripts para la lectura de datos e inserción en las bases de datos, para una mejor comprensión y depuración de los mismos se optó por la instalación de RStudio, entorno gráfico para el desarrollo de código en R.

También ha sido necesario instalar MySQL Server, para poder disfrutar del almacenamiento en bases de datos. Ha sido necesario configurar el usuario *root* dentro de MySQL, y se ha creado otro usuario, este último será el que se utilizará para la creación, y modificación de las bases de datos a través de los scripts. Como se comentó en el apartado 2.5, en el ordenador local o de trabajo se instalará el programa MySQL Workbench, el cuál permite el acceso a bases de datos se encuentren de forma local o en remoto. Permite consultar y visualizar de forma muy cómoda las bases de datos existentes en nuestro servidor.

Por otro lado para el correcto funcionamiento del servidor y la completa seguridad del mismo se ha instalado el paquete *UFW*, se trata de un firewall muy sencillo de utilizar por su flexibilidad y útil por todas las opciones que ofrece para configurarlo correctamente. Es una gran ayuda para este caso dado que se va a necesitar exponer ciertos puertos de la maquina, con este paquete se tiene constancia de los puertos abiertos en todo momento, y la posibilidad de

cerrarlos en caso de que ya no sea necesario. A su vez, la instalación del paquete *Apache2*, indispensable a la ahora de desplegar la aplicación web donde se van a representar todos las variables obtenidas de los diferentes dispositivos, dado que uno de los cometidos principales del servidor es alojar la aplicación web.

Para el posterior desarrollo tanto de la aplicación controladora del *back-end* y la SPA (*Single Page Aplication*) es necesario la instalación del entorno *Node.js*. Se puede obtener y descargar a través de su página oficial, o a través de la linea de comandos, para ello es necesario añadir el PPA (*Personal Package Archive*), repositorio proporcionado en la web oficial de *Node.js*, una vez añadido el repositorio, se instalará como un paquete más, usando el gestor de paquetes del propio servidor. Existe un gestor propio de *Node.js*, *npm* (*Node Package Manager*), imprescindible a la hora de desarrollar aplicaciones. Este se encarga de añadir paquetes a las aplicaciones, nos da la opción de ejecutarla para conocer si esta funcionando como es debido antes de hacer la build final y desplegarla en producción. Cabe destacar que *npm* ayuda a la hora de construir la compilación final de la aplicación, antes de desplegarla.

3.2. Scripts en R para almacenar los datos en BBDD

En esta sección se va comentar el trabajo que se ha llevado a cabo para la implementación de los scripts en el lenguaje R, para la automatización del proceso de lectura y adaptación de los datos incluidos en los archivos recibidos de todos los dispositivos que se van a utilizar para la monitorización de la ETSIT. Una vez leídos los archivos y obtenido los datos, se insertan a la base de datos correspondiente de cada edificio, mediante una *query* (sentencia).

Existen diversos dispositivos en cada uno de los edificios, la mayoría de ellos son medidores de energía, pero hay algunos otros medidores de temperatura, humedad y CO_2. Por esta razón existen diferentes tipos de scripts, para cada uno de los tipos de medidores dentro de cada edificio. No solo existen diferentes tipos de scripts por los diferentes tipos de medidores, sino también por las diferentes funciones que desempeñan.

En el conjunto de ficheros que conforman todos los archivos para la lectura e inserción de datos en las diversas bases de datos, encontramos diferentes tipos de scripts, según su funcionalidad:

- **Ficheros para la creación de tablas:** sirven para crear nuevas tablas en las bases de datos estructuradas y almacenar en ellas todos los datos existentes pertenecientes a un módulo de adquisición concreto. Estos ficheros solo deberían ejecutarse una vez al inicio de la aplicación, hacen

uso de las funciones contenidas en el fichero *functions.R* para lograr su finalidad. En ellos únicamente se incluye la información referente al modulo de adquisición al que se asocian (fichero de texto con los datos en bruto, cantidad de datos, fechas y horas de las medidas) y de la tabla que crean (nombre de la tabla, base de datos a la que pertenece, nombre de las columnas de la tabla). Además, opcionalmente, pueden incluir cualquier tipo de operación que se requiera aplicar sobre los datos de forma masiva, como pueden ser cambios de escala, factores de transformación, integrales, etc... En el sistema MONITORING ROBOLABO, todos estos ficheros cuentan con el prefijo *create* en su nombre.

- **Ficheros para la actualización de tablas:** sirven para actualizar tablas existentes en las bases de datos estructuradas y almacenar en ellas los nuevos datos pertenecientes a un modulo de adquisición concreto. Estos ficheros deben ejecutarse cada cinco minutos, también hacen uso de las funciones contenidas en el fichero *functions.R* para lograr su finalidad. En ellos se incluye la información referente al modulo de adquisición al que se asocian (fichero de texto con los datos en bruto, cantidad de datos, fechas y horas de las medidas) y de la tabla que actualizan (nombre de la tabla, base de datos a la que pertenece, nombre de las columnas de la tabla). Además, opcionalmente, pueden incluir cualquier tipo de operación que se requiera aplicar sobre los datos de forma masiva, como pueden ser cambios de escala, factores de transformación, integrales..., así como operaciones que requieran comprobar las etiquetas temporales de datos de una o varias tablas para generar un eje temporal común a todas ellas. En el sistema MONITORING ROBOLABO, todos estos ficheros cuentan con el prefijo *update* en su nombre.

- **Ficheros auxiliares:** son aquellos que ofrecen soporte a la realización de acciones concretas para el manejo de los datos en bruto de la aplicación. El fichero más relevante de esta categoría es el fichero nombrado *functions.R*, una librería completa con todas las funciones necesarias para extraer los datos de los ficheros de texto recibidos de los dispositivos, comprobar su validez y trasladarlos a contenedores más apropiados para su tratamiento masivo, las matrices de R. En resumen, este fichero dota de funcionalidad al resto de ficheros. Tratando de hacer el sistema lo más abstracto posible, extrayendo de los archivos de creación y actualización, las funciones que se repiten en cada uno de los archivos, dejando las funciones en un solo fichero (*functions.R*), de este modo el resto de archivos se encuentran más organizados y legibles.

Para lograr la interacción de los scripts en R con las bases de datos, en este caso el gestor instalado en el servidor es MySQL, es necesario el uso del

paquete *RMySQL* para la interacción del código de R con las bases de datos. Este paquete incluye métodos para el manejo de las bases de datos, la ejecución de sentencias desde los propios ficheros de R, y la lectura y escritura de las tablas existentes en las bases de datos. Una librería muy potente, que ayuda a la integración de las bases de datos almacenadas y gestionadas, en este caso por MySQL, desde los scripts de R.

Una vez ejecutados todos los scripts de R, queda generara una estructura en MySQL de la siguiente forma: existe una base de datos por cada edificio a monitorizar, dentro de la base de datos una serie de tablas donde se almacenan en orden cronológico los datos obtenidos de los diferentes dispositivos. Además de existir una tabla por cada dispositivo, existe una tabla de potencia y otra de energía. Estas dos últimas se calculan en base al resto de tablas, ya que se encargan de realizar la suma total de la potencia consumida total de un edificio, en el caso de que hubiese paneles solares, sumaría la cantidad de energía obtenida a través de los mismos. En la Figura 3.1 se muestra cómo queda la estructura de las bases de datos.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| ETSIT              |
| edificio_A         |
| edificio_B         |
| edificio_C         |
| edificio_D         |
| edificio_IES       |
| edificio_MAGICBOX  |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
```

Figura 3.1: Estructura final de las bases de datos

Como se ha comentando anteriormente, existe una base de datos por cada edificio, y cada una de ellas en su interior contiene una tabla por cada uno de

los medidores que se encuentran en ese edificio. Adicionalmente existe una base de datos con el nombre ETSIT, contiene dos tablas: una de energía y otra de potencia. Ambas tablas son las sumas totales de todos los edificios de la escuela. Estas dos tablas se generan con un script que realiza una consulta a las tablas *pro_power* y *pro_energy* de todas las bases de datos, y suma las datos con la misma marca de tiempo (*timestamp*). Obteniendo así las tablas con la suma total del consumo eléctrico de energía y potencia de todos los edificios de la ETSIT.

```
mysql> show tables;
+----------------------+
| Tables_in_edificio_A |
+----------------------+
| general              |
| meter01              |
| meter02              |
| meter03              |
| meter04              |
| meter05              |
| pro_energy           |
| pro_power            |
+----------------------+
```

Tablas del Edificio A

