

Construcción aplicación web

Breve descripción:

En este componente formativo se realizará una aplicación web completa (Back-end y Front-end). Se utilizarán tecnologías del lado del servidor como Node.js, MongoDB, Express y del lado del cliente AngularJS y Materialize. La aplicación consumirá un API REST con el propósito de hacer el registro de empleados en una empresa.

Julio 2024

Tabla de contenido

Introducción	4
1. Construcción del back-end de la aplicación web	6
1.1. Instalación y configuración de herramientas de Node.js	6
1.2. Desarrollo del index.js	7
1.3. Configuración e instalación de la base de datos MongoDB	12
1.4. Configuración de database.js	14
1.5. Configuración de las rutas del proyecto con Node.js	16
1.6. Desarrollo de controladores para la API REST	19
1.7. Realización de pruebas de la API REST con Postman	24
2. Construcción del front-end de la aplicación web	29
2.1. Instalación y configuración de AngularJS	29
2.2. Generación de código con ng	31
2.3. Incorporación de Materialize al proyecto web	34
2.4. Configuración de app.component.html	35
2.5. Configuración de los módulos app.module.ts	36
2.6. Configuración del modelo empleado.ts	37
2.7. Creación del servicio empleado.service.ts	39
2.8. Desarrollo del componente empleados.component.ts	41

2.9. Desarrollo de la interfaz HTML empleados.component.html.....	44
2.10. Probando la aplicación completa.	47
Síntesis	54
Material complementario	55
Glosario	56
Referencias bibliográficas	57
Créditos	58

Introducción

El desarrollo de aplicaciones web modernas requiere la integración de diversas tecnologías y herramientas que permitan crear soluciones robustas, escalables y eficientes. En este componente, se detalla el proceso de construcción de una aplicación web utilizando la arquitectura MEAN, que comprende MongoDB, Express.js, Angular y Node.js. Este conjunto de tecnologías permite desarrollar aplicaciones completas del lado del servidor (back-end) y del cliente (front-end), facilitando la creación de sistemas integrados.

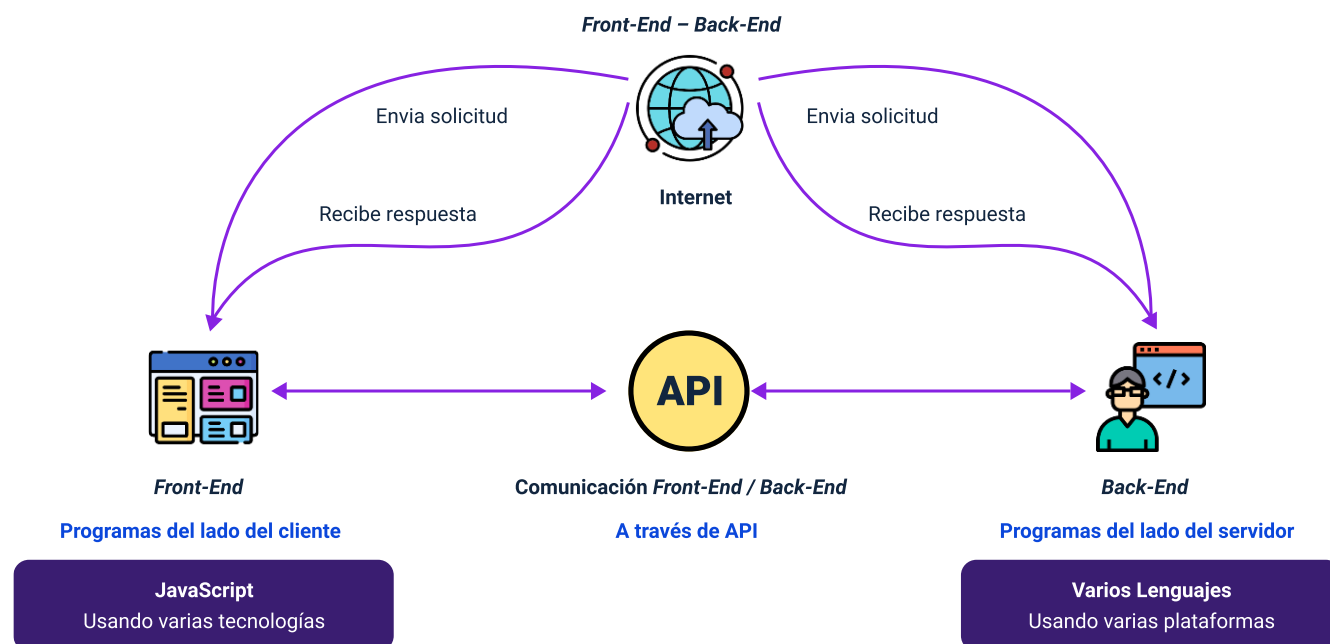
El enfoque principal de este proyecto es la gestión de empleados, abarcando desde la configuración inicial de herramientas y la estructura del proyecto, hasta el desarrollo de controladores, modelos y rutas para la API REST. Además, se explica cómo configurar y utilizar Angular para la creación de componentes y servicios en el front-end, asegurando una interacción fluida y dinámica con el servidor.

Cada sección del componente proporciona instrucciones detalladas y ejemplos de código que guían paso a paso el proceso de desarrollo. Desde la instalación de Node.js y Express, la configuración de la base de datos MongoDB, hasta la integración de Materialize para mejorar la interfaz de usuario, se cubren todos los aspectos necesarios para construir una aplicación web funcional y profesional.

Esto permitirá tener un mayor conocimiento sobre las tecnologías web actuales, especialmente en el uso de las técnicas del lenguaje, los recursos para probar y testear la aplicación, las nuevas formas de sintaxis de algunos lenguajes, los componentes y módulos que harán el trabajo más fácil a los desarrolladores. En últimas palabras, este

componente apoya de manera significativa el primer paso para generar competencias y habilidades con el fin de generar nuevas aplicaciones web.

Figura 1. Imagen con el diagrama que ilustra la comunicación entre FrontEnd y BackEnd



1. Construcción del back-end de la aplicación web

Se inicia con una estructura de proyecto MEAN (Mongo, Express, Angular y Node.js) para el proyecto que abarca tanto el front-end como el back-end. Para la instalación de las primeras herramientas como Node.js y Express.

1.1. Instalación y configuración de herramientas de Node.js

Se utiliza Visual Studio Code (VS Code) con la siguiente estructura de directorios para el back-end, detallada a continuación:

GESTION-EMPLEADOS

 backend

 controllers

 models

 routes

 database.js

 index.js

node_modules

package-lock.json

package.json

Como se presentó anteriormente, se crea la carpeta GESTION-EMPLEADOS, que será la principal o raíz del proyecto. Dentro de esta, se crea la carpeta back-end, en la cual se crean otros directorios y archivos. De tal forma, esta será la estructura que apoyará el desarrollo de la base de la aplicación. Una vez creadas las carpetas y

archivos, como controllers, models, routes, database.js e index.js, se abre la terminal del VS Code y se digita el siguiente comando para crear un proyecto Node.js: npm init -yes

PROBLEMS | OUTPUT | TERMINAL | DEBUG CONSOLE

Windows PowerShell

Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma

<https://aka.ms/pscore6>

PS D:\gestion-empleados> npm init --yes

Una vez creado el proyecto, se requieren todas las bondades del framework de Node.js, para lo cual es necesario Express. Express generará una serie de componentes que ayudarán mucho al desarrollo de esta aplicación. Por lo tanto, se instala Express con la siguiente línea de comandos desde la terminal del VS Code: npm install express.

1.2. Desarrollo del index.js

El siguiente paso es iniciar con el desarrollo del archivo principal del back-end, que se llama index.js como se presenta a continuación:

```
const express = require('express')
```

```
const morgan = require('morgan');
```

```
const cors = require('cors');
```

```
const app = express(); // la constante app tendrá ahora todo el funcionamiento del servidor
```

```
const { mongoose } = require('./database'); // no se quiere todo el archivo sino la
conexión/** * Se crea una REST API, es la manera de decirle al servidor que reciba y
envíe datos */
```

```
// Configuraciones
```

```
app.set('port', process.env.PORT || 3000);
```

```
app.use(morgan('dev'));
```

```
app.use(express.json()); // método que ayuda a convertir el código para que el
servidor pueda entender lo que viene del cliente.
```

```
app.use(cors({origin: 'http://localhost:4200'})); // método para comunicar con el
cliente
```

```
// rutas de nuestro servidor
```

```
app.use('/api/empleados',require('./routes/empleado.routes'));
```

```
// Iniciando el servidor
```

```
app.listen(app.get('port'), () => { // esta es una mejor manera de configurar el
puerto
```

```
console.log('server activo en el puerto', app.get('port'));
```

```
});
```


A continuación, se presentan los siguientes pasos para el desarrollo del index.js:

a) Desarrollo del archivo principal del #[em back-end]

El siguiente paso es iniciar con el desarrollo del archivo principal del back-end, que se llama index.js.

b) Código del archivo index.js

Se explica el código del archivo index.js para el proyecto. A medida que avance este componente, se explicará cada línea de código que se requiera ejecutar.

c) Requerimiento de Express

Se inicia con el requerimiento de Express y se almacena en una constante en JavaScript llamada express. De esta forma, se tendrá acceso a todo el funcionamiento del servidor con la línea de código: `const app = express();`.

d) Configuración del puerto

Una vez creada la constante app, se pasa a la configuración del puerto por donde va a escuchar y recibir el servidor todo tipo de solicitudes por parte del cliente, según se muestra en la siguiente línea: `app.set('port', process.env.PORT)`

e) Uso de set

Set es para crear una variable que va a ser accedida desde cualquier parte de la aplicación. La estructura es la siguiente: (nombre var, valor). process es porque cuando se despliegue la app no se va a tener la opción de definir el puerto, sino que el mismo servicio de la nube ayudará a definirlo, en caso de que el puerto 3000 esté ocupado.

f) Instalación de Nodemon

Con el fin de revisar los cambios en el servidor de manera automática, se tiene planeada la instalación de algunas herramientas que harán más fácil el desarrollo sobre Node.js, por lo que se instala Nodemon como una utilidad que monitorea de manera constante los cambios en el código fuente que se está desarrollando y de manera automática reinicia el servidor. La línea de código utilizada en la terminal de VS Code es: `npm install nodemon -D`.

g) Configuración de Nodemon

Esto significa que Nodemon será instalado como una dependencia de desarrollo y no como dependencia general del proyecto. Se configura el archivo `package.json` de acuerdo a las instrucciones.

h) Iniciar el servidor

Se inicia a ejecutar el servidor con la siguiente línea de comando: `npm run dev`.

