ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

“DISEÑO E IMPLEMENTACIÓN DEL CATÁLOGO DE INFRAESTRUCTURA DE LA GERENCIA DE INFRAESTRUCTURA FÍSICA DE LA ESPOL”

INFORME DE MATERIA INTEGRADORA

Previo a la obtención del Título de:

INGENIERO EN COMPUTACIÓN

HÉCTOR FRANKLIN JÚPITER QUINDE

BAKKE ANDRÉS MEDINA ABARCA

GUAYAQUIL – ECUADOR

AÑO: 2017

AGRADECIMIENTOS

En esta obra quiero agradecer en primer lugar a mi abuela que en paz descansa, mi madre, hermanos y demás familiares por todos los consejos que me dieron, para continuar adelante con mis metas y mis sueños.

Quiero agradecer a mi tío Robert Roybal Montoya y mi tía María Luisa Abarca de Roybal por ser pilares fundamentales durante toda mi carrera, que me dieron todo su apoyo y esfuerzo para que yo como persona salga a la sociedad como un profesional responsable y de bien.

Agradezco también a mis profesores especialmente al nuestro tutor PhD. Luis Eduardo Mendoza siempre estuvo allí, para este sueño se cumpla.

Bakke Medina

En primer lugar, a Dios por haberme guiado por el camino de la felicidad hasta ahora, en segundo lugar, a cada uno de los que son parte de mi familia a mi padre Ángel Júpiter, mi madre Juana Quinde, mi segunda madre mi abuela que en paz descanse, a mis hermanos, a todos mis tíos y a mi enamorada Juliana Baquerizo por siempre haberme dado su fuerza y apoyo incondicional que me han ayudado y llevado hasta donde estoy ahora.

Por último a mis profesores a quienes les debo gran parte de mis conocimientos, gracias a su paciencia y enseñanza, a mi compañero de tesis porque en esta armonía grupal lo hemos logrado y finalmente a nuestro tutor quien nos ayudó en todo momento, PhD. Luis Eduardo Mendoza.

Héctor Júpiter

DEDICATORIA

El presente proyecto lo dedico a mis ángeles en el cielo que siempre están conmigo a lo largo de mi vida y carrera profesional.

Bakke Medina

A mi madre con mucho amor y cariño le dedico todo mi esfuerzo y trabajo puesto para la realización de este proyecto.

Héctor Júpiter

TRIBUNAL DE EVALUACIÓN

PhD. LUIS MENDOZA MORALES
PROFESOR EVALUADOR

PhD. CARLOS MONSALVE ARTEAGA
PROFESOR EVALUADOR

DECLARACIÓN EXPRESA

"La responsabilidad y la autoría del contenido de este Trabajo de Titulación, nos corresponde exclusivamente; y damos nuestro consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual"

.....

Héctor Júpiter Quinde

.....

Bakke Medina Abarca

RESUMEN

A lo largo de los años, la infraestructura física de la Escuela Superior Politécnica del Litoral (ESPOL) ha ido creciendo contando con nuevas estructuras físicas, renovación de componentes y puestos de trabajo de los profesores, entre otros, que en conjunto representan una masiva cantidad de información importante para la institución, que no es mantenida de manera correcta.

La finalidad de este proyecto fue desarrollar un sistema web que automatiza la administración de la información de la infraestructura física de la ESPOL, categorizando los componentes para proporcionar una consistencia de la información dentro de un catálogo. Mediante una interfaz web, el sistema es accesible para los trabajadores de la ESPOL. Además, sigue los estándares y servicios proporcionado por la Gerencia de Tecnología y Sistemas de Información. El sistema fue implementado con los Framework Angular 2 y Nodejs.

Este proyecto fue llevado a cabo en conjunto con la Gerencia de Infraestructura Física de la ESPOL, estableciendo la configuración del catálogo de la forma en que están organizados los componentes y los espacios físicos de la ESPOL.

ÍNDICE GENERAL

CAPÍTULO 1

1. SITUACIÓN ESPECÍFICA DEL PROBLEMA A RESOLVER.

En el marco del crecimiento de las organizaciones, se debe considerar como un aspecto relevante el detalle de la infraestructura física de una empresa.

El catálogo e inventario de edificios, comedores, aulas, oficinas, con sus respectivos mobiliarios y sus componentes, que facilite su ubicación y la toma de decisiones ante una posible remodelación, cambio de lugar o construcción de un nuevo edificio, es importante para una organización cuya actividad depende de la infraestructura física.

El catálogo de la infraestructura consta de una cantidad masiva de datos que a lo largo de los años ha ido creciendo dentro de la Escuela Superior Politécnica del Litoral (ESPOL) al contar con nuevas estructuras físicas, renovación de componentes, puestos de trabajos de los profesores, que en conjunto representan una masiva cantidad de información importante la cual no es mantenida de manera correcta.

1.1 Situación actual de la infraestructura física de la ESPOL.

La ESPOL es una instalación de educación superior que posee diversos campus con muchas edificaciones o instalaciones físicas, en donde cada una de ellas contiene salas que pueden ser aulas, laboratorios, auditorios, baños, oficinas de profesores, salas especiales y salas de eventos múltiples. Cada una de estas edificaciones no posee un control en el registro de sus componentes, así como de la estructura física en cada edificio, aula, planta, así como la categorización de sus componentes.

La ESPOL actualmente no mantiene un registro de cómo está conformada cada edificación. No cuenta con una sistematización de la información que permita facilitar la tarea de obtención de datos ni de poder gestionar la información de los edificios nuevos ni que es lo que contiene cada uno de ellos.

Actualmente la ESPOL mantiene su inventario de infraestructura física mediante el uso de las herramientas de Office como Excel, que sirven de ayuda al personal administrativo en la categorización de todos los componentes existentes dentro de las salas, laboratorios, cubículos y pisos.

1.2 Soluciones similares

Se pueden encontrar que alrededor del mundo se han desarrollado diversos sistemas web tipo Enterprise Resource Planning (ERP) para el manejo de inventario, tanto de libre uso como de licenciamiento, o mejor conocidos como de pago, que contienen módulos más complejos que permiten obtener un mejor funcionamiento, basándose en los requerimientos y peticiones de los usuarios. Entre ellos nos podemos encontrar con: Pulpo WMS, Oddo, PMI, ECAF.

Pulpo WMS. Es un sistema integrado en la nube que permite la gestión de productos físicos dentro de un almacén para el control de su inventario. También permite tener un inventario virtual usando una Oculus VR y control de bluetooth; también permite probar ciertos algoritmos hechos por terceros y así beneficiar al sistema de inventario [1].

Oddo. Es un modelo de código abierto para las diferentes integraciones al mundo del ERP, que abstrae toda la gestión de proyectos, gestión de almacenes, manufactura, gestión de activos, campañas de marketing, entre otros, que no requiere algún tipo de paga ya que es totalmente gratuito en versiones para la comunidad [2].

PMI. Es un software para el mantenimiento y gestión de activos, el cual proporciona el soporte necesario para la gestión, monitoreo y control del mantenimiento preventivo, correctivo y predictivo de los activos e infraestructura con el objetivo de optimizar los costos de mantenimiento, aumentar la producción y la vida útil [3].

ECAF. Software encargado de la gestión y control de activos fijos y bienes de control. Este software es multiplataforma con lo cual se garantiza que funcionará en cualquier tipo de plataforma, incluso para dispositivos móviles, flexible lo cual permitirá un crecimiento en un futuro [4].

Si bien los ERP antes mencionados ayudan de alguna forma a tener un control y un inventario, están orientados al manejo de inventarios de elementos físicos, más no del inventario de infraestructura física. Por ello, la decisión de desarrollar el proyecto e implementar algún tipo de solución informática que permita la catalogación de toda la infraestructura bajo los estándares de desarrollo y de la

infraestructura tecnológica de la Gerencia de Tecnología y Sistemas de la Información (GTSI) de la ESPOL.

Si bien la propuesta desarrollada a través de esta Materia Integradora no resuelve del todo lo que se requiere a nivel de reportes o identificación de los componentes que pueden estar asociados a la infraestructura física de la ESPOL, el sistema ha sido preparado para la futura integración con el sistema de manejo de Activos Fijos de la ESPOL. Por ello, se deja abierto para una posible ampliación del sistema y elaboración de reportes, así como para la integración de otros sistemas pertenecientes a la institución.

CAPÍTULO 2

2. ANÁLISIS Y DISEÑO DE LA SOLUCIÓN PARA EL SISTEMA DE LA INFRAESTRUCTURA FÍSICA

La solución para el problema que se ha encontrado, se basa en el desarrollo de un sistema que agilice los procedimientos de registro de la infraestructura física, así como la categorización de sus respectivos componentes, como son: la iluminación, mobiliario y equipos, entre otros, para brindar así un soporte más detallado con información específica de los edificios dentro de un campus. Con este sistema se pretende solucionar el problema de la distribución y administración de la infraestructura física que tiene la ESPOL.

Para implementar la solución decidimos utilizar el Framework de Express.js y Node.js porque nos proveen de servicios, controladores y rutas para el manejo del servidor en conjunto con Angular 2. Este último nos ayuda en el manejo del Document Object Model (DOM) y mediante su API nos facilita la conexión con Express.js y Node.js a consumir el API REST.

2.1 Objetivos del desarrollo del sistema web

2.1.1 Objetivos generales

Diseñar e implementar un sistema web para registrar, automatizar, administrar y visualizar la información de la infraestructura física de la ESPOL y el contenido de sus edificaciones, categorizando los componentes para proporcionar una consistencia de la información dentro de un catálogo.

Automatizar la catalogación de las estructuras físicas de la ESPOL y de sus componentes de forma descentralizada, para agilizar la búsqueda e ingreso de información a través de una aplicación web, identificando los roles de administrador y administrativo.

2.1.2 Objetivos específicos

- Construir un sistema web, para la administración de la información de las edificaciones y sus detalles específicos.
- Catalogar los distintos componentes que pueden estar en una infraestructura física y sus características.
- Seleccionar las solicitudes o cambios para nuevos componentes o elementos de infraestructura.
- Registrar los usuarios del sistema.
- Establecer una estructura de datos para los componentes de la ESPOL.