```
mysql> show tables;
+----------------------+
| Tables_in_edificio_C |
+----------------------+
| general              |
| line1                |
| line2                |
| line3                |
| meter01              |
| meter05              |
| meter06              |
| meter07              |
| meter08              |
| meter09              |
| meter10              |
| meter11              |
| meter12              |
| meter13              |
| meter14              |
| pro_energy           |
| pro_power            |
| threep               |
+----------------------+
```

Tablas del Edificio D

Figura 3.2: Contenido de las bases de datos de los edificios A y D

3.3. Desarrollo de una API REST

En este punto del desarrollo existen unas bases de datos, cada una asociada a un edificio de las ETSIT, cargadas con toda la información recibida por los diferentes dispositivos. Ahora surge la necesidad de poder exportar la información de una manera sencilla y eficaz. El principal usuario/cliente será la aplicación web, que se encargará de hacer una petición a este servicio, para poder obtener la información y poder representarla en la aplicación web.

Para la realización de esta aplicación de *back-end* se ha utilizado el framework Expres.js, comentado en el Capítulo 2. Se ha decido emplear framework en el desarrollo de la API por su versatilidad, sencillo mantenimiento,

y su liviana ocupación de la memoria del servidor. En primer lugar se ha de tener en cuenta el uso de la aplicación, el cliente realizará una petición a una de las rutas disponibles. En este caso serán peticiones de tipo GET, a unas rutas concretas, las cuales devolverán la información que ha sido requerida por el cliente. En la Figura 3.3 indica el ámbito en el que va a trabajar esta aplicación.

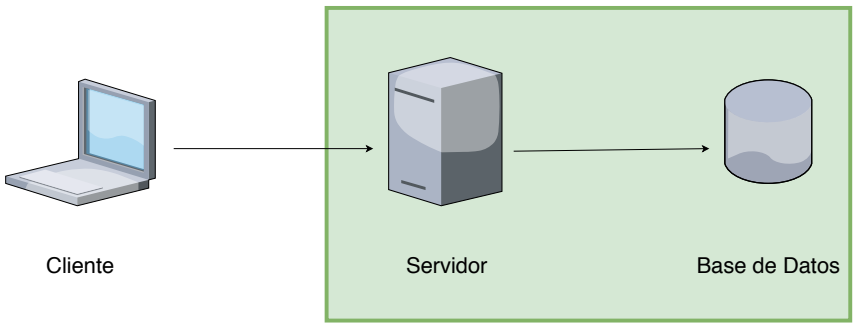

Figura 3.3: Ámbito de trabajo de la aplicación de *back-end*

A continuación se van a comentar las acciones realizadas para llevar a cabo dicha aplicación. Una vez creado un proyecto básico de *Node.js* tenemos un archivo *package.json*. En él se describen las dependencias del proyecto y opcionalmente se pueden añadir diferentes scripts para la ejecución de diversos comandos más complejos como por ejemplo la ejecución de la aplicación durante el desarrollo de la misma cuando aun no se encuentra en producción. Una vez descritas las dependencias necesarias para nuestro proyecto se pueden instalar las mismas ejecutando del comando `npm install` en la carpeta raíz del sistema, las dependencias se instalan en forma de paquetes en el directorio *node_ modules*, ahora el proyecto estaría preparado para empezar a desarrollarse.

Se describen a continuación los diferentes ficheros que componente el proyecto y su funcionalidad dentro de la aplicación:

- **server.js:** se trata del fichero principal de la aplicación en el que se especifican una serie de parámetros como el puerto en el que se va a servir nuestra API, un método para poder decodificar las rutas introducidas por el cliente en *UTF-8*, el formato en el que se va a efectuar la respuesta desde el servidor al cliente y la señalización donde se encuentran las rutas (también llamadas *endpoints*). Es en este archivo donde se encuentran las configuraciones necesarias, y el archivo de entrada a nuestra aplicación, ya que posteriormente será este archivo el que se ponga en ejecución una vez este listo para desplegar en producción.

- **routes.js:** en este archivo se describen las rutas que se encuentran disponibles en la aplicación de *back-end*, estos son los *endpoints* activos. Son a estas rutas a las que el cliente realiza las peticiones para acceder a la información que se almacena en las bases de datos. El formato de las rutas es muy sencillo, están compuestas como mínimo por dos parámetros, y como máximo tres.

  El primer parámetro de la ruta será el nombre del edificio del cual se quiere obtener información, puede ser cualquiera de los siguientes: *edificioA, edificioB, edificioC, edificioD, edificioIES, edificioMagicbox, etsit* (las rutas son sensibles a minúsculas y mayúsculas, por lo que si no se introducen correctamente la aplicación no devolverá información). El segundo parámetro se refiere al tipo de información que se necesita, potencia o energía. Los datos de potencia que se devuelven es uno cada minuto desde que se pusieron en marcha cada uno de los medidores, en cambio la información de la energía que se devuelve es un dato por día, de la suma total de la energía consumida por el edificio durante las veinticuatro horas de cada día. Para cada uno de los edificios existe el par de rutas descritas. Los formatos de las rutas descritas hasta ahora son los siguientes:

  - */{edificio}/pro_power*
  - */{edificio}/pro_energy*

  Existe una ruta adicional por cada edificio, esta contiene un tercer parámetro, para poder diferenciarla de las anteriores. Al realizar una petición a esta ruta se obtienen los datos mes a mes del consumo de energía del edificio que se requiera. La ruta tiene el siguiente aspecto:

  - */{edificio}/pro_energy/month*

  Cada una de las rutas tienen asociada un controlador diferente, este controlador es el que atiende las peticiones realizadas por el cliente. Según a la ruta a la que acceda el cliente, estará haciendo una petición u otra de información, por ello cada ruta esta asociada a un controlador diferente, como se puede observar en la Figura 3.4.

```
app.route('/edificioA/pro_power')
    .get(edificioA.list_all_pro_power);
app.route('/edificioA/pro_energy')
    .get(edificioA.list_all_pro_energy);
app.route('/edificioA/pro_energy/month')
    .get(edificioA.list_all_pro_energy_by_month);

app.route('/edificioB/pro_power')
    .get(edificioB.list_all_pro_power);
app.route('/edificioB/pro_energy')
    .get(edificioB.list_all_pro_energy);
app.route('/edificioB/pro_energy/month')
    .get(edificioB.list_all_pro_energy_by_month);

app.route('/edificioC/pro_power')
    .get(edificioC.list_all_pro_power);
app.route('/edificioC/pro_energy')
    .get(edificioC.list_all_pro_energy);
app.route('/edificioC/pro_energy/month')
    .get(edificioC.list_all_pro_energy_by_month);
```

```
app.route('/edificioD/pro_power')
    .get(edificioD.list_all_pro_power);
app.route('/edificioD/pro_energy')
    .get(edificioD.list_all_pro_energy);
app.route('/edificioD/pro_energy/month')
    .get(edificioD.list_all_pro_energy_by_month);

app.route('/edificioIES/pro_power')
    .get(edificioIES.list_all_pro_power);
app.route('/edificioIES/pro_energy')
    .get(edificioIES.list_all_pro_energy);
app.route('/edificioIES/pro_energy/month')
    .get(edificioIES.list_all_pro_energy_by_month);

app.route('/edificioMagicbox/pro_power')
    .get(edificioMagicbox.list_all_pro_power);
app.route('/edificioMagicbox/pro_energy')
    .get(edificioMagicbox.list_all_pro_energy);
app.route('/edificioMagicbox/pro_energy/month')
    .get(edificioMagicbox.list_all_pro_energy_by_month);
```

Figura 3.4: Rutas de acceso a la API

- **Edificio*XXX*Controller.js** Los controladores se encuentran organizados por edificios, es decir, existe un archivo de tipo *controller* por cada uno de los edificios. Al existir tres tipos de ruta por edificio, existen tres métodos controladores por edificio. Este archivo en concreto se encarga de llamar a uno de los métodos del modelo, los cuales se encargan de lanzar las *querys* contra las bases de datos, como se verá más adelante. Este mismo archivo es el que gestiona la respuesta que se le envía al cliente, la información si está disponible o el error correspondiente si no se puede acceder a ella.

- **dbEdificio*XXX*.js** Al igual que el archivo principal (server.js), este se trata de un archivo de configuaración para poder acceder a las bases de datos. Importa un paquete de *Node.js*, llamado *mysql*. Con este paquete importado se crea un objeto de tipo `connection`, donde se le indican parametros como el host donde se encuentra la bases de datos (en este caso: 'localhost'), usuario y contraseña para poder acceder MySQL y por último, debemos indicar la base de datos a la que se quiere conectar. Por ello se necesita un fichero de este tipo por cada edificio del que se puede obtener información.

- **Edificio*XXX*.js** Este es el archivo encargado de realizar las consultas a las bases de datos, gracias al archivo anteriormente comentado donde se crea un objeto de tipo conexión y al paquete, previamente importado, *mysql* de *Node.js*. Este tipo de fichero también es conocido como modelo, ya que accede a los datos en bruto. Para poder realizar las consultas a las bases de datos se utiliza un método que incorporan los objetos de tipo `connection`, que se llama: `query`. Llamando a ese método, se realiza

la consulta, y se devuelve el resultado al método controlador, que fue quien previamente invocó este método, el cual posteriormente se encarga de enviársela al cliente. Hay un archivo de este tipo por cada uno de los edificios.

Se ha usado un repositorio en GitHub, para guardar el proyecto de forma remota, esto permite una gran flexibilidad a la hora de desplegar el proyecto en producción en el servidor. Al realizar el proyecto bajo control de versiones es necesario crear un archivo *.gitignore*, este se encarga de ignorar los archivos o directorios para el control de versiones. En la Figura 3.5 se muestra la estructura final del proyecto.

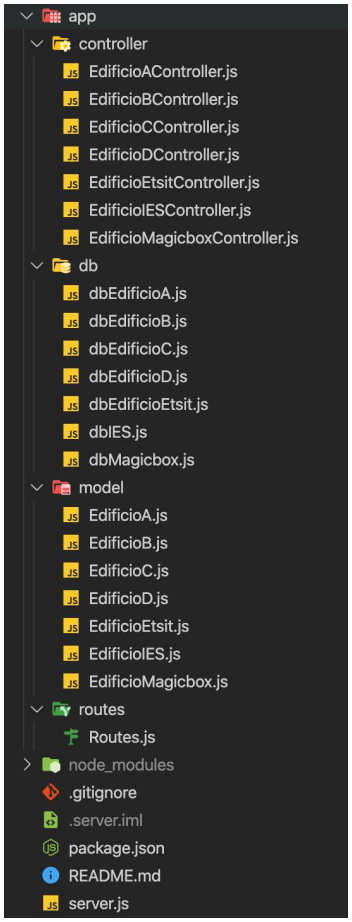

Figura 3.5: Estructuración del proyecto de la aplicación de *back-end*

3.4. Desarrollo de la aplicación web

En este punto del desarrollo se ha terminado de elaborar todas las acciones relacionadas con el *back-end*, almacenamiento e inserción de las variables en las bases de datos, y la creación de la API para servir los datos almacenados en las

bases de datos. Por ello ahora se necesita crear una aplicación web para poder visualizar todos los datos obtenidos por los diversos dispositivos instalados a lo largo de la red eléctrica de la ETSIT. Por ultimo queda desarrollar el ámbito del front, como se discutió en el segundo capítulo, existen diversas tecnologías para la realización de una web.

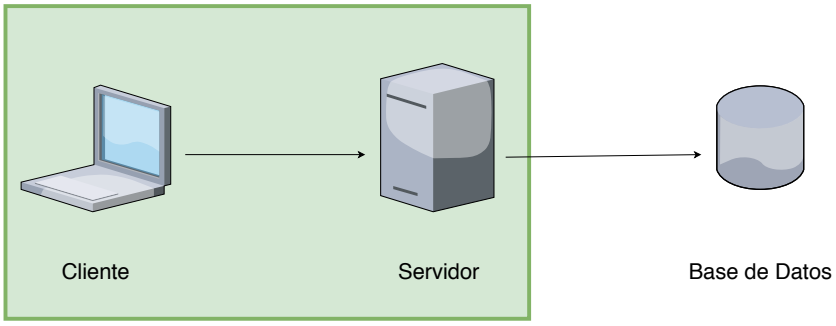

Figura 3.6: Ámbito de trabajo de la aplicación de *front-end*

Se ha decidido utilizar la librearía *React* para la realización de la aplicación web de este proyecto. Algunas de las razones por las que se ha decido emplear esta librería han sido: la facilidad para su programación por componentes, que permite una alta abstracción del código, permitiendo la reutilización de los componentes, y otra es la renderización automática de los componentes. A continuación se comentará el desarrollo de esta SPA, para la visualización de la información almacenada en las bases de datos.

Al igual que se ha comentado en el apartado anterior, el desarrollo de la aplicación se va a hacer en un ordenador personal, y cuando haya versiones operativas y funcionales, podrá clonarse el proyecto en el servidor desde GitHub, para desplegar allí la aplicación web, estará accesible a través del puerto *TCP* 80. Para poder comenzar a desarrollar la aplicación web se necesita que el ordenador donde se va a desarrollar y por supuesto la maquina donde se alojará la aplicación dispongan de *Node.js* y *npm*, el gestor de paquetes de *Node.js*.

Con el comando `npx` se pueden ejecutar paquetes binarios de *Node.js*. Y gracias a uno de los paquetes desarrollado por el equipo de Facebook Open Source, equipo encargado del mantenimiento de la librería de *react*, la creación de una aplicación básica de *react* con toda su estructura y lista para empezar a desarrollar se puede obtener con la ejecución de un solo comando:

```
npx create-react-app {nombre de la app}
```

Al ejecutar este comando se obtiene un directorio, con el nombre de la aplicación seleccionado, en su interior se encuentra un proyecto básico de *react*, a partir del cual se puede empezar a desarrollar el código necesario para la creación de la aplicación que se quiera crear. Una vez ejecutado el comando para la creación del proyecto básico, se puede poner en marcha y ver en que consiste. Para ello debemos entrar en el directorio creado por el comando anterior y arrancar la aplicación, eso se logra ejecutando los siguientes dos comandos.

