Grado en Ingeniería Informática

Trabajo de Final de Grado

Aplicación para Nodo IoT

Autor:
ALDANE

Supervisado por:
J. L. M. P.
Tutorizado por:
G. F. L

Resumen

Este documento detalla el proyecto de fin de grado realizado por el estudiante de la Universidad Jaume I, durante la estancia en prácticas en la empresa *IoTsens*.

El proyecto consiste en desarrollar una aplicación *web* que muestre de forma gráfica los datos que llegan a un nodo *IoT* (Internet Of Things), es un dispositivo informático capaz de recopilar información procedente de una red de sensores, estos datos los almacena y envía a otros equipos para su posterior análisis.

Para la realización de este proyecto se ha realizado un análisis, diseño e implementación siguiendo una metodología ágil, donde cada tarea se identifica con un *sprint*.

Palabras clave

Internet de las cosas, Macrodatos, Desarrollo ágil.

Keywords

Internet of things, Big Data, Agile development.

Índice general

1. Introducción **9**

1.1. Contexto y motivación del proyecto 9

1.2. Objetivos del proyecto 10

1.2.1. Alcance funcional 10

1.2.2. Alcance organizativo 10

1.2.3. Alcance informático 11

1.3. Descripción del proyecto 11

1.3.1. Herramientas 11

1.3.2. Hardware 12

1.4. Estructura de la memoria 14

2. Planificación del proyecto **15**

2.1. Metodología 15

2.2. Planificación 16

2.3. Estimación de recursos y costes del proyecto 17

2.4. Seguimiento del proyecto 20

3. Análisis y diseño del sistema **23**

3.1. Análisis del sistema 23

3.1.1. Actores . . . 23
3.1.2. Definición de requisitos . . . 24
3.1.3. Casos de uso . . . 25
3.2. Diseño de la arquitectura del sistema . . . 29
3.2.1. Diseño del nodo . . . 29
3.2.2. Diseño de la base de datos . . . 31
3.2.3. Diseño *API REST* . . . 34
3.3. Diseño de la interfaz . . . 35

4. Implementación y pruebas **39**
4.1. Detalles de implementación . . . 39
4.1.1. *API* . . . 39
4.1.2. *DAO* . . . 40
4.1.3. Código *web* . . . 41
4.2. Verificación y validación . . . 41

5. Conclusiones **43**

Bibliografía **45**

A. Detalles de implementación **47**
A.1. *API* . . . 47
A.2. *DAO* . . . 50
A.3. Código *web* . . . 52
A.4. Verificación y validación . . . 55

B. Documentación del código **57**

Índice de figuras

1.1. Gateway. 12

1.2. Sensor de sonido. 13

1.3. Sensor de temperatura. 13

1.4. Sensor de calidad del aire. 14

3.1. Diagrama de casos de uso. 25

3.2. Esquema del funcionamiento del nodo. 30

3.3. Diseño de la base de datos. 31

3.4. Mapa de memoria. 32

3.5. Canales. 32

3.6. Propiedades del canal. 33

3.7. Diseño lógico de la aplicación. 34

3.8. Página de *login* 35

3.9. Página de visualización de datos. 36

3.10. Página de visualización de datos. 37

3.11. Página de administración de dispositivos. 37

A.1. Cabeceras de la *API*. 48

A.2. Lectura de operación desde la *API*. 48

A.3. Método *"send_response"*. 49

A.4. Operaciones de la *API* 49

A.5. Constructor de la clase *DAO*. 50

A.6. Método "getCodecs". 51

A.7. Método "get". 51

A.8. Método "deleteChannelType". 52

A.9. Método "delete". 52

A.10.Parámetros *Ajax*. 53

A.11.Petición *Ajax*. 53

A.12.Formulario. 54

A.13.Vista del formulario. 54

A.14.Validación de formulario. 55

Índice de tablas

2.1. Pila del producto. 16

2.2. Coste recursos humanos. 18

2.3. Coste recursos software. 18

2.4. Coste recursos hardware. 19

2.5. Coste total del proyecto. 20

3.1. Actores del sistema. 23

3.2. Definición de requisitos. 24

3.3. CU01. Dar de alta usuario. 26

3.4. CU02. Consultar y modificar usuarios. 26

3.5. CU03. Añadir un nuevo dispositivo. 27

3.6. CU04. Modificar dispositivos. 27

3.7. CU05. Exportar información de dispositivos. 28

3.8. CU06. Consultar datos de los sensores. 28

3.9. CU07. Exportar los datos recogidos. 29

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

Giditek es el área de transformación digital de Grupo Gimeno, se encarga de potenciar la tecnología, apoyar su digitalización y generar y desarrollar nuevos negocios basados en la revolución digital. Dentro de esta área podemos encontrar la participación en *UVAX Concepts* y la unidad de negocio de *IoTsens* (lugar en el que se realizará el proyecto).

IoTsens es una empresa proveedora de soluciones verticales del Internet de las Cosas (*IoT*). Recolectan, integran, almacenan y analizan la información de soluciones diferenciando tres ramas principales:

- **Industria inteligente**: es una solución para el control y gestión remota de actividades de plantas industriales que permite la integración, almacenamiento y análisis de datos procedentes de los sensores situados en el área de actividad, permitiendo modificar el estado y el funcionamiento de motores.
- **Agua inteligente**: es un método integral de lectura de contadores a distancia que recoge los datos de manera remota y automática. Lo que consigue es una gestión rápida y eficaz de la red de abastecimiento detectando posibles averías.
- **Ciudad inteligente**: es una solución para la gestión remota y el control de dispositivos inteligentes (sensores) instalados en distintas ciudades o núcleos urbanos. Ofrece distintas herramientas para recopilar información de los sensores instalados, almacenar los datos, generar cálculos y analizarlos.

La motivación de este proyecto nace de la necesidad de tener una plataforma *web* para administrar los dispositivos conectados a un nodo *IoT*. Los beneficios que supone esta plataforma es que se puede replicar en cada nodo y así cada cliente lo puede configurar en torno a sus necesidades y consultar la información que está recogiendo.

1.2. Objetivos del proyecto

El principal objetivo de este proyecto es desarrollar una aplicación *web* totalmente integrada con el nodo *IoT* para poder configurarlo y obtener los datos de los sensores conectados a él y ser capaz de mostrarlos de forma interactiva.

El objetivo principal se puede desglosar en los siguientes subobjetivos:

- Adquirir conocimiento en entornos *Big data* y tiempo real.
- Desarrollar aplicaciones *backend* con bases de datos de alto rendimiento.
- Entender el estado del arte de aplicaciones *IoT*.

1.2.1. Alcance funcional

El alcance funcional hace referencia a las tareas que se pueden realizar gracias a la plataforma que vamos a desarrollar:

- Permitir que el administrador dé de alta usuarios.
- Permitir que el administrador añada nuevos dispositivos.
- Permitir que el administrador añada distintas configuraciones para los dispositivos.
- Permitir que cualquier usuario registrado pueda consultar los datos recogidos por los sensores.
- Permitir exportar los datos recogidos por los sensores y exportar la información de los dispositivos.

1.2.2. Alcance organizativo

A nivel organizativo, este proyecto involucra a varios sectores de la empresa. El departamento más implicado es el de *hardware*, que se va a encargar de configurar el nodo para que recoja los datos a través de los sensores y de desarrollar una aplicación *web* integrada para configurar el nodo y visualizar los datos.

Indirectamente, también afecta al departamento de *software*, ya que los datos que recoja el nodo también tienen que ser enviados a la aplicación *smartcity*, la cual es desarrollada por este departamento para tratar y analizar los datos de todos los nodos.

1.2.3. Alcance informático

El alcance informático hace referencia al diseño e implementación de la plataforma *web* que estará instalada en el propio nodo y que está conectada a una base de datos donde almacena todos los datos recogidos y las distintas configuraciones creadas por el administrador.

1.3. Descripción del proyecto

Este proyecto consiste en desarrollar una aplicación *web* que muestre de forma gráfica los datos que llegan a un nodo *IoT* (Internet Of Things), es un dispositivo informático capaz de recopilar información procedente de una red de sensores, estos datos los almacena y envía a otros equipos para su posterior análisis. También es necesario reenviar estos datos a una plataforma de *smartcity* para su posterior tratamiento y análisis.

El proyecto que se presenta en este documento se va a desarrollar con distintas tecnologías. Por un lado, tenemos la parte de *frontend*, la cual será desarrollada principalmente con *JavaScript*, donde utilizaremos librerías como *eCharts* para mostrar los datos recogidos por el nodo con gráficos y *jQuery* para facilitar la tarea de tratamiento de datos y ayudarnos con la parte gráfica de la plataforma. Por lo tanto, en esta parte utilizaremos también *HTML* y *CSS* para desarrollar las vistas.

Por otro lado, tenemos el *backend* que se desarrollará en *PHP*, que se encargará de todas las peticiones que lleguen desde el *frotend*. Para esta parte hemos creado una interfaz de programación de aplicaciones (*API*) que nos permite acceder a la base de datos desarrollada con *PostgreSQL* y así obtener los datos que nos solicitan o añadir, eliminar o modificar la información correspondiente.

Un sistema como este permitiría que el tiempo dedicado a configurar y recoger información del nodo disminuyera, ya que automatiza el proceso de recogida y tratamiento de datos de los sensores y permite configurar el nodo de forma remota mediante una aplicación *web* intuitiva.

La situación de la que partimos para el desarrollo de la plataforma es únicamente de una plantilla *web* que tiene los estilos y logos correspondientes a la empresa.

1.3.1. Herramientas

Las herramientas que hemos utilizado para el desarrollo son:

- **GitLab:** es una herramienta para el control de versiones basado en Git que permite gestionar y realizar diversas tareas como la gestión del código fuente, el mantenimiento de la seguridad y el seguimiento
- **Visual Studio Code:** es un editor de código fuente independiente que se ejecuta en Windows, macOS y Linux. Para desarrolladores *web*, con extensiones para admitir casi

cualquier lenguaje de programación.

- **Oracle VirtualBox:** es un software de virtualización que nos permite simular el entorno del nodo.
- **Redmine:** es una herramienta para la gestión de proyectos, que con sus diversas funcionalidades permite a los usuarios de diferentes proyectos realizar el seguimiento y organización de los mismos.

1.3.2. Hardware

Todas las herramientas anteriores hacen referencia al software utilizado para el desarrollo del proyecto. A continuación, vamos a ver en cuanto a hardware, qué dispositivos estamos utilizando para poder recoger todos estos datos.

- ***Gateway:*** equipo diseñado para actuar como pasarela entre los dispositivos y la plataforma, permitiendo dar cobertura de red al edificio, figura 1.1.

Figura 1.1: Gateway.

- **Sensor de sonido:** desarrollado para monitorizar los niveles de ruido en zonas acústicamente saturadas (ZAS) y, así, tomar acciones para minimizar las posibles consecuencias derivadas de prolongadas exposiciones [1]. Este dispositivo se corresponde con la figura 1.2.

Figura 1.2: Sensor de sonido.

- **Sensor de temperatura:** controla los parámetros ambientales clave para mantener unas condiciones óptimas en interiores [2]. Este dispositivo se corresponde con la figura 1.3.

Figura 1.3: Sensor de temperatura.

- **Sensor de calidad del aire**: evalúa las condiciones del aire a través de la monitorización de los principales parámetros ambientales [3]. Este dispositivo se corresponde con la figura 1.4.

Figura 1.4: Sensor de calidad del aire.

1.4. Estructura de la memoria

La estructura de la memoria se compone de cinco capítulos:

- Capítulo 1, *Introducción*: describe el proyecto y define los objetivos que se pretenden alcanzar.
- Capítulo 2, *Planificación*: explica la metodología utilizada para el desarrollo del proyecto y describe la planificación. También muestra cómo se realiza el seguimiento del proyecto y la estimación de recursos y costes.
- Capítulo 3, *Análisis y diseño*: especifica los requisitos y realiza el modelado del sistema. También describe los componentes y su relación.
- Capítulo 4, *Implementación y pruebas*: explica el trabajo de programación, patrones, algoritmos y las pruebas de verificación y validación realizadas.
- Capítulo 5, *Conclusión*: conclusiones finales sobre el proyecto, abarcando el ámbito formativo, el ámbito profesional y el ámbito personal.

Capítulo 2

Planificación del proyecto

En este capítulo se explica la metodología utilizada para el desarrollo del proyecto, la planificación inicial con los pasos a desarrollar, la estimación de recursos y el seguimiento que se ha llevado a cabo.

2.1. Metodología

La metodología que se ha utilizado para desarrollar el proyecto ha sido la metodología ágil (tipo Scrum) adaptada a los objetivos de la empresa.

Al comienzo del proyecto, siguiendo la metodología, hemos dejado claro las tareas que hay que realizar para alcanzar los objetivos del proyecto. En la primera reunión, el director del proyecto marca las tareas y deja claro los requisitos del proyecto.

A partir de ahí, en cada reunión se analiza el progreso de cada integrante y se comentan las dificultades que han podido surgir o se aportan ideas que puedan mejorar la aplicación.

Cada tarea del proyecto se identifica con un *sprint* en la metodología *Scrum* y hay que asignarle una prioridad, una estimación del esfuerzo necesario, una descripción y se puede añadir tanto información adicional como observaciones, personas asignadas, etcétera.