2.2 Metodología para el sistema de infraestructura

2.2.1 Diagrama de casos de uso

Debido a la cantidad de casos de uso que se lograron obtener, se optó por elaborar un diagrama de paquetes que agrupa las funcionalidades por los roles más importantes, con el objetivo de que se cubra el contenido general del sistema y así poder mostrar el uso de la aplicación de una mejor manera.

El rol de administrador tiene como principal tarea la de crear los distintos elementos del catálogo de la infraestructura: campus, tipos de salas, componentes y sus características. Además, la aprobación y rechazo de la creación de nuevos elementos para el catálogo, proveniente de solicitudes hechas por los administrativos. El rol del administrativo se encarga de generar la conformación de los distintos tipos de edificaciones tomando como base los elementos ya creados en el catálogo, y administrados de forma descentralizada por las distintas unidades académicas dentro de la ESPOL.

En estos diagramas (ver Figuras 2.1 a 2.3) se han identificado 6 actores, los cuales son:

- CAS
    - Servicio de autenticación de usuario de la ESPOL.
- Trabajador ESPOL
    - Actor padre del Administrador y el Administrativo, donde adquieren las mismas características pertenecientes a los trabajadores dentro de la ESPOL.
- CargoDedicacion WebService
    - Servicio web proporcionado por GTSI para la obtención de los roles para la distinción del usuario dentro del sistema.
- Academico WebService
    - Servicio web proporcionado por GTSI para la obtención de nombre y apellidos de los trabajadores de la ESPOL.
- Administrador o Admin
    - Actor que crea el catálogo dentro el sistema.
- Administrativo(Custodio)
    - Actor que crea la infraestructura física dentro de la ESPOL.

Paquetes:

- Gestión de Catálogo
    - Casos de usos de la administración del catálogo, del usuario y de las solicitudes, por parte del administrador.
- Gestión de Infraestructura
    - Casos de usos de la creación de la infraestructura física mediante el usuario administrativo.

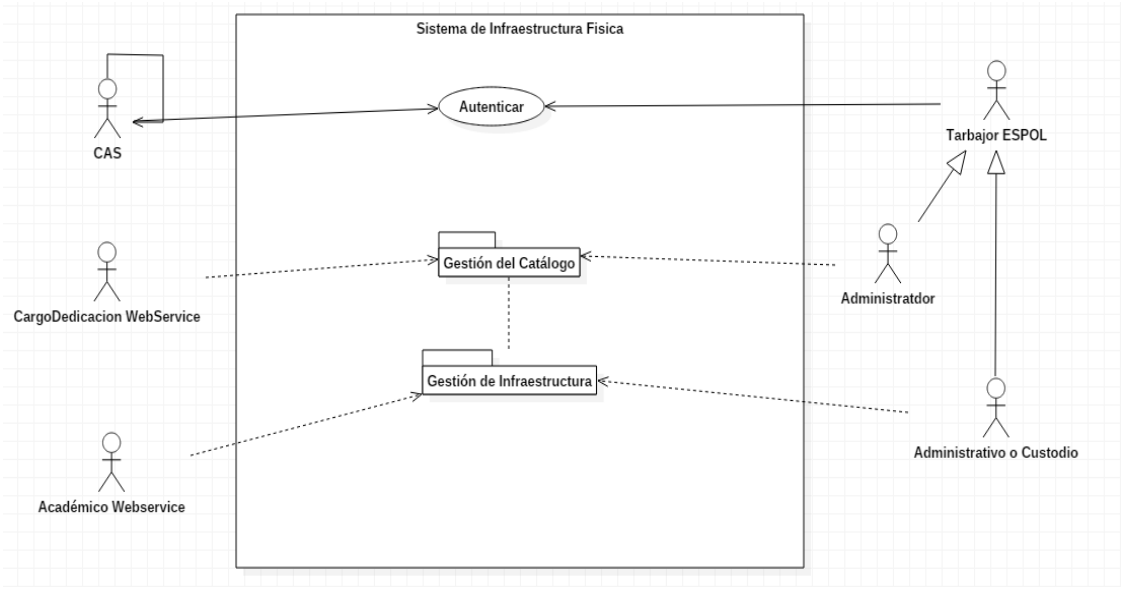

Figura 2.1: Diagrama del sistema de infraestructura física.

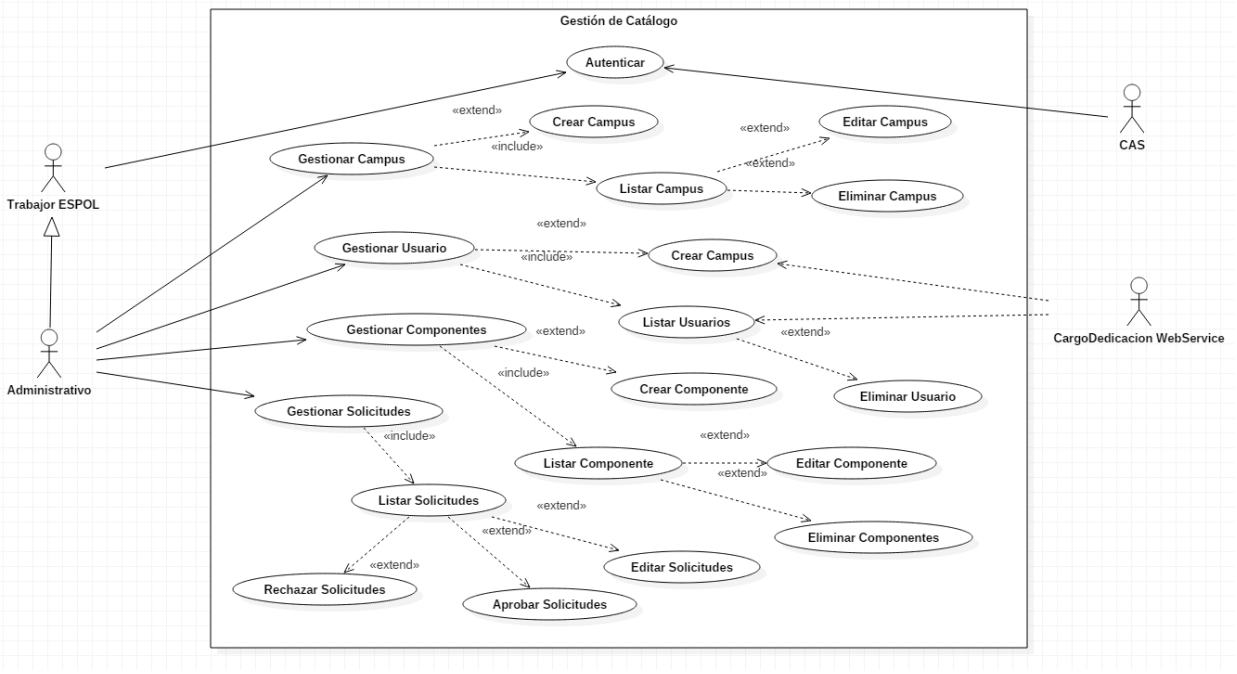

Figura 2.2: Diagrama de casos de uso de la gestión de catálogo.

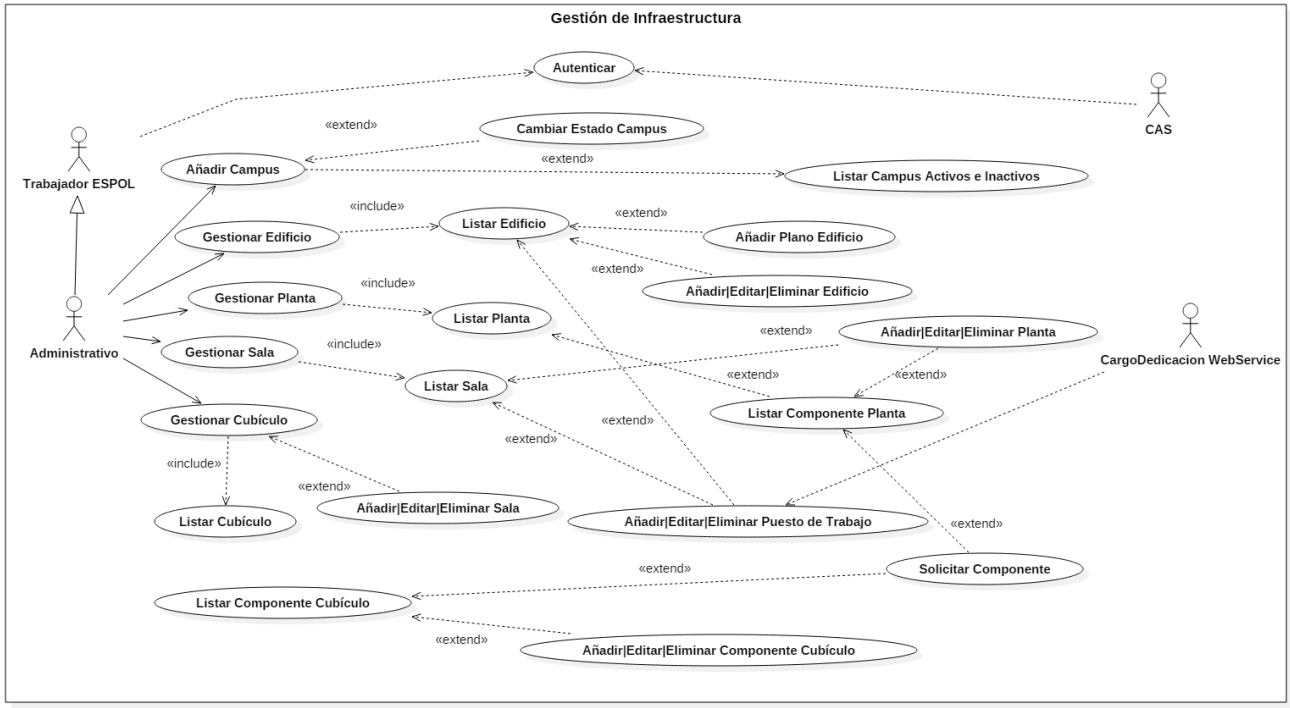

Figura 2.3: Diagrama de casos de uso de la gestión de infraestructura.