Estructura `package.json`:

```
{  
  
  "name": "gestion-empleados",  
  
  "version": "1.0.0",  
  
  "description": "",  
  
  "main": "index.js",  
  
  "scripts": {
```

```
"dev": "nodemon BackEnd/index.js"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

  "cors": "^2.8.5",

  "express": "^4.17.1",

  "mongoose": "^5.12.14",

  "morgan": "^1.10.0"

},

"devDependencies": {

  "nodemon": "^2.0.7"

}

}
```

Ahora, se construyen las Middleware dentro del archivo index.js, encargadas de procesar los datos, es decir, cuando se solicitan o envían al servidor. El servidor debe entender los datos que le llegan del cliente (AngularJS toma los datos del cliente y los envía en formato JSON) y es por eso que se necesita algún tipo de conversión. Por lo

tanto, se requiere del uso del formato JSON ejecutando la siguiente instrucción:

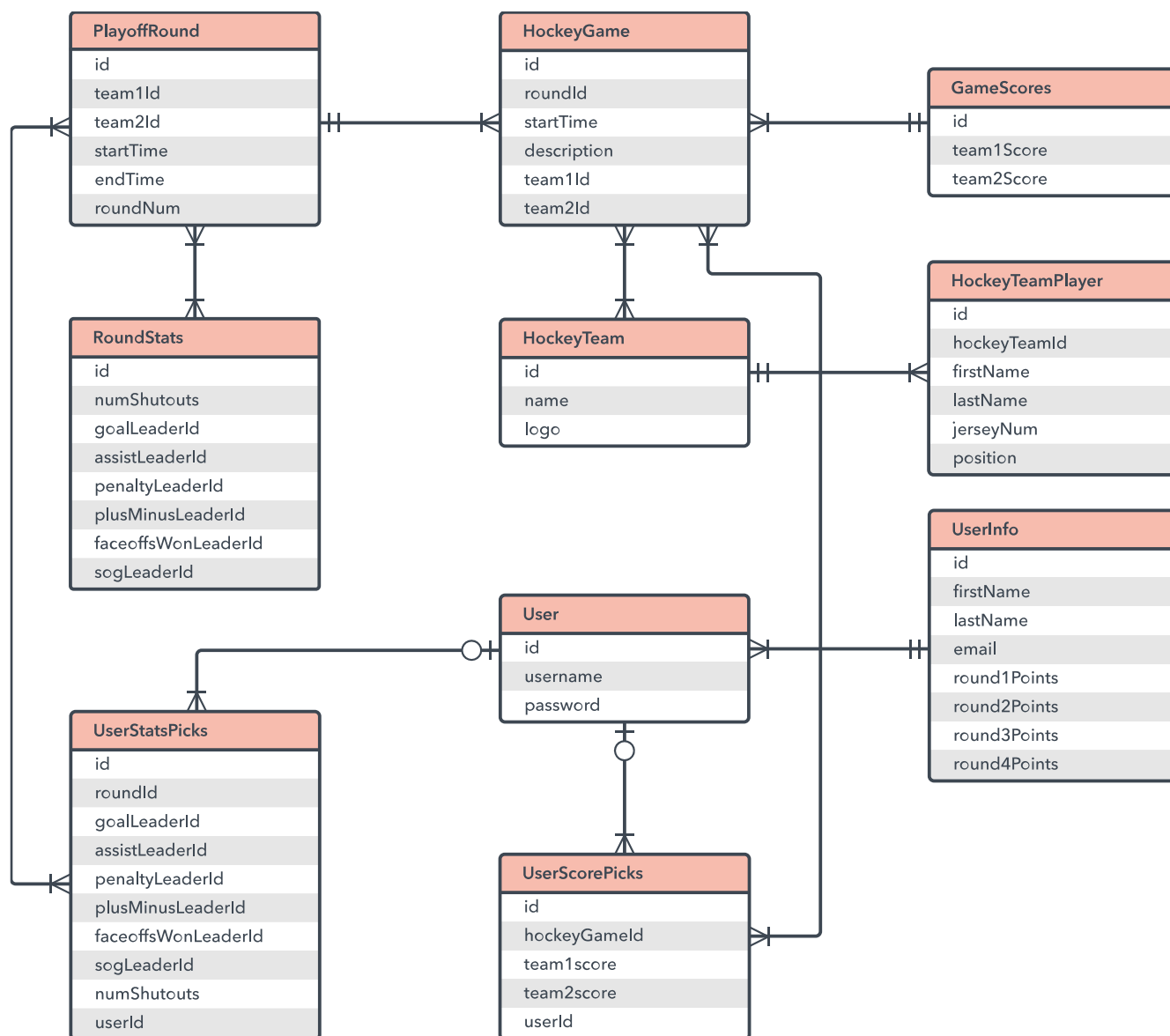
```
app.use(express.json());
```

Para registrar la transacción de mensajes por consola, se puede utilizar una herramienta útil llamada Morgan, que ayuda a verificar en consola lo que el usuario está solicitando. Esta herramienta es opcional para el desarrollo de la aplicación, no obstante, se deja la línea de comando para su instalación y utilización: `npm install morgan`. `app.use(morgan('dev'));`

1.3. Configuración e instalación de la base de datos MongoDB

A continuación, se realizará la conexión de la aplicación del servidor a la base de datos, por lo que se inicia configurando e instalando los recursos necesarios antes de desarrollar. En ese sentido, se instala MongoDB de manera persistente en el equipo de desarrollo. La siguiente figura representa de manera general el diagrama entidad-relación de una base de datos.

Figura 2. Diagrama entidad-relación



De manera opcional, se puede instalar la extensión de MongoDB dentro del entorno de VS Code, permitiendo una mayor integralidad en el desarrollo. Estas herramientas facilitan la celeridad en el proceso de construcción de sistemas de software. Por lo tanto, en el material de apoyo se dejará información para acceder a la guía de instalación de esta extensión, denominada extensión VS Code y MongoDB.

localhost:27017 connected

- admin
- config
- empleados
- local

Posteriormente, para realizar la conexión a la base de datos y definir la estructura de los datos de la aplicación a desarrollar, conocida técnicamente como esquemas, se instala la herramienta Mongoose con la siguiente línea de comando desde la terminal de VS Code: `npm install mongoose`

1.4. Configuración de database.js

Con todas las herramientas necesarias para la conexión de la aplicación a la base de datos, se empieza a desarrollar y configurar el archivo `database.js`

Con respecto al código del archivo `database.js`, se indica que es necesario que el módulo se conecte a MongoDB, por eso se utiliza Mongoose. Al llamarse el módulo, se almacena en una constante. Además, se define la dirección de la conexión de la base de datos en una constante denominada `URI`, y se utiliza una promesa para obtener la información de la base de datos y corroborar si hay conexión o no.

```
const mongoose = require('mongoose');  
  
const URI = 'mongodb://localhost/empleados';  
  
mongoose.connect(URI)  
  
  .then(db => console.log('DB is connected'))
```

```
.catch(err => console.error(err));
```

```
module.exports = mongoose;
```

Por ser un módulo que se utiliza en toda la aplicación, se exporta. De esta forma, la constante Mongoose devolverá la conexión. De manera adicional, en el archivo index.js se invoca o llama el archivo database.js, especificando que lo que se requiere es la conexión. Solo resta probar si la conexión está activa, por lo que se inicia el servidor con la línea de comando `npm run dev`, como se indicó anteriormente.