```
cd {nombre de la app}
```

```
npm start
```

Una vez dentro del directorio de la aplicación, y ejecutando el último comando comentado, se arrancará la aplicación básica que se ha creado. Cuando termine de compilarse, inmediatamente después se abrirá una ventana de nuestro navegador, y se mostrará la aplicación como se puede observar en la Figura 3.7. La aplicación en desarrollo se sirve de manera local en el puerto 3000 (siempre que este se encuentre libre), encontramos la aplicación en la siguiente dirección:

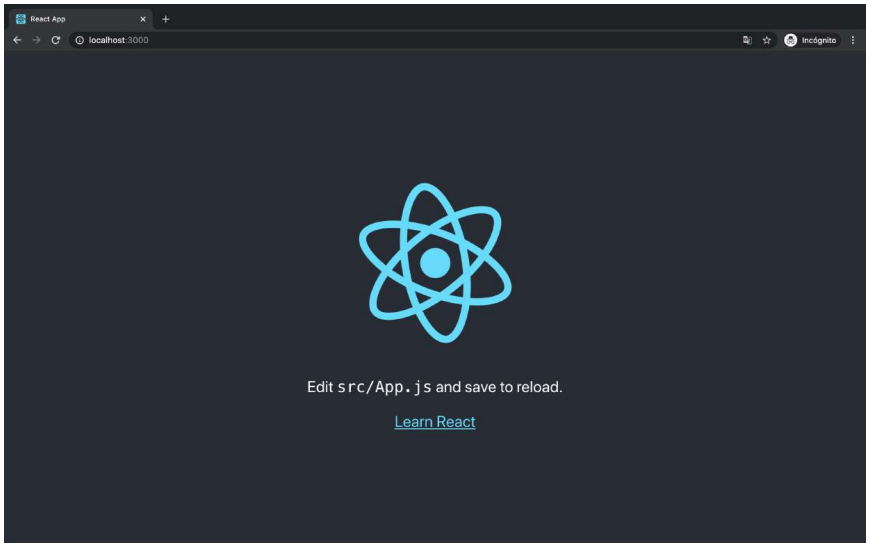

Figura 3.7: Aplicación básica de *react*

Dentro del directorio creado se pueden encontrar archivos clave a la hora de empezar el desarrollo de la aplicación con la librería de *react*. Como se trata de un proyecto de *Node.js*, y esta administrado con el gestor de paquetes *npm*, en el directorio encontramos el archivo *package.json*, donde se describen

las dependencias necesarias para el proyecto. Cuando la aplicación es creada automáticamente con el paquete `create-react-app`, el directorio proyecto viene bajo control de versiones Git, a partir de ahora se puede realizar *commits*, y guardarlo en un repositorio remoto, siempre que se pueda disponer de él. A continuación se muestran los ficheros más importantes que se pueden encontrar dentro del directorio de la aplicación:

- **\node_modules:** es el directorio donde se encuentran descargados los diferentes paquetes que la aplicación va a necesitar (dependencias).

- **index.html:** archivo html, será el provisto al navegador para que pueda mostrar la aplicación web. Como se puede observar esta completamente vacío, en el solo se encuentra un * *, con una referencia de etiqueta *id=root*. Para los efectos prácticos toda le aplicación se encuentra englobada dentro de esta etiqueta.

- **index.js:** este archivo es el nexo de unión entre la librería de *react*, es decir la aplicación en si, y el archivo html, el cual será el que se sirva a los navegadores. En su interior se encuentra la instrucción para que todo lo que se encuentra el archivo *app.js*, que veremos a continuación, sea mostrado en el contenedor * *, comentado anteriormente.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Figura 3.8: Contenido del archivo *index.js*

- **app.js:** en este fichero se encuentra todo el contenido que aparece en la pagina web de la Figura 3.7. Se importa la librería de *react*, se crea la función App, y en su interior y dentro de la función *return*, se desarrolla código html, el cual será el que se inserte en dentro de la etiqueta * *, ubicada en el archivo *index.html*. Por último lo que se debe de hacer es exportar como un módulo la función App.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

Figura 3.9: Contenido del archivo *app.js*

Una vez tenemos la base de un proyecto de *react*, se debe plantear como se va a estructurar la aplicación web, y para ello que componentes vamos a utilizar, para poder crear un código lo más abstracto posible. Se debe en la medida de los posible reutilizar los componentes creados, es una de las mayores ventajas que tiene esta librería.

La idea principal de este proyecto es poder plasmar de forma intuitiva y eficaz la información de cada uno de los edificios de la ETSIT almacenados en las bases de datos. Para ello se va a utilizar un sistema de *routing*, gracias a un paquete de *Node.js*, que se puede instalar en nuestro proyecto con el comando `npm install react-router-dom`, automáticamente se añadirá a la lista de dependencias necesarias para la ejecución del proyecto, en el fichero *package.json*, descargando los paquetes que fueran necesarios en el directorio *node_modules*.

Se va a utilizar el paquete *react-router-dom*, para crear una ruta para cada uno de los edificios, esas paginas englobaran un contenido común entre todos los edificios, el componente *Header*, estará presente en todas las páginas la aplicación. Existe otro componente común dentro de la pagina exclusiva de cada uno de los edificios, este será el contenedor donde se ilustran los datos en diferentes gráficas.

En la pagina principal se presentará el componente *Header* en la parte superior, seguidamente debajo una foto principal de portada, que a su vez será

un botón con el que se podrá navegar a la pagina donde aparezcan las gráficas correspondientes a los datos globales de la ETSIT. En la parte inferior de la pagina principal de la aplicación, se encuentra una cuadricula de dos filas y tres columnas para representar con una foto cada uno de los edificios de la ETSIT. Y al igual que la foto principal de la web, clicando en cualquiera de estas fotos se navegará a la pagina exclusiva de cada uno de los edificios, donde se visualizarán los datos de consumo eléctrico a través de gráficas.

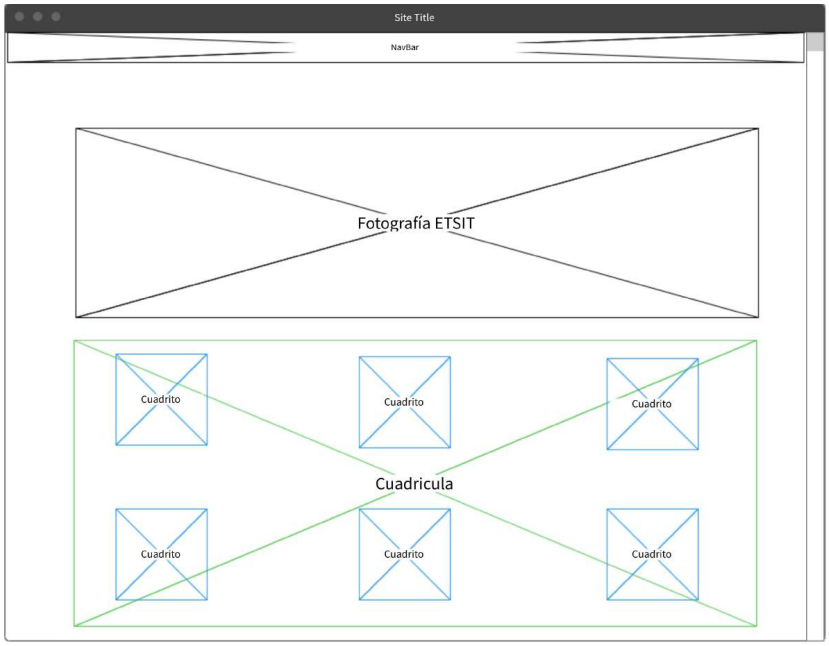

Figura 3.10: Mockup de la pagina principal de la aplicación web

En la figura anterior se muestra una maqueta de cómo quedaría la pantalla principal de la aplicación web, como se observa en la figura, existe un componente cuadricula, el cual engloba a una serie de elementos, estos reciben el nombre de *cuadrito*. El elemento es el mismo que se pinta seis veces en este caso, en cada una de las situaciones en el se representa un edificio diferente y está enlazado a una página correspondiente de su edificio. El componente *cuadrito* es un claro ejemplo de la reutilización de componentes en *react*.

Ahora que se tiene una visión global del funcionamiento y un esquema de sus componentes, se van a detallar los componentes más importantes de la aplicación web.

- **App.jsx:** este es el archivo principal de la aplicación. En este fichero es donde se realizan las llamadas a la API, para poder obtener la información almacenada en las bases de datos. Una de las ventajas que tiene usar esta librería, es que las llamadas se realizan de forma asíncrona, esto implica que la carga inicial no se verá afectada en cuento a tiempo, aun que las

llamas a las APIs no se hayan completado. Al haber incorporado el módulo *react-router-dom*, es en este fichero donde se deben especificar las rutas que se van a implementar en la aplicación y que componente debe representar cuando se acceda a una de esas rutas.

- **Home.jsx:** este fichero es el que se representa en la página principal de la aplicación, contiene la cuadricula anteriormente comentada, y desde este fichero se navega a cada uno de los ficheros en los cuales se muestran los datos en las diferentes gráficas.

- **Cuadricula.jsx:** este componente se inserta dentro de su componente padre, en este caso: *Home*. Este componente es el encargado de crear un grid e ir generando un componente *Cuadrito* por cada uno de los edificios existentes.

- **Cuadrito.jsx:** este componente recibe una array de su padre en el que aparece información, y se crea el componente gracias a los datos que le ofrece el padre, cuando este le crea.

- **Header.jsx:** este componente se encuentra siempre en la parte superior de la aplicación web, se reutiliza en las diferentes pantallas de la aplicación, consiste en una barra de navegación para poder desplazarse entre las diferentes páginas de la aplicación.

El resto de los componentes importantes que existen dentro del proyecto son los componentes asociados a la representación de la información de cada uno de los edificios. Existe un componente por edificio, asociado a una ruta diferente cada uno. Estos componentes están formados por el componente *Header*, y a continuación una barra de navegación, en la cual se puede elegir el tipo de gráfica se quiere observar. Como se comentó en el apartado anterior, la API devuelve tres tipos de información diferentes por cada edificio, la energía consumida por meses, la energía consumida al día, y la potencia utilizada, de esta última información se tiene un dato por cada minuto.

En la Figura 3.11 se distingue la botonera en la parte superior para poder intercambiar las gráficas y seleccionar la que se necesite en cada momento. E inmediatamente debajo se observa la representación gráfica de la información almacenada en la base de datos, la cual ha sido obtenida mediante llamadas a la API. En este caso se muestra la gráfica del consumo energético por meses del Edificio A de la ETSIT.

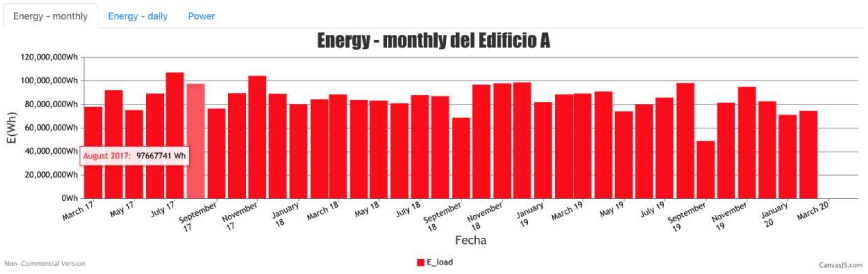

Figura 3.11: Gráfica de la energía consumida al mes en el Edificio A

Una vez realizado todo el desarrollo de las diferentes acciones a llevar a cabo para realizar el proyecto, se obtiene un sistema completo tanto en *back-end* y *front-end*. Los datos enviados por los dispositivos son recibidos en el ordenador en ficheros de texto plano, con la ayuda de los scripts en R, se recoge y almacena la información en bases de datos, ordenadas por cada uno de los edificios de la ETSIT. Los datos son expuestos para los clientes a través de la API que se ha creado, siendo su principal cliente la aplicación web, quien hace una petición al *back-end*, para almacenar los datos y representarlos mediante gráficas. En la siguiente figura se observa la página principal de la aplicación web, a partir de la cual se puede seleccionar de que edificio se quieren obtener datos y ver las distintas gráficas.

Figura 3.12: Página principal de la aplicación web

A la hora de realizar el despliegue en producción de las dos aplicaciones, tanto la aplicación en Express, que gestiona la API, y la aplicación web, se debe de tener en cuenta diferentes aspectos. La aplicación realizada con Express, se va a exponer en el puerto 8000, se debe habilitar este puerto con el paquete previamente instalado, para controlar de una forma sencilla el *firewall* del servidor. Una vez se tenga el puerto habilitado, se debe lanzar la aplicación de tal forma que siempre este activa, esperando las peticiones de los clientes. Por ello se ha decido instalar un paquete de *Node.js*, de forma global con el siguiente comando:

```
npm install -g forever
```

Una vez instalado el paquete forever, solo se debe ir al directorio donde se encuentra la aplicación de *back-end* e iniciar la aplicación con el paquete *forever* que se acaba de instalar, de la siguiente forma:

```
forever start server.js
```

De esta formar se tiene ejecutando el servicio en segundo plano, pero siempre activo, y se reiniciará si existe algún error. Realizar el despliegue con este gestor de procesos, asegura que el script seguirá ejecutándose continuamente.

A la hora de realizar el despliegue de la aplicación web el servidor, para que esta se encuentre disponible para todos los usuarios interesados, se deben de realizar una serie de acciones previas. Para ello se realizará la instalación de uno de los servidores de paginas web más famosos por su versatilidad y gran fiabilidad, se trata del software *Apache2*, para poder instalarlo ejecutaremos el siguiente comando, ya que el software está disponible en los repositorios de instalación de Debian.

```
apt install apache2
```

A continuación se debe habilitar la opción *WWW* mediante el gestor de *firewall* previamente instalado, mediante la siguiente instrucción:

```
ufw allow 'WWW'
```

Seguidamente se deberá clonar en el directorio *\var\www\{nombre del dominio}* el proyecto de la aplicación web lista para ser desplegada desde el repositorio remoto, esta se encuentra alojada en un repositorio privado de GitHub.

Una vez clonado el repositorio, accedemos al directorio principal de la aplicación y ejecutamos el comando para poder compilar la aplicación de *react* lista para ser desplegada correctamente, esta acción se lleva a cabo con el siguiente comando:

```
npm run build
```

El comando ha creado un archivo index.html, el cual será servido como nuestra pagina web principal, y *Apache2* es la aplicación encargada de servir la aplicación web alojada en el servidor. Por último se debe reiniciar el servicio *Apache2*, para que este comience a servir la nueva página web creada con el comando anterior, para ello se ejecutará la siguiente instrucción:

```
service apache2 restart
```

Capítulo 4

Validación y pruebas

Durante la elaboración de este TFG se ha comprobado que cada uno de los componentes que integran el sistema final funcionan correctamente y lo hacen de manera óptima. El proyecto está compuesto por tres elementos totalmente independientes: bases de datos, API y aplicación web. Esto ha permitido un desarrollo individualizado de cada uno de ellos, permitiendo gran libertad a la hora de la validación del trabajo realizando las pruebas en cada uno de los diferentes elementos.

4.1. Pruebas para la población de las Bases de datos

Las primeras pruebas y comprobaciones se realizan sobre la ejecución de los scripts de R, encargados de la lectura de los archivos en texto plano, para la posterior inserción de la información en las bases de datos. Debido a la gran cantidad de datos que se leen, su almacenamiento y su posterior inserción en una base de datos, el proceso es algo tedioso.

Para ser consciente de la cantidad de datos que estos scripts deben de procesar, se van a comentar los ficheros recibidos. La mayoría de los dispositivos son medidores de potencia comerciales de corriente trifásica, los datos que estos dispositivos generan por minuto, se envían y almacenan en ficheros de texto plano, son los siguientes: tensión (Vi) y corriente (Ii) de cada una de las tres fases, corriente en el neutro (In), la potencia aparente (PaTi), la potencia instantánea (Papi), el coseno de la fase (CFi) y la frecuencia (Fi) de cada una de las fases. A todos estos datos le tenemos que sumar tres columnas más: la marca de tiempo (*timestamp*), la fecha y la hora (ver Figura 4.1).

unix_ref	date	time	V1	V2	V3	I1	I2	I3	IN
1486076400	2017-02-03	00:00:02	23836	23848	23659	66	73	77	37
1486076460	2017-02-03	00:01:02	23807	23833	23624	66	73	62	38
1486076520	2017-02-03	00:02:02	23774	23820	23590	67	81	56	44
1486076580	2017-02-03	00:03:02	23752	23771	23528	74	75	56	45
1486076640	2017-02-03	00:04:02	23716	23768	23515	71	74	54	44
1486076700	2017-02-03	00:05:02	23679	23739	23477	67	82	54	47
1486076760	2017-02-03	00:06:01	23657	23725	23469	65	77	55	43
1486076820	2017-02-03	00:07:01	23706	23744	23548	61	73	64	39
1486076880	2017-02-03	00:08:01	23684	23748	23526	61	81	53	46
1486076940	2017-02-03	00:09:01	23680	23742	23511	61	74	53	42

Parte 1

PaT1	PaT2	PaT3	Pap1	Pap2	Pap3	CP1	CP2	CP3	F1	F2	F3
16020	19840	19840	31500	34820	36480	506	570	542	5002	5002	5002
16040	19420	15360	31820	35240	29360	500	560	522	5000	4999	4999
16920	21820	14600	32200	38640	26800	519	566	553	4998	4997	4998
16500	20660	14620	35200	36000	26740	455	578	554	4998	4997	4997
16020	19520	14300	33960	35280	25800	471	549	566	4997	4997	4997
15900	21600	14240	32040	39180	25780	476	550	563	4999	4999	4999
15560	19360	14040	30880	36820	26140	495	536	543	4997	4997	4997
13840	18220	15160	28920	35080	30360	464	514	496	4999	5000	4999
14400	20480	12420	29100	38680	25060	481	529	494	4998	4998	4998
14360	18200	12400	29120	35380	25020	482	514	490	4999	4997	4999

Parte 2

Figura 4.1: Información de la base de datos de un medido del Edificio A

Los primeros medidores se instalaron en 2015 y progresivamente se fueron añadiendo más hasta 2017, por este motivo sería arbitrario hablar del tiempo que tardan los scripts en ejecutarse y poblar las bases de datos, ya que los datos son proporcionales a los años de funcionamiento de medidores. Cada fichero, genera 1.440 registros por día (uno por minuto), lo que supone 525.600 registros por año. Esto da una idea de la magnitud de datos a procesar en cada uno de los medidores. Por este motivo las pruebas del procesado e inserción de datos se va a medir en intervalos de un año.

Cada uno de los registros generados por minuto consta de 22 columnas. Los scripts recogen los datos almacenado en los ficheros provenientes de los medidores, se organizan en una matriz, y, dependiendo de la línea en la que son recogidos, los datos se someten a una operación matemática para unificar las unidades. Una vez que se ha realizado todo el procedimiento, se abre una conexión con la base de datos para crear la tabla dentro de la base de datos correspondiente sino existe, y rellenarla con la matriz creada en el script. Después de haber tomado diferentes medidas, el tiempo de ejecución de los scripts para poblar la base de datos con la información de un año, es de entre un minuto y medio y dos minutos. Esta diferencia se debe a las operaciones que se realizan en los scripts para acomodar los datos a las mismas unidades.

La mayoría de los dispositivos instalados a lo largo de la ETSIT son medidores de energía, en cambio existen algunos medidores de temperatura, humedad y CO_2. Estos medidores se encuentran instalados en el edificio Magicbox. Es relevante comentar la diferencia de tiempos en poblar la base de datos de los medidores de energía con 22 columnas por minuto, con respecto a estos medidores, los cuales contienen de 5 a 8 columnas por minuto. Se produce un cambio considerable en el tiempo de ejecución del script encargado de la recolección de la información e inserción en la base de datos. El tiempo medio de ejecución e inserción de la información en el intervalo de un año se encuentra por debajo del minuto. Menos de la mitad del tiempo necesario para poblar la base de datos de un medidor de energía, debido a que los anteriores deben de poblar la base de datos con más del doble de columnas.

4.2. Pruebas de validación de la API

Como se ha comentado a lo largo del Capítulo 3, cuando la API recibe una petición de un cliente esta se encarga de procesar y ejecutar una consulta a la base de datos según la petición recibida. Estas peticiones son correctamente respondidas, y se ha podido comprobar con la aplicación de *PostMan* (ver Figura 4.2) a lo largo del desarrollo de la aplicación de *back-end*.

Figura 4.2: Respuesta de la API a una petición

En la Figura 4.2 se observa el correcto funcionamiento de la API, y su respuesta en formato JSON. Pero para realizar una validación de la calidad de esta aplicación, se ha utilizado un paquete Node.js, llamado *loadtest*. Este paquete permite medir el tiempo de respuesta de la API y se le pueden asignar diferentes parámetros para la realización de los test. Los parámetros utilizados para la llevar a cabo los test han sido: el número de peticiones a realizar a la API (*opción -n*), y la concurrencia (*opción -c*), es decir, el número de peticiones que se realizan simultáneamente. El comando para la realización de las pruebas es el siguiente.