La forma de trabajar en *IoTsens* es utilizando *Microsoft Teams* tanto para la organización de las reuniones como para conversar entre compañeros. En nuestro caso, al comienzo del proyecto las reuniones las llevábamos a cabo de forma presencial, una vez que entramos más a fondo desarrollando el código de la aplicación las realizábamos de forma remota. Esto lo veremos más a fondo en el apartado de seguimiento del proyecto.

2.2. Planificación

La planificación del proyecto se realizó teniendo en cuenta la metodología ágil, como he explicado en el punto anterior. Para la planificación de este proyecto hay que tener en cuenta que el alumno dispone de 300 horas para su realización.

En nuestro caso, la pila del producto queda como se muestra en la tabla 2.1.

Identificador	Descripción	Estimación	Predecesoras
1	**Inicio**	**23 horas**	
1.1	Definición del proyecto	8 horas	
1.2	Identificar objetivos y alcance	8 horas	1.1
1.3	Establecer calendario de reuniones	7 horas	1.2
2	**Planificación**	**42 horas**	
2.1	Diseñar la base de datos	10 horas	1.2
2.2	Definir interfaz gráfica	4 horas	1.2
2.3	Estudio de las tecnologías	15 horas	2.1, 2.2
2.4	Identificación del patrón DAO	17 horas	2.3
3	**Implementación**	**175 horas**	
3.1	Puesta en marcha del servidor	15 horas	2.4
3.2	Implementar base de datos	20 horas	3.1
3.3	Implementar el objeto de acceso a los datos	30 horas	3.2
3.4	Implementar la API	30 horas	3.3
3.5	Implementar vistas	80 horas	3.4
4	**Pruebas**	**30 horas**	
4.1	Definición de pruebas	10 horas	3.5
4.2	Validación de pruebas	20 horas	4.1
5	**Correcciones**	**30 horas**	
5.1	Correcciones en los fallos detectados	30 horas	4.2

Tabla 2.1: Pila del producto.

Con esta planificación cumplimos con las horas que disponemos y podemos distinguir los diferentes *sprints* con los objetivos, qué hay que ir realizando y que tareas tenemos que tener hechas antes de poder continuar.

A continuación voy a explicar cada fase de las anteriores y cómo la enfocamos en cada caso:

- **Inicio**: es el comienzo del proyecto y cuándo me uno a la empresa. Ellos ya tenían unas bases para el proyecto y me explican más a fondo en lo que vamos a trabajar. Entonces dejamos definido el proyecto e identificamos los objetivos, también se comenta cómo vamos a realizar las reuniones y definimos el calendario.

- **Planificación**: En este apartado ya empezamos a dar forma al proyecto, dejando definido un primer diseño de la base de datos (más adelante explicaré los cambios que hemos realizado), el diseño de la interfaz *web* (partíamos de una plantilla básica con los logos de

la empresa), aclaramos qué tecnologías vamos a utilizar para cada parte y diseñamos el método de acceso a los datos.

- **Implementación**: Para poder realizar la implementación tuvimos que instalar un servidor para ir comprobando el correcto funcionamiento de todo lo que vamos desarrollando. A partir de aquí ya entramos en materia de programación implementando la base de datos en *PostgreSQL*, implementando el *DAO* y la *API* en *PHP* y, por último, desarrollando las vistas utilizando *HTML, CSS y JavaScript*.

- **Pruebas**: Una vez que tenemos la aplicación desarrollada y en marcha, elaboramos distintas pruebas para ver que el funcionamiento es el adecuado. Estas pruebas se realizan para buscar diferentes tipos de fallos: fallos graves que pueden comprometer la seguridad del sistema o fallos más leves en las vistas.

- **Correcciones**: Al terminar las pruebas tenemos que corregir los distintos fallos que aparecen, que no tienen por qué ser fallos graves del sistema, pero sí hay algunos detalles que corregir.

2.3. Estimación de recursos y costes del proyecto

Para realizar este apartado vamos a diferenciar entre tres tipos de recursos: software, hardware y humanos. Y al final haremos un balance con la suma de los tres.

Hay que tener en cuenta que para valorar el coste del proyecto hay que hacerlo como si se tratase de un proyecto profesional.

- **Recursos humanos**: para evaluar los costes de recursos humanos tenemos que analizar qué personal está involucrado en el desarrollo del proyecto. En nuestro caso somos un equipo de tres personas en las que se divide el jefe de proyecto, un desarrollador *hardware* y un desarrollador *software*. El estudiante es el encargado del desarrollo *software* y a pesar de que tiene adjudicada una ayuda de 1.200 € por las 300 horas de trabajo, vamos a suponer que es un programador que está cobrando la media del sector en Castellón. Disponemos del coste total en la tabla 2.2.

Recurso	Horas	Coste/Hora	Coste Total
Programador junior	300	8 €/h	2.400 €
Desarrollador hardware	300	11,25 €/h	3.37 5€
Jefe de proyecto	50	15 €/h	750 €
Total			6.525 €

Tabla 2.2: Coste recursos humanos.

- **Recursos software**: en este apartado tenemos que incluir cualquier programa que hayamos utilizado tanto para desarrollar código como para poder realizar las reuniones semanales y el seguimiento del proyecto. Por lo que el desglose lo tenemos en la tabla 2.3.

Recurso	Meses	Coste/Mes	Coste Total
Visual Studio Code	3	Licencia gratuita	0 €
GitLab	3	19 €/mes/usuario	171 €
Microsoft empresas	3	10,5 €/mes/usuario	94,5 €
Redmine	3	Licencia gratuita	0 €
Total			265,5 €

Tabla 2.3: Coste recursos software.

- **Recursos hardware**: hay que incluir tanto los equipos utilizados para desarrollar el proyecto como el *hardware* del producto final (el nodo y los sensores). Por lo que la estimación está disponible en la tabla 2.4.

Recurso	**Cantidad**	**Coste**	**Coste Total**
Ordenador de sobremesa	3	400 €	1.200 €
Monitores	7	50 €	350 €
Teclados	4	15 €	60 €
Ratón	4	10 €	40 €
Nodo	1	300 €	300 €
Sensor de sonido	1	40 €	40 €
Sensor de aire	1	40 €	40 €
Sensor de temperatura	1	40 €	40 €
Total			2.070 €

Tabla 2.4: Coste recursos hardware.

Como podemos observar en las tablas 2.2, 2.3, 2.4, tenemos un extracto con los costes de cada recurso que abarca el proyecto, obteniendo el costo total del proyecto, tabla 2.5.

Recurso	**Coste**
Recursos humanos	6.525 €
Recursos software	265,5 €
Recursos hardware	2.070 €
Total	8.860,5 €

Tabla 2.5: Coste total del proyecto.

Viendo el resumen final podemos concluir que la parte más costosa del proyecto está relaciona con el personal involucrado en el proyecto, ya que al estar trabajando tres personas se aumentan los costes.

El segundo recurso más costos es el de *hardware*, ya que hay que proveer a las tres personas de equipos informáticos para poder trabajar y, además, el producto final involucra a los distintos sensores que hemos instalado y al nodo para poder almacenar la información.

Finalmente, tenemos el *software* como aspecto más económico, ya que para desarrollar el código disponemos de programas lo suficientemente potentes para realizar el trabajo y que además son de código abierto. Los programas que son de pago están relacionados con la comunicación y gestión de equipo y con el control de versiones del código.

2.4. Seguimiento del proyecto

Para realizar el seguimiento del proyecto, el jefe de equipo decidió que la mejor forma de hacerlo era mediante dos reuniones semanales con una duración inicial de treinta minutos cada una, que podía extenderse en caso de que no hubiésemos terminado con el tema del día.

Al comienzo de la estancia realizábamos estas reuniones de forma presencial para definir los objetivos del proyecto y las primeras ideas y estructuras que queríamos llevar a cabo. De esta forma era más fácil comunicarse entre nosotros y coger notas o diseñar esquemas.

Según avanzaba el proyecto y empezábamos a desarrollar código, cambiamos las reuniones de forma presencial a forma remota porque así era más sencillo poder compartir pantalla e ir mostrando fragmentos de código o incluso las vistas que había avanzado cada uno. De esta

manera podíamos estar haciendo uso de la *web* y mostrando el código que nos interesase en ese momento.

Capítulo 3

Análisis y diseño del sistema

En este capítulo se explica el análisis y diseño del sistema, apoyándonos en el diagrama de casos de uso, en el diseño de la base de datos y en el diseño elegido para la interfaz *web*.

3.1. Análisis del sistema

3.1.1. Actores

Antes de definir los requisitos y de generar el diagrama de casos de uso tenemos que concretar los actores que van a interactuar con el sistema.

En nuestro caso, al ser una aplicación *web* para la configuración de dispositivos y para la consulta de datos recogidos por el nodo, tuvimos la necesidad de crear dos actores que interactúan con la aplicación, se muestran en la tabla 3.1.

Identificador	Actor	Descripción
A01	Usuario administrador	El usuario administrador podrá acceder a la visualización de datos recogidos y gracias a su rol de administrador podrá añadir y configurar dispositivos.
A02	Usuario	El usuario sin permisos de administrador únicamente podrá visualizar los datos recogidos por el nodo.

Tabla 3.1: Actores del sistema.

3.1.2. Definición de requisitos

La definición de requisitos sirve para describir los servicios que ha de ofrecer el sistema y las restricciones asociadas a su funcionamiento.

Como ya tenemos definidos los actores, ahora es más sencillo identificar que requisitos tiene que cumplir el sistema para cada actor, como se muestra en la tabla 3.2.

Administrador	- Dar de alta nuevos usuarios. - Consultar y modificar usuarios existentes. - Añadir nuevos dispositivos. - Cambiar la configuración de los dispositivos. - Exportar información de los dispositivos. - Consultar todos los datos recogidos. - Exportar los datos.
Usuario	- Consultar todos los datos recogidos. - Exportar los datos.

Tabla 3.2: Definición de requisitos.

Una vez que tenemos definidos los requisitos, podemos pasar a representar los actores y su relación con la funcionalidad del sistema mediante el diagrama de casos de uso.

3.1.3. Casos de uso

El diagrama de casos de uso es una forma de diagrama de comportamiento en lenguaje de modelado unificado (UML), con la que se representan sistemas y procesos de programación orientada a objetos. Por lo tanto, UML no es un lenguaje de programación, sino un lenguaje de modelado, es decir, un método estandarizado para representar sistemas planificados o ya existentes. En este diagrama, todos los objetos involucrados se estructuran y se relacionan entre sí [4].

Se ha realizado el diagrama, figura 3.1, utilizando la herramienta MagicDraw [5], este software permite desarrollar código en diversos lenguajes y además facilita el modelado de datos mediante diferentes diagramas.

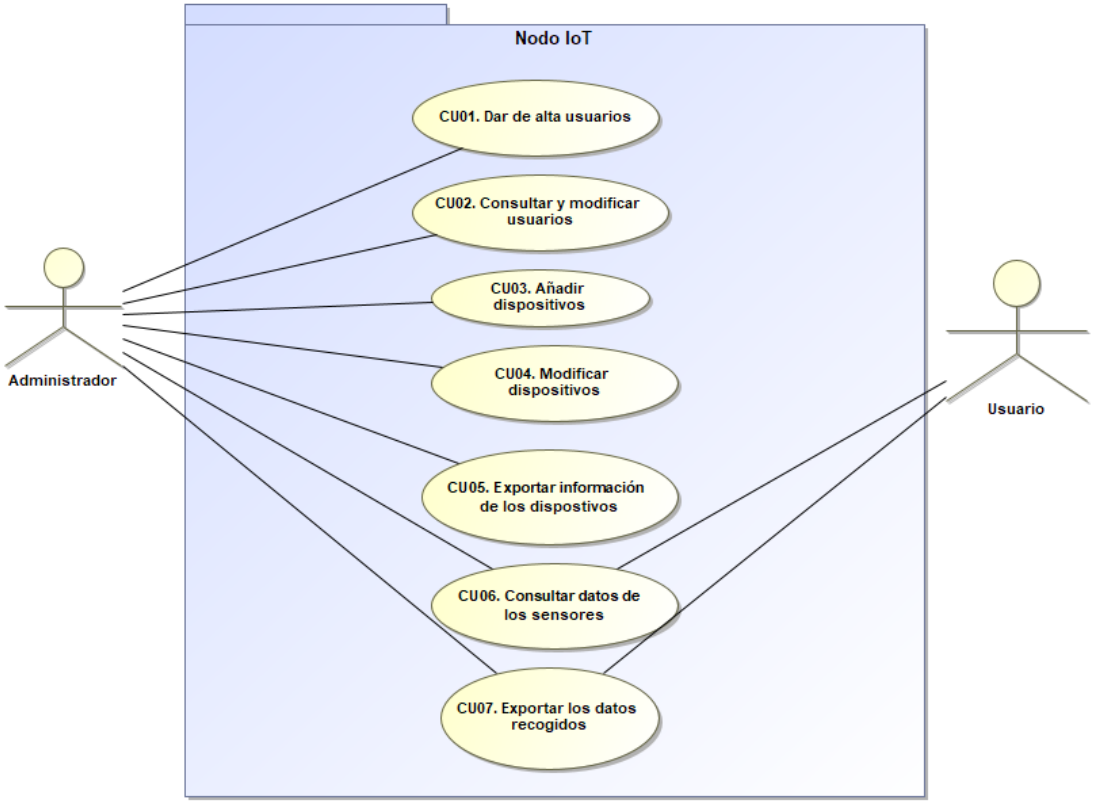

Figura 3.1: Diagrama de casos de uso.