2.3 Estructura de la base de datos

La base de datos fue diseñada para su correcto funcionamiento en el ambiente Oracle, puesto que es uno de los requisitos para el de desarrollo del sistema por parte de la institución. Se encontraron algunas dificultades al momento de diseñar la base de datos entre ellas tenemos:

- La creación o petición de los respectivos componentes por parte de los usuarios dentro del sistema, la cual se solucionó otorgando un estado de confirmación para los componentes que se requerían, denotando con un “0” a una petición pendiente y “1” a una petición aceptada. Esta implementación se realizó para no crear una nueva tabla denominada peticiones que iba a contener los mismos campos que tiene un componente con la única diferencia que este tenía el campo extra de descripción o motivo por el cual se hacia la petición. Dicho campo fue incluido en la tabla actual de componente.

- Una de las complicaciones más destacadas dentro de la elaboración de la base de datos fue la declaración de la tabla componente, puesto que esta misma se iba a usar en todas las edificaciones que lo necesitaban; además, es una tabla importante para la parte de activo fijo, con lo cual hemos añadido una tabla general de componente y una tabla intermedia que ayude en la creación de los múltiples componentes dentro de la infraestructura. La tabla general contendrá las cualidades básicas de los componentes, el tipo al que pertenecen, su característica, el nombre de tal forma que pueda ser usado donde se lo requiera. La tabla intermedia, por otra parte, contendrá las referencias de las edificaciones en las que vayan a ser añadidas. Además, contendrá el campo del código de activo fijo el cual servirá para una futura integración con activo fijo; esto ayudará a mantener una contabilidad de los componentes que se encuentran dentro de la ESPOL.

El modelo entidad relación que se obtuvo al finalizar el levantamiento de los requerimientos del cliente se muestra en la Figura 2.4.

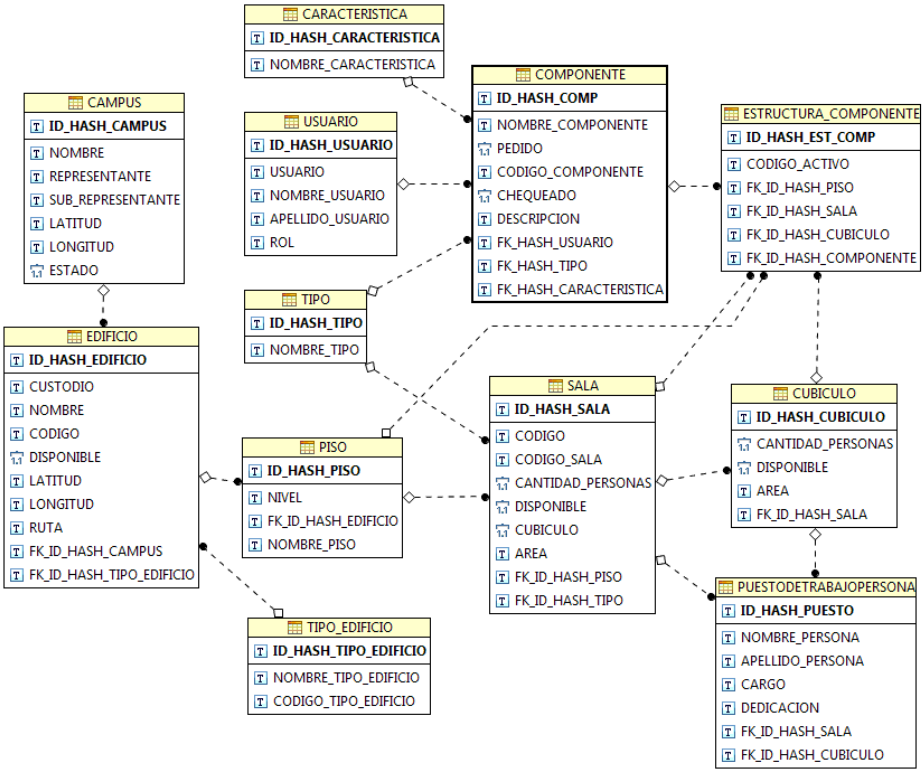

Figura 2.4: Diseño Entidad Relación de la base de datos.

2.4 Tecnologías y componentes utilizados

Para el desarrollo del sistema para la catalogación de la infraestructura física se usaron las siguientes tecnologías: NodeJs, Express, Oracle DataBase 11g y Angular 2.

NodeJS: Es un Framework para la capa del servidor, usando JavaScript como lenguaje base y compiladores de C++, el cual se usa para optimizar de mejor manera (ver Figura 2.5) el uso de callbacks. Dicho Framework es asíncrono y de arquitectura orientada a eventos lo cual es altamente escalable para los servidores web que hoy en día se utilizan para los envíos de peticiones por usuario [4].

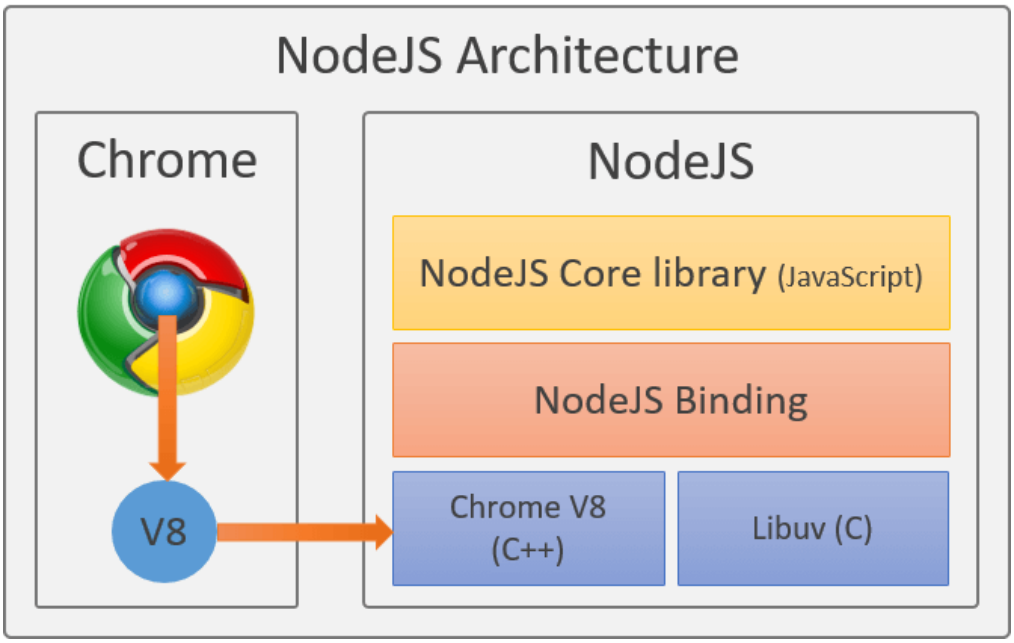

Figura 2.5: Arquitectura de NodeJS, en donde se comunica con el motor V8 para él envió de datos, usando el compilador de C++ [4].

Express: Es un Framework que trabaja junto con NodeJS para proporcionar extensiones o middleware al Framework como dar servicio HTTP, manejador de sesión, cookieParser y más. Estos middlewares se conectan con app.use (). Estas son algunas tareas que se pueden hacer con Express las cuales son Routing con las URL, gestionar las sesiones usando las cookies y analizar los

tipos de solicitudes mediante todo tipo de datos y el manejo de errores con las solicitudes no autorizadas (ver Figura 2.6) [4].

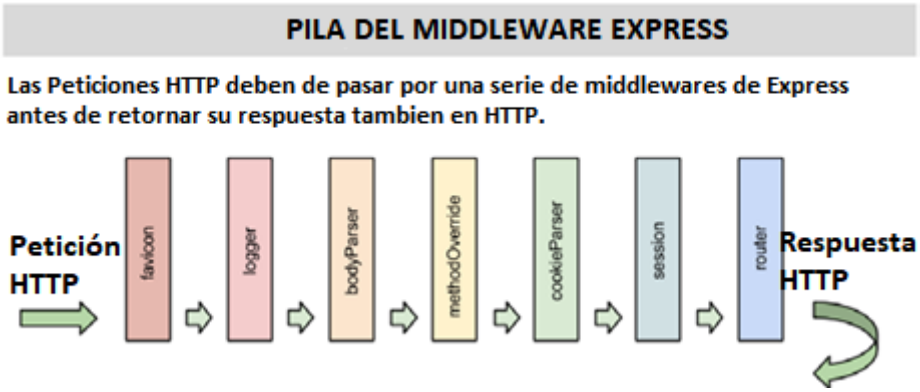

Figura 2.6: Funcionamiento del Framework Express [5].

Oracle Data base Express: Es una base de datos gratuita para desarrollar aplicaciones web con Node.js, Python, PHP, Java, .NET, el cual brinda a los diferentes tipos de aplicaciones la facilidad de información y utilización mediante la basta documentación que ellos poseen para solucionar problemas de TI.

En nuestro caso, se usará el Oracle DataBase 11g Version2 para las pruebas locales junto a NodeJS versiones superiores a 4, el cual es estable para dichas versiones. [6]

En la Figura 2.7 se muestra la conexión de NodeJS con Oracle, en la cual mostramos una función llamada ejecutarQuery, esta función se ejecuta el query o procedure para, que nos devuelva la respuesta almacenada en la base del Oracle [6].

```
exports.ejecutarQuery = function (query, req, res){
    oracledb.getConnection(connAttrs, function (err, connection) {
        if (err) {
                // Error connecting to DB
                res.set('Content-Type', 'application/json');
                res.status(500).send(JSON.stringify({
                        status: 500,
                        message: "Error connecting to DB",
                        detailed_message: err.message
                }));
                return;
        }
        connection.execute(query, {}, {
                outFormat: oracledb.OBJECT // Return the result as Object
        }, function (err, result) {
                if (err) {
                        res.set('Content-Type', 'application/json');
                        res.status(500).send(JSON.stringify({
                                status: 500,
                                message: "Error getting campus",
                                detailed_message: err.message
                        }));
                } else {
                        res.contentType('application/json').status(200);
                        console.log(result);
                        res.send(JSON.stringify(result.rows));
                }
                // Release the connection
        connection.release(
            function (err) {
                    if (err) {
                            console.error(err.message);
                    } else {
                            console.log("GET /campus : Connection released");
                    }
            });
        });
    });
}
```

Figura 2.7: Conexión Oracle.