A continuación, se presenta el resultado de la terminal del VS Code:

```
[nodemon] restarting due to changes...
```

```
[nodemon] starting `node backend/index.js`
```

```
(node:78784) DeprecationWarning: current URL string parser is deprecated, and
will be removed in a future version. To use the new parser, pass option {
useNewUrlParser: true } to MongoClient.connect.
```

```
(Use `node --trace-deprecation ...` to show where the warning was created)
```

```
(node:78784) [MONGODB DRIVER] Warning: Current Server Discovery and
Monitoring engine is deprecated, and will be removed in a future version. To use
the new Server Discover and Monitoring engine, pass option {
useUnifiedTopology: true } to the MongoClient constructor.
```

```
server activo en el puerto 3000
```

```
DB is connected
```

1.5. Configuración de las rutas del proyecto con Node.js

El siguiente paso es configurar las rutas o URL por donde se van a enviar o recibir los datos, por lo que se crea un archivo llamado `empleado.route.js` dentro de la carpeta `routes`. En este primer apartado, se desea revisar el funcionamiento de las rutas, por lo que, a manera de prueba, se presenta el siguiente código:

```
/**
 * Vamos a crear rutas del servidor
 * creamos un módulo por eso utilizamos express
 * vamos a utilizar como nuestra rest api para
 * enviar y recibir datos en formato json
 */
const express = require('express');
const router = express.Router();

/* generamos un ejemplo cuando le soliciten algo al span servidor por el método
GET */
router.get('/', (req, res) => {
  res.json({
    status: 'API REST funcionando'
  });
})
```



```
module.exports = router;
```

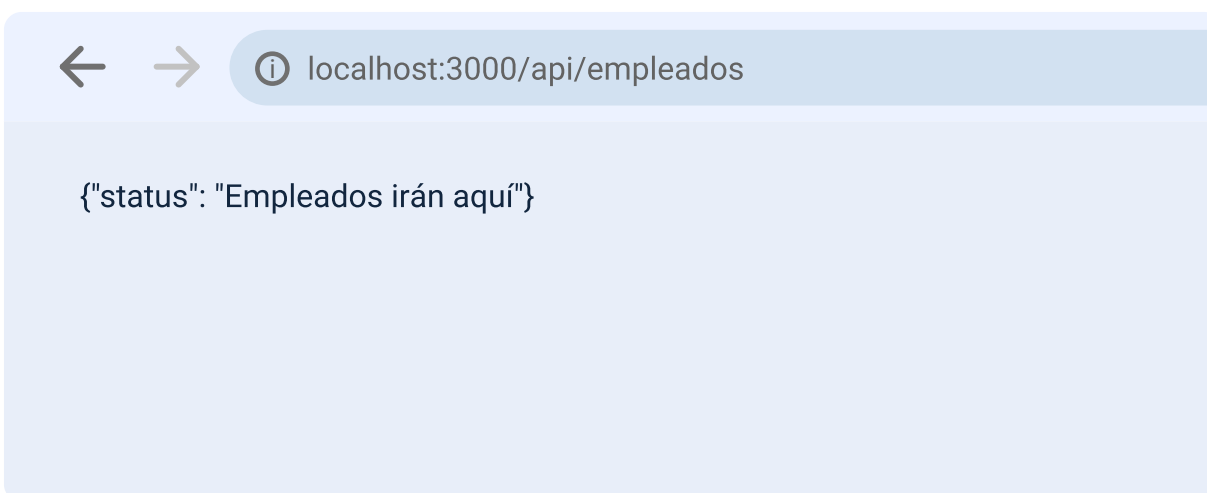
De manera adicional, se puede observar el archivo `index.js`, en el cual se establece la ruta por donde se van a recibir o enviar los datos desde el servidor. Para este caso de estudio, será <http://localhost:3000/api/empleados>

Para la definición de esta ruta, es necesario requerir el archivo `empleado.routes.js` ejecutando la siguiente instrucción:

```
app.use('/api/empleados', require('./routes/empleado.routes'));
```

Al hacer el llamado en el servidor, se vería de la siguiente manera:

Figura 3. Respuesta del servidor



Ya visto un ejemplo del funcionamiento de las rutas, es necesario comprender su comportamiento. Cada solicitud debe ir asociada a una ruta, y esta a su vez a un intercambio de información o de datos, es decir, de interacción con la base de datos. En ese sentido, es más que necesaria la construcción del modelo `empleado.js` con datos específicos para los empleados dentro de la carpeta `models`.

Para construir el modelo empleado.js, primero se debe crear un esquema que defina la estructura de los datos que se almacenarán en la base de datos. Aquí un ejemplo de cómo podría verse el archivo empleado.js:

```
const mongoose = require('mongoose');
```

```
const {Schema} = mongoose;
```

```
const EmpleadoSchema = new Schema({
```

```
  name: {type: String, require: true},
```

```
  position: {type: String, require: true},
```

```
  office: {type: String, require: true},
```

```
  salary: {type: Number, require: true},
```

```
});
```

```
module.exports = mongoose.model('Empleado', EmpleadoSchema);
```

Se requiere Mongoose y se almacena en una constante denominada de la misma forma, que se utilizará para definir los esquemas de datos. Para esto, se empieza a modelar con la información, como nombre y cargo, entre otros. En ese sentido, Mongoose le indica a MongoDB cómo va a lucir la estructura de empleados.

1.6. Desarrollo de controladores para la API REST

Para mantener una estructura organizada y facilitar el crecimiento de la aplicación, se recomienda el uso de controladores. Los controladores son responsables de definir los métodos y acciones necesarios para las rutas de la aplicación. En este caso, se creará el archivo `empleado.controller.js` dentro de la carpeta `controllers`.

A continuación, se presenta un ejemplo de cómo podría estructurarse el controlador para manejar las operaciones básicas de la API REST:

```
/**
 * Se coloca el controlador como un objeto y luego se exporta como
 * se requiere primero el modelo empleado
 */
const Empleado = require('../models/empleado');

const empleadoCtrl = {};

/**
 * DEFINO LOS MÉTODOS
 */

// Obtener todos los empleados
empleadoCtrl.getEmpleados = async (req, res) => {
  const empleados = await Empleado.find();
  res.json(empleados);
}
```

```
}

// Crear empleados

empleadoCtrl.createEmpleados = async (req, res) => {

  const empleado = new Empleado(req.body);

  await empleado.save();

  res.json({

    'status': 'Empleado guardado'

  });

}

// Conseguir un único empleado

empleadoCtrl.getUnicoEmpleado = async (req, res) => {

  const empleadoUnico = await Empleado.findById(req.params.id);

  res.json(empleadoUnico);

}


// Actualizar empleado

empleadoCtrl.editarEmpleado = async (req, res) => {

  const { id } = req.params;

  const empleadoEdit = {
```

```
name: req.body.name,

position: req.body.position,

office: req.body.office,

salary: req.body.salary

};

await Empleado.findByIdAndUpdate(id, {$set: empleadoEdit}, {new: true});

res.json({status: 'Empleado Actualizado'});

}

// Eliminar empleado

empleadoCtrl.eliminarEmpleado = async (req, res) => {

  await Empleado.findByIdAndDelete(req.params.id);

  res.json({status: 'Empleado Eliminado'});

}

// exporto el módulo

module.exports = empleadoCtrl;
```

El desarrollo de controladores para la API REST se realiza:

- **Explicación del código**

Se inicia explicando de manera sucinta el código presentado anteriormente. En primera instancia, se requiere el modelo realizado denominado `empleado.js`, por lo cual se almacena en una constante `Empleado`.

- **Uso de funciones JavaScript**

En cada uno de los métodos se utilizan funciones nativas y nuevas de JavaScript. Anteriormente, se utilizaban funciones de tipo callback que existían como parámetros de los métodos, después aparecieron las promesas con la función `then`, pero ahora existen funciones que están dedicadas a realizar de manera más eficiente el procesamiento de los datos y mejorar los tiempos de respuesta como lo son `async` y `await` que hacen parte de la última versión de JavaScript.

- **Método obtener empleados**

Para el método de obtener empleados se utiliza la función `async`, la cual entrega en formato JSON todo lo que encuentre de empleados. Se utiliza la función `await`, porque se espera que tarde un poco en buscar. Se utiliza uno de los métodos intrínsecos del modelo `Empleado`, que en este caso es `find()`.

- **Método crear empleado**

El método de crear empleado tiene una naturaleza similar al anterior. En este método se crea una nueva instancia de empleado `new Empleado`. Luego, se guarda ese nuevo registro con `save()`. Es importante notar que hasta el momento no se ha creado la base de datos en MongoDB, pero al almacenar un registro, se crea la base de datos.

- **Método editar empleado**

Para el método de editar empleado se presenta una sintaxis interesante. Primero, se escribe de otra forma para capturar el id del usuario que se quiere actualizar { id }. Además, se crea una constante empleadoEdit para pasarle los datos traídos del cliente y finalmente se utiliza await para esperar en la acción de encontrar por id y actualizar a través de la función set, que se utilizará para modificar los datos en última instancia.

- **Exportación del módulo**

Finalmente, al crear un módulo, se exporta para que sea utilizado en cualquier parte de la aplicación.

A continuación, se definen las rutas, por lo que se configura y desarrolla la lógica de esta sección en el archivo empleados.routes.js dentro de la carpeta routes:

```
plaintext
```

```
/**
```

```
*
```

```
* creamos un módulo por eso utilizamos express
```

```
*/
```

```
const express = require('express');
```

```
const router = express.Router();
```

```
const empleadoCtrl = require('../controllers/empleado.controller');
```

```
router.get('/', empleadoCtrl.getEmpleados); // Rutas más limpias (obtener  
empleados)  
  
router.post('/', empleadoCtrl.createEmpleados); // guardar  
  
router.get('/:id', empleadoCtrl.getUnicoEmpleado); // obtiene un único empleado  
  
router.put('/:id', empleadoCtrl.editarEmpleado); // Actualizar datos (uno a la vez)  
  
router.delete('/:id', empleadoCtrl.eliminarEmpleado);  
  
  
module.exports = router;
```

Con respecto al código anterior, se observa que se requiere el uso del controlador, por lo que se almacena en una constante llamada `empleadoCtrl`, y de esta forma se pueden utilizar todos los métodos de este, es decir, la gestión o CRUD sobre el empleado.