CU01: Dar de alta un nuevo usuario
Autor: ALDANE Revisor: H. M. P.
Descripción: Añadir un nuevo usuario al sistema.
Secuencia de pasos normal: 1. Iniciar sesión con la cuenta de administrador. 2. Acceder al apartado de administración. 3. Ir a *Users Manager* y darle a crear nuevo usuario.
Excepción: No puede haber campos vacíos en el formulario de dar de alta.
Precondición: La cuenta que realice los pasos anteriores tiene que ser administrador.

Tabla 3.3: CU01. Dar de alta usuario.

CU02: Consultar y modificar usuarios
Autor: ALDANE Revisor: H. M. P.
Descripción: Consultar usuarios existentes.
Secuencia de pasos normal: 1. Iniciar sesión con la cuenta de administrador. 2. Acceder al apartado de administración. 3. Ir a *Users Manager* y se mostrará un listado con los usuarios existentes y un botón para poder modificar el usuario deseado.
Precondición: La cuenta que realice los pasos anteriores tiene que ser administrador.

Tabla 3.4: CU02. Consultar y modificar usuarios.

CU03: Añadir dispositivos
Autor: ALDANE Revisor: H. M. P.
Descripción: Añadir un nuevo dispositivo.
Secuencia de pasos normal: 1. Iniciar sesión con la cuenta de administrador. 2. Acceder al apartado de administración. 3. Ir a *Añadir dispositivo* y rellenar la información correspondiente.
Excepción: El nombre del dispositivo tiene que ser único y no puede haber campos vacíos.
Precondición: La cuenta que realice los pasos anteriores tiene que ser administrador.

Tabla 3.5: CU03. Añadir un nuevo dispositivo.

CU04: Modificar dispositivos
Autor: ALDANE Revisor: H. M. P.
Descripción: Modificar dispositivos.
Secuencia de pasos normal: 1. Iniciar sesión con la cuenta de administrador. 2. Acceder al apartado de administración. 3. Ir a *Dispositivos* y se mostrará un listado con los dispositivos existentes. 4. Seleccionar el dispositivo a modificar y cambiar los campos deseados.
Precondición: La cuenta que realice los pasos anteriores tiene que ser administrador y tiene que haber dispositivos creados previamente.

Tabla 3.6: CU04. Modificar dispositivos.

CU05: Exportar información de dispositivos
Autor: ALDANE Revisor: H. M. P.
Descripción: Ofrece la posibilidad de descargar la información de los dispositivos en formato CSV.
Secuencia de pasos normal: 1. Iniciar sesión con la cuenta de administrador. 2. Acceder al apartado de administración. 3. Ir a *Dispositivos* y se mostrará un listado con los dispositivos existentes. 4. Al final hay un botón para descargar la información de cada dispositivo.
Precondición: La cuenta que realice los pasos anteriores tiene que ser administrador y tiene que haber dispositivos creados previamente.

Tabla 3.7: CU05. Exportar información de dispositivos.

CU06: Consultar datos de los sensores
Autor: ALDANE Revisor: H. M. P.
Descripción: Poder consultar los datos recogidos por los sensores y mostrarlos mediante gráficas.
Secuencia de pasos normal: 1. Iniciar sesión y acceder al apartado de gráficos. 2. Rellenar el formulario para consultar un intervalo de tiempo. 3. Se mostrará un gráfico y la posibilidad de ver los datos formateados en tablas.
Excepción: El formulario tiene que estar correctamente cumplimentado.

Tabla 3.8: CU06. Consultar datos de los sensores.

CU07: Exportar los datos recogidos
Autor: ALDANE Revisor: H. M. P.
Descripción: Poder exportar los datos recogidos, en un intervalo de tiempo, a un fichero *CSV* para poder realizar operaciones con ellos.
Secuencia de pasos normal: 1. Iniciar sesión y acceder al apartado de gráficos. 2. Rellenar el formulario para consultar un intervalo de tiempo. 3. Se mostrará un gráfico y debajo un botón para poder descargar la información.
Excepción: El formulario tiene que estar correctamente cumplimentado.
Precondición: Es necesario que haya almacenados datos de los sensores para poder exportar la información.

Tabla 3.9: CU07. Exportar los datos recogidos.

3.2. Diseño de la arquitectura del sistema

En este apartado se van a diferenciar tres tipos de diseños. Por un lado, tenemos el diseño del nodo donde veremos y entenderemos el funcionamiento de los sensores y de la transmisión de la información. Por otro lado, está el diseño de la base de datos y cómo almacenamos toda la información que recogen los sensores y lo relacionamos con los dispositivos correspondientes. Y, finalmente, el diseño de la aplicación *web* basado en una arquitectura *REST*.

3.2.1. Diseño del nodo

Como vimos anteriormente, el objetivo de este proyecto es desarrollar una aplicación *web* para poder configurar el nodo y poder visualizar los datos recogidos. Por lo tanto, como podemos deducir, es un proyecto enfocado al apartado de *software*, pero está directamente relacionado con todos los dispositivos (sensores, nodo) que son la base para poder realizar la aplicación *web*. Por este motivo es necesario explicar el funcionamiento del *hardware*, figura 3.2, para entender cómo llega la información desde el sensor hasta la base de datos.

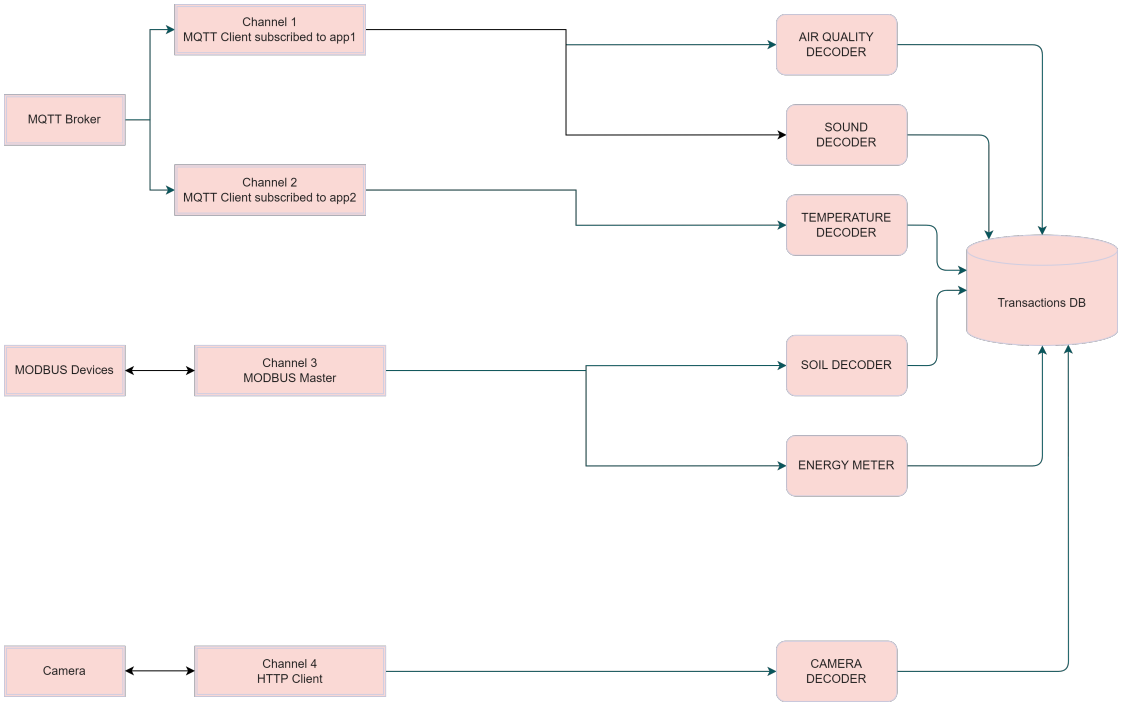

Figura 3.2: Esquema del funcionamiento del nodo.

Como podemos observar en la figura 3.2, la primera columna empezando por la izquierda hace referencia a los sensores, podemos diferencias tres tipos:

- **Dispositivos *MQTT:*** hace referencia a todos los dispositivos que utilizan el protocolo *MQTT* para la transmisión de datos. En nuestro caso el sensor de aire, de sonido y de temperatura.
  MQTT es un protocolo de red ligero, de publicación y suscripción, de máquina a máquina. Está diseñado para conexiones con ubicaciones remotas que tienen dispositivos con restricciones de recursos o ancho de banda de red limitado [6].

- **Dispositivos *MODBUS:***: al contrario que en el caso anterior, este tipo de dispositivos utilizan el protocolo de comunicaciones *MODBUS*.
  MODBUS es un protocolo de comunicaciones situado en los niveles 1, 2 y 7 del modelo *OSI*, basado en la arquitectura cliente/servidor (*TCP/IP*). Es el protocolo más utilizado en la automatización industrial y el medio más común para conectar dispositivos electrónicos automatizados [7].

- **Cámaras de vídeo**: Por último, tenemos las cámaras de vídeo que utilizan el protocolo HTTP (*Hiper Text Transfer Protocol*).

En la segunda columna encontramos información del canal por el que se transmiten los datos. Cada canal tiene los siguientes campos: nombre, tipo de protocolo que utiliza, propiedades del canal y si está activado. Toda esta información se verá más clara en el siguiente punto cuando hablemos del diseño de la base de datos.

En la tercera columna tenemos el códec que se utiliza en cada caso para decodificar el flujo de datos digitales y poder escribir la información en la base de datos. Algunos ejemplos de códecs que hemos utilizado son *IoTBus*, *Milesight* y *Modbus*.

3.2.2. Diseño de la base de datos

Ahora que ya entendemos el funcionamiento de los dispositivos y cómo se transfiere la información, vamos a explicar la forma en la que almacenamos todos los datos. Para entender el almacenamiento de los datos partimos de la siguiente figura 3.3.

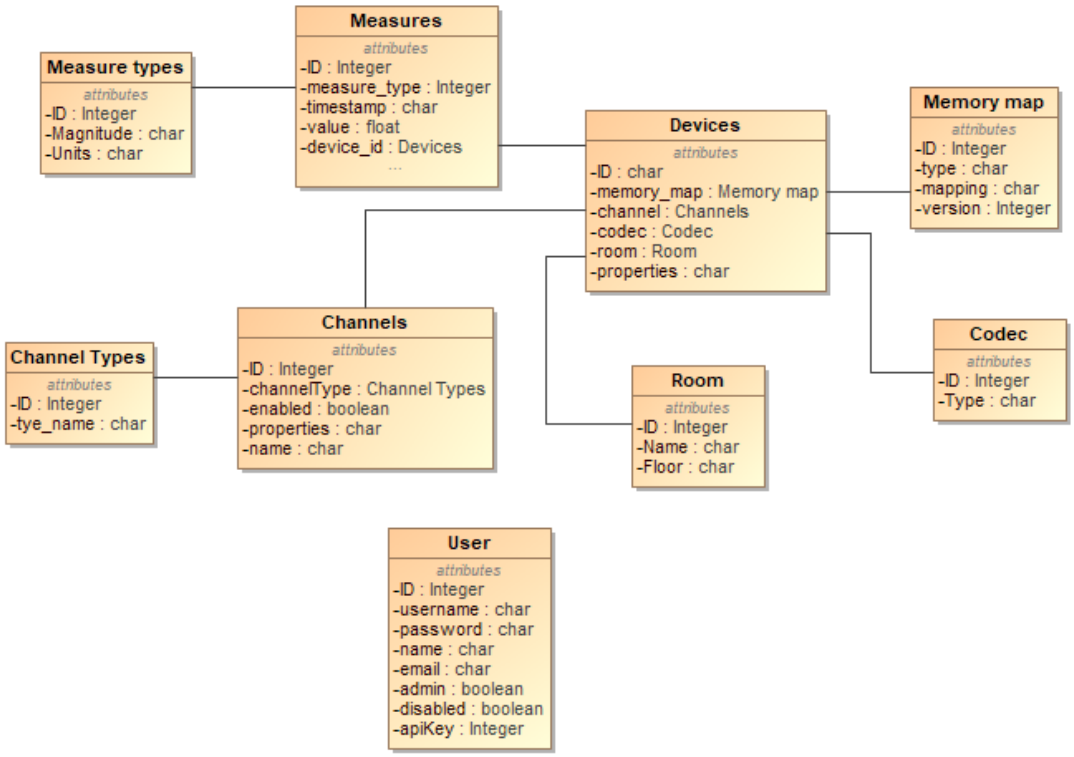

Figura 3.3: Diseño de la base de datos.

Para entender la relación de la base de datos vamos a empezar viendo la clase "dispositivo". Esta clase tiene los siguientes atributos:

- ***ID:*** es el identificador del dispositivo y es de tipo *string*, ya que el *ID* es el nombre del dispositivo y debe ser único. En nuestro caso, los dispositivos 0004A30B0050B9D4, 0004A30B004F93E2 hacen referencia a dos sensores de temperatura.