Angular 2: Es un módulo escrito en JavaScript basado en AngularJS o Angular 1. Con la ayuda de los desarrolladores de Google se creó Angular 2. Sirve para la creación de páginas web complejas ayudando con una sintaxis completamente diferente a la de AngularJS, usando como código el TypeScript para las operaciones más complejas, como servicios, componentes, pero dando una mejora sustancial al desarrollo web y reduciendo las compatibilidades con los navegadores existentes [7].

Angular 2 utiliza componentes, componentes de comunicación, plantilla, metadatos, enlace de datos y servicios [7].

2.5 Módulos del framework nodejs junto con el framework angular 2

NodeJS es un framework para back-end, mientras que Angular 2 es un framework para front-end, los cuales se combinan para dar un esquema MVC en ambas partes, dando así un ambiente de desarrollo más limpio, optimizando la carga que recibe el usuario en el navegador. En la Figura 2.8 se puede apreciar como Angular 2 hace llamadas al API REST de NodeJS, para esta pueda hacer las consultas en la base de datos de Oracle [7].

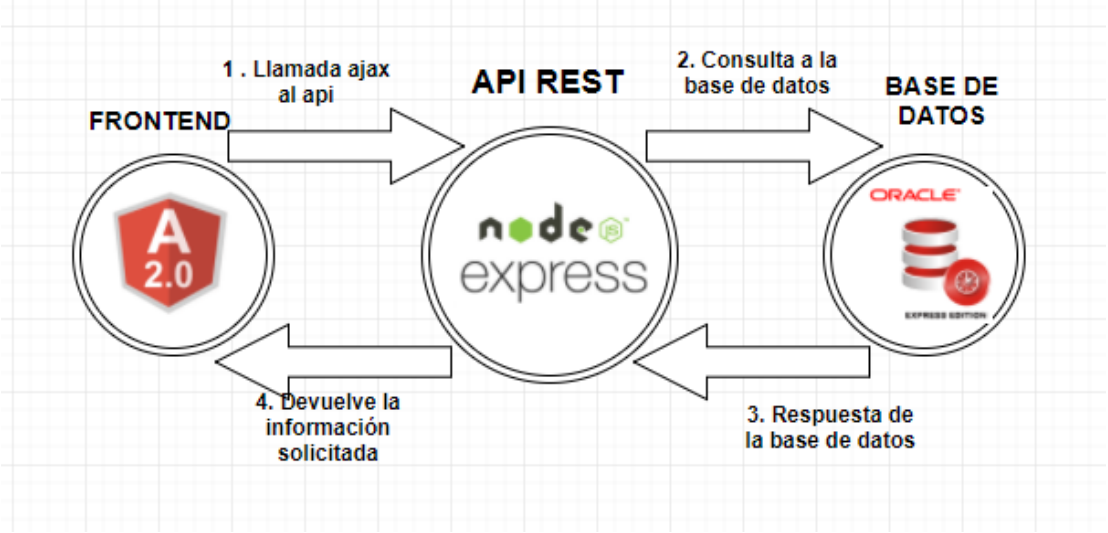

Figura 2.8: Funcionamiento de Angular 2 junto NodeJS y Oracle.

CAPÍTULO 3

3. IMPLEMENTACIÓN PARA EL SISTEMA DE INFRAESTRUCTURA FÍSICA

La implementación del Sistema de Infraestructura Física consiste en establecer una interacción con los distintos tipos de usuarios, en este caso serían el administrativo (custodio) y el administrador, usando el Central Autenthication Service (CAS), que se usará para el ingreso al sistema, (ver Figura 3.1), el cual nos devolverá, si se ingresa con éxito, el nombre del usuario.

Figura 3.1: Página de inicio del CAS uso exclusivo de la ESPOL.

EL nombre del usuario que provee el CAS es insuficiente para la obtención de más información personal, y fue necesario pedir a la Gerencia de Tecnología y Servicios de la Información (GTSI) de la ESPOL un servicio web que nos proporcione dicha información.

La GTSI nos brindó dos servicios web, el primero es para acceder a la información más detallada (https://ws.espol.edu.ec/saac/wsSODIS.asmx?WSDL) y el segundo (http://ws.espol.edu.ec/saac/Catalogo.asmx?WSDL) para hallar el cargo y dedicación del personal de la ESPOL cuando se cree un nuevo puesto de trabajo dentro de la sala o cubículo.

Con esto se plantearon muchas dudas acerca de cómo se maneja el CAS para el ingreso al sistema junto con Angular2, pero se las solucionó gracias a la cas-client librería externa para NodeJS [8].

Pudimos crear un esquema de paquetes para el mejor entendimiento de cómo están siendo utilizados los componentes junto con las herramientas ya descritas en el capítulo anterior.

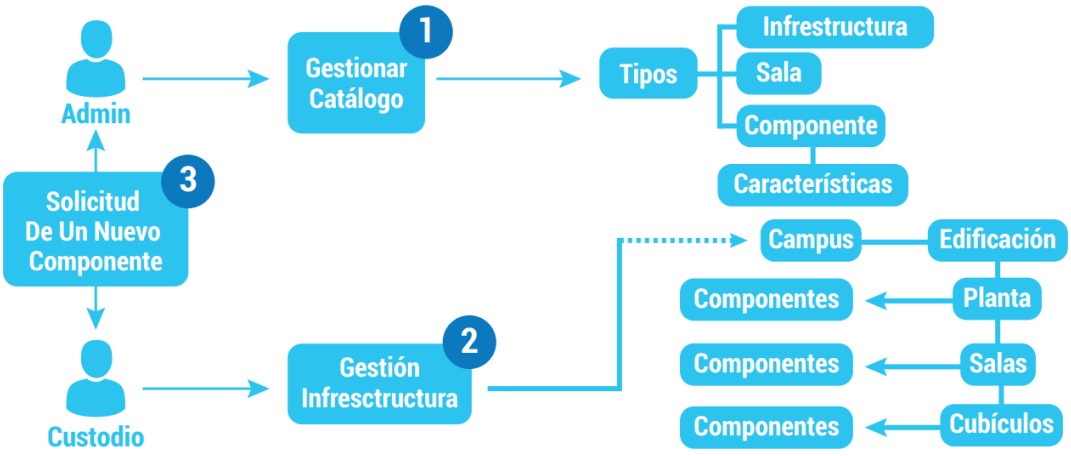

Figura 3.2: Esquema del modelo del funcionamiento del sistema de infraestructura física.

En la Figura 3.2 se puede apreciar la metodología de la infraestructura física en tres partes:

1) Gestión del catálogo con los tipos de infraestructura, sala, componente y característica.

2) Gestión de la Infraestructura con la creación de toda la infraestructura física dentro de la unidad académica como sus componentes.

3) Solicitud de un nuevo componente para el envío de solicitudes, que facilita al administrador la aprobación o rechazo, dependiendo si es un componente válido para el sistema y para el catálogo.

3.1 Vistas o componentes del administrador

Para el administrador las vistas más importantes para describirlas son: el Catálogo y las Peticiones.

Cada uno de estos componentes tiene su propio Hyper Text Markup Language (HTML) que es la estructura en etiquetas junto con el Cascading Style Sheets (CSS) que es el estilo de la página y el TypeScripts (TS) que es la funcionalidad para interactuar con dicho componente.

3.1.1 Catálogo

Para el desarrollo del catálogo creamos tres secciones que son tipo de componente para la edificación, tipo de características y nuevo componente.

3.1.1.1 Tipo de componente para la edificación

Esta sección del catálogo está destinada al tipo de edificación (ver Figura 3.3).

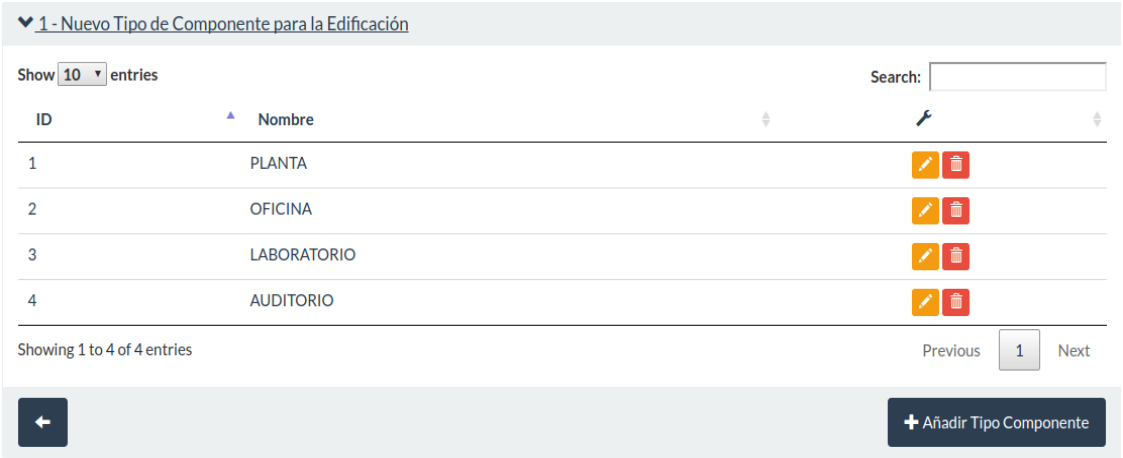

Figura 3.3: Vista de añadir el tipo de componente.

3.1.1.2 Tipo de característica para el componente

En esta sección del desarrollo, los componentes tienen un tipo de característica lo cual se complicó al principio, porque existen componentes que no poseen una característica pero otras sí, por lo que se vio importante definir la característica llamada “NO APLICA” (ver Figura 3.4).