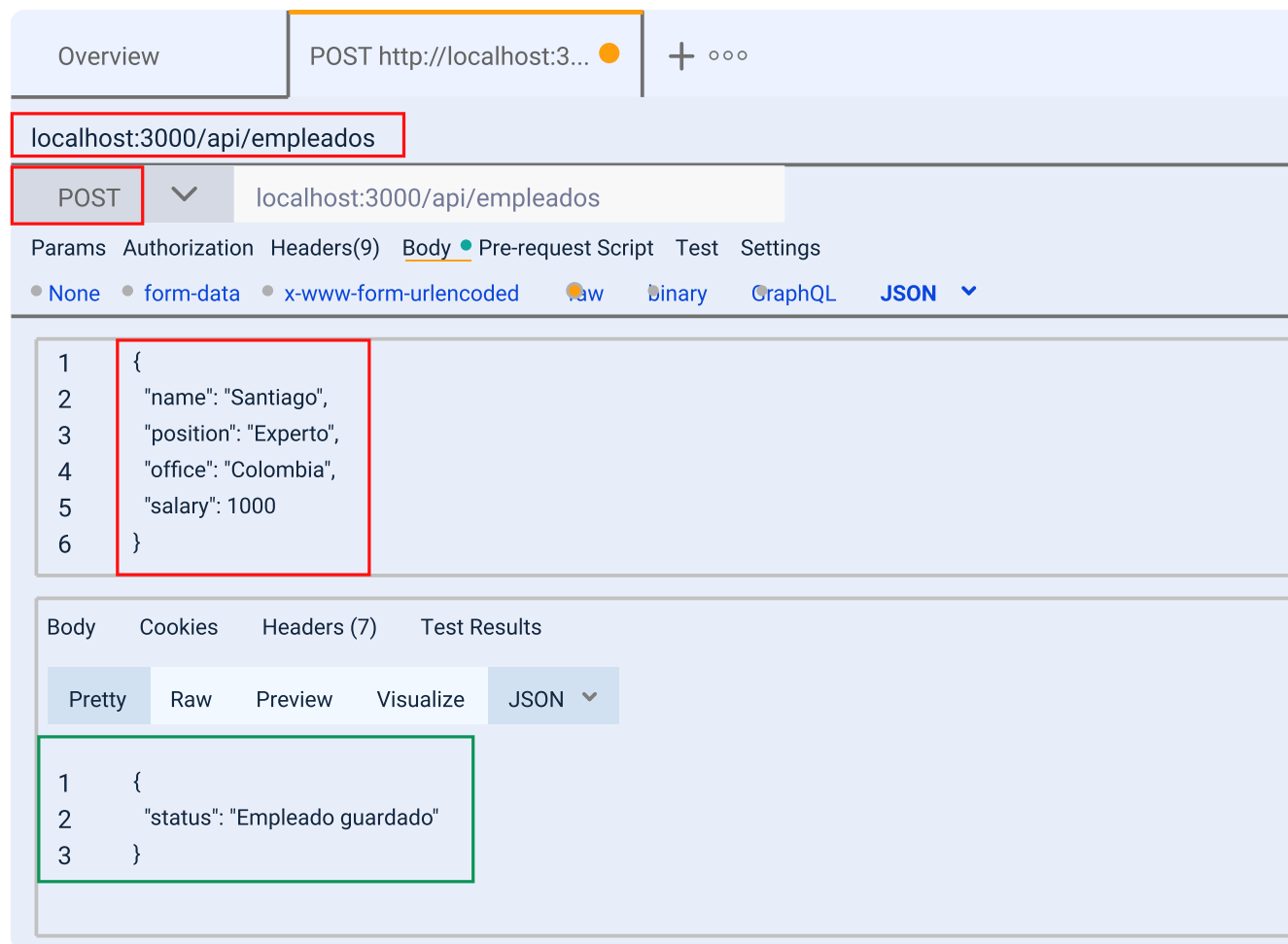
En ese sentido, se observa que el objeto `router` llama a cada uno de los métodos por defecto de petición HTTP como GET, POST, PUT, entre otros, asociados a los métodos realizados con el controlador. De manera que se tienen rutas mucho más limpias y apoyadas con la buena práctica del uso del controlador.

1.7. Realización de pruebas de la API REST con Postman

Una vez finalizada la construcción del modelo, controlador y rutas, se procede a realizar las pruebas respectivas para evaluar el funcionamiento de la API REST. Para ello, se utilizará la herramienta Postman.

Se comienza insertando datos utilizando el método POST, el cual llamará a la ruta para crear empleados.

Figura 4. Pruebas con Postman

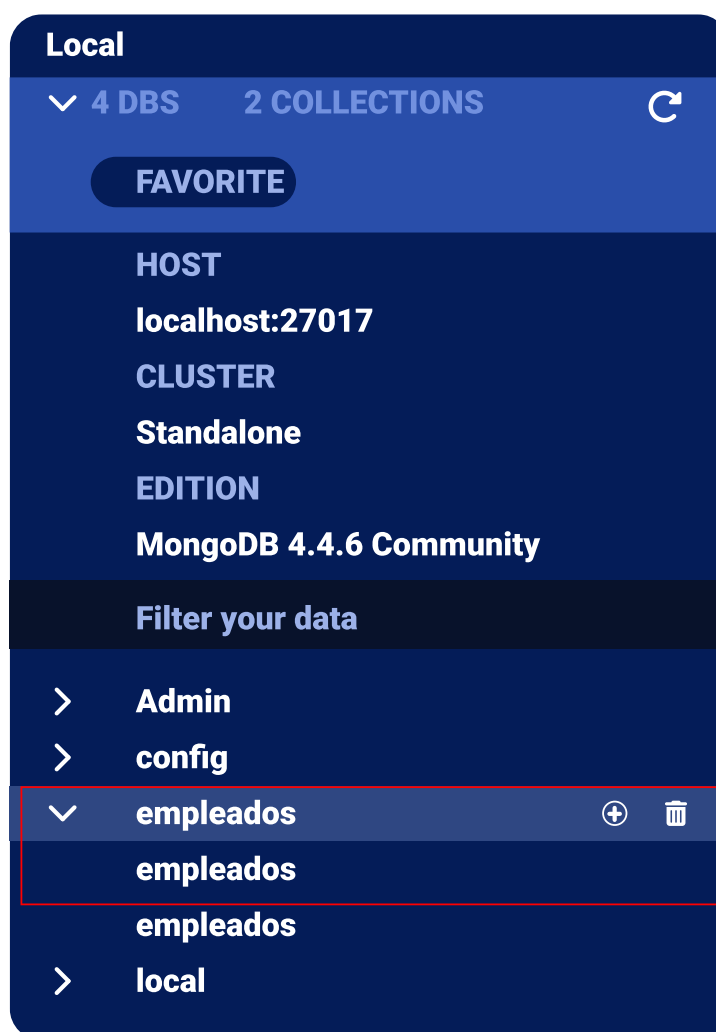


En la figura anterior se pueden apreciar los datos que se envían al servidor, dado que Postman permite realizar esa emulación del cliente. Los datos que se envían están acordes al esquema definido en el modelo de empleado.js, como lo son `name`, `position`, `office` y `salary`. La respuesta en consola, que se programa como `status`, es que el Empleado ha sido guardado.

En este orden de ideas, es necesario revisar la base de datos y verificar que efectivamente se ha guardado la información y si realmente MongoDB la ha creado.

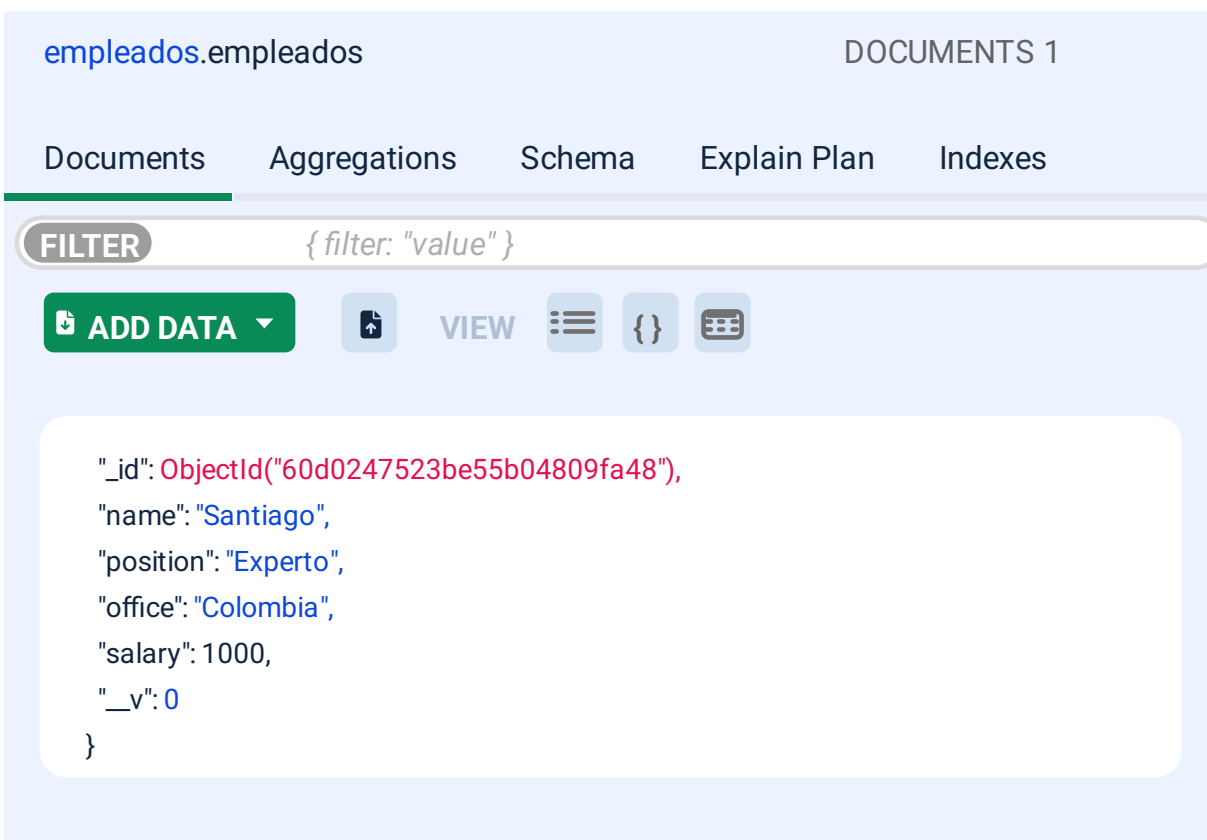
En la siguiente figura se presenta la imagen que representa la interfaz gráfica de MongoDB y el resultado de verificación de la creación de la base de datos.