```
loadtest -c 10 -n 50 {ruta al end-point}
```

En el comando anterior se han seleccionado las siguientes opciones: 50 peticiones y con 10 peticiones simultáneas. Como se comentó en el Capítulo 3, existen tres tipos de endpoints, y para cada uno de ellos la consulta que se realiza a la base de datos es diferente, variando el tamaño de las respuestas. Se va a coger el Edificio A como referente para realizar las pruebas. Y para uno de los *end-points* se van a realizar 50 peticiones con una concurrencia de 10 peticiones simultaneas.

- ***/edificioA/pro_ energy:*** este *end-point* devuelve un dato de energía consumida por día. Se ha utilizado el siguiente comando para comprobar la calidad de la API;

```
loadtest -c 10 -n 50 {/edificioA/pro_energy}
```

```
INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
INFO
INFO Target URL:          http://138.100.21.252:8000/edificioA/pro_energy
INFO Max requests:        50
INFO Concurrency level:   10
INFO Agent:               none
INFO
INFO Completed requests:  50
INFO Total errors:        0
INFO Total time:          0.442080446 s
INFO Requests per second: 113
INFO Mean latency:        79.1 ms
INFO
INFO Percentage of the requests served within a certain time
INFO   50%      71 ms
INFO   90%      119 ms
INFO   95%      132 ms
INFO   99%      146 ms
INFO  100%      146 ms (longest request)
```

Figura 4.3: Test del *end-point "pro_ energy"*

Observando el resultado mostrado en la Figura 4.3, se puede ver el tiempo

total que ha necesitado la API para poder responder a las 50 peticiones. El tiempo total necesitado ha sido menor de medio segundo, y con una capacidad para ejecutar alrededor de 113 respuestas por segundo.

- ***/edificioA/pro_ energy/month:*** la respuesta de esta petición contiene un dato por mes, la consulta a la base de datos realiza la suma de todos los días del mes, respondiendo con un solo dato. Se ha ejecutado el siguiente comando para la realización de la prueba a este *end-point*;

```
loadtest -c 10 -n 50 {/edificioA/pro_energy/month}
```

```
INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
INFO
INFO Target URL:          http://138.100.21.252:8000/edificioA/pro_energy/month
INFO Max requests:        50
INFO Concurrency level:   10
INFO Agent:               none
INFO
INFO Completed requests:  50
INFO Total errors:        0
INFO Total time:          0.144791395 s
INFO Requests per second: 345
INFO Mean latency:        23.5 ms
INFO
INFO Percentage of the requests served within a certain time
INFO   50%      22 ms
INFO   90%      32 ms
INFO   95%      35 ms
INFO   99%      37 ms
INFO  100%      37 ms (longest request)
```

Figura 4.4: Test del *end-point "pro_ energy/month"*

En la Figura 4.4 se observan los resultados obtenidos de las pruebas realizadas a esta petición. Se responden las 50 peticiones en unos 140 milisegundo, aproximadamente la API es capaz de resolver unas 345 peticiones por segundo. Y el tiempo de respuesta individual por petición oscila entre los 22 y 37 milisegundos. Son resultados excelentes debido al pequeño tamaño de las respuestas. Cabe destacar que esto es así gracias al lenguaje SQL, el cual optimiza al máximo las consultas complejas como esta a sus bases de datos.

- ***/edificioA/pro_ power:*** la respuesta a esta petición contiene un dato por cada minuto de cada uno de los medidores del edifico. Son respuestas con gran cantidad de datos, pero se encuentran muy optimizadas en tiempo. Para la realización de este test se han utilizado diferentes parámetros debido al gran peso de las peticiones. Se ha utilizado el siguiente comando;