- ***Memory map:*** en función de cada dispositivo dispone de un mapa de memoria con la configuración necesaria para cada tipo de medida que recoja. Un ejemplo de mapa de memoria es el de la figura 3.4.

Battery	channel_id	1
	channel_type	117
	data_len	1
	measure_type	6
Temperature	channel_id	3
	channel_type	103
	data_len	2
	measure_type	1
Humidity	channel_id	4
	channel_type	104
	data_len	1
	measure_type	3
Contact_status	channel_id	6
	channel_type	0
	data_len	1
	measure_type	10

Figura 3.4: Mapa de memoria.

- ***Codec:*** indicamos qué tipo de códec utiliza para decodificar los datos como explicamos en el apartado anterior.
- ***Room:*** almacenamos la ubicación de cada dispositivo dentro del edificio, para ello indicamos el nombre de la sala y en qué planta se encuentra.
- ***Channels:*** para poder explicar los canales lo vamos a hacer basándonos en las figuras 3.5 y 3.6.

Enabled		Channel name	Channel type
true	Properties	TCP modbus simulador	MODBUS master
true	Properties	PRE_Lora	Chirpstack MQTT client
true	Properties	PRO_FERIAS	Chirpstack MQTT client

Figura 3.5: Canales.

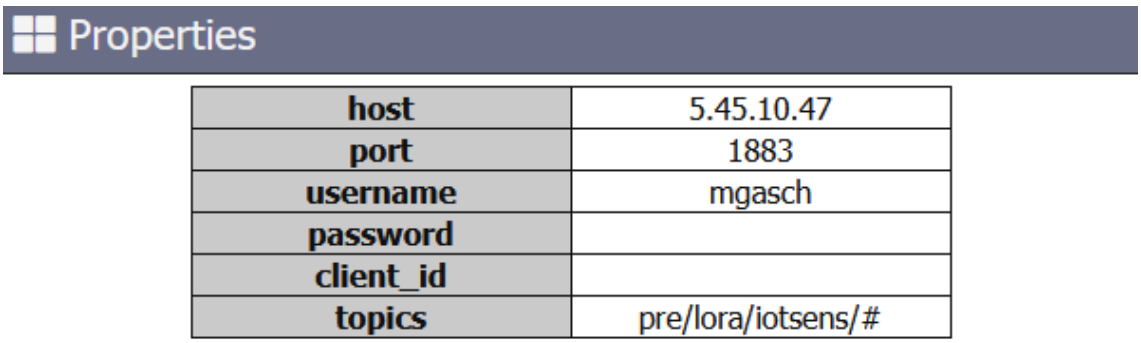

Figura 3.6: Propiedades del canal.

Como podemos observar en la figura 3.5 tenemos información de si el dispositivo se encuentra activado, el nombre del canal y qué tipo de canal está utilizando para transmitir la información. Como vemos, en las propiedades 3.6 tenemos la dirección *IP* y el puerto que utiliza, el usuario y contraseña (censurada por motivos de seguridad) que utiliza para poder realizar la transmisión, el *client id* que en este caso está vacío y distintas etiquetas para identificar rápidamente el canal.

Por otro lado, tenemos las medidas, donde almacenamos lo siguiente:

- ***ID:*** el identificador único para cada medida recogida.
- ***Measure type:*** es el tipo de medida recogida, para saber qué estamos midiendo. Algunos ejemplos son: Temperatura (grados Celsius), Humedad (porcentaje de humedad relativa), CO_2 (partículas por billón).
- ***Timestamp:*** es la fecha y hora en la que se realizó la medición.
- ***Value:*** valor recogido en ese momento.
- ***Device id:*** dispositivo que realizó la medida.

Por último tenemos una tabla "usuario" donde almacenamos los usuarios dados de alta en el sistema y con distintas propiedades según el usuario. Usuario, contraseña, nombre completo, dirección de correo, si tiene permisos de administrador, si el usuario está activado y una *API key* para poder acceder a los recursos de la *API*.

3.2.3. Diseño *API REST*

Una vez que hemos visto el funcionamiento del nodo y dónde se almacenan los datos, vamos a pasar a ver el funcionamiento de la aplicación *web* 3.7. Para realizar la lógica de la aplicación *web* se decidió utilizar la interfaz *API REST*.

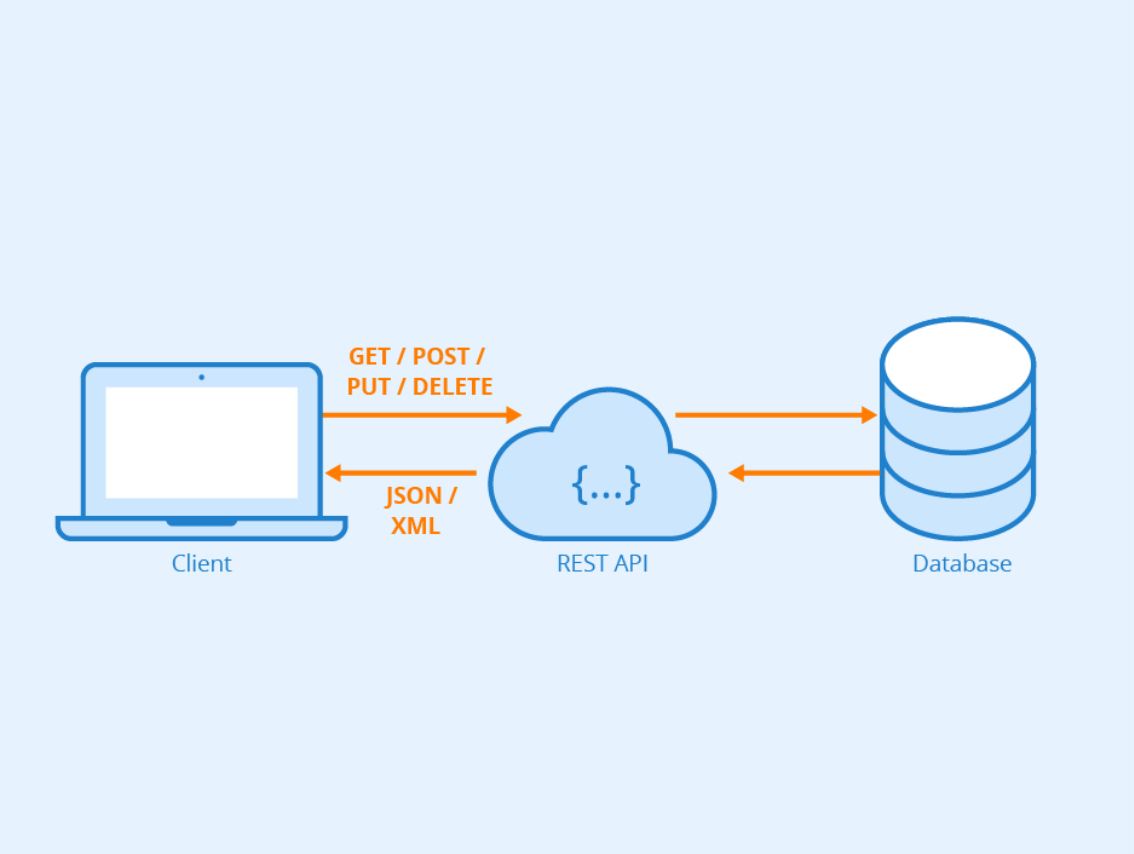

Figura 3.7: Diseño lógico de la aplicación.
(Fuente: [8]).

REST es una interfaz para conectar varios sistemas basados en el protocolo *HTTP* y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como *XML* y *JSON* [9].

En nuestro caso hemos utilizado el formato *JSON* debido a que es algo más ligero que *XML* y es uno de los formatos más utilizados en la actualidad.

Basándonos en la figura 3.7, cuando el usuario hace uso de la aplicación, realiza peticiones a nuestra *API*. Estas peticiones las llevamos a cabo gracias a la librería *Ajax* de *JavaScript* (los detalles de implementación los veremos más a fondo en el próximo capítulo) y pueden ser de tipo *GET, POST, PUT* y *DELETE* en función de la operación que esté realizando el usuario. Cuando esta petición le llega a la *API* (los posibles errores se validan en el lado del usuario), accede al objeto de acceso a los datos y realiza la operación requerida sobre la base de datos. Una vez que ha obtenido la información de la base de datos, procede a enviárselos al navegador del usuario en formato *JSON*. Una vez allí los datos son interpretados y en función de la operación

realizada se mostrará un mensaje al usuario o se representarán los datos solicitados.

3.3. Diseño de la interfaz

Para realizar esta parte, el jefe de proyecto decidió seguir los estándares fijados por la empresa para que el diseño sea similar a otras aplicaciones creadas previamente. Por lo que se nos proporcionó una plantilla con los estilos e imágenes de la empresa.

Aunque partíamos de una plantilla, en las primeras reuniones dejamos definidos los objetivos que tenía que cumplir la interfaz:

- Una interfaz adecuada a los usuarios y a las actividades a realizar.
- Que el tiempo utilizado en aprender a usar la aplicación sea el menor posible
- Minimizar la tasa de errores y, en caso de que suceda alguno, avisar al usuario con el problema concreto y no con un mensaje de error genérico.
- Proporcionar rapidez en el uso y en la respuesta de la interfaz.

Vamos a ver distintas vistas de la aplicación y a explicar algunos detalles en referencia a la interfaz.

En la figura 3.8, disponemos de la vista para acceder a la plataforma. Tenemos el formulario de acceso con el logotipo identificativo de la empresa, una imagen de fondo de una ciudad interconectada y el formulario con los campos de usuario y contraseña para poder acceder.

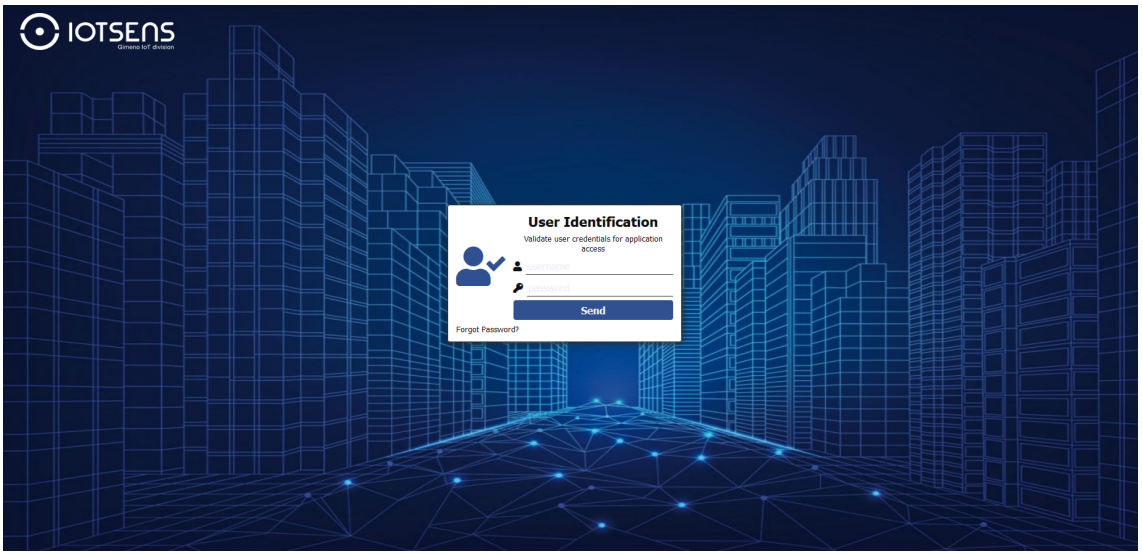

Figura 3.8: Página de *login*

En la figura 3.9 tenemos la página para visualizar los datos. En la parte superior tenemos la cabecera que es igual para el resto de vistas y, un poco más abajo, un formulario para consultar los datos recogidos en un intervalo de tiempo y también debemos elegir el dispositivo que queremos consultar y el tipo de medida. Estos dos últimos campos, dispositivo y tipo de medida, están relacionados, ya que si primero seleccionas un dispositivo, la lista de tipos de medidas se actualiza con los tipos que recoge ese dispositivo y viceversa: si seleccionas primero el tipo de medida se actualiza la lista de dispositivos para poder seleccionar, únicamente, los dispositivos que recogen ese tipo de medida.