Figura 5. Panel MongoDB



La siguiente figura muestra el resultado de la verificación de un registro de empleado almacenado en MongoDB.

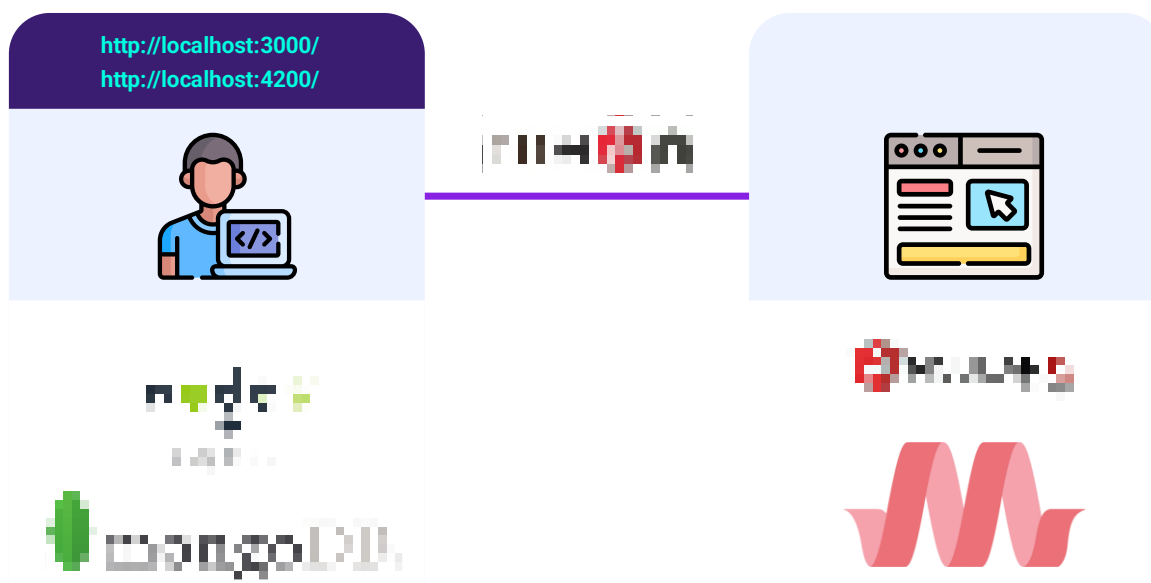
Figura 6. Registro de empleado almacenado en MongoDB



Como es evidente, la API REST funciona correctamente. Se ha creado la base de datos empleado y se tiene almacenado el primer registro del empleado Santiago. De igual forma, se confirma que los demás métodos HTTP, como GET, PUT o DELETE, también funcionan. Por lo tanto, se realiza la invitación a probar la API con Postman.

Hasta este punto se ha realizado el desarrollo del back-end, por lo que se tiene la base de datos lista para proceder con la construcción del front-end, que será el que interactuará con el servidor. A continuación, se presenta una figura que ilustra la estructura general del proyecto.

Figura 7. Resumen estructura del proyecto web



2. Construcción del front-end de la aplicación web

Para esta sección se plantea generar una sola vista, que corresponde al formulario de registro de empleados. Por esta razón, no es necesario generar sketch o maquetación para evaluar la navegación entre distintos componentes. No obstante, se emplean buenas prácticas de diseño amparadas en las técnicas y tecnologías como Materialize, que dará alcance a varias reglas de estilo para los sistemas interactivos de software.

plaintext

```
PS D:\gestion-empleados> npm install -g @angular/cli
```

```
npm WARN deprecated request@2.88.2: request has been deprecated, see  
https://github.com/request/request/issues/3142
```

```
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
```

```
npm WARN deprecated uuid@3.4.0: Please upgrade to version 7 or higher. See  
https://v8.dev/blog/math-random for details.
```

2.1. Instalación y configuración de AngularJS

A continuación, se presenta la estructura de directorios del proyecto Angular creado con el comando `ng new frontend`:

GESTION-EMPLEADOS

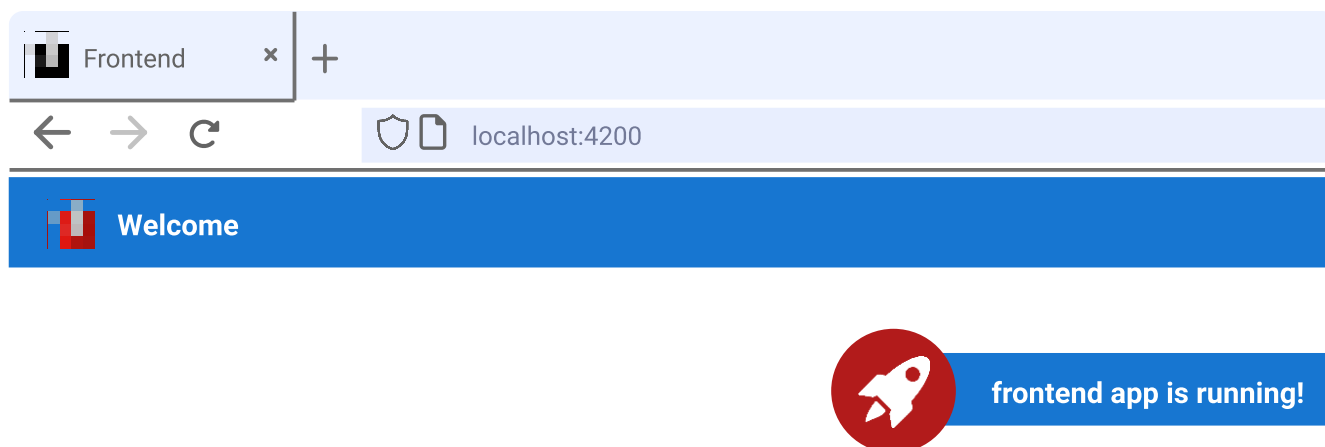
└─ backend

└─ frontend

```
| ├── node_modules
| ├── src
| ├── .browserslistrc
| ├── .editorconfig
| ├── .gitignore
| ├── angular.json
| ├── karma.conf.js
| ├── package-lock.json
| ├── package.json
| ├── README.md
| ├── tsconfig.app.json
| ├── tsconfig.json
| └── tsconfig.spec.json
```

A continuación, se debe ingresar a la carpeta del front-end e iniciar el servidor digitando en la terminal del VS Code la siguiente línea de comandos: `ng serve`. Es importante recordar que para ingresar por la terminal a la carpeta de front-end se hace con el comando `cd`, es decir, `cd Frontend/`. Una vez dentro de la carpeta front-end se ejecuta el comando `ng serve`, lo que inicia un servidor de desarrollo para Angular. En ese sentido, el servidor se inicializa en el puerto 4200 (<http://localhost:4200/>), como se representa en la siguiente figura:

Figura 8. Estado servidor Front-end



2.2. Generación de código con ng

Después de iniciar el servidor Angular, se revisa la estructura de directorios de este framework. Aunque hay muchos archivos, es necesario concentrarse en la carpeta ubicada en la ruta frontend/src/app. En la carpeta app se realizará todo el código necesario para el proyecto.

Adicionalmente, para dar coherencia al proyecto, dentro de la carpeta app se crean tres carpetas más: models, services y components, para tener bien definido el proyecto.

GESTION-EMPLEADOS

└─ backend

└─ frontend

| └─ node_modules

| └─ src

| | └─ app

```
| | | └─ components
| | | └─ models
| | | └─ services
| | | └─ app-routing.module.ts
| | | └─ app.component.css
| | | └─ app.component.html
| | | └─ app.component.spec.ts
| | | └─ app.component.ts
| | | └─ app.module.ts
```

Haciendo uso de una de las características del framework, se genera el código ingresando a las carpetas que se acaban de crear y en cada una se ejecutan las siguientes líneas de código respectivamente:

- **Generar componente empleados**

Dentro de la carpeta components se genera un componente llamado empleados con Angular CLI: `ng g c empleados`.

- **Crear clase empleado**

Dentro de la carpeta models se crea la clase empleado con: `ng g class empleado`.

- **Crear servicio empleado**

Dentro de la carpeta services se crea un servicio llamado empleado con la siguiente línea: ng g s empleado.