```
loadtest -c 5 -n 20 /edificioA/pro_power
```

```
INFO Target URL:          http://138.100.21.252:8000/edificioA/pro_power
INFO Max requests:        20
INFO Concurrency level:   5
INFO Agent:               none
INFO
INFO Completed requests:  20
INFO Total errors:        0
INFO Total time:          271.937875579 s
INFO Requests per second: 0
INFO Mean latency:        64076.4 ms
INFO
INFO Percentage of the requests served within a certain time
INFO   50%      68119 ms
INFO   90%      80890 ms
INFO   95%      102147 ms
INFO   99%      102147 ms
INFO  100%      102147 ms (longest request)
```

Figura 4.5: Test del *end-point "pro_power"*

Como se puede comprobar en la Figura 4.5, las peticiones a este *end-point* son considerablemente más lentas que las anteriores. Esto se debe a la cantidad de datos que se deben transferir en la respuesta. En el caso concreto del edificio A, se tienen un total de cinco medidores, y a ello le debemos añadir la suma, la fecha y hora, y el *timestamp*. Se transmiten nueve datos por minuto, y existen datos desde aproximadamente inicios de 2017. Aproximadamente la respuesta está formada por 1.500.000 objetos JavaScript, dado que el formato de la respuesta es un JSON. Debido a la optimización de la API y la ágil respuesta de la base de datos a la consulta, resuelve las 20 peticiones en aproximadamente 270 segundos. Esto significa que la API tiene un tiempo de respuesta frente a archivos grandes de aproximadamente 1,5 millones de objetos JavaScript de unos 13 segundos por petición.

4.3. Pruebas de la renderización de la Aplicación web

Una de las ventajas por la que se decidió el uso de la librería de React para el desarrollo de la aplicación web, fue la gestión interna que realiza sobre la vida de los componentes que conforman la aplicación web. Es importante entender la administración que lleva acabo React sobre sus componentes. Por ello se va a comentar brevemente la gestión interna que lleva a cabo cuando accedemos a la aplicación.

Cuando se accede a la aplicación web desarrollada, representa gráficamente

los componentes que se encuentran listos para ser "pintados", lo hace desde el componente padre a los componentes hijos. Una vez se ha cargado la pagina inicial, en el caso de este TFG, una pagina de inicio simple, empieza a realizar las diferentes llamadas a los *end-points* de la API, cuando esta recibe los datos los almacena en el estado, y envía las correspondientes *props* a los componentes hijos. En la pagina principal no existe ningún hijo que necesite de los datos recibidos de la API, por lo que visualmente no se notará ningún cambio visual.

Es cuando accedemos a ver las gráficas de cada uno de los edificios, cuando los datos recibidos de la API son usados. Estos datos son previamente almacenados en el estado de React. De esta forma se mitiga el tiempo de espera a la respuesta de la API, siendo inapreciable el tiempo de espera para la visualización de los datos.

El renderizado de los datos dentro de las gráficas se realiza de forma óptima gracias a la librería elegida para representar las gráficas, *canvas.js*. Debido a la gran cantidad de datos que se deben de representar, el tiempo en renderizar los datos debe minimizarse, y esta librería es capaz de ello. Como se puede leer en su documentación [28], es capaz de representar gráficamente 100.000 datos en alrededor de 200 milisegundos (ver Figura 4.6).

Figura 4.6: Test del tiempo en representar gráficamente 100.000 puntos

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

Llevar a cabo el presente Trabajo Fin de Grado ha sido una gran labor, con alguna que otra dificultad. Existen multitud de formas de llevar acabo un proyecto software, al igual que existen diversos lenguajes de programación para llevar a termino un mismo proyecto de como se ha comentado anteriormente. Por ello, el estudio previo ha realizar antes de formalizar la elección como se va a trabajar es importante, debido a las consecuencias que puede acarrear durante la realización del proyecto.

Este TFG ha permitido llevar a la práctica un sistema completo (*full-stack*), compuesto por varios módulos, independientes entre sí, pero necesarios para el funcionamiento global del proyecto. Gracias a la individualidad de los módulos se ha conseguido un sistema liviano para la memoria y robusto en funcionalidades.

Los scripts de R, en apenas pocos minutos son capaces de leer, en algunos casos, casi 3 millones de registros, los cuales tienen alrededor de 20 columnas. Hacer las operaciones pertinentes para su correcta y posterior inserción en su correspondiente tabla de la base de datos.

En vistas a la representación gráfica de los datos previamente almacenados, se decidió la creación de una interfaz de programación de aplicación, también conocida como API. Esto supone la segregación entre el *back-end* y el *front-end*, permitiendo el uso de los datos no solo para la aplicación web donde se muestran las gráficas, sino para futuras aplicaciones que requieran el uso de las variables ambientales y eléctricas recogidas por los diferentes dispositivos.

El *front-end* ha sido diseñado para la representación gráfica de los diversos datos recibidos por los dispositivos. Dentro de la aplicación web existen diferentes tipos de gráficas para la representación de la diversa información

a monitorizar.

El proyecto final se ha configurado en uno de los servidores del laboratorio de Robótica y Control de la ETSIT. Permitiendo que este servicio se encuentre disponible para todo los usuarios interesados en consultar la información recogida por cada uno de los dispositivos instalados, que se muestran en intuitivas gráficas donde observar el consumo energético de cada uno de los edificios de la ETSIT.

5.2. Líneas futuras

Este proyecto puede contener diversas líneas futuras con las que poder continuar el desarrollo de este sistema, dotándolo de nuevas funcionalidades, incluso mejorando el rendimiento de los diferentes componentes. A continuación se realizan algunas de las propuestas con las que continuar:

- **Descarga de datos.** Incluyendo algún tipo de botón cercano a la gráfica, que permita la descarga de los datos representados en ese instante, además de la posibilidad de descargar un fichero de tipo imagen con la representación de la gráfica actual.

- **Optimización de tiempo en las llamadas a la API.** Actualmente la cantidad de datos que ofrece la API es grande, en algunos edificios llega a los 2.500.000 de datos, en una sola petición, y se tarda en obtener respuesta al rededor de unos 11 segundos. No es un tiempo excesivo teniendo en cuenta la gran cantidad de datos con los responde la API. Aun así, se sugiere un optimización del tiempo de respuesta, minimizando el tiempo de carga de las gráficas.

- **Aplicación móvil.** Se ofrece la posibilidad de la creación de una aplicación móvil bien con un lenguaje unificado para dispositivos Android e iOS, o bien la creación de aplicaciones nativas para cada uno de los lenguajes. Debido a la independencia actual que existe en el sistema, la aplicación ejecutaría las llamadas necesarias a la API, obteniendo así los datos en crudo. Después solo sería necesaria la representación gráfica en la app.

- **Estudio del consumo eléctrico, para ahorrar energía.** Dado que la API esta disponible, y es un artefacto totalmente independiente de la aplicación web, se plantea la posibilidad de la realización de un estudio con ayuda del Big Data y Machine Learning, para obtener datos valiosos y relevantes sobre el consumo eléctrico de la ETSIT, para poder ahorrar energía o conseguir una mejor eficiencia en la red eléctrica.

References

[1] Juan Almorza Daza Eloy Rafael Sanz Tapia, Lourdes Benítez Sánchez-Cid. Monitorización y gestión de dispositivos, servicios y aplicaciones, 2006. Last time accesed 13 Junio 2020.

[2] Pedro J. Saturno Hernández. Qué, cómo y cuándo monitorizar: marco conceptual y guía metodológica, 1998. Last time accesed 10 Mayo 2020.

[3] F.J.R.E.Y. MARTINEZ, E.V. GOMEZ, and J.M.R.E.Y. HERNÁNDEZ. *Eficiencia energética de los edificios. Sistema de gestión energética ISO 50001. Auditorías energéticas*. Paraninfo, 2018.

[4] Departamento de Gestión de Cuidados. Protocolo de monitorización básica en pacientes críticos, 2015. Last time accesed 10 Junio 2020.

[5] PandoraFMS. Network monitoring system , what do you need to know about it?, 2017. Last accessed 23 Mayo 2020.

[6] J. Turnbull. *Pro Nagios 2.0*. Books for professionals by professionals. Apress, 2006.

[7] Agencia Europea del Medio Ambiente. Índice europeo de calidad del aire, 2017. Last time accesed 29 Mayo 2020.

[8] Álvaro Gutiérrez Álvarez. Análisis para la gestión de la demanda.

[9] Solar Decahtlon Euerope. *Development Manual*. Solar Decahtlon Euerope, 2013.

[10] E. Paradis. *R para principiantes*. Institut des Sciences de l Évolution. Universit Montpellier, 2002.

[11] Python ORG. History and license. history of the software, 2000. Last accessed 28 Mayo 2020.

[12] W. McKinney. *Python for Data Analysis*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.

[13] Google Inc. ¿por qué usar tensorflow? Last accessed 30 Mayo 2020.

[14] Red Hat. What are apis? Last accessed 30 Mayo 2020.

[15] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019.

[16] G. Mak, D. Rubio, and J. Long. *Spring Recipes: A Problem-Solution Approach*. Expert's voice in open source. Apress, 2010.

[17] Angular. Angular docs. Last accessed 2 Junio 2020.

[18] Facebook Open Source. Levantado el estado. Last accessed 3 Junio 2020.

[19] DOM. Last accessed 13 Junio 2020.

[20] Facebook Open Source. Virtual DOM and Internals – React. Last accessed 3 Junio 2020.

[21] O'Reilly. Learning react native. Last accessed 3 Junio 2020.

[22] Microsoft. Vs code. Last accessed 3 Junio 2020.

[23] Rstudio. Rstudio. Last accessed 3 Junio 2020.

[24] Oracle. Mysql workbench. Last accessed 3 Junio 2020.

[25] PostMan. Postman. Last accessed 3 Junio 2020.

[26] GitHub Inc. Github. Last accessed 3 Junio 2020.

[27] Debian. Descargar distribución debian. Last accessed 13 Junio 2020.

[28] Documentación canvas.js. Last accessed 13 Junio 2020.

[29] El perfil del ingeniero de telecomunicación.

Apéndice A

Aspectos éticos, económicos, sociales y ambientales

Después de la realización de este Trabajo Fin de Grado, se ha llevado a cabo un análisis sincero del impacto producido por este proyecto sobre la sociedad, la economía, la ética y el medio ambiente. Se comentan a continuación más detalladamente cada uno de ellos.

A.1. Impacto social

Este proyecto tendrá un impacto directo sobre los administradores de sistemas de la ETSIT, ya que de forma gráfica y visual podrán obtener información sobre el consumo eléctrico, y así tomar decisiones en cuanto al consumo realizado por cada uno de los edificios de manera individualizada. Incluso en algunos casos, poder observar datos de una zona concreta, como por ejemplo el consumo de la biblioteca, el aire acondicionado de la biblioteca o el consumo de la cafetería.

A.2. Impacto económico

El despliegue de este proyecto permite la monitorización de variables eléctricas de consumo y producción, implica que gracias a este proyecto se podrán observar y tomar decisiones para obtener un consumo de energía más eficiente, lo que permitirá un ahorro económico considerable. También se monitorizan la energía obtenida con los paneles fotovoltaicos, permitiendo reconocer la eficiencia de los paneles, para poder incrementar ese consumo posibilitando incrementar el ahorro de gasto eléctrico.

A.3. Impacto ético

El desarrollo de este proyecto no tiene un impacto directo sobre la ética, debido al hecho de que no se establece ninguna relación con ningún usuario, ni ninguna persona, a demás no se tratan datos personales

A.4. Impacto ambiental

Quizás el impacto medio ambiental sea el más importante y elevado de este proyecto. Debido que con un análisis exhaustivo de los datos obtenidos de cada uno de los edificios, se formaran decisiones en cuanto a los cambios necesarios para obtener la mayor eficiencia energética posible, disminuyendo el consumo de energía desde centrales contaminantes. Teniendo también la posibilidad de hacer más eficiente la producción de energía eléctrica con los paneles fotovoltaicos ya instalados en la ETSIT, obteniendo una cantidad mayor de energía producida, implicando un menor consumo de energía obtenida a través de procesos contaminantes.

Apéndice B

Presupuesto económico

Este Trabajo de Fin de Grado ha sido realizado en colaboración con el Laboratorio de Robótica y Control (ROBOLABO), a lo largo de 6 meses, usando los recursos que se me han proporcionado. A continuación se desglosa el presupuesto realizado sobre los costes de material y mano de obra.

- **Costes de personal**: Los costes de personal se han calculado conforme al los datos obtenidos del Mapa socio-económico del titulado en Ingeniería de Telecomunicaciones [29]. El documento ha sido desarrollado por el Colegio Oficial de Ingenieros de Telecomunicación (COIT) y por el IDC. Se obtiene que el salario medio de un graduado en sus primeros años de vida laboral recibe una remuneración media de 23.555 €/anuales, estimando el coste por hora trabajada aproximadamente de 12,50 €/hora. A su vez para obtener la remuneración del director de trabajo, se ha estimado el salario anual medio de una persona con 15 años de experiencia en el sector. Resultado un salario aproximado de 51.220 €/anuales, con una remuneración de 27 €/hora.

  Con los salarios anteriormente comentados y la estimación de horas dedicadas al proyecto a la hora del desarrollo de todos sus componentes se han estimado las horas invertidas y el presupuesto final de costes de personal (ver Tabla B.1).

	Coste horario (€)	**Horas**	**Total (€)**
Director del trabajo	27	30	810
Estudiante de ingeniería	12.5	350	4375
TOTAL			**5185**

Tabla B.1: Costes de personal.

- **Costes de recursos materiales**: Para la realización del proyecto ha sido necesario el uso de un servidor web, instalado en el Laboratorio

de robótica y control (ROBOLABO), el servidor era de la marca Dell, con un procesador Intel Xeon E-2124, 8GB de memoria RAM y 1TB de almacenamiento.También ha sido necesario el uso de un ordenador personal, en este caso se ha utilizado un MacBook Pro con un procesador Intel Core i5 a 2.7GHz, 8GB de RAM y almacenamiento SSD de 256GB. El ordenador personal ha sido necesario para poder llevar a cabo el desarrollo de los diferentes componentes existentes dentro del proyecto. La vida de estos elementos es mayor que la duración de este TFG. Aproximadamente estos equipos tienen unos 5 años de vida por lo que en el presupuesto solo se contabilizará la parte proporcional de este presente año. (ver Tabla B.2).

	Tiempo de vida (años)	**Uds.**	**Coste (€)**	**Amortización (€/mes)**	**Uso (meses)**	**Total (€)**
Servidor Dell	5	1	967	16.12	6	96.70
MacBook Pro	5	1	1299	21.65	6	129.90
TOTAL						**226.60**

Tabla B.2: Costes de recursos materiales.

Una vez realizado el desglose de presupuesto diferenciando el coste en salarios para el personal, y el presupuesto para los equipos informáticos, tanto para el desarrollo de los diversos componentes que forman este proyecto como el servidor necesario, para el despliegue. Se muestra a continuación la suma total de los gastos desglosados anteriormente.

	Coste
Costes de personal	5185 €
Costes de material	226.60 €
Subtotal	**5411.60 €**
IVA (21 %)	1136.44 €
Total	**6548.04 €**

Tabla B.3: Costes totales.

Apéndice C

Manual del desarrollador

En este anexo se van a mostrar los pasos básicos para poder entender mejor las aplicaciones del *back-end* y el *front-end*, en caso de necesitar modificar alguna de sus partes, o continuar con el desarrollo de cualquiera de las dos partes.

Es necesaria la instalación de Node.js para el desarrollo de ambas aplicaciones. Es un software multiplataforma y se puede obtener de su página web: `https://nodejs.org/es/download/`

C.1. Manual de la Aplicación de Express.js

Es una aplicación realizada con Node.js, por lo que en ella existe el pacakge.json, y en el se encuentran la descripción de las dependencias necesarias.

Una vez dentro del directorio en el que se encuentra el desarrollo de la aplicación, lo primero que se debe de hacer es ejecutar el comando:

```
npm install
```

Con él nos aseguraremos que todas las dependencias se encuentren correctamente instaladas. Se debe saber que el archivo de entrada de la aplicación es *server.js*, en él se describen ciertos parámetros esenciales de la aplicación, como por ejemplo en el puerto que va a estar disponible.