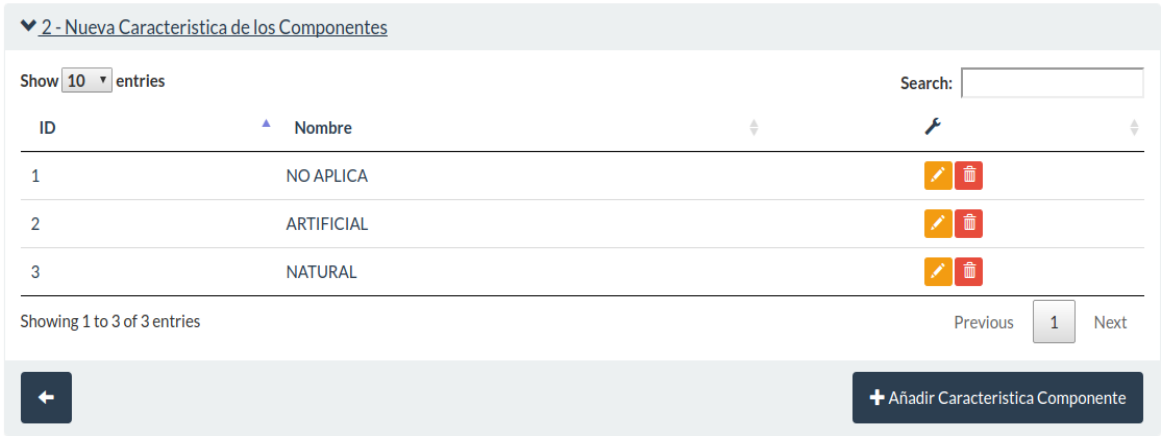

Figura 3.4: Vista del Añadir una nueva Característica al Componente.

3.1.1.3 Nuevo componente

En esta sección del desarrollo, se crea el nuevo componente asignándole su característica y su tipo (ver Figura 3.5).

Figura 3.5: Vista para crear un nuevo componente.

3.1.2 Peticiones

Vista que visualiza todas las peticiones hechas por los usuarios administrativos o custodios de nuevos componentes que no han sido creados por el administrador, en donde pueden editarse antes de su aprobación o simplemente ser rechazadas (ver Figura 3.6).

Figura 3.6: Vista las peticiones hechas por el custodio para aprobarlas o rechazarlas.

3.2 Vistas o componentes del administrativo (custodio)

Para el usuario administrativo o custodio, se describirá en las siguientes secciones los problemas u observaciones más relevantes obtenidas:

- Campus
- Edificios
- Sala y cubículo

3.2.1 Campus

La funcionalidad principal de esta vista es habilitar un campus creado por parte del administrador dentro del sistema, en donde luego será añadida su respectiva infraestructura; sin embargo, se encontró un problema al tratar de asignar un campus a un administrativo sin que este se repita. Para esto se asignó un estado al campus para habilitarlo; el botón para Añadir Campus se deshabilita o habilita dependiendo del estado en que se encuentre, con el fin de que el usuario administrativo no pueda asignar dos veces el mismo campus (ver Figura 3.7).

Figura 3.7: Vista para añadir una estructura previamente creada por el administrador mas no podrá editarla.

3.2.2 Edificios

En la vista del edificio el problema que hubo fue con la subida de archivos AutoCAD, puesto que son los planos de los edificios existentes dentro de la ESPOL. Por ello, se hicieron tres funciones para el trabajo de subir archivo, crear la ruta del archivo y la descarga del plano que son SubirArchivoAutocad, CrearRutaPlano, DescargaPlano. Estas funciones se encuentran en el lado de server en controladores con el nombre edificio.server.controller.js (ver Figura 3.8).

```
exports.SubirArchivoAutocad = function(req, res, next){
    res.json(req.files.map(file => {
    return {
        originalName: file.originalname,
        filename: file.filename
      }
  }));
}

exports.CrearRutaPlano = function(req, res,next){
    "use strict";
    console.log(req.body);
    var query = "UPDATE EDIFICIO SET  "+
                "EDIFICIO.RUTA = :RUTA "+
                "WHERE EDIFICIO.ID_HASH_EDIFICIO = :ID_HASH_EDIFICIO";//query o procedure
    var queryValue = [req.body.RUTA, req.body.ID_HASH_EDIFICIO];
    oracleModelFunction.ejecutarQueryParam(query, queryValue, req, res);
}

exports.DescargarPlano = function(req, res, next){
    res.download("./uploads/"+req.carPoolSession.username+"/edificios/"+req.params.RUTA,req.params.RUTA);
}
```

Figura 3.8: Funciones para la subida, descarga y creación de ruta del archivo AutoCAD.

3.2.3 Sala y cubículos

En la vista de sala y cubículo se presentó la dificultad de añadir un usuario al puesto de trabajo, lo que obstaculizó conocer si el usuario ya estaba ocupando ese puesto de trabajo, o si se repetía el usuario (ver Figura 3.9).

Para solucionar esta situación, se verificó en cada sala y cubículo con la función “VerificarPuestoTrabajo” (que recibe el usuario y nos retorna un 0 si esa persona no ha sido asignada a un puesto de trabajo y un 1 si esa persona ya está asignada), ver Figura 3.10.

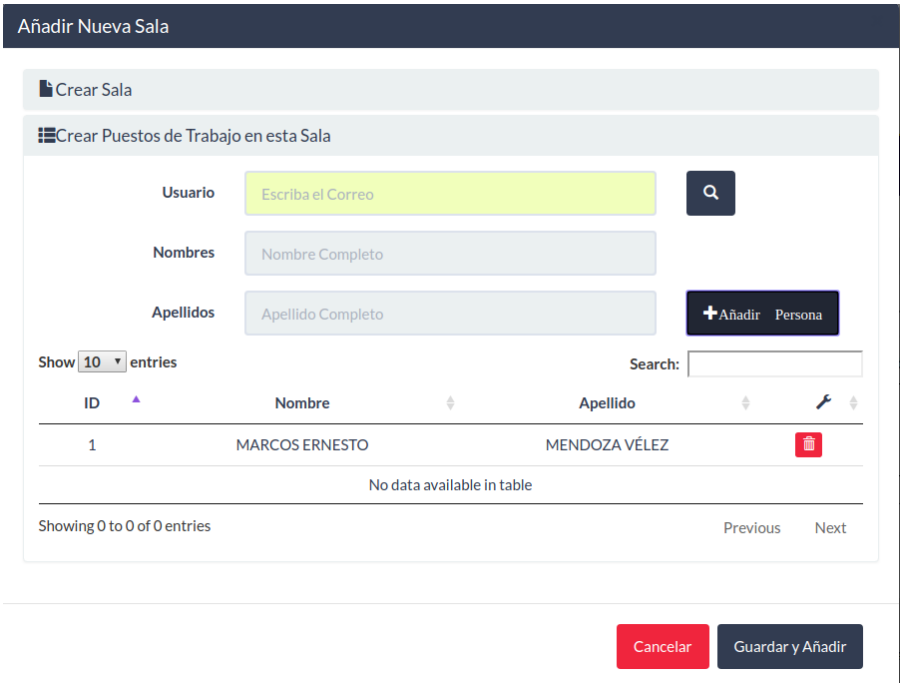

Figura 3.9: Vista para crear o añadir la sala y los puestos de trabajo.

```
exports.VerificarPuestoTrabajo= function(req, res, next){
    "use strict";
    console.log(req.body);
    var query = "SELECT COUNT(*) AS CANTIDAD FROM PUESTODETRABAJOPERSONA "+
    "WHERE PUESTODETRABAJOPERSONA.NOMBRE_PERSONA ||' '|| PUESTODETRABAJOPERSONA.APELLIDO_PERSONA = :NOMBRE";//query o procedure
    var queryValue = [req.params.NOMBRE];
    oracleModelFunction.ejecutarQueryParam(query, queryValue, req, res);
}
```

Figura 3.10: Función VerificarPuestoTrabajo.

3.2.4 Solicitud de nuevo componente

En esta parte tuvimos la obligación de crear un campo en la tabla del HTML denominado “no info”, el cual sirve de ayuda cuando el usuario ha dejado vacío este campo dentro de la solicitud del nuevo componente (ver Figura 3.11).

```
<tbody>
  <tr data-status="completed" *ngFor="let val of solicitudesComponente;let i = index">
    <td class="hidden-xs">{{i+1}}</td>
    <td>{{val.NOMBRE_TIPO}}</td>
    <td>{{val.NOMBRE_COMPONENTE}}</td>
    <td>{{val.NOMBRE_CARACTERISTICA}}</td>
    <td *ngIf="val.DESCRIPCION != ' '">{{val.DESCRIPCION}}</td>
    <td *ngIf="val.DESCRIPCION == ' '">no info</td>
    <td *ngIf="val.DESCRIPCION != ' '"><p class="text-warning">Pendiente</p></td>
    <td *ngIf="val.DESCRIPCION == ' '"><p class="text-success">Aprobado</p></td>
  </tr>
```

Figura 3.11: Selección de sentencia IF para establecer el campo vacío con “no info”.

3.3 Servicios

Los servicios de Angular 2 sirven para realizar acciones concretas, como se dan en algunos casos: proveer datos a algún componente dentro de la aplicación, hacer las peticiones al servidor haciendo como función de servicio web sin la necesidad de hacer una aplicación aparte ni estar ejecutando un servicio web para consumirlo, sino que la misma aplicación se comportará como una, dando la facilidad como lo son las API REST [7].

Cada una de estas funciones implementa lo que es un Observable que se basa en dos patrones que son el “Observer” e “Iterator”. De esta manera podemos pensar en arrays, eventos de click, llamadas http al servidor de una manera uniforme, con el fin de tratar todo tipo de administración de la información de entrada y salida, al cual se le puede agregar más operaciones para el procesamiento del flujo de los datos [7].

```
// las llamadas devuelven observables
leerDatos(): Observable<Response> {
  // Se declara cómo va a ser la llamada
  // ocultando los pormenores a los consumidores
  return this.http
    .get(`${this.urlBase}/recurso`);
  // En este momento aún no se efectuó la llamada
}
```

Figura 3.12: Ejemplo del servicio que devuelve lo datos que se llaman desde la urlBase que contiene el link al servidor [7].