Para mayor precisión con relación a las instrucciones ejecutadas, a continuación se presenta la evidencia de las operaciones realizadas:

```
PS D:\gestion-empleados\frontend> cd .\src\app\components\
```

```
PS D:\gestion-empleados\frontend\src\app\components> ng g c empleados
```

```
CREATE src/app/components/empleados/empleados.component.html (24 bytes)
```

```
CREATE src/app/components/empleados/empleados.component.spec.ts (647  
bytes)
```

```
CREATE src/app/components/empleados/empleados.component.ts (287 bytes)
```

```
CREATE src/app/components/empleados/empleados.component.css (0 bytes)
```

```
UPDATE src/app/app.module.ts (498 bytes)
```

```
empleados.component.css
```

```
empleados.component.html
```

```
empleados.component.spec.ts
```

```
empleados.component.ts
```

2.3. Incorporación de Materialize al proyecto web

Se incorpora Materialize al proyecto mediante la inclusión de los CDN (red de distribución de contenidos) de hojas de estilos, JavaScript e íconos en el documento index.html del Front-end, por lo que la ruta al archivo es FrontEnd/src/index.html

```
html

<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>FrontEnd</title>

  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">

  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css"
>

  <link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">

</head>

<body>
```

```
<app-root></app-root>
```

```
<script
```

```
src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>
```

```
</body>
```

```
</html>
```

2.4. Configuración de app.component.html

Ahora se inicia con la modificación del archivo app.component.html, dado que en este archivo se llaman los componentes de la aplicación que, para el caso de estudio propuesto, se denomina empleados.

```
html
```

```
<nav class="nav-wrapper blue">
```

```
<div class="container">
```

```
<a href="/" class="brand-logo">GESTIÓN DE EMPLEADOS</a>
```

```
</div>
```

```
</nav>
```

```
<div class="container p-4">
```

```
<app-empleados></app-empleados>
```

```
</div>
```

El código se evidencia que se adiciona la etiqueta <app-empleados>, indicando que se hará uso de ese componente.

2.5. Configuración de los módulos `app.module.ts`

A continuación, se configura uno de los archivos más importantes del proyecto, el cual se encarga de cargar los módulos de la aplicación. Es necesario entender la naturaleza de la aplicación que se está desarrollando, ya que esto permitirá definir qué módulos son necesarios para el óptimo funcionamiento de la aplicación.

En este caso de estudio, la aplicación permitirá la adición de empleados utilizando un API REST, por lo que es evidente el uso de un formulario en la interfaz, además de la comunicación que debe tener el front-end con el servidor.

typescript

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { FormsModule } from '@angular/forms';

import { HttpClientModule } from '@angular/common/http';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { EmpleadosComponent } from
'./components/empleados/empleados.component';
```

```
@NgModule({
```

```
declarations: [  
  
  AppComponent,  
  
  EmpleadosComponent  
  
],  
  
imports: [  
  
  BrowserModule,  
  
  FormsModule,  
  
  AppRoutingModule,  
  
  HttpClientModule  
  
],  
  
providers: [],  
  
bootstrap: [AppComponent]  
  
})  
  
export class AppModule { }
```

2.6. Configuración del modelo empleado.ts

Una vez realizada la configuración en los módulos de la aplicación, se inicia la construcción del modelo de la aplicación. La codificación se realiza dentro del archivo `empleado.ts`, cuya ruta es `models/empleado.ts`.

Dentro de este archivo se define la estructura de datos que está asociada a lo que se describió en el back-end.

typescript

```
export class Empleado {  
  
    constructor(_id = "", name = "", position = "", office = "", salary = 0) {  
  
        this._id = _id;  
  
        this.name = name;  
  
        this.position = position;  
  
        this.office = office;  
  
        this.salary = salary;  
  
    }  
  
    _id: string; // Sub guión id porque los datos van a venir de MongoDB  
  
    name: string;  
  
    position: string;  
  
    office: string;  
  
    salary: number;  
  
}
```

En el código anterior, se definen los parámetros que serán utilizados para el envío de datos hacia el servidor; por lo tanto, se comienza con la inicialización de las variables a través de un constructor y con la declaración del tipo de datos de cada una.

2.7. Creación del servicio `empleado.service.ts`

El objetivo general de esta parte del proyecto es agregar los datos ya definidos a la REST API. Para ello, se enfoca en generar el servicio necesario, desarrollando el archivo `empleado.service.ts`. Este servicio permite definir los métodos reutilizables en cualquier parte de la aplicación. En este caso, se trata de funciones que gestionan los datos, como agregar, eliminar y editar, entre otros.

A continuación, se presenta el código del archivo `empleado.service.ts`.

typescript

```
import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Empleado } from '../models/empleado';

@Injectable({
  providedIn: 'root'
})

export class EmpleadoService {

  selectedEmpleado: Empleado;
```

```
empleados: Empleado[];

readonly URL_API = 'http://localhost:3000/api/empleados';

constructor(private http: HttpClient) {

    this.selectedEmpleado = new Empleado();

    this.empleados = [];

}

getEmpleados() {

    return this.http.get(this.URL_API);

}

postEmpleado(Empleado: Empleado) {

    return this.http.post(this.URL_API, Empleado);

}

putEmpleado(Empleado: Empleado) {

    return this.http.put(this.URL_API + `/${Empleado._id}`, Empleado);

}

deleteEmpleado(_id: string) {

    // Solo se necesita el id, no todo lo del empleado

    return this.http.delete(this.URL_API + `/${_id}`); // utilizamos el método delete

}
```



```
}
```

En el código anterior se resalta la importancia de haber definido primeramente el módulo `empleado.ts`, ya que esto establece una estructura clara de datos. Por esta razón, se importa el modelo y se asegura la comunicación con el servidor a través de la modificación del archivo `app.modules.ts` con la incorporación del módulo `HTTPClient`.

Además, se establece como atributo de solo lectura la URL de la API a la cual se conectará el cliente, en este caso, la URL del servidor ya configurada en la primera sección de este componente es <http://localhost:3000/empleados>.

A continuación, se procede con la definición de los métodos para la gestión de los datos de empleado, generando así cuatro métodos: `getEmpleados`, `postEmpleados`, `putEmpleado` y `deleteEmpleado`. Para el caso de estudio, se analiza el método `postEmpleado`, que recibe como parámetro un objeto de tipo `Empleado`. El retorno es la adición de un nuevo empleado (se envían todos los datos) a través del método HTTP POST:

```
return this.http.post(this.URL_API, Empleado);
```

Es fundamental asegurarse de que en cada método se envíe la URL del servidor y los datos necesarios en relación con el método correspondiente.

2.8. Desarrollo del componente `empleados.component.ts`

Definidos los servicios, es necesario abordar la lógica del componente. A continuación, se enfoca en el archivo `empleados.component.ts`. El código del archivo se presenta a continuación:

```
import { Component, OnInit } from '@angular/core';
```

```
import { EmpleadoService } from '../services/empleado.service';

import { NgForm } from '@angular/forms';

import { Empleado } from '../models/empleado';

declare var M: any;

@Component({
  selector: 'app-empleados',
  templateUrl: './empleados.component.html',
  styleUrls: ['./empleados.component.css'],
  providers: [EmpleadoService]
})

export class EmpleadosComponent implements OnInit {

  constructor(public empleadoService: EmpleadoService) { }

  ngOnInit(): void {

  }
}
```

```
agregarEmpleado(form?: NgForm) {  
  
    this.empleadoService.postEmpleado(form?.value)  
  
    .subscribe(res => {  
  
        this.resetForm(form);  
  
        M.toast({ html: 'Guardado satisfactoriamente' });  
  
    });  
  
}  
  
  
resetForm(form?: NgForm) {  
  
    // Limpiar el formulario, recibe un formulario como parámetro  
  
    if (form) {  
  
        form.reset();  
  
        this.empleadoService.selectedEmpleado = new Empleado();  
  
    }  
  
}  
  
}
```

En el código anterior se destaca la importancia de haber creado y configurado previamente los módulos y archivos para la construcción de este componente. Para su funcionamiento adecuado, es necesario contar con EmpleadoService, el modelo Empleado y el módulo ya definido como NgForm.

Una vez se importan estos módulos, se enfocan los esfuerzos en los métodos. Para el caso de estudio, se destaca el método de agregar empleado. Este método toma como parámetro la existencia de datos del formulario y, si es verdadero, llama al método del servicio empleado (previamente construido) `PostEmpleado`, entregando los valores para que sean enviados al servidor y almacenados en la base de datos de MongoDB.

Además, como ejemplo de este componente, se añade un plus: en cada adición de datos en el formulario, este se puede limpiar automáticamente mediante la función `resetForm`, que resetea o limpia los campos del formulario utilizando la función intrínseca `reset()`.

Finalmente, para mejorar la interfaz, se agrega un complemento de Materialize llamado `Toast`, que no es más que una notificación que se utiliza para indicar al usuario que el empleado se ha guardado correctamente.

2.9. Desarrollo de la interfaz HTML `empleados.component.html`

Después de definir la lógica del componente empleado, lo único que falta es la construcción del formulario. Por lo tanto, se enfoca ahora en el desarrollo del archivo `empleados.component.html`, que se presenta a continuación:

```
<div class="container">

  <div class="row">

    <div class="col s5">

      <div class="card">

        <div class="card-content">
```

```
<form #empleadoForm="ngForm"
  (ngSubmit)="agregarEmpleado(empleadoForm)">

  <div class="row">

    <div class="input-field col s12">

      <input type="text" name="name" #name="ngModel"
        [(ngModel)]= "empleadoService.selectedEmpleado.name" placeholder="Ingrese su
        nombre">

    </div>

    <div class="input-field col s12">

      <input type="text" name="position" #position="ngModel"
        [(ngModel)]= "empleadoService.selectedEmpleado.position" placeholder="Ingrese su
        cargo">