```
server.js > ...
1   const express = require('express');
2   const app = express();
3   const bodyParser = require('body-parser');
4   const port = process.env.PORT || 8000;
5   const cors = require('cors')
6
7   app.use(cors());
8
9   app.listen(port);
10
11  console.log('API server started on: ' + port);
12
13  app.use(bodyParser.urlencoded({ extended: true }));
14  app.use(bodyParser.json());
15
16  const routes = require('./app/routes/Routes'); //importing routes
17  routes(app);
```

Figura C.1: Código del archivo *server.js*

Cuando se ha configurado el puerto donde se va a exponer la API, se debe abrir dicho puerto en el firewall, con el siguiente comando:

```
ufw allow {nº del puerto}
```

Como se ve en la Figura C.1 el archivo principal utiliza el fichero almacenado en el ruta */app/routes/*, llamado *Routes.js*. En este fichero se encuentran la diferentes rutas de acceso a la API, cada ruta esta conectada con un método controlador (ver Figura C.2).

```
app.route('/edificioA/pro_power')
    .get(edificioA.list_all_pro_power);
app.route('/edificioA/pro_energy')
    .get(edificioA.list_all_pro_energy);
app.route('/edificioA/pro_energy/month')
    .get(edificioA.list_all_pro_energy_by_month);

app.route('/edificioB/pro_power')
    .get(edificioB.list_all_pro_power);
app.route('/edificioB/pro_energy')
    .get(edificioB.list_all_pro_energy);
app.route('/edificioB/pro_energy/month')
    .get(edificioB.list_all_pro_energy_by_month);

app.route('/edificioC/pro_power')
    .get(edificioC.list_all_pro_power);
app.route('/edificioC/pro_energy')
    .get(edificioC.list_all_pro_energy);
app.route('/edificioC/pro_energy/month')
    .get(edificioC.list_all_pro_energy_by_month);
```

```
app.route('/edificioD/pro_power')
    .get(edificioD.list_all_pro_power);
app.route('/edificioD/pro_energy')
    .get(edificioD.list_all_pro_energy);
app.route('/edificioD/pro_energy/month')
    .get(edificioD.list_all_pro_energy_by_month);

app.route('/edificioIES/pro_power')
    .get(edificioIES.list_all_pro_power);
app.route('/edificioIES/pro_energy')
    .get(edificioIES.list_all_pro_energy);
app.route('/edificioIES/pro_energy/month')
    .get(edificioIES.list_all_pro_energy_by_month);

app.route('/edificioMagicbox/pro_power')
    .get(edificioMagicbox.list_all_pro_power);
app.route('/edificioMagicbox/pro_energy')
    .get(edificioMagicbox.list_all_pro_energy);
app.route('/edificioMagicbox/pro_energy/month')
    .get(edificioMagicbox.list_all_pro_energy_by_month);
```

Figura C.2: Código del archivo *Routes.js*

Existe un controlador por cada edificio del que se tienen datos almacenados. En ellos se redireccionan las peticiones recibidas en las rutas, a los modelos. Todos los controladores se organizan en un directorio (ver Figura C.3).

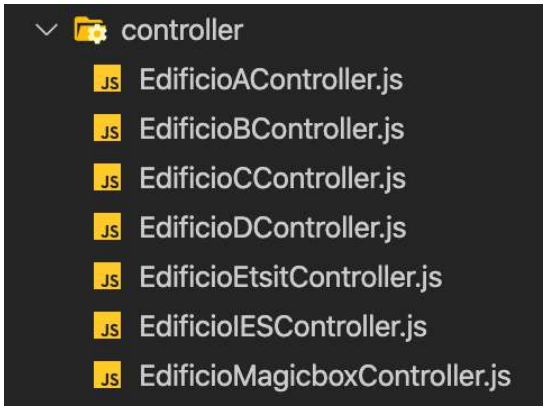

Figura C.3: Directorio Controllers

Cuando cada uno de los controladores reciben la petición realizada a la ruta, mandan la petición al modelo y este se conecta a la base de datos, y lanza una consulta a la base de datos. El resultado de la petición se convierte a formato *JSON*, y se devuelve al cliente.

Existe un directorio que contiene las conexiones a cada una de las bases de datos. Un archivo por cada edificio, ya que las bases de datos se encuentran organizadas por edificios (ver Figura C.4).

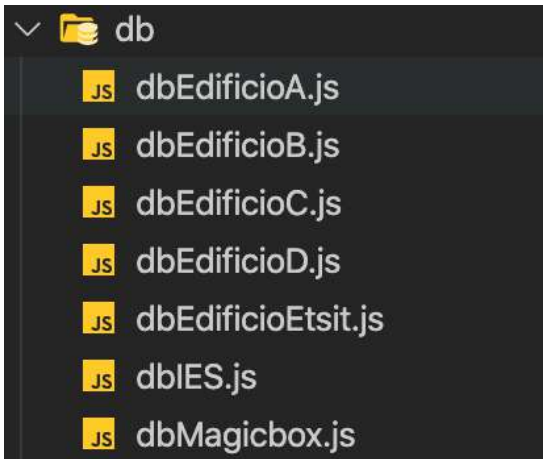

Figura C.4: Directorio de las conexiones a las bases de datos

En cada uno de los ficheros de la Figura C.4, se encuentra la configuración necesaria para el acceso a la base de datos correspondiente, en la Figura C.5 se observa la configuración realizada para el acceso a la base de datos del edificio A.

```
'user strict';

var mysql = require('mysql');

//local mysql db connection
var connection = mysql.createConnection({
    host     : 'localhost',
    user     : 'usuario',
    password : 'contraseña',
    database : 'edificio_A'
});

connection.connect(function(err) {
    if (err) throw err;
});

module.exports = connection;
```

Figura C.5: Contenido del fichero *dbEdificioA.js*

Se utiliza el paquete *mysql*, para poder realizar la conexión a las bases de datos, debido a que la información se encuentra almacenada en el gestor de bases de datos MySQL. Debido a que esta aplicación de *Express.js* se ejecuta en el servidor donde se encuentran las bases de datos, el parámetro *host* toma el valor *'localhost'*.

El fichero de la Figura C.5 es utilizado por el modelo para poder realizar la conexión a la base de datos, el modelo se encarga de realizar la consulta a la base de datos, y devolver el resultado al controlador, y este entregará la respuesta al cliente, realizando el camino inverso. En el modelo existen 3 métodos:

- Obtener los datos de la tabla *pro_power*.
- Obtener los datos de la tabla *pro_energy*.
- Obtener la suma de cada mes de la tabla *pro_energy*.

```
'user strict';
var sql = require('../db/dbEdificioA.js');

let EdificioA = function (edificio) {

};
EdificioA.getAll_pro_power = function (result) {
    sql.query("Select unix_ref, L_EdA, L_2, L_3, L_4, L_5 FROM general", function (err, res) {

        if(err) {
            console.log("error: ", err);
            result(null, err);
        }
        else{
            console.log('Devolviendo tabla pro_power');

            result(null, res);
        }
    });
};

EdificioA.getAll_pro_energy = function (result) {
    sql.query("Select * FROM pro_energy", function (err, res) {

        if(err) {
            console.log("error: ", err);
            result(null, err);
        }
        else{
            console.log('Devolviendo tabla pro_energy');

            result(null, res);
        }
    });
};

EdificioA.getAll_pro_energy_by_month = function (result) {
    sql.query("SELECT MONTH(date) as month, YEAR(date) as year, sum(E_load) as E_load from pro_energy GROUP BY MONTH(date), YEAR(date);", function (err,res){
        if(err) {
            console.log("error: ", err);
            result(null, err);
        }
        else{
            console.log('Devolviendo tabla pro_energy');

            result(null, res);
        }
    });
};


module.exports = EdificioA;
```

Figura C.6: Contenido del fichero *EdificioA.js*

En la Figura C.6 se comprueba los tres métodos anteriormente comentados. Los dos métodos existentes para el acceso a la tabla *pro_ energy* son iguales para todos los edificios. En cambio, el método de acceso a la tabla *pro_ power* varía en función de los medidores existentes en el edificio. Para realizar correctamente esta consulta se debe primero comprobar los diferentes medidores que existen en cada uno de los edificios. Realizando así la consulta correcta sobre cada uno de los edificios.

Para poder arrancar la aplicación es necesario ejecutar uno de los siguientes comandos en el el directorio principal de la aplicación.

```
node server.js
```

Como se trata de una aplicación Node.js, se puede iniciar con el comando

anterior. En cambio como se observa en la Figura C.7, en el package.json se ha incluido una dependencia a un paquete llamado: *nodemon*, este paquete permite que no se tenga que reiniciar la aplicación cuando se realice algún cambio el código. Para que sea más fácil la ejecución de este comando se ha creado un script (ver Figura C.7).

Figura C.7: Contenido del fichero *package.json*

Con la incorporación del paquete *nodemon* al proyecto, la aplicación se puede iniciar con el comando:

```
npm start
```

Que sería lo mismo que ejecutar: `nodemon server.js`. Esto facilita el trabajo, ahora no es necesario reiniciar la aplicación cuando se realiza algún cambio sobre cualquiera de los ficheros, gracias e este paquete se reinicia automáticamente.

C.2. Manual de la Aplicación de React

Para la creación de una aplicación de React se debe de instalar el paquete *create-react-app* de manera global para poder crear la plantilla básica de React, y partir desde ahí comenzar a programar. Para ello esta vez se va a utilizar un gestor de paquetes de Node.js diferente al anteriormente mencionado (*npm*), usaremos el gestor *yarn*, tiene una optimización mejor de los paquetes para el uso de la librería de React.

Añadimos el paquete *create-react-app* y creamos nuestra aplicación.