Cabe mencionar en el Capítulo 2 en donde se habló un poco de los servicios, y aunque existan muchas directivas y/o servicios. Estas nos ayudan a encontrar un problema fácilmente y solucionarlo, con ello se vuelven más automatizables para ver los errores de forma más rápida y eficiente.

Los servicios nos han permitido tener una implementación y funcionalidad más configurable para la previa llamada de la API REST [7].

Una operación de simple escritura REST seria así:

```
escribirDatos(unDato): Observable<Response> {
  // Los envíos de información deben configurarse a mano
  // esto es fácilmente generalizable y reutilizable
  let body = JSON.stringify(unDato);
  let headers = new Headers({ 'Content-Type': 'application/json' });
  let options = new RequestOptions({ headers: headers });
  // declarar la llamada y retornar el observable
  // las variables de configuración y los datos, van como parámetros
  if (unDato._id) {
    return this.http
      .put(`${this.urlBase}/recurso/${unDato._id}`, body, options);
  } else {
    return this.http
      .post(`${this.urlBase}/recurso`, body, options);
  }
}
```

Figura 3.13: Ejemplo de escritura en donde se envían los datos en formato json. [7]

En nuestro proyecto se tienen los siguientes servicios pero no describimos todos.

- campus.service.ts
- características.service.ts
- componente.service.ts
- cubículo.service.ts
- edificio.service.ts
- estructura.service.ts
- planta.service.ts
- puesto-trabajo.service.ts
- sala.service.ts
- tipo-comoponente.service.ts
- tipo-edificio.service.ts
- usuario-admin.service.ts

3.3.1 Servicio campus

Los servicios utilizados:

- getCampusActivos() - getCampusInactivos()
    - Obtener todos los campus del sistema de acuerdo con el estado que se le ha asignado, siendo 1 para los que se encuentran activos y 0 para los que no.
- updateEstadoCampus(campus)
    - Actualizar el estado del campus de activo a inactivo y viceversa, recibe los datos del campus.ts.

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import { Constants } from './general/constantes';
import 'rxjs/add/operator/map';

@Injectable()
export class CampusService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "campus_server/";

  getCampusActivos(): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+'campus_activo').map(res => res.json());
  }

  getCampusInactivos(): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+'campus_inactivo').map(res => res.json());
  }

  addCampus(campus): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'campus', JSON.stringify(campus),this.options);
  }

  deleteCampus(campus): Observable<any> {
    return this.http.put(Constants.SERVER_API+this.ruta+`campus/${campus.ID_HASH_CAMPUS}`, this.options);
  }

  updateEstadoCampus(campus): Observable<any> {
    return this.http.put(Constants.SERVER_API+this.ruta+`campus_actualizar/${campus.ID_HASH_CAMPUS}`, this.options);
  }

  getCampus(): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+'campus').map(res => res.json());
  }

  editCampus(campus): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'edit_campus', JSON.stringify(campus),this.options);
  }

  deleteCampusAdmin(ID_HASH_CAMPUS): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_campus_admin/${ID_HASH_CAMPUS}`, this.options);
  }

}
```

Figura 3.13: Funciones del servicio de campus.service.ts.

3.3.2 Servicio componente

- addComponente(componente)
  - Añadir un nuevo componente al sistema.
- getComponente()
  - Obtener todos los componentes del sistema.
- deleteComponente(ID_HASH_COMP)
  - Borrar un componente dentro del sistema.
- getComponenteTipo(ID_HASH_TIPO)
  - Obtener todos los tipos de componentes existentes.
- getComponentePiso()
  - Obtener todos los componentes que pertenecen a una planta o piso.
- getTodasSolicitudesComponentes()
  - Obtener todas las solicitudes de los componentes existentes.
- approveComponente(ID_HASH_COMP)
  - Aprobar un componente que este en la lista de componentes solicitados.
- editComponente(componente)
  - Editar un componente existente.
- getComponenteTipoString(TIPO)
  - Obtener los componentes de los cubículos.

```
@Injectable()
export class ComponenteService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "componente/";

  addComponente(componente): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'componente_nuevo', JSON.stringify(componente),this.options);
  }

  getComponente(): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+'obtener_componente').map(res => res.json());
  }

  deleteComponente(ID_HASH_COMP): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_componente/${ID_HASH_COMP}`, this.options);
  }

  getComponenteTipo(ID_HASH_TIPO): Observable<any>{
    return this.http.get(Constants.SERVER_API+this.ruta+`obtener_componente_tipo/${ID_HASH_TIPO}`).map(res => res.json());
  }

  getComponentePiso(): Observable<any>{
    return this.http.get(Constants.SERVER_API+this.ruta+'obtener_componente_tipo_piso').map(res => res.json());
  }

  getSolicitudesComponentes(): Observable<any>{
    return this.http.get(Constants.SERVER_API+this.ruta+'obtener_solicitud_componentes').map(res => res.json());
  }

  getTodasSolicitudesComponentes(): Observable<any>{
    return this.http.get(Constants.SERVER_API+this.ruta+'obtener_todas_solicitudes').map(res => res.json());
  }


  approveComponente(ID_HASH_COMP): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+`aprobar_componente/${ID_HASH_COMP}`, this.options);
  }

  editComponente(componente): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'editar_componente', JSON.stringify(componente),this.options);
  }

  getComponenteTipoString(TIPO): Observable<any>{
    return this.http.get(Constants.SERVER_API+this.ruta+`obtener_componente_tipo_cubiculo/${TIPO}`).map(res => res.json());
  }
}
```

Figura 3.14: Funciones del servicio componente.service.ts

3.3.3 Servicio cubículo

- addCubiculoGetId(cubiculo)
  - Añadir un cubículo a la sala
- getCubiculo(ID_HASH_SALA)
  - Obtener todos los cubículos de una sala por su ID.
- deleteCubiculo(ID_HASH_CUBICULO)
  - Borrar cubículo por ID.
- editCubiculo(cubiculo)
  - Editar un cubículo por ID.

- getCubiculoId(ID_HASH_CUBICULO)
    - Obtener el cubículo solo con el ID.
- getCubiculoComponente(ID_HASH_CUBICULO)
    - Obtener todos los componentes dentro del cubículo.
- addCubiculoComponente(cubiculoComponente)
    - Añadir un componente dentro de los cubículos.
- deleteCubiculoComponente(cubiculoComponente)
    - Borrar el componente dentro del cubículo.
- getCubiculoComponenteTodo(ID_HASH_CUBICULO)
    - Obtener todos los componentes de los cubículos.

```
@Injectable()
export class CubiculoService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "cubiculo/";

  addCubiculoGetId(cubiculo): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'cubiculo_nuevo_id', JSON.stringify(cubiculo),this.options);
  }

  getCubiculo(ID_HASH_SALA): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`obtener_cubiculo/${ID_HASH_SALA}`).map(res => res.json());
  }

  deleteCubiculo(ID_HASH_CUBICULO): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_cubiculo/${ID_HASH_CUBICULO}`, this.options);
  }

  editCubiculo(cubiculo): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'cubiculo_editar', JSON.stringify(cubiculo),this.options);
  }

  getCubiculoId(ID_HASH_CUBICULO){
    return this.http.get(Constants.SERVER_API+this.ruta+`cubiculo_obtener_cubiculo/${ID_HASH_CUBICULO}`).map(res => res.json());
  }

  getCubiculoComponente(ID_HASH_CUBICULO){
    return this.http.get(Constants.SERVER_API+this.ruta+`cubiculo_obtener_componente/${ID_HASH_CUBICULO}`).map(res => res.json());
  }

  addCubiculoComponente(cubiculoComponente){
    return this.http.post(Constants.SERVER_API+this.ruta+'cubiculo_nuevo_cubiculoComponente', JSON.stringify(cubiculoComponente),this.options);
  }

  deleteCubiculoComponente(cubiculoComponente){
    return this.http.post(Constants.SERVER_API+this.ruta+'cubiculo_borrar_cubiculoComponente', JSON.stringify(cubiculoComponente),this.options);
  }

  getCubiculoComponenteTodo(ID_HASH_CUBICULO){
    return this.http.get(Constants.SERVER_API+this.ruta+`cubiculo_obtener_cubiculoComponente_todo/${ID_HASH_CUBICULO}`).map(res => res.json());
  }

}
```

Figura 3.15: Funciones del servicio cubiculo.service.ts

3.3.4 Servicio edificio

- addEdificio(edificio)
  - Añadir un edificio al campus.
- getEdificios(FK_ID_HASH_CAMPUS)
  - Obtener los edificios pertenecientes al campus.
- deleteEdificios(FK_ID_HASH_CAMPUS)
  - Borrar edificio dentro del campus.
- deleteEdificio(ID_HASH_EDIFICIO)
  - Borrar la referencia del edificio en el campus.
- addPlanoEdificio(plano)
  - Subir un plano al edificio.
- getPlanoEdificio(RUTA)
  - Obtener el plano dentro del edificio.
- editEdificio(edificio)
  - Editar los atributos del edificio.