    </div>

    <div class="input-field col s12">

      <input type="text" name="office" #office="ngModel"
        [(ngModel)]= "empleadoService.selectedEmpleado.office" placeholder="Ingrese su
        lugar de trabajo">

    </div>

    <div class="input-field col s12">
```

```
<input type="text" name="salary" #salary="ngModel"
[(ngModel)]= "empleadoService.selectedEmpleado.salary" placeholder="Ingrese su
salario">

</div>

<div class="card-action">

  <div class="input-field col s12">

    <button class="btn right"
(click)="resetForm(empleadoForm)">Limpiar</button>

    <button class="btn">Guardar</button>

  </div>

</div>

</div>

</div>

</form>

</div>

</div>

</div>

<div class="col s7">

</div>

</div>

</div>
```

En el código anterior se presenta la estructura base de un formulario en HTML, el cual se adhiere a clases y reglas provistas por Materialize para mejorar la apariencia de la interfaz. Además, se muestran cuatro inputs en los cuales el usuario debe ingresar datos, y al final se presentan dos botones: uno para enviar el formulario y otro para limpiarlo. A continuación, se destaca una línea de código específica:

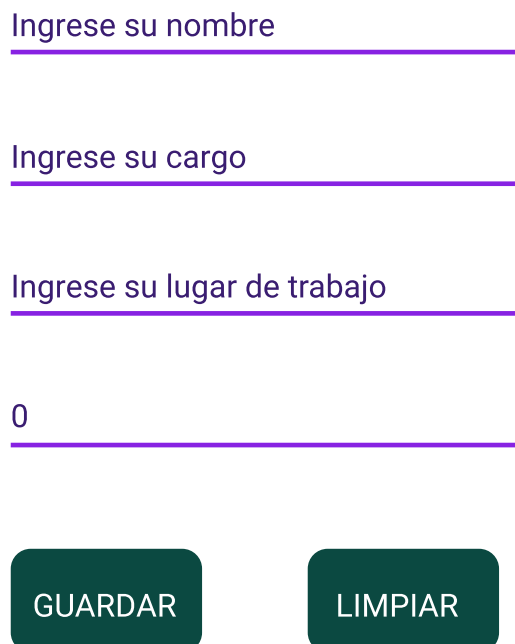
```
<form #empleadoForm="ngForm" (ngSubmit)="agregarEmpleado(empleadoForm)">
```

Esta línea define que todos los campos del formulario serán almacenados en una sola variable (empleadoForm), y estos campos serán procesados por el método agregarEmpleado, método que fue definido anteriormente en el componente de empleado.

2.10. Probando la aplicación completa.

Proyecto web terminado Finalizada la codificación de cada uno de los archivos del front-end, es necesario revisar cómo ha quedado la aplicación. Sin embargo, este paso no se realiza solo al final, sino que, debido a la concisión de este componente, se presenta en esta instancia. Como buena práctica, en cada bloque de desarrollo o línea se debe revisar el resultado obtenido, es decir, trabajar de manera paralela. Una muestra de la interfaz de la aplicación se ilustra en la siguiente figura:

Figura 9. Registro empleados

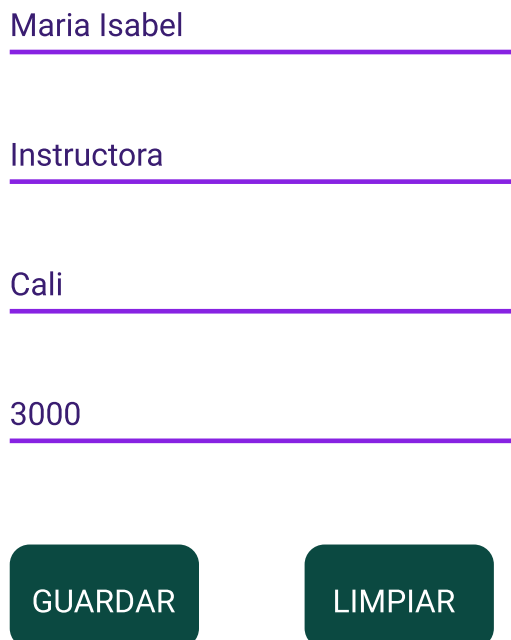


Formulario de registro de empleados:

- Caja de texto con placeholder: "Ingrese su nombre"
- Caja de texto con placeholder: "Ingrese su cargo"
- Caja de texto con placeholder: "Ingrese su lugar de trabajo"
- Caja de texto con placeholder: "0"
- Botón: GUARDAR
- Botón: LIMPIAR

En la muestra anterior de la interfaz se ilustra un formulario en el cual se solicitan los datos al usuario para el registro de un empleado. Es importante notar el uso de los placeholders en cada caja de texto, cuya función es guiar o indicar qué tipo de datos se deben registrar. De esta manera, la interfaz quedaría como se muestra en la siguiente figura:

Figura 10. Registro empleado



Maria Isabel

Instructora

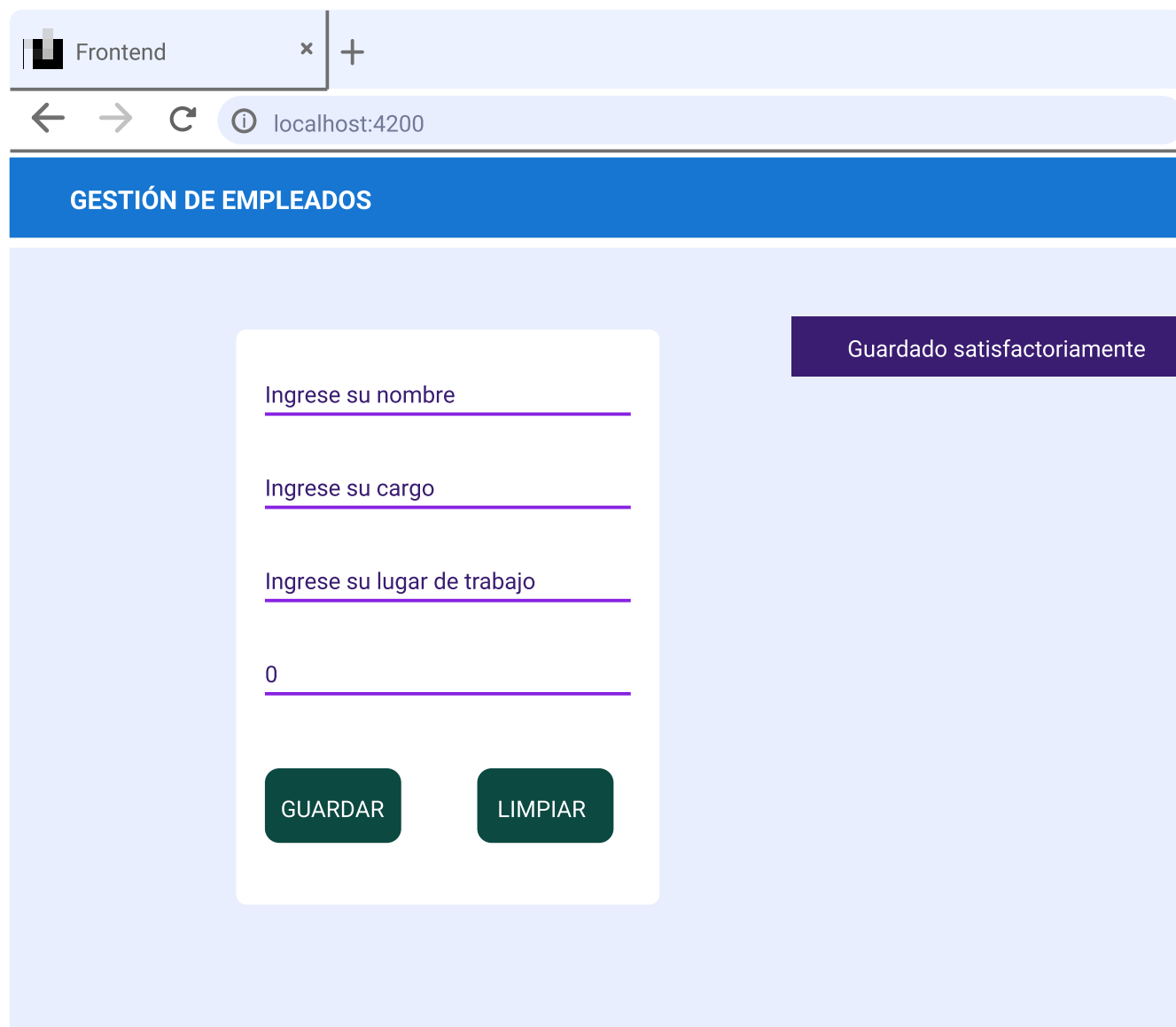
Cali

3000

GUARDAR LIMPIAR

Al momento de dar clic en guardar, se almacena la información y se muestra la notificación de Materialize Toast, indicando que el usuario se ha guardado exitosamente. Además, se limpia el formulario mediante la función `reset()`, tal como se aprecia en la figura.

Figura 11. Notificación Toast de Materialize



The screenshot shows a web browser window with a single tab titled 'Frontend'. The address bar displays 'localhost:4200'. The page has a blue header with the text 'GESTIÓN DE EMPLEADOS'. In the center, there is a white form with four input fields: 'Ingrese su nombre', 'Ingrese su cargo', 'Ingrese su lugar de trabajo', and a numeric field containing '0'. Below the fields are two buttons: 'GUARDAR' and 'LIMPIAR'. A dark purple toast notification in the top right corner reads 'Guardado satisfactoriamente'.

Para verificar que los datos incorporados en el formulario se hayan almacenado correctamente en la base de datos empleado de MongoDB, se utilizará el panel de control de MongoDB, como se aprecia en los siguientes códigos:

```
json
```

```
{
```

```
"_id": ObjectId("60d2aeaa21a176987f0157485"),  
  
"name": "Julian",  
  
"position": "Ingeniero",  
  
"office": "Cali",  
  
"salary": 5000,  
  
"__v": 0  
}  
  
{  
  
  "_id": ObjectId("60d2aeaa21a176987f0157487"),  
  
  "name": "Julian",  
  
  "position": "Ingeniero",  
  
  "office": "Cali",  
  
  "salary": 5000,  
  
  "__v": 0  
}  
  
{  
  
  "_id": ObjectId("60d2aeaa21a176987f0157489"),  
  
  "name": "Jaime",  
  
  "position": "Oficial",
```

```
"office": "Juzgados",  
  
"salary": 8000,  
  
"__v": 0  
}  
  
{  
  
  "_id": ObjectId("60d64392eadf31b1d99e64f4"),  
  
  "name": "María Isabel",  
  
  "position": "Instructora",  
  
  "office": "Cali",  
  
  "salary": 3000,  
  
  "__v": 0  
}
```

Evidencia de registro en MongoDB:

```
{  
  
  "_id": ObjectId("60d64392eadf31b1d99e64f4"),  
  
  "name": "María Isabel",  
  
  "position": "Instructora",  
  
  "office": "Cali",  
  
  "salary": 3000,
```

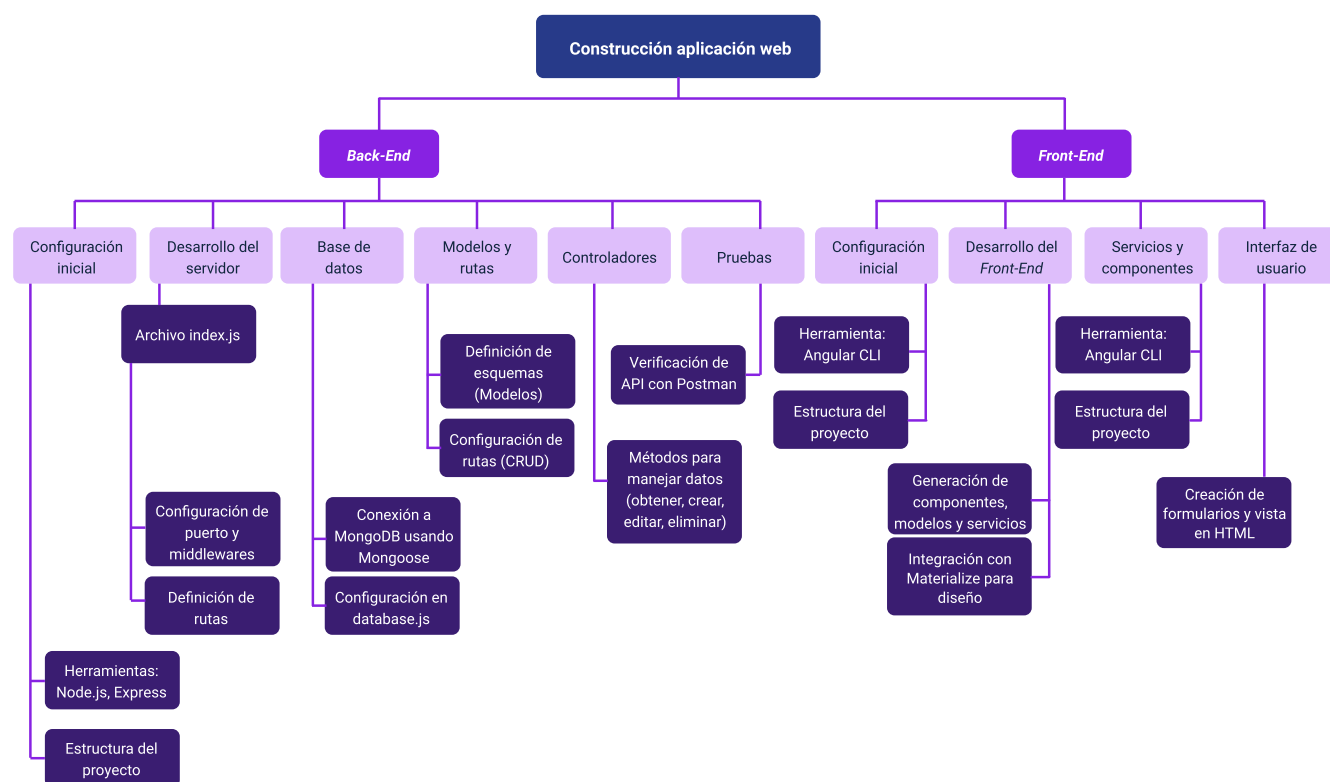
```
"__v": 0  
}
```

En los anteriores códigos se puede evidenciar que los datos se guardan correctamente en la base de datos, indicando que la aplicación desde el front-end funciona de manera idónea, así como la API REST del proyecto. Con esto se culmina este componente.

Se espera que haya sido de aprendizaje el uso de tecnologías tanto del lado del cliente como del lado del servidor. Este es uno de los ejercicios más completos y precisos que se pueden realizar en materia de esta disciplina del desarrollo web. En ese sentido, se exhorta a seguir estudiando más alternativas sobre este tipo de desarrollos que son muy demandados en la industria del software.

Síntesis

A continuación, se muestra un mapa conceptual con los elementos más importantes desarrollados en este componente.



Material complementario

Tema	Referencia	Tipo de material	Enlace del recurso
Descripción Back-end y Fron-end	EDteam. (2020). ¿Qué es Backend y Front-end? (guía completa) [video]. YouTube.	Video	https://www.youtube.com/watch?v=50RbVujPPGs&feature=youtu.be
Instalación MongoDB	MongoDB. (2024). Install MongoDB Community Edition on Windows.	Instalador	https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/
Instalación Node.js y Express	Coding, G. (2020). Cómo instalar Node.js en Windows 10 [video]. YouTube.	Video	https://www.youtube.com/watch?v=BgtB31gXkoA&feature=youtu.be
Guía de instalación Angular CLI	Angular. (2024). Installing Angular CLI.	Página web	https://angular.io/cli
Guía de instalación Postman	Limitless Minds. (2019). Instalación Postman y prueba de los métodos GET [video]. YouTube.	Video	https://www.youtube.com/watch?v=440yQGvINkk
Guía de Materialize	Materializecss. (2024). Getting Started - Materialize.	Página web	https://materializecss.com/getting-started.html
Guía extensión VS Code y MongoDB	Developer Soapbox. (2020). How to use Visual Studio Code as your MongoDB IDE [video]. YouTube.	Video	https://www.youtube.com/watch?v=wM7NJtQ0F6U
Proyecto web	GitLab. (2021). Gestion-empleados.	Software	https://gitlab.com/jonathanga/gestion-empleados

Glosario

Async: cuando se llama a una función `async`, se devuelve un elemento de tipo `promesa` o `promise`. Cuando la función `async` devuelve un valor, la promesa se resolverá con el valor devuelto. Si la función `async` genera una excepción o algún valor, la promesa se rechazará con el valor generado.

Await: una función `async` puede contener una expresión `await`, de hecho es una buena práctica. Para lo cual pausa la ejecución de la función asíncrona y espera la resolución de la promesa pasada y, a continuación, reanuda la ejecución de la función `async` y devuelve el valor resuelto.

CDN: una red de distribución de contenido. Es un tipo de sistema informático en red, el cual consiste en distribuir o repartir parte de la información en diferentes servidores. De esta forma el usuario recibe el contenido de la página del servidor más cercano.

Middleware: un `middleware` en `Node.js` es un bloque de código completo que se ejecuta entre la petición que hace el usuario (`request`) hasta que la petición llega al servidor.

Promise: el objeto `promise` (promesa) es usado para computaciones asíncronas. Así que una promesa representa un valor que puede estar disponible ahora o más adelante.

Referencias bibliográficas

Vortexbird. (2017). Nodemon. <https://vortexbird.com/nodemon/>

Créditos

Nombre	Cargo	Centro de Formación y Regional
Milady Tatiana Villamil Castellanos	Responsable del Ecosistema	Dirección General
Olga Constanza Bermúdez Jaimes	Responsable de Línea de Producción	Centro de Servicios de Salud - Regional Antioquia
Santiago Muñoz de la Rosa	Experto Temático	Centro de Teleinformática y Producción Industrial - Regional Cauca
Paola Alexandra Moya Peralta	Evaluadora Instruccional	Centro de Servicios de Salud - Regional Antioquia
Andrés Felipe Herrera Roldán	Diseñador de Contenidos Digitales	Centro de Servicios de Salud - Regional Antioquia
Edwin Sneider Velandia Suárez	Desarrollador Fullstack	Centro de Servicios de Salud - Regional Antioquia
Edgar Mauricio Cortés García	Actividad Didáctica	Centro de Servicios de Salud - Regional Antioquia
Jaime Hernán Tejada Llano	Validador de Recursos Educativos Digitales	Centro de Servicios de Salud - Regional Antioquia
Margarita Marcela Medrano Gómez	Evaluador para Contenidos Inclusivos y Accesibles	Centro de Servicios de Salud - Regional Antioquia
Daniel Ricardo Mutis Gómez	Evaluador para Contenidos Inclusivos y Accesibles	Centro de Servicios de Salud - Regional Antioquia