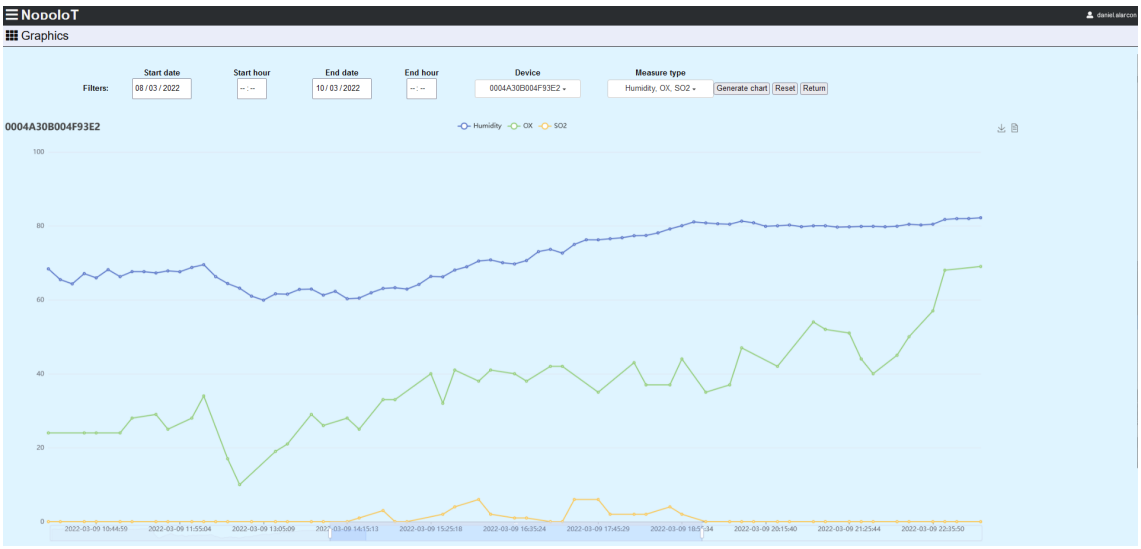

Figura 3.9: Página de visualización de datos.

En la figura 3.10 podemos observar el menú de administrador con las distintas opciones:

- ***Add device:*** nos lleva al formulario para añadir un nuevo dispositivo.
- ***Codecs:*** nos permite administrar los codecs que tenemos creados. Si queremos añadir uno nuevo tenemos la opción de hacerlo en la página de “Añadir dispositivo".
- ***Channels types:*** es la página correspondiente para administrar los tipos de canales.
- ***Measures types:*** en esta página podemos administrar y añadir tipos de medidas.
- ***Maps:*** nos muestra la información de los mapas de memoria y nos permite gestionarlos.
- ***Channels:*** es una vista muy similar a la de los mapas de memoria pero con la información relativa a los canales.
- ***Room:*** nos permite gestionar las distintas ubicaciones donde queremos situar los dispositivos.
- ***Devices:*** es una página con toda la información de los dispositivos y con distintos botones para poder administrarlos. Ahora más adelante mostraremos esta ventana.
- ***Users manager:*** para tener el control de todos los usuarios y poder dar de alta usuarios nuevos.

- ***Home:*** es el último botón de este menú que nos permite volver al inicio de la aplicación.

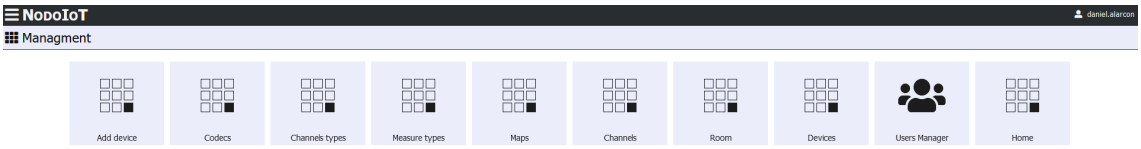

Figura 3.10: Página de visualización de datos.

Como hemos comentado anteriormente, tenemos la vista para administrar los dispositivos, como observamos en la figura 3.11.

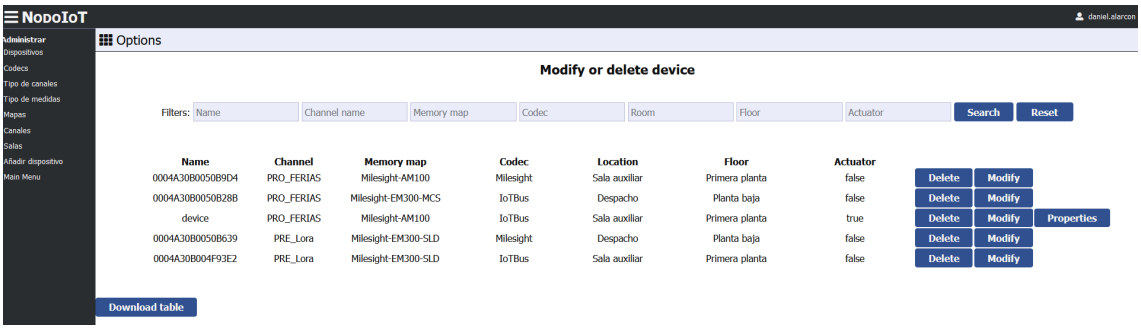

Figura 3.11: Página de administración de dispositivos.

En esta vista disponemos de diferentes opciones. En primer lugar, en el lateral izquierdo tenemos un menú de fácil acceso para movernos por las distintas opciones de configuración comentadas en el apartado anterior 3.3.

Mirando la imagen, en la parte más céntrica podemos encontrar un pequeño formulario por si necesitamos buscar o filtrar los dispositivos por alguna característica específica. Y, justo debajo, tenemos una tabla con toda la información de los dispositivos y con los correspondientes botones para poder modificar algún dato o incluso borrarlos.

Y, finalmente, en la parte inferior izquierda disponemos del botón para poder descargar la información de la tabla en formato *CSV*.

Capítulo 4

Implementación y pruebas

En este capítulo se explica cómo ha sido la implementación de la aplicación, explicando distintas partes del código y las pruebas que hemos realizado para asegurar el correcto funcionamiento de la aplicación.

4.1. Detalles de implementación

Para explicar los detalles de la implementación vamos a diferenciar entre tres partes: explicación del desarrollo de la *API*, por otro lado, veremos el objeto de acceso a los datos (*DAO*) y, finalmente, los detalles de implementación de la página *web*.

4.1.1. *API*

El lenguaje de programación escogido para desarrollar la *API* fue *PHP*, como se comentó en capítulos anteriores, esto es debido a que necesitamos que sea lo más ligero posible para que se adapte a las necesidades del nodo.

El funcionamiento de la *API* es el siguiente:

- Recibe una petición solicitada por el usuario.
- Analiza el tipo de operación y lee los datos correspondientes. En caso de que la petición sea errónea, devuelve un error al usuario.
- Una vez que tiene la información necesaria, realiza la solicitud del usuario.
- Cuando la operación ha sido realizada, devuelve un mensaje y la información correspondiente.

En el apartado A.1 del anexo se adjunta código de ejemplo para entender fase por fase cómo se resuelven las peticiones.

4.1.2. *DAO*

El objeto de acceso a los datos nos permite separar la lógica de acceso a los datos de la aplicación. De esta manera, obtenemos métodos más concretos y con una utilidad definida, en caso de juntarlo todo en la misma clase pueden quedar métodos más extensos y confusos. Al tener separada la lógica de la *API* y del *DAO* nos permite utilizar ambas clases para cualquier uso externo de la aplicación.

Esta clase está documentada con los comentarios necesarios para cumplir los estándares de codificación. De esta manera, al tener el código correctamente comentado, podemos generar automáticamente la documentación en la que vienen explicadas todas las partes que hemos desarrollado. Esto es muy útil porque las personas que vayan a utilizar este código o vayan a ampliarlo e incluso mejorarlo pueden encontrar la documentación con todo explicado y entender qué realiza cada parte. La documentación para la clase *DAO* se ha añadido en el anexo B. Se han decidido mantener los estilos y numeraciones del propio documento, de ahí el motivo de añadirlo al final, en el anexo.

Al comienzo del proyecto se deja definido el tipo de base de datos que queremos utilizar (*PostgreSQL*), aunque se comentó que en el futuro este tipo se podía cambiar por *MySQL*, por lo que para realizar esta parte decidimos utilizar una librería genérica que nos permite acceder a distintos tipos de base de datos.

El funcionamiento de esta clase es el siguiente:

- Para poder utilizar las funciones hay que crear un objeto de esta clase, pasando como parámetro el tipo de base de datos.
- La clase con ese parámetro inicializa la conexión con el servidor de base de datos.
- Una vez que tenemos el objeto creado, ya podemos llamar a los distintos métodos para realizar la operación correspondiente en ese momento.
- Cuando queremos terminar de realizar operaciones hay que llamar al método correspondiente para finalizar la conexión.

En el apartado A.2 del anexo se adjunta código de ejemplo para entender cómo se realiza la conexión con la base de datos y cómo se ejecutan las diferentes consultas.

4.1.3. Código *web*

Para desarrollar las vistas de la página *web* se han utilizado diversos lenguajes: *HTML, CSS, JavaScript y PHP.*

- *HTML:* se ha utilizado para dar forma y estructura a la *web.*
- *CSS:* se utiliza para manejar el diseño y la presentación. Los estilos utilizados vienen definidos por la propia empresa para seguir la marca de negocio. Muchos de estos estilos utilizan *bootstrap* que es una biblioteca o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones *web*. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño [10].
- *JavaScript:* utilizado principalmente para realizar las peticiones al servidor mediante *Ajax* y utilizando la librería de *jQuery*. Gracias a estas bibliotecas, una vez que se obtienen los datos, nos permiten modificar el documento *HTML* para mostrar la información al usuario.
- *PHP:* se ha utilizado para servir las páginas *web* al usuario. Al estar enfocado al lado del servidor nos permite mostrar estas páginas con distintos parámetros configurados en función del usuario.

Al igual que los apartados anteriores, en el anexo A.3 encontramos ejemplos utilizados en el proyecto que demuestran el desarrollo de esta parte.

4.2. Verificación y validación

La verificación y validación de la aplicación se puede diferenciar en dos partes: las comprobaciones que se realizan en el código para evitar errores y las pruebas finales que hemos realizado para comprobar que el funcionamiento de la aplicación es el adecuado.

Respecto a las comprobaciones sobre el código, cada vez que el usuario rellena un formulario o introduce algún dato, se realiza una comprobación para asegurar que la información introducida es válida. La verificación se realiza en el lado del cliente, lo que supone una mayor velocidad, ya que no hace falta una conexión con el servidor para asegurarnos que los parámetros son correctos. De esta forma, nos aseguramos que cuando se realiza una petición al servidor los datos ya han sido validados y se puede proceder con la solicitud. En el anexo A.4 encontramos un ejemplo de validación de datos de un formulario.

Finalmente, para la realización de pruebas, se han realizado varias fases:

- Cuando se realiza un nuevo formulario o una nueva funcionalidad en la *API* y en el *DAO*, los propios desarrolladores realizamos pruebas con todos los casos posibles para comprobar que se tratan correctamente los posibles errores. Al tener distintas clases hay que realizar pruebas a cada clase.

Empezamos comprobando que la clase *dao.php* ejecuta las sentencias *SQL*, modifica la base de datos y obtiene la información correctamente. A continuación, siguiendo el orden de desarrollo, se comprueba la *API* para asegurarnos que tramita las peticiones del cliente y llama a los métodos correspondientes del *DAO* para realizar las operaciones.

Finalmente, realizamos las pruebas de la vistas comprobando que son capaces de identificar los posibles errores en los formularios y notifican correctamente al usuario con un mensaje explícito. Además, nos aseguramos de que las peticiones son enviadas al servidor de forma correcta y que las respuestas son tratadas adecuadamente.

Para la ejecución de estas pruebas no se ha utilizado ningún software automatizado, han sido los desarrolladores los que han probado manualmente cada posible caso de error para asegurar el correcto funcionamiento.

- Una vez que la aplicación tiene las vistas desarrolladas y ha superado las pruebas de los desarrolladores, se le pide a distintos usuarios que accedan y utilicen las distintas funcionalidades. De esta forma, obtenemos retroalimentación del diseño de la página *web* por si es necesario cambiar algún estilo o añadir una nueva funcionalidad y comprobamos si salta algún error que no hayamos tratado.

  Estas pruebas realizadas a usuarios ajenos al desarrollo de la aplicación nos permiten obtener distintas opiniones para poder implementar mejoras o modificar alguna parte del diseño si varios usuarios han coincidido en el mismo punto.

- Por último, *IoTsens* tiene un grupo de personas para realizar pruebas más a fondo sobre una aplicación una vez que está desarrollada, así, antes de lanzarla al mercado, se aseguran de que la plataforma es capaz de reaccionar y tratar todos los errores que puedan surgir.

  Estas pruebas que realiza el departamento intentan buscar casos más específicos de error. Por ejemplo, se simula una conexión inestable y con pérdidas de paquetes, de esta manera, cuando lanzas una petición al servidor el tiempo de respuesta es muy alto y la aplicación ha de tenerlo en cuenta para que no se quede en espera mucho tiempo y mostrar el mensaje adecuado.

Capítulo 5

Conclusiones

El resultado de este proyecto es una aplicación que nos permite visualizar los datos recogidos por el nodo y, además, podemos configurar los distintos dispositivos conectados a él. Como este proyecto ha abarcado tantas áreas, podemos sacar distintas conclusiones.

A nivel formativo este proyecto me ha servido para aprender el funcionamiento del internet de las cosas y cómo se realiza toda la conectividad entre los distintos dispositivos. También he profundizado en lenguajes como *PHP* y *JavaScript*, lo que ha hecho que aprenda cómo es el funcionamiento de una página *web* y cómo se comunica con la parte del servidor y se trata la información.