```
@Injectable()
export class EdificioService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "edificio/";

  addEdificio(edificio): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'edificio', JSON.stringify(edificio),this.options);
  }

  getEdificios(FK_ID_HASH_CAMPUS): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`edificio/${FK_ID_HASH_CAMPUS}`).map(res => res.json());
  }

  deleteEdificios(FK_ID_HASH_CAMPUS): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`edificio/${FK_ID_HASH_CAMPUS}`, this.options);
  }

  deleteEdificio(ID_HASH_EDIFICIO): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`edificio_borrar/${ID_HASH_EDIFICIO}`, this.options);
  }

  addPlanoEdificio(plano): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'ruta_plano_edificio', JSON.stringify(plano),this.options);
  }

  getPlanoEdificio(RUTA): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`descargar_plano_edificio/${RUTA}`).map(res => res.json());
  }

  editEdificio(edificio): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'editar_edificio', JSON.stringify(edificio),this.options);
  }

}
```

Figura 3.16: Funciones del servicio edificio.service.ts

3.3.5 Servicio estructura

- addEstructuraComponente(estructuraComponente)
    - Añadir una estructura al usuario.

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import { Constants } from './general/constantes';
import 'rxjs/add/operator/map';

@Injectable()
export class EstructuraComponenteService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
    private options = new RequestOptions({ headers: this.headers });

    constructor(private http: Http) { }

    ruta = "estructuraComponente/";

    addEstructuraComponente(estructuraComponente): Observable<any> {
      return this.http.post(Constants.SERVER_API+this.ruta+'estructura_componente_nuevo', JSON.stringify(estructuraComponente),this.options);
    }
}
```

Figura 3.17: Funciones del servicio estructura.service.ts

3.3.6 Servicio planta

Este servicio además de las operaciones básicas del CRUD, incluye otras operaciones para el correcto funcionamiento:

- addPiso(piso)
    - Añadir un piso al edificio.
- getPiso(ID_EDIFICIO)
    - Obtener todos los pisos en planta dentro del edificio.
- getPisoCantidad(ID_EDIFICIO)
    - Obtener la cantidad del piso.
- deletePiso(ID_HASH_PISO)
    - Eliminar el piso.
- editPiso(piso)
    - Editar el piso.
- getPisoComponente(ID_PISO)
    - Obtener todos los componentes de la planta por ID.
- getComponentePiso()
    - Obtener todos los componentes de la planta.
- addPlantaComponente(plantaComponente)

- Añadir una componente a la planta.

- getPlantaComponente(ID_HASH_PISO)
  - Obtener de la planta ese componente por ID.
- getPlantaComponenteTodo(ID_HASH_PLANTA)
  - Obtener los componentes de la planta.
- deletePlantaComponente(cubiculoComponente)
  - Borrar una planta en el edificio.

```
@Injectable()
export class PlantaService {

    private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
    private options = new RequestOptions({ headers: this.headers });

    constructor(private http: Http) { }

    ruta = "planta/";
    ruta_estructura_componente = "estructuraComponente/";
    ruta_componente = "componente/";

    addPiso(piso): Observable<any> {
      return this.http.post(Constants.SERVER_API+this.ruta+'piso_nuevo', JSON.stringify(piso),this.options);
    }

    getPiso(ID_EDIFICIO): Observable<any> {
      return this.http.get(Constants.SERVER_API+this.ruta+`piso_obtener/${ID_EDIFICIO}`).map(res => res.json());
    }

    getPisoCantidad(ID_EDIFICIO): Observable<any> {
      return this.http.get(Constants.SERVER_API+this.ruta+`piso_cantidad/${ID_EDIFICIO}`).map(res => res.json());
    }

    deletePiso(ID_HASH_PISO): Observable<any> {
      return this.http.delete(Constants.SERVER_API+this.ruta+`piso_borrar/${ID_HASH_PISO}`, this.options);
    }

    editPiso(piso): Observable<any> {
      return this.http.post(Constants.SERVER_API+this.ruta+'piso_edit', JSON.stringify(piso),this.options);
    }

    getPisoComponente(ID_PISO): Observable<any> {
      return this.http.get(Constants.SERVER_API+this.ruta_estructura_componente+`estructura_componente_obtener_todo/${ID_PISO}`).map(res => res.json());
    }

    getComponentePiso(): Observable<any> {
      return this.http.get(Constants.SERVER_API+this.ruta_componente+'obtener_componente_tipo_piso').map(res => res.json());
    }

    addPlantaComponente(plantaComponente){
      return this.http.post(Constants.SERVER_API+this.ruta+'planta_nuevo_plantaComponente', JSON.stringify(plantaComponente),this.options);
    }

    getPlantaComponente(ID_HASH_PISO){
      return this.http.get(Constants.SERVER_API+this.ruta+`planta_obtener_componente/${ID_HASH_PISO}`).map(res => res.json());
    }

    getPlantaComponenteTodo(ID_HASH_PLANTA){
      return this.http.get(Constants.SERVER_API+this.ruta+`planta_obtener_plantaComponente_todo/${ID_HASH_PLANTA}`).map(res => res.json());
    }

    deletePlantaComponente(cubiculoComponente){
      return this.http.post(Constants.SERVER_API+this.ruta+'planta_borrar_plantaComponente', JSON.stringify(cubiculoComponente),this.options);
    }

}
```

Figura 3.18: Funciones del servicio planta.service.ts.

3.3.7 Servicio puesto-trabajo

- addPuestoTrabajo(puesto_trabajo)
  - Añadir un puesto de trabajo.
- getPuestoTrabajo(ID_HASH_SALA)

    - Obtener los puestos de trabajo de la sala por ID.
- getPuestoTrabajoCubiculo(ID_HASH_CUBICULO)
    - Obtener los puestos de trabajo dentro de los cubículos.
- verificarPuestoTrabajo(NOMBRE)
    - Verificar el puesto de trabajo si ya existe.
- deletePuestoTrabajo(ID_HASH_PUESTO)
    - Borrar un puesto de trabajo.

```
@Injectable()
export class PuestoTrabajoService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "puestoTrabajo/";

  addPuestoTrabajo(puesto_trabajo): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'puesto_trabajo_nuevo', JSON.stringify(puesto_trabajo),this.options);
  }

  getPuestoTrabajo(ID_HASH_SALA): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`puesto_trabajo_obtener/${ID_HASH_SALA}`).map(res => res.json());
  }

  getPuestoTrabajoCubiculo(ID_HASH_CUBICULO): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`puesto_trabajo_cubiculo_obtener/${ID_HASH_CUBICULO}`).map(res => res.json());
  }

  verificarPuestoTrabajo(NOMBRE): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`puesto_trabajo_verificar/${NOMBRE}`).map(res => res.json());
  }

  deletePuestoTrabajo(ID_HASH_PUESTO): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_puestoTrabajo/${ID_HASH_PUESTO}`, this.options);
  }

}
```

Figura 3.19: Funciones del servicio puesto-trabajo.service.ts.

3.3.8 Servicio sala

- addSala(sala)
    - Añadir una sala a la planta.
- getSala(ID_PISO)
    - Obtener todas las salas dentro de la planta.
- deleteSala(ID_HASH_SALA)
    - Borrar una sala dentro de la planta.
- editSala(sala)
    - Editar una sala dentro de la planta.

- getSalaId(ID_HASH_SALA)
    - Obtener una sala por ID.
- getSalaComponente(ID_HASH_SALA)
    - Obtener el componente dentro de la sala.
- addSalaComponente(salaComponente)
    - Añadir un componente dentro de la sala.
- deleteSalaComponente(salaComponente)
    - Borrar un componente dentro de la sala.
- getSalaComponenteTodo(ID_HASH_SALA)
    - Obtener todos los componentes dentro de la sala.

```
@Injectable()
export class SalaService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "sala/";

  addSala(sala): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'sala_nuevo', JSON.stringify(sala),this.options);
  }

  getSala(ID_PISO): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+`sala_obtener/${ID_PISO}`).map(res => res.json());
  }

  deleteSala(ID_HASH_SALA): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`sala_eliminar/${ID_HASH_SALA}`, this.options);
  }

  editSala(sala): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'sala_editar', JSON.stringify(sala),this.options);
  }

  getSalaId(ID_HASH_SALA){
    return this.http.get(Constants.SERVER_API+this.ruta+`sala_obtener_sala/${ID_HASH_SALA}`).map(res => res.json());
  }

  getSalaComponente(ID_HASH_SALA){
    return this.http.get(Constants.SERVER_API+this.ruta+`sala_obtener_componente/${ID_HASH_SALA}`).map(res => res.json());
  }

  addSalaComponente(salaComponente){
    return this.http.post(Constants.SERVER_API+this.ruta+'sala_nuevo_salaComponente', JSON.stringify(salaComponente),this.options);
  }

  deleteSalaComponente(salaComponente){
    return this.http.post(Constants.SERVER_API+this.ruta+'sala_borrar_salaComponente', JSON.stringify(salaComponente),this.options);
  }

  getSalaComponenteTodo(ID_HASH_SALA){
    return this.http.get(Constants.SERVER_API+this.ruta+`sala_obtener_salaComponente_todo/${ID_HASH_SALA}`).map(res => res.json());
  }

}
```

Figura 3.20: Funciones del servicio sala.service.ts.

3.3.9 Servicio tipo-componente

- addTipoComponente(tipoComponente)
  - Añadir un nuevo tipo de componente.
- getTipoComponente()
  - Obtener todos los tipos de componente.
- deleteTipoComponente(ID_HASH_TIPO)
  - Borrar el tipo de componente.
- editTipo(tipo)
  - Editar el tipo de componente.

```
@Injectable()
export class TipoComponenteService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "tipo/";

  addTipoComponente(tipoComponente): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'nuevo_tipo', JSON.stringify(tipoComponente),this.options);
  }

  getTipoComponente(): Observable<any> {
    return this.http.get(Constants.SERVER_API+this.ruta+'obtener_tipo').map(res => res.json());
  }

  deleteTipoComponente(ID_HASH_TIPO): Observable<any> {
    return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_tipo/${ID_HASH_TIPO}`, this.options);
  }

  editTipo(tipo): Observable<any> {
    return this.http.post(Constants.SERVER_API+this.ruta+'edit_tipo', JSON.stringify(tipo),this.options);
  }
}
```

Figura 3.21: Funciones del servicio tipo-componente.service.ts.