```
yarn global add create-react-app
```

```
yarn create-react-app {nombre-de-la-app}
```

Si se desea añadir alguna dependencia adicional al proyecto se ejecutará el siguiente comando:

```
yarn add {nombre-de-la-dependencia}
```

Al igual que el proyecto de *Express.js* para asegurar que todas las dependencias se encuentren correctamente instaladas. Se ejecutará el comando:

```
yarn install
```

De este modo todas las dependencias quedarán instaladas en el directorio *node_modules*.

En el directorio principal de la aplicación se encuentran diversos directorios y ficheros, los más importantes son:

- **package.json:** este archivo contiene la definición de las dependencias necesarias para el correcto funcionamiento de la aplicación. También incluye las instrucciones necesarias que arrancan los scripts encargados de iniciar la aplicación.
- **node_modules:** en este directorio se encuentran descargadas todas las dependencias incluidas en el fichero *package.json*.
- **src:** en este directorio se alojan todos los componentes utilizados en la aplicación de *front-end*. El fichero principal de la aplicación es *App.jsx*, en él se encuentran descritas las diferentes vistas a reenderizar según la ruta seleccionada.

- **public:** este es el directorio donde se encuentran los archivos visuales necesarios para mostrar en cada una de las vistas, como por ejemplo las fotos.

- **edificios.ts:** este es un archivo en formato TypeScript. Incluye un JSON con toda la información relevante de los edificios de los que se tienen datos. Incluye información detallada de cada edificio, como por ejemplo las diferentes variables a representar en las distintas gráficas de cada uno de ellos.

- **assets:** en este directorio se encuentran los fichero necesarios para la utilización de la librería *CanvasJS*.

Todo lo comentado anteriormente se muestra en la Figura C.8, en la que se muestra el contenido del directorio de la aplicación.

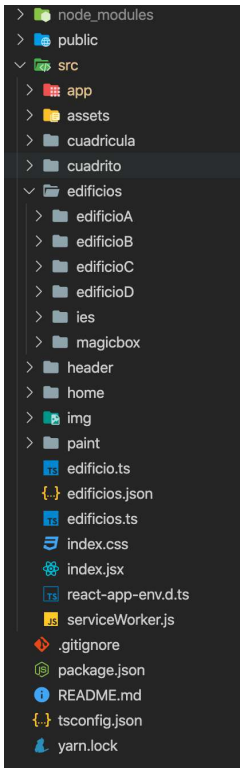

Figura C.8: Contenido del directorio de la aplicación de React

El archivo principal de toda pagina web es *index.html* y para esta aplicación no es diferente. La diferencia es que este archivo carece de ningún contenido, solo contiene un * * vacío en el que se insertará la aplicación web.

El componente principal, el cual se incrusta en el * * del archivo *index.html*, de esta aplicación se encuentra en el fichero *App.jsx*. En él se

encuentran las rutas disponibles en la aplicación, y los componentes que debe renderizar con cada una de las rutas (ver Figura C.9). Esto es posible gracias al paquete previamente instalado *react-router-dom*.

```
yarn add react-router-dom
```

```
return (
  <div className="App">
    <Router>
      <Switch>
        <Route path='/' exact component={Home} />
        <Route path='/magicbox'   render={() => <Magicbox data={this.state.magicbox}/>} />
        <Route path='/ies'        render={() => <Ies data={this.state.ies}/>} />
        <Route path='/edificioA'  render={() => <EdificioA data={this.state.edificioA}/>} />
        <Route path='/edificioB'  render={() => <EdificioB data={this.state.edificioB}/>} />
        <Route path='/edificioC'  render={() => <EdificioC data={this.state.edificioC}/>} />
        <Route path='/edificioD'  render={() => <EdificioD data={this.state.edificioD}/>} />
      </Switch>
    </Router>
  </div>
);
```

Figura C.9: Contenido del fichero *App.jsx*

El componente que renderiza la vista principal se encuentra en el fichero *Home.jsx*, y como se puede ver en la Figura C.10, contiene el componente *<Header/>*, la imagen de la portada, un título, un subtítulo y por último se añade el componente *<Cuadricula/>*.

```
function Home(){
    return(
        <div>
            <Header/>
            <img className='portada' src={portada} alt="Foto principal de la ETSIT"/>
            <h1 className='tituloETSIT'>Monitorización de la ETSIT</h1>
            <h6>Pulse sobre el edificio sobre el que quiera obtener datos.</h6>
            <Cuadricula/>
        </div>

    );
}
```

Figura C.10: Contenido del fichero *Home.jsx*

Ahora se mostrará el contenido del componente *Cuadricula*, en la Figura C.11, para poder comprender mejor su funcionamiento.

```
function Cuadricula() {
    const edificios = EDIFICIOS.filter((e, i) =>
        e.name !== "ETSIT"
    ).map((e, i)=>
        <Cuadrito key={i} edificio={e}/>
    );
    return (
        <div className='cuadro'>
            <div className='grid'>
                {edificios}
            </div>
        </div>
    );
}
```

Figura C.11: Contenido del fichero *Cuadricula.jsx*

Se realiza un *map* al archivo edificios, para obtener todos los edificios que no sean la ETSIT, ya que esta tendrá su punto de entra en otro lugar, no en la cuadricula. Y por cada uno de los edificios se crea un componente *<Cuadrito/>*, al que se le pasa como *props* la información del edificio que va a representar. Todos ellos se guardan en la variable edificios, esto será lo que se muestre. Pero para ello necesitamos saber qué es un componente *<Cuadrito/>* (ver Figura C.12).

```
function Cuadrito(props) {
    return (
        <div className='cuadrito'>
            <Link to={{
                pathname:`/${props.edificio.id}`
            }}>
                <img className='cuadrito' src={require(`../img/${props.edificio.id}.jpg`)} alt="Imagen de Edificio" />
            </Link>
            <h5>{props.edificio.name}</h5>
        </div>

    );
}
```

Figura C.12: Contenido del fichero *Cuadrito.jsx*

Cada componente *<Cuadrito/>* es un enlace a la ruta del edificio que representa, muestra la fotografía del edificio que representa, y un título con el nombre del edificio. El resultado final una vez renderizado del componente *<Cuadricula/>*, compuesto por cada uno de los componentes *<Cuadrito/>*, se muestra en la aplicación como en el de la Figura C.13

Figura C.13: Componente <*Cuadricula*/> una vez renderizado

Existe una ruta por cada uno de los edificios. Cuando se accede a cada una de ellas, se renderiza el componente asociado a esa ruta. Los componentes son muy parecidos y los forman los mismos componentes, varía los datos recibidos por la API. Cada componente solo recibe los de su edificio.

Se va a describir como está compuesto y desarrollado el componente del Edificio A, ya que el resto de los componentes pertenecientes a cada uno de los edificios funcionan de la misma manera.

Existen tres variables principales: *dataPoints_X*, en ellas se almacena un objeto, con las propiedades para representar la gráfica como se desee. Una variable para cada tipo de gráfica: *energy_daily, energy_month, power*. Uno de las propiedades más importantes a insertar en este objeto es la información a representar, que como se puede ver en la Figura C.14 se optiene de hacer un *map* al array almacenado en `props.data`. En `props.data` se encuentran almacenados tres arrays diferentes: `power, energy, energy_month`.

```
const dataPoints = {
    zoomEnabled: true,
    zoomType: "xy",
    animationEnabled: true,
    dataPointWidth: 30,

    title: {
        text: edificio.subtab[0].name
    },
    axisY: {
        title: "E(Wh)",
        suffix: "Wh",
        includeZero: false
    },
    axisX: {
        title: "Fecha",
        valueFormatString: "MMMM YY",
        includeZero: false
    },
    legend: {
        horizontalAlign: "center",
        verticalAlign: "bottom"
    },
    data: [{
        type: "column",
        color: "red",
        name: `${edificio.subtab[0].blocks.var.subvar.name}`,
        xValueFormatString: "MMMM YYYY",
        yValueFormatString: "# Wh",
        showInLegend: true,
        dataPoints: this.props.data.energy_month.map(d => {
            return {x: new Date(d.year, d.month), y: parseFloat(d[edificio.subtab[0].blocks.var.subvar.varName])}
        })}]
};
```

Figura C.14: Contenido de la variable datapoints

En la figura C.15 se muestra el contenido que se renderiza del componente *<EdificioA/>*. Este fichero está compuesto por el componente *<Header>*, también incluye una NavBar para poder cambiar de gráfica entre las tres disponibles: energy, energy_monthly, power. Por último, según la pestaña seleccionada, se pinta el componente *<CanvasJSChart/>* y se le pasa como parámetro el objeto correspondiente a la gráfica seleccionada, ya que existe un objeto *datapoint* con las características de cada una de las gráficas a representar.

```
return (
    <div>
        <Header/>
        <Nav variant="tabs" defaultActiveKey="link" onSelect={(selectedKey) => this.setState(()=> ({tab: selectedKey}))}>
            <Nav.Item>
                <Nav.Link eventKey="link">{edificio.subtab[0].name}</Nav.Link>
            </Nav.Item>
            <Nav.Item>
                <Nav.Link eventKey="link-1">{edificio.subtab[1].name}</Nav.Link>
            </Nav.Item>
            <Nav.Item>
                <Nav.Link eventKey="link-2">{edificio.subtab[2].name}</Nav.Link>
            </Nav.Item>
        </Nav>
        <CanvasJSChart options = {this.state.tab === "link"? dataPoints : this.state.tab === "link-1"? dataPoints1: dataPoints2}
                    onRef={ref => this.chart = ref}
        />
    </div>
);
```

Figura C.15: Contenido del método return del fichero *EdificioA.jsx*

C.2.1. Iniciar aplicación durante su desarrollo

Para poder iniciar la aplicación a lo largo del desarrollo de forma local, para comprobar que los componentes funcionan como es debido, se puede hacer con el siguiente comando:

```
yarn start
```

Una vez iniciada se podrá acceder a ella entrando en cualquier navegador en la siguiente ruta: `http://localhost:3000/`.

C.2.2. Desplegar la aplicación en el servidor

Para poder realizar el despliegue de la aplicación web el servidor, se deben de realizar una serie de acciones previas. Para ello se realizará la instalación de *Apache2*, para poder instalarlo ejecutaremos el siguiente comando:

```
apt install apache2
```

A continuación se debe habilitar la opción *WWW* mediante el gestor de *firewall* previamente instalado, mediante la siguiente instrucción:

```
ufw allow 'WWW'
```

Seguidamente se deberá clonar en el directorio $\backslash var \backslash www \backslash \{nombre\ del\ dominio\}$. Una vez clonado el repositorio, accedemos al directorio principal de la aplicación y ejecutamos el comando para poder compilar la aplicación de React lista para ser desplegada de React correctamente, esta acción se lleva a cabo con el siguiente comando:

```
yarn run build
```

El comando ha creado un archivo index.html, el cual será servido como nuestra pagina web principal, y *Apache2* es la aplicación encargada de servir la aplicación web alojada en el servidor.

C.2.3. Modificar la aplicación una vez desplegada

Para realizar algún cambio a la aplicación una vez esta se encuentra desplegada en el servidor *Apache2*, debemos acceder al directorio donde la

aplicación se encuentre almacenada, modificar los ficheros que se considere necesarios, a continuación se debe ejecutar el comando:

```
yarn run build
```

Para poder obtener una imagen lista para producción de la aplicación web. Una vez se ha realizado esta operación, la página estaría lista para servirse. Ahora se debe reiniciar el servicio *Apache2*, para que este comience a servir la aplicación modificada. Para ello necesitamos acceder al perfil *root* con el siguiente comando:

```
su -
```

Ahora ya podremos reiniciar el servicio *Apache2* y tener lista de nuevo la aplicación.

```
service apache2 restart
```