A nivel profesional ha sido una gran experiencia, ya que he tenido la oportunidad de incorporarme en una empresa con un gran equipo detrás. Lo que ha hecho que aprenda a trabajar en proyectos reales con más compañeros, aportando nuevas ideas y colaborando entre nosotros.

Finalmente, a nivel personal ha sido un proyecto con el que me siento satisfecho y, orgulloso de haber sacado adelante. Me he sentido muy cómodo con los compañeros y ha habido muy buena relación, lo que ha hecho que la comunicación sea más fluida.

Bibliografía

[1] P. web IoTsens, "Sensor de sonido." https://www.iotsens.com/producto/sensor-de-sonido/. [Consulta: 25 de mayo de 2022].

[2] P. web IoTsens, "Sensor de co2 interior." https://www.iotsens.com/producto/sensor-co2-interior/. [Consulta: 25 de mayo de 2022].

[3] P. web IoTsens, "Sensor de calidad del aire." https://www.iotsens.com/producto/sensor-de-calidad-del-aire/. [Consulta: 25 de mayo de 2022].

[4] Ionos, "Diagrama de casos de uso." https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/diagrama-de-casos-de-uso/. [Consulta: 27 de mayo de 2022].

[5] "Magicdraw." https://www.magicdraw.com/. [Consulta: 27 de mayo de 2022].

[6] "Información mqtt." https://en.wikipedia.org/wiki/MQTT. [Consulta: 1 de junio de 2022].

[7] "Información modbus." https://es.wikipedia.org/wiki/Modbus. [Consulta: 1 de junio de 2022].

[8] "Seobility." https://www.seobility.net/es/wiki/API_REST. [Consulta: 11 de junio de 2022].

[9] "Rest." https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/. [Consulta: 3 de junio de 2022].

[10] "Bootstrap." https://es.wikipedia.org/wiki/Bootstrap_(framework). [Consulta: 3 de junio de 2022].

[11] "Doxygen." https://doxygen.nl. [Consulta: 10 de junio de 2022].

Anexo A

Detalles de implementación

A.1. *API*

Vamos a proceder con la explicación de las partes más importantes del código, empezaremos viendo los parámetros definidos y las distintas cabeceras que hemos creado.

En la mitad superior de la figura A.1 dejamos definidas las rutas de acceso a distintos ficheros de la aplicación, de esta forma cuando queramos hacer una redirección, en vez de tener que escribir cada vez la ruta, solo hace falta poner la variable a la que nos queremos dirigir.

En las variables siguientes recuperamos los valores correspondientes en cada caso y los filtramos para remover caracteres especiales.

En la mitad inferior definimos distintas cabeceras, esto sirve para cuando enviemos la información solicitada al usuario; las cabeceras de los paquetes *HTTP* llevarán definidos estos parámetros.

```php
<?php

//
// API
//

$app_config["rootFolder"]   = "/var/www/sketch";
$app_config["urlPrefix"]    = "/sketch";
$app_config["apiPrefix"]    = $app_config["urlPrefix"] ."/api";
$app_config["homePage"]     = $app_config["urlPrefix"] ."/index.html";
$app_config["loginPage"]    = $app_config["urlPrefix"] ."/login/user_login.php";

$_GET    = filter_input_array( INPUT_GET,    FILTER_SANITIZE_STRING );
$_POST   = filter_input_array( INPUT_POST,   FILTER_SANITIZE_STRING );
$_COOKIE = filter_input_array( INPUT_COOKIE, FILTER_SANITIZE_STRING );
$_SERVER = filter_input_array( INPUT_SERVER, FILTER_SANITIZE_STRING );

//// cache control
header("Cache-Control: no-store, max-age=0");
//header("Cache-Control: no-store, no-cache, must-revalidate, max-age=0");
//header("Pragma: no-cache");
//header("Expires: ". gmdate("D, d M Y H:i:s") ." GMT");
header("Last-Modified: ". gmdate("D, d M Y H:i:s") ." GMT");

//// cors control
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Headers: Origin, Accept, Content-Type, X-Requested-With, Access-Control-Request-Method");
header("Access-Control-Allow-Methods: GET, POST, HEAD, OPTIONS, PUT, DELETE");
header("Allow: GET, POST, HEAD, OPTIONS, PUT, DELETE");

//// mime type
header("Content-Type: application/json; charset=utf-8");
```

Figura A.1: Cabeceras de la *API*.

Antes de realizar ninguna operación tenemos que recibir la información del usuario para saber qué operación quiere realizar.

Como podemos observar en la figura A.2, en la primera línea importamos un fichero con distintas funcionalidades. El fichero "dao.php" nos permite realizar las operaciones sobre la base de datos como veremos más adelante 4.1.2.

```php
require_once("dao.php");

if( strlen( $post_json = file_get_contents("php://input") ) == 0 ) {
    send_response(400,array( "code" => "400", "text" => "fail input post" )));
    exit;
} elseif( !( $post_data = json_decode( $post_json, true, 4 ) ) ) {
    send_response(400,array( "code" => "400", "text" => "fail decode post" )));
    exit;
};
```

Figura A.2: Lectura de operación desde la *API*.

En las siguientes líneas encontramos dos condiciones. La primera de ellas convierte la información que nos llega a tipo *string* y, en caso de que la longitud sea cero, significa que no sabemos qué tenemos que hacer y devolvemos el error con un método que ahora veremos su utilidad.

En la siguiente condición intentamos convertir el texto recibido (que debería ser de tipo *JSON*) a un vector llamado "post_data", en caso de que el texto no sea de tipo *JSON* o contenga algún error de sintaxis y no permita leerlo, devolvemos un error indicando el problema.

El método "send_response" lo tenemos en la figura A.3.

```
function send_response( $http_code, $send_message ) {
    $http_status = array (
        200 => "200 OK",
        400 => "400 Bad Request",
        403 => "403 Forbidden",
        500 => "500 Internal Server Error",
    );

    http_response_code( $http_code );

    header("Status: ". $http_status[ $http_code ]);
    echo json_encode( $send_message );
};
```

Figura A.3: Método *"send_response"*.

Este método recibe dos parámetros, uno con el código del estado y otro con el mensaje que tenemos que enviar. Añade la cabecera correspondiente al estado del paquete y escribe el mensaje que tiene que devolver. Este método también se utilizará cada vez que queremos devolver información al usuario, como vamos a ver ahora.

Una vez llegados a este punto, la petición del usuario es correcta y procedemos a realizar lo que solicita como observamos en la figura A.4.

```
switch( strtolower( $_SERVER["QUERY_STRING"] ) ) {

    case "getcodecs":
        $dao = new DAO($post_data["bbdd"]);
        $codecs = $dao->getCodecs();
        send_response(200, $codecs);
        break;

    case "deletechanneltype":
        $dao = new DAO($post_data["bbdd"]);
        $vari = $dao->deleteChannelType($post_data["channeltype"]);
        if(!$vari){
            send_response(400, array( "code" => "400", "text" => "error when deleting channel type" ));
            break;
        }
        send_response(200, array( "code" => "200", "text" => "channel type deleted" ));
        break;
```

Figura A.4: Operaciones de la *API*

En la primera sentencia, dentro de "switch" leemos la operación que quiere realizar el usuario. Esta operación se declara en *JavaScript* con las peticiones *Ajax*, como vemos en la figura A.11.

Una vez que hemos leído la operación, con las sentencias "case", nos vamos al apartado correspondiente para realizarla. En la imagen aparecen dos, aunque en nuestro caso hay muchas más, pero como son todas muy similares se ha decidido poner esas dos de ejemplo.

En la línea “case getcodecs”, significa que hemos recibido una petición para consultar todos los códecs existentes. Para ello creamos un nuevo objeto *DAO* que recibe como parámetro el tipo de la base de datos. Esto está hecho así para poder trabajar con diferentes bases de datos, como se explica en el apartado 4.1.2.

Posteriormente, llamamos al método correspondiente del objeto que hemos creado para poder obtener todos los códecs (método definido en A.6). Y, finalmente, haciendo uso del método “send_response” enviamos la información al cliente.

El segundo tipo de operación que vemos es para eliminar un tipo de canal. Al igual que antes, inicializamos el objeto para acceder a la base de datos. Llamamos al método correspondiente pasándole como parámetro la información que hemos decodificado previamente (en este caso, la información es el *ID* del tipo de canal que vamos a eliminar).

Al ser una operación de borrado de datos puede ser que haya algún error al eliminar debido a que el tipo de canal esté como clave foránea en otra tabla. Para asegurarnos, comprobamos mediante la condición *if* si se ha podido eliminar; en caso afirmativo, respondemos con el mensaje correspondiente y, en caso contrario, avisamos de que no se ha podido eliminar.

A.2. *DAO*

En la figura A.5, observamos la definición de la clase y dentro tenemos un objeto privado que es donde almacenaremos la conexión con la base de datos.

```php
<?php

class DAO {

    /**
     * @access protected
     * @var PDO
     */
    private $mbd;

    /**
     * @param string $bbdd Nombre del tipo de base de datos
     */
    function __construct($bbdd){         //pgsql
        $this->mbd = new PDO("$bbdd:host=localhost;port=5432;dbname=nodo;", "postgres", "postgresql"); // PDO Connection
    }
```

Figura A.5: Constructor de la clase *DAO*.

Un poco más abajo, tenemos el constructor de la clase que recibe como parámetro el tipo de base de datos que queremos inicializar. Dentro de este método asignamos a la variable anterior un nuevo objeto *PDO* (librería utilizada para acceder a la base de datos). Este objeto recibe como parámetros el tipo de base de datos, la dirección *IP* (en este caso la del propio equipo), el puerto por el que se conecta (*PostgreSQL* por defecto utiliza el puerto 5432), el nombre de la base de datos, y el usuario y contraseña (se han puesto los valores por defecto para no revelar la autenticación original).

Ahora vamos a explicar diferentes métodos utilizados en esta clase. Para hilarlo con los métodos vistos en la *API*, A.4, vamos a ver cómo son tratadas esas dos llamadas.

Como observamos en la figura A.6, tenemos el método para obtener toda la información de los códecs. En la primera línea del método creamos la variable que almacena la sentencia *SQL* para ejecutarla en la base de datos. En la segunda línea nos apoyamos de un método auxiliar para realizar las consultas de obtención de datos, figura A.7.

```php
/**
 * Obtiene toda la información de los codecs
 *
 * @return array|false $this->get($query) Devuelve un array con la información de los codecs
 * o false en caso de no haber.
 */
public function getCodecs(){                                    //SELECT CODEC
    $query = "SELECT * FROM codecs";
    return $this->get($query);
}
```

Figura A.6: Método "getCodecs".

```php
/**
 * Ejecuta una instrucción (query) para consultar información
 *
 * @param string $query Instrucción que va a ejecutar en la base de datos
 * @return array|false $this->mbd->query($query)->fetchAll()
 * Devuelve un array con la información obtenida o, false en caso de no haber.
 */
private function get($query){                                   //SELECT GENERAL
    return $this->mbd->query($query)->fetchAll();
}
```

Figura A.7: Método "get".

De esta forma cada método genera su propia sentencia *SQL* para obtener datos y esta sentencia es ejecutada por este método genérico. A continuación, vamos a ver cómo se realiza para las operaciones de borrado de datos.

En la figura A.8 tenemos el método correspondiente para eliminar un tipo de canal. Este método recibe como parámetro el *ID* y a continuación genera la sentencia *SQL* correspondiente para eliminar el tipo de canal. Al igual que antes, tenemos un método genérico para ejecutar las sentencias de borrado, figura A.9.

```php
/**
 * Elimina un tipo de canal en función del id
 *
 * @param int $id ID a eliminar en la tabla tipo de canal
 * @return int $this≥add($query) Devuelve el ID de la fila eliminada.
 */
public function deleteChannelType($id){                                    //DELETE CHANNEL TYPE
    $query = "DELETE FROM channel_types WHERE id = $id";
    return $this->delete($query);
}
```

Figura A.8: Método "deleteChannelType".

```php
/**
 * Ejecuta una instrucción (query) para eliminar información
 *
 * @param string $query Instrucción que va a ejecutar en la base de datos
 * @return int|boolean $this->mbd->prepare($query)->execute() Devuelve el ID de la fila
 * eliminada o false en caso de error.
 */
private function delete($query){                                           //DELETE GENERAL
    return $this->mbd->prepare($query)->execute();
}
```

Figura A.9: Método "delete".

Este método recibe la sentencia y la ejecuta, devuelve un valor entero en caso de que se haya podido eliminar el dato o *false* en caso de que se haya producido algún error.

A.3. Código *web*

Esta parte de código es bastante más extensa que los apartados anteriores, por lo que vamos a explicar los puntos más importantes del código *web*.

Vamos a empezar explicando las peticiones que realizamos con *Ajax*. En la figura A.10 definimos distintos parámetros. Es decir, todas la peticiones que realizamos en adelante utilizando *Ajax* llevarán estos parámetros definidos. Los parámetros más importantes los encontramos al final de la imagen: indicamos que el tipo de datos va en formato *JSON* y añadimos una cabecera con el token del usuario para poder acceder a la *api*.

```
$.ajaxSetup( {
    timeout: 60000,
    async: true,
    cache: true,
    crossDomain: true,
    processData: false,
    dataType: "json",
    contentType: "application/json; charset=utf-8",
    headers: {  "X-Session-Token":  sessionStorage.getItem("session_token")     },
} );
```

Figura A.10: Parámetros *Ajax*.