3.3.10 Servicio tipo-edificio

- addTipoEdificio(tipoEdificio)
  - Añadir un nuevo tipo de edificio que cree el administrador.
- getTipoEdificio()
  - Obtener todos los tipos de edificio existente.
- deleteTipoEdificio(ID_HASH_TIPO_EDIFICIO)
  - Borrar el tipo de edificio por su ID.
- editTipoEdificio(tipoEdificio)
  - Editar el edificio existente

```
@Injectable()
export class TipoEdificioService {
  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
    private options = new RequestOptions({ headers: this.headers });

    constructor(private http: Http) { }

    ruta = "tipo_edificio/";

    addTipoEdificio(tipoEdificio): Observable<any> {
      return this.http.post(Constants.SERVER_API+this.ruta+'nuevo_tipo_edificio', JSON.stringify(tipoEdificio),this.options);
    }

    getTipoEdificio(): Observable<any> {
      return this.http.get(Constants.SERVER_API+this.ruta+'obtener_tipo_edificio').map(res => res.json());
    }

    deleteTipoEdificio(ID_HASH_TIPO_EDIFICIO): Observable<any> {
      return this.http.delete(Constants.SERVER_API+this.ruta+`eliminar_tipo_edificio/${ID_HASH_TIPO_EDIFICIO}`, this.options);
    }

    editTipoEdificio(tipoEdificio): Observable<any> {
      return this.http.post(Constants.SERVER_API+this.ruta+'edit_tipo_edificio', JSON.stringify(tipoEdificio),this.options);
    }

}
```

Figura 3.22: Funciones del servicio tipo-edificio.service.ts.

3.3.11 Servicio usuario-admin

- addUsuario(usuario)
  - Añadir un nuevo usuario al sistema.
- checkUsuario(USUARIO)
  - Verificar si el usuario ya existe en el sistema.
- deleteUsuario(ID_HASH_USUARIO)
  - Borrar el usuario ya existente.
- getUsuario()
  - Obtener todos los usuarios ya existentes en el sistema.
- getUsuarioWebService(USUARIO)
  - Obtener al usuario desde el desde el web-service de la ESPOL.
- getUsuarioCookie()
  - Obtener la cokkie desde el cliente.
- getLogoutLink()
  - Enviar el link del CAS.
- getRolUsuario(USUARIO)
  - Obtener el rol del usuario.

- getlogin()
  - Enviar al CAS de la ESPOL.
- getlogout()
  - Desloguear(cerrar sesión) al usuario desde el cliente, destruyendo la cookie almacenada en el navegador.
- getLocalStorage()
  - Obtener desde la cache del navegador el usuario.

```
@Injectable()
export class UsuarioAdminService {

  private headers = new Headers({ 'Content-Type': 'application/json', 'charset': 'UTF-8' });
  private options = new RequestOptions({ headers: this.headers });

  constructor(private http: Http) { }

  ruta = "usuario_server/";

  addUsuario(usuario): Observable<any> {
    return this.http.post(Constants.SERVER_API + this.ruta + 'nuevo_usuario', JSON.stringify(usuario), this.options);
  }

  checkUsuario(USUARIO): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + `verifica_usuario/${USUARIO}`).map(res => res.json());
  }

  deleteUsuario(ID_HASH_USUARIO): Observable<any> {
    return this.http.delete(Constants.SERVER_API + this.ruta + `borrar_usuario/${ID_HASH_USUARIO}`, this.options);
  }

  getUsuario(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'obtener_usuario').map(res => res.json());
  }

  getUsuarioWebService(USUARIO): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + `obtener_usuario_webservice/${USUARIO}`).map(res => res.json());
  }

  getUsuarioCookie(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'obtener_cookie_user_secret').map(res => res.json());
  }

  getLogoutLink(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'logout').map(res => res.json());
  }

  getRolUsuario(USUARIO): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + `rol_usuario/${USUARIO}`).map(res => res.json());
  }

  getlogin(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'login',this.options).map(res => res.json());
  }

  getlogout(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'logout',this.options).map(res => res.json());
  }

  getLocalStorage(): Observable<any> {
    return this.http.get(Constants.SERVER_API + this.ruta + 'obtener_local_storage',this.options).map(res => res.json());
  }

}
```

Figura 3.23: Funciones del servicio usuario-admin.service.ts

3.4 Funcionamiento del servidor en NodeJS

El funcionamiento del servidor NodeJS como se había mencionado en el capítulo 2, es el de una plataforma que se enfoca en el backend para el desarrollo de aplicaciones web basado en JavaScript. Esta herramienta fue importante para el desarrollo, brindándonos una librería llamada Express, que nos proporciona un conjunto de características para dar más flexibilidad a nuestra aplicación web [8].

3.5 Módulo Oracle utilizado en el server

El módulo utilizado para la conexión de Oracle y NodeJS es el oracledb versión 1.13 para versiones mayores a 4.6 de NodeJS, Esta es una extensión que facilita el trabajo con NodeJS, en GIT existe una comunidad que se encarga de dar el soporte a este driver, así como proveer de varios ejemplos a su disposición [9].

En este módulo se importa la librería oracledb y la configuración en donde se encuentran el usuario y la contraseña específicos de la base que se está usando (ver Figura 3.24).

En la Figura 3.25 mostramos como es la ejecución de las consultas sin parámetros, esto nos facilita para hacer consultas SELECT en un procedure, mientras en la Figura 3.26 nos facilita hacer INSERT, UPDATE y DELETE.

```
var oracledb = require('oracledb');
var config = require('./oracleconfig');
var connAttrs = {
    "user" : config.user,
    "password" : config.password,
    "connectString" : "localhost/XE"
}
```

Figura 3.24: Ejemplo del archivo oracleconfig.js llamando los atributos user y password.

```
exports.ejecutarQuery = function (query, req, res){
    oracledb.getConnection(connAttrs, function (err, connection) {
        if (err) {
                // Error connecting to DB
                res.set('Content-Type', 'application/json');
                res.status(500).send(JSON.stringify({
                        status: 500,
                        message: "Error connecting to DB",
                        detailed_message: err.message
                }));
                return;
        }
        connection.execute(query, {}, {
                outFormat: oracledb.OBJECT // Return the result as Object
        }, function (err, result) {
                if (err) {
                        res.set('Content-Type', 'application/json');
                        res.status(500).send(JSON.stringify({
                                status: 500,
                                message: "Error",
                                detailed_message: err.message
                        }));
                } else {
                        res.contentType('application/json').status(200);
                        console.log(result);
                        res.send(JSON.stringify(result.rows));
                }
                // Release the connection
        connection.release(
            function (err) {
                    if (err) {
                            console.error(err.message);
                    } else {
                            console.log("GET : Connection released");
                    }
            });
        });
    });
}
```

Figura 3.25: Función para ejecutar el query o procedure, que accede a la base de datos para devolver la consulta sin parámetros en formato json.

```
exports.ejecutarQueryParam = function(query, queryValue, req, res){
	oracledb.getConnection(connAttrs, function (err, connection) {
		var queryResult = [];
		if (err) {
			// Error connecting to DB
			res.set('Content-Type', 'application/json').status(500).send(JSON.stringify({
				status: 500,
				message: "Error connecting to DB",
				detailed_message: err.message
			}));
			return;
		}
		connection.execute(query,queryValue, {
				autoCommit: true,
				outFormat: oracledb.OBJECT // Return the result as Object
			},
			function (err, result) {
			if (err) {
				res.set('Content-Type', 'application/json');
				res.status(500).send(JSON.stringify({
					status: 500,
					message: "Error ",
					detailed_message: err.message
				}));
			} else {
				res.contentType('application/json').status(200);
				console.log(result);
				res.send(JSON.stringify(result.rows));
			}
			// Release the connection
		connection.release(
			function (err) {
				if (err) {
					console.error(err.message);
				} else {
					console.log("Post  : Connection released");
				}
			});
		});
	});
}
```

Figura 3.26: Función para ejecutar el query o procedure, que accede a la base de datos para devolver la consulta con parámetros en formato json.

CONCLUSIONES Y RECOMENDACIONES

El sistema web nos permite catalogar las distintas edificaciones y componentes dentro de la ESPOL, para ello se elaboró una estructura entre los distintos tipos de componentes del catálogo que posee la Gerencia de Infraestructura Física de la ESPOL, actualizando así el ingreso al catálogo solo de tipos ya existentes.

Con ello se pudo gestionar toda la infraestructura física y de sus componentes, dinamizando el proceso de envío de solicitudes por parte del custodio y la respuesta del administrador para que se tenga un control en añadir un nuevo componente al catálogo.

El sistema se puede mejorar implementando un módulo de reportería más avanzada puesto que está abierto para requerir de otros sistemas de la ESPOL.

Cuando alguna librería dentro del package.json esté deprecada o no se le dé algún tipo de mantenimiento, se la deberá actualizar considerando el nombre de las funciones que se estén usando en este proyecto.

Si se desea tener un mejor rendimiento al momento de añadir más componente a las tablas del HTML, se las deberá actualizar o recurrir a una nueva librería, puesto que en este proyecto se usa la librería básica para el uso de la tabla de datos (angular-datatable) [10].

BIBLIOGRAFÍA

[1] Pulpo WMS: Sistema de Gestión de Inventarios [Online] disponible en: https://pulpowms.com/es/education/.

[2] O. Kuryan, Odoo v9 Community vs Enterprise [Online] disponible en:

https://xpansa.com/odoo/odoo-community-vs-odoo-enterprise/.

[3] Software para la Gestión y Mantenimiento de los Activos [Online] disponible en: http://www.ingelsi.com.ec/pmi-software-para-la-gestion-y-mantenimiento-de-activos/.

[4] O. Blancarte, Introducción a Nodejs[Online] disponible en:

https://www.oscarblancarteblog.com/2017/05/29/introduccion-a-nodejs-2/.

[5] DeveloperIQ, Software Technology magazine [Online] disponlible en: http://developeriq.in/articles/2015/feb/09/node-expressjs-and-mongoose-part-ii/.

[6] C. Álvarez Caules, Angular 2 y el futuro de las arquitecturas web – Arquitectura Java [Online] disponible en: http://www.arquitecturajava.com/angular-2-y-el-futuro-de-la-web/.

[7] A. Basalo, Comunicaciones http observables con Angular 2 [Online] disponible en: http://academia-binaria.com/comunicaciones-http-observables-con-angular2/.

[8] Express – Infraestructura de aplicaciones web Node.js [Online] disponible en: http://expressjs.com/es/.

[9] C. Jones, Oracle/node-oracledb [Online] disponible en: https://github.com/oracle/node-oracledb.

[10] L. Lin, Angular DataTables [Online] disponible en: https://l-lin.github.io/angular-datatables/#/welcome.