Una vez que hemos definido las distintas opciones podemos realizar peticiones como se muestra en la figura A.11.

```
$.ajax({
        type: "GET",
        url: "<?= $app_config["apiPrefix"] ?>/data_base.php?getcodecs",
        data: JSON.stringify({
            "bbdd": "pgsql",
        }),
        })
        .always( function(){
            viewOverlay.overlayHide();
        })
        .fail( function( jqXHR, textStatus, errorThrown){
            viewOverlay.overlayHide();
            console.log("text_status: "+ textStatus +", xhr_status: "+ jqXHR.status +", xhr_status_text: "+ jqXHR.statusText);
            viewOverlay.messageShow("Error al consultar códecs");
        })
        .done( function( respData, textStatus, jqXHR ){
            codecs = respData;
            for(var i in respData){
                $(".codec select").append('<option value="'+respData[i].id+'">'+respData[i].type +'</option>');
            };
        });
```

Figura A.11: Petición *Ajax*.

- El primer campo, *type*, sirve para indicar el tipo de petición que vamos a realizar. Como queremos obtener todos los códecs es una petición de tipo *get*.
- En la dirección *url* indicamos a dónde va dirigida la petición y añadimos al final "?getcodecs", este será el parámetro que analice la *API* para saber qué operación tiene que realizar.
- En el campo data, le pasamos el tipo de base de datos que queremos utilizar, en este caso *PostgreSQL*.
- En el método *always* indicamos qué hacer cuando obtengamos respuesta de la petición, independientemente de si nos devolvió un error o la información que buscamos. En nuestro caso, cuando se lanza una petición, salta un *overlay* para que el usuario no pueda utilizar la aplicación hasta que haya obtenido una respuesta (en el apartado de verificación y validación 4.2 se analizará más a fondo). Por lo tanto, en este método, como ya tenemos una respuesta, ocultamos el *overlay*.
- En el método *fail* indicamos qué hacer en caso de que nos devuelva un error la petición. Como vemos en el código, escribimos un mensaje por consola para que el desarrollador sepa qué está pasando y al usuario le mostramos el error correspondiente.

- En caso de que la petición haya resultado exitosa, añadimos a la vista la información recibida para que el usuario la pueda visualizar.

En la figura A.12 tenemos el formulario que utilizamos para añadir nuevos tipos de medidas. Es un formulario que utiliza estilos de *bootstrap* y tiene únicamente dos campos, magnitud y unidad. El texto, en vez de escribirlo directamente en un idioma, utiliza un diccionario con todas las palabras que existentes en la *web*. De esta manera, la aplicación se muestra en el idioma que haya elegido el usuario. Finalmente, tenemos dos botones, el botón *Add* para lanzar la petición y añadir un nuevo tipo de medida o el botón *Return* para volver atrás como se puede observar en la figura A.13.

```
<div id="modal_add" class="modal_template" style="display: none;">
    <div class="modal_header">
        <div><i class="modal_icon fas fa-th-large"></i></div>
        <div class="modal_display"><?= lang_get("Add")?></div>
    </div>
    <div class="modal_content">
        <div class="modal_grid_col2">
            <div class="text_label"><?= lang_get("Magnitude")?></div>
            <input id="magnitud" type="text" class="input_field">
        </div>
        <div class="modal_grid_col2">
            <div class="text_label"><?= lang_get("Units")?></div>
            <input id="unit" type="text" class="input_field">
        </div>
    </div>
    <div class="modal_footer">
            <button id="btn_accept"   class="text_bold"><?= lang_get("Add")?></button>
            <button id="btn_cancel"   class="text_bold"><?= lang_get("Return")?></button>
    </div>
</div>
```

Figura A.12: Formulario.

Figura A.13: Vista del formulario.

Para llegar a esta ventana tenemos que ir al apartado de administración, luego seleccionar tipos de medidas y pulsar el botón de añadir un nuevo tipo. Esta ventana aparecerá como un *overlay* por encima de la *web*, evitando que el usuario realice más acciones a la vez.

A.4. Verificación y validación

Para hilar con la figura A.12, vamos a explicar la verificación utilizada en los formularios y los *overlays* utilizados para evitar varias peticiones seguidas:

Dentro de la ventana A.13 tenemos dos botones que en el código vienen representados como los métodos *on* en la figura A.14. El primero de ellos hace referencia al botón de *Return* y lo que hace el botón es ocultar la ventana con el formulario.

```
$("#modal_add")
    .on("click", "#btn_cancel", function() {
        $("#modal_add").hide();
        viewOverlay.windowHide();
    })
    .on("click", "#btn_accept", function() {
        viewOverlay.overlayShow(0);

        if( $("#magnitud").val().trim().length == 0 ) {
            viewOverlay.messageShow( "<?= lang_get("ERROREMPTY")?>");
            viewOverlay.overlayHide();
            return false;
        };

        $.ajax( {
            type: "POST",
            url: "<?= $app_config["apiPrefix"] ?>/data_base.php?addmeasuretype",
            data: JSON.stringify( {
                        "magnitude": $("#magnitud").val(),
                        "units": $("#unit").val(),
                        "bbdd": "pgsql",
                    }),
        })
```

Figura A.14: Validación de formulario.

El segundo método *on* se ejecuta cuando el usuario le ha dado al botón de *Add*. La primera acción que realizamos es mostrar un *overlay*. La utilidad de lanzar un *overlay* en los formularios es para que el usuario no pueda seguir utilizando la aplicación de fondo mientras procesamos una petición y no nos envíe desde un mismo equipo muchas peticiones. De esta manera, el nodo tiene más soltura para responder las peticiones y así evitamos que pueda llegar a colapsarse la aplicación. En la condición encontramos cómo validamos el campo magnitud (solo sé valida este campo porque las unidades pueden estar vacías), ya que es un campo obligatorio y su longitud tiene que ser mayor que cero. En caso de recibir este campo vacío, se muestra el error correspondiente al usuario y, en caso de que esté correcto, se lanza la petición al servidor.

Anexo B

Documentación del código

En este anexo se adjunta la documentación generada sobre la clase *DAO*. Esto es una muestra de lo que se puede llegar a conseguir en un proyecto si se desarrolla y comenta el código correctamente. Es una funcionalidad muy útil porque cualquier persona que se vaya a unir al proyecto o vaya a trabajar en él puede encontrar la información rápidamente, por lo que su adaptación a la aplicación será más adecuada.

Esta documentación ha sido generada utilizando la herramienta *Doxygen*. Es una herramienta de código abierto que nos permite generar documentación para diversos lenguajes [11].

A partir de esta hoja se encuentra el documento manteniendo la integridad original de los estilos y de su propia numeración.

Documentación DAO

Generado por Doxygen 1.9.4

1 Índice de clases 1

1.1 Lista de clases . . . 1

2 Documentación de las clases 3

2.1 Referencia de la Clase DAO . . . 3

2.1.1 Documentación del constructor y destructor . . . 4

2.1.1.1 __construct() . . . 4

2.1.2 Documentación de las funciones miembro . . . 4

2.1.2.1 addChannel() . . . 4

2.1.2.2 addChannelType() . . . 5

2.1.2.3 addCodec() . . . 5

2.1.2.4 addDevice() . . . 5

2.1.2.5 addMeasuresType() . . . 6

2.1.2.6 addMeasureType() . . . 6

2.1.2.7 addMemoryMap() . . . 7

2.1.2.8 addSala() . . . 7

2.1.2.9 addUser() . . . 7

2.1.2.10 cerrarConexion() . . . 8

2.1.2.11 deleteChannel() . . . 8

2.1.2.12 deleteChannelType() . . . 8

2.1.2.13 deleteCodec() . . . 9

2.1.2.14 deleteDevice() . . . 9

2.1.2.15 deleteMeasure() . . . 9

2.1.2.16 deleteMeasureByDate() . . . 10

2.1.2.17 deleteMeasuresType() . . . 10

2.1.2.18 deleteMemoryMap() . . . 10

2.1.2.19 deleteSala() . . . 11

2.1.2.20 getAllDevices() . . . 11

2.1.2.21 getAllFromChannels() . . . 11

2.1.2.22 getChannelById() . . . 11

2.1.2.23 getChannels() . . . 12

2.1.2.24 getChannelTypes() . . . 12

2.1.2.25 getChannelTypesById() . . . 12

2.1.2.26 getCodecs() . . . 13

2.1.2.27 getCodecsById() . . . 13

2.1.2.28 getDevices() . . . 13

2.1.2.29 getMagnitudes() . . . 14

2.1.2.30 getMeasureByDate() . . . 14

2.1.2.31 getMeasureChart() . . . 14

2.1.2.32 getMeasures() . . . 15

2.1.2.33 getMeasuresTypes() . . . 15

2.1.2.34 getMeasuresTypesById() . . . 15

2.1.2.35 getMemoryMaps() 15
2.1.2.36 getMemoryMapsById() 16
2.1.2.37 getSalas() 16
2.1.2.38 getSourcesMeasures() 16
2.1.2.39 updateChannel() 16
2.1.2.40 updateChannelType() 17
2.1.2.41 updateCodec() 17
2.1.2.42 updateDevice() 18
2.1.2.43 updateMap() 18
2.1.2.44 updateMeasureType() 18
2.1.2.45 updateSala() 19

Índice alfabético **21**

Capítulo 1

Índice de clases

1.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

DAO 3

Capítulo 2

Documentación de las clases

2.1. Referencia de la Clase DAO

Métodos públicos

- __construct (bbdd)
- addCodec (type)
- addChannelType (type)
- addMeasureType (magnitude)
- addMeasuresType (magnitude, units)
- addMemoryMap (type, mapping, version)
- addChannel (type, enabled, properties, name)
- addDevice (id, memoryMap, channel, codec, sala, json)
- addSala (nombre, planta)
- addUser (username, password, name, email, admin, activo, apiKey)
- deleteCodec (id)
- deleteChannelType (id)
- deleteMeasuresType (id)
- deleteMemoryMap (id)
- deleteChannel (id)
- deleteDevice (id)
- deleteMeasure (id)
- deleteMeasureByDate (start, end)
- deleteSala (id)
- getCodecs ()
- getCodecsById (id)
- getChannelTypes ()
- getChannelTypesById (id)
- getMeasuresTypes ()
- getMeasuresTypesById (id)
- getMagnitudes ()
- getMemoryMaps ()
- getMemoryMapsById (id)
- getChannels ()
- getAllFromChannels ()
- getChannelById (id)
- getDevices ()
- getAllDevices ()

- getMeasures ()
- getSalas ()
- getMeasureByDate (start, end, source)
- getMeasureChart (start, end, source, magnitude)
- getSourcesMeasures ()
- updateCodec (id, codec)
- updateChannelType (id, channeltype)
- updateChannel (id, channeltype, enabled, properties, name)
- updateMap (id, type, mapping, version)
- updateMeasureType (id, magnitude, units)
- updateSala (id, nombre, planta)
- updateDevice (id, memoryMap, channel, codec, sala, json)
- cerrarConexion ()

2.1.1. Documentación del constructor y destructor

2.1.1.1. __construct()

`DAO::__construct (bbdd)`

Constructor para inicializar la conexión.

Parámetros

string	bbdd Nombre del tipo de base de datos

2.1.2. Documentación de las funciones miembro

2.1.2.1. addChannel()

`DAO::addChannel (type, enabled, properties, name)`

Añadir un canal.

Parámetros

int	type ID del tipo de canal
boolean	enabled Si está activado o desactivado
json	properties Propiedades del canal en formato JSON
string	name Nombre del canal

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.2. addChannelType()

```
DAO::addChannelType (type)
```

Añadir un tipo de canal.

Parámetros

string	type Nombre del tipo de códec

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.3. addCodec()

```
DAO::addCodec (type)
```

Añadir un nuevo códec.

Parámetros

string	type Nombre del tipo de códec

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.4. addDevice()

```
DAO::addDevice (id, memoryMap, channel, codec, sala, json)
```

Añadir un dispositivo.

Parámetros

string	id ID del dispositivo que, a la vez, es su nombre
boolean	memoryMap ID del mapa de memoria que tiene el dispositivo
int	channel ID del tipo de canal que utiliza el dispositivo
string	codec ID del códec que tiene el dispositivo
string	sala ID de la sala en la que se ubica
string	json propiedades del dispositivo

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.5. addMeasuresType()

```
DAO::addMeasuresType (magnitude, units)
```

Añadir una medida con su respectiva unidad.

Parámetros

string	magnitude Nombre de la magnitud
string	units Nombre de la unidad

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.6. addMeasureType()

```
DAO::addMeasureType (magnitude)
```

Añadir una medida sin unidades.

Parámetros

string	magnitude Nombre de la magnitud

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.7. addMemoryMap()

```
DAO::addMemoryMap (type, mapping, version)
```

Añadir un mapa de memoria.

Parámetros

string	type Tipo de mapa de memoria
string	mapping Datos del mapa de memoria
int	version Versión del mapa de memoria

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.8. addSala()

```
DAO::addSala (nombre, planta)
```

Añadir una sala.

Parámetros

string	nombre Nombre de la sala en la que se sitúa el dispositivo
string	planta Planta en la que se sitúa el dispositivo

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.9. addUser()

```
DAO::addUser (username, password, name, email, admin, activo, apiKey)
```

Añadir un usuario.

Parámetros

string	username Nombre de usuario

Parámetros

string	password Contraseña del usuario
string	name Nombre del usuario
string	email Dirección de correo electrónico del usuario
boolean	admin Si tiene permisos de administrador
boolean	activo Si el usuario está activo
string	apiKey Clave para acceder a la api

Devuelve

false|string thisadd(query) Devuelve el ID de la fila insertada en la tabla o, false en caso de que no se haya podido insertar.

2.1.2.10. cerrarConexion()

```
DAO::cerrarConexion ( )
```

Cierra la conexión con la base de datos.

2.1.2.11. deleteChannel()

```
DAO::deleteChannel (id)
```

Elimina un canal en función del ID.

Parámetros

int	id ID a eliminar en la tabla canales

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.12. deleteChannelType()

```
DAO::deleteChannelType (id)
```

Elimina un tipo de canal en función del ID.

Parámetros

int	id ID a eliminar en la tabla tipo de canal

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.13. deleteCodec()

```
DAO::deleteCodec (id)
```

Elimina un códec en función del ID.

Parámetros

int	id ID a eliminar en la tabla codec

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.14. deleteDevice()

```
DAO::deleteDevice (id)
```

Elimina un dispositivo en función del ID.

Parámetros

int	id ID a eliminar en la tabla dispositivo

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.15. deleteMeasure()

```
DAO::deleteMeasure (id)
```

Elimina una medida en función del ID.

Parámetros

int	id ID a eliminar en la tabla medida

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.16. deleteMeasureByDate()

```
DAO::deleteMeasureByDate (start, end)
```

Elimina una medida en función del intervalo de tiempo.

Parámetros

DateTime	start Fecha a partir de la cual comenzaremos a eliminar medidas
DateTime	end Fecha hasta la que eliminaremos medidas

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.17. deleteMeasuresType()

```
DAO::deleteMeasuresType (id)
```

Elimina un tipo de medida en función del ID.

Parámetros

int	id ID a eliminar en la tabla tipo de medida

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.18. deleteMemoryMap()

```
DAO::deleteMemoryMap (id)
```

Elimina un mapa de memoria en función del ID.

Parámetros

int	id ID a eliminar en la tabla mapa de memoria

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.19. deleteSala()

```
DAO::deleteSala (id)
```

Elimina una sala determinada.

Parámetros

int	id ID a eliminar en la sala

Devuelve

int thisadd(query) Devuelve el ID de la fila eliminada.

2.1.2.20. getAllDevices()

```
DAO::getAllDevices ( )
```

Obtiene toda la información de los dispositivos y, en vez de devolver los ID de las claves foráneas, consulta sus valores.

Devuelve

array|false thisget(query) Devuelve un array con la información de los dispositivos o, false en caso de no haber.

2.1.2.21. getAllFromChannels()

```
DAO::getAllFromChannels ( )
```

Obtiene toda la información de los canales y, en vez de devolver los ID de las claves foráneas, consulta sus valores.

Devuelve

array|false thisget(query) Devuelve un array con la información de los canales o, false en caso de no haber.

2.1.2.22. getChannelById()

```
DAO::getChannelById (id)
```

Obtiene toda la información de un canal mediante el ID.

Parámetros

int	id ID del tipo de canal que queremos la información

Devuelve

array|false thisget(query) Devuelve un array con la información del canal o, false en caso de no haber.

2.1.2.23. getChannels()

```
DAO::getChannels ( )
```

Obtiene toda la información de los canales.

Devuelve

array|false thisget(query) Devuelve un array con la información de los canales o, false en caso de no haber.

2.1.2.24. getChannelTypes()

```
DAO::getChannelTypes ( )
```

Obtiene toda la información de los canales.

Devuelve

array|false thisget(query) Devuelve un array con la información de los tipos de canales o, false en caso de no haber.

2.1.2.25. getChannelTypesById()

```
DAO::getChannelTypesById (id)
```

Obtiene el nombre de un tipo de canal mediante el ID.

Parámetros

int	id ID del tipo de canal que queremos la información

Devuelve

array|false thisget(query) Devuelve un array con la información del tipo de canal o, false en caso de no haber.

2.1.2.26. getCodecs()

```
DAO::getCodecs ( )
```

Obtiene toda la información de los códecs.

Devuelve

array|false this->get(query) Devuelve un array con la información de los códecs o false en caso de no haber.

2.1.2.27. getCodecsById()

```
DAO::getCodecsById (id)
```

Obtiene el tipo de un códec mediante el ID.

Parámetros

int	id ID del códec que queremos la información

Devuelve

array|false thisget(query) Devuelve un array con la información del códec o, false en caso de no haber.

2.1.2.28. getDevices()

```
DAO::getDevices ( )
```

Obtiene toda la información de los dispositivos.

Devuelve

array|false thisget(query) Devuelve un array con la información de los dispositivos o, false en caso de no haber.

2.1.2.29. getMagnitudes()

```
DAO::getMagnitudes ( )
```

Obtiene el ID y la magnitud de los tipos de medidas.

Devuelve

array|false thisget(query) Devuelve un array con la información de los tipos de medidas o, false en caso de no haber.

2.1.2.30. getMeasureByDate()

```
DAO::getMeasureByDate (start, end, source)
```

Obtiene toda la información de medidas en un intervalo de tiempo y de un dispositivo.

Parámetros

DateTime	start Fecha a partir de la cual comenzaremos a consultar medidas
DateTime	end Fecha hasta la que consultaremos medidas
string	source Dispositivo del que se quiere consultar las medidas

Devuelve

array|false thisget(query) Devuelve un array con la información de las medidas o, false en caso de no haber.

2.1.2.31. getMeasureChart()

```
DAO::getMeasureChart (start, end, source, magnitude)
```

Obtiene el valor, la fecha, el dispositivo y tipo de medida de las distintas medidas en un intervalo de tiempo y de un dispositivo.

Parámetros

DateTime	start Fecha a partir de la cual comenzaremos a consultar medidas
DateTime	end Fecha hasta la que consultaremos medidas
string	source Dispositivo del que se van a consultar las medidas
int	magnitude Tipo de medida

Devuelve

array|false thisget(query) Devuelve un array con la información de las medidas o, false en caso de no haber.

2.1.2.32. getMeasures()

```
DAO::getMeasures ( )
```

Obtiene toda la información de las medidas.

Devuelve

array|false thisget(query) Devuelve un array con la información de las medidas o, false en caso de no haber.

2.1.2.33. getMeasuresTypes()

```
DAO::getMeasuresTypes ( )
```

Obtiene toda la información de los tipos de medidas.

Devuelve

array|false thisget(query) Devuelve un array con la información de los tipos de medidas o, false en caso de no haber.

2.1.2.34. getMeasuresTypesById()

```
DAO::getMeasuresTypesById (id)
```

Obtiene toda la información de un tipo de medida mediante el ID.

Parámetros

int	id ID del tipo de medida que queremos la información

Devuelve

array|false thisget(query) Devuelve un array con la información del tipo de medida o, false en caso de no haber.

2.1.2.35. getMemoryMaps()

```
DAO::getMemoryMaps ( )
```

Obtiene toda la información de los mapas de memorias.

Devuelve

array|false thisget(query) Devuelve un array con la información de los mapas de memoria o, false en caso de no haber.

2.1.2.36. getMemoryMapsById()

```
DAO::getMemoryMapsById (id)
```

Obtiene toda la información de un mapa de memoria mediante el ID.

Parámetros

int	id ID del tipo de medida que queremos la información

Devuelve

array|false thisget(query) Devuelve un array con la información de los mapas de memoria o, false en caso de no haber.

2.1.2.37. getSalas()

```
DAO::getSalas ( )
```

Obtiene toda la información de las salas.

Devuelve

array|false thisget(query) Devuelve un array con la información de las medidas o, false en caso de no haber.

2.1.2.38. getSourcesMeasures()

```
DAO::getSourcesMeasures ( )
```

Obtiene los nombres de dispositivos de la tabla medidas.

Devuelve

array|false thisget(query) Devuelve un array con la información de los dispositivos de las medidas o, false en caso de no haber.

2.1.2.39. updateChannel()

```
DAO::updateChannel (id, channeltype, enabled, properties, name)
```

Modificar un canal.

Parámetros

int	id ID del canal a modificar
int	channeltype ID del tipo de canal
boolean	enabled Si está activado o desactivado
json	properties Propiedades del canal en formato JSON
string	name Nombre del canal

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.40. updateChannelType()

```
DAO::updateChannelType (id, channeltype)
```

Modificar un tipo de canal.

Parámetros

int	id Id del tipo de canal a modificar
string	type Nombre del tipo de canal

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.41. updateCodec()

```
DAO::updateCodec (id, codec)
```

Modificar un códec.

Parámetros

int	id ID del códec a modificar
string	type Nombre del tipo de códec

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.42. updateDevice()

```
DAO::updateDevice (id, memoryMap, channel, codec, sala, json)
```

Modificar un dispositivo.

Parámetros

string	id ID del dispositivo que a la vez es su nombre
boolean	memoryMap ID del mapa de memoria que tiene el dispositivo
int	channel ID del tipo de canal que utiliza el dispositivo
string	codec ID del códec que tiene el dispositivo
string	sala ID de la sala en la que se ubica
string	json propiedades del dispositivo

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.43. updateMap()

```
DAO::updateMap (id, type, mapping, version)
```

Modificar un mapa de memoria.

Parámetros

int	id ID del mapa a modificar
string	type Tipo de mapa de memoria
string	mapping Datos del mapa de memoria
int	version Versión del mapa de memoria

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.44. updateMeasureType()

```
DAO::updateMeasureType (id, magnitude, version)
```

Modificar un tipo de medida.

Parámetros

int	id ID del mapa a modificar
string	magnitude Nombre de la magnitud
string	units Nombre de la unidad

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

2.1.2.45. updateSala()

```
DAO::updateSala (id, nombre, planta)
```

Modificar una sala.

Parámetros

int	id ID del mapa a modificar
string	nombre Nombre de la sala en la que se sitúa el dispositivo
string	planta Planta en la que se sitúa el dispositivo

Devuelve

false|int this->update(query) Devuelve el ID de la fila modificada en la tabla o, false en caso de que no se haya podido modificar.

La documentación para esta clase fue generada a partir del siguiente fichero:

- dao.php

Índice alfabético

__construct
DAO, 4

addChannel
DAO, 4
addChannelType
DAO, 5
addCodec
DAO, 5
addDevice
DAO, 5
addMeasuresType
DAO, 6
addMeasureType
DAO, 6
addMemoryMap
DAO, 7
addSala
DAO, 7
addUser
DAO, 7

cerrarConexion
DAO, 8

DAO, 3
__construct, 4
addChannel, 4
addChannelType, 5
addCodec, 5
addDevice, 5
addMeasuresType, 6
addMeasureType, 6
addMemoryMap, 7
addSala, 7
addUser, 7
cerrarConexion, 8
deleteChannel, 8
deleteChannelType, 8
deleteCodec, 9
deleteDevice, 9
deleteMeasure, 9
deleteMeasureByDate, 10
deleteMeasuresType, 10
deleteMemoryMap, 10
deleteSala, 11
getAllDevices, 11
getAllFromChannels, 11
getChannelById, 11
getChannels, 12
getChannelTypes, 12
getChannelTypesById, 12
getCodecs, 13
getCodecsById, 13
getDevices, 13
getMagnitudes, 13
getMeasureByDate, 14
getMeasureChart, 14
getMeasures, 15
getMeasuresTypes, 15
getMeasuresTypesById, 15
getMemoryMaps, 15
getMemoryMapsById, 16
getSalas, 16
getSourcesMeasures, 16
updateChannel, 16
updateChannelType, 17
updateCodec, 17
updateDevice, 18
updateMap, 18
updateMeasureType, 18
updateSala, 19
deleteChannel
DAO, 8
deleteChannelType
DAO, 8
deleteCodec
DAO, 9
deleteDevice
DAO, 9
deleteMeasure
DAO, 9
deleteMeasureByDate
DAO, 10
deleteMeasuresType
DAO, 10
deleteMemoryMap
DAO, 10
deleteSala
DAO, 11

getAllDevices
DAO, 11
getAllFromChannels
DAO, 11
getChannelById
DAO, 11
getChannels
DAO, 12

getChannelTypes
DAO, 12
getChannelTypesById
DAO, 12
getCodecs
DAO, 13
getCodecsById
DAO, 13
getDevices
DAO, 13
getMagnitudes
DAO, 13
getMeasureByDate
DAO, 14
getMeasureChart
DAO, 14
getMeasures
DAO, 15
getMeasuresTypes
DAO, 15
getMeasuresTypesById
DAO, 15
getMemoryMaps
DAO, 15
getMemoryMapsById
DAO, 16
getSalas
DAO, 16
getSourcesMeasures
DAO, 16

updateChannel
DAO, 16
updateChannelType
DAO, 17
updateCodec
DAO, 17
updateDevice
DAO, 18
updateMap
DAO, 18
updateMeasureType
DAO, 18
updateSala
DAO, 19