**FORMATO PARA EL DESARROLLO DE COMPONENTE FORMATIVO**

| PROGRAMA DE FORMACIÓN | Programación de *software* |
| --- | --- |

| COMPETENCIA | 220501096 - Desarrollar la solución de *software* de acuerdo con el diseño y metodologías de desarrollo. | RESULTADOS DE APRENDIZAJE | 220501096-03 - Codificar el *software* empleando el lenguaje de programación seleccionado. |
| --- | --- | --- | --- |

| NÚMERO DEL COMPONENTE FORMATIVO | 013 |
| --- | --- |
| NOMBRE DEL COMPONENTE FORMATIVO | Construcción aplicación web |
| BREVE DESCRIPCIÓN | En este componente formativo se realizará una aplicación *web* completa (*Backend* y *Frontend*). 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. |
| PALABRAS CLAVE | AngularJS, MEAN, MongoDB, NodeJS, web |

| ÁREA OCUPACIONAL | 2 - Ciencias Naturales, aplicadas y relacionadas |
| --- | --- |
| IDIOMA | Español |

1. **TABLA DE CONTENIDOS**

**Introducción**

1. **Construcción del Backend de la aplicación web**
   1. Instalación y configuración de herramientas de Node.js
   2. Desarrollo del index.js
   3. Configuración e instalación de la base de datos MongoDB
   4. Configuración de database.js
   5. Configuración de las rutas del proyecto con Node.js
   6. Desarrollo de controladores para la API REST
   7. Realización de pruebas de la API REST con Postman
2. **Construcción del Front-end de la aplicación web**
   1. Instalación y configuración de AngularJS
   2. Generación de código con ng
   3. Incorporación de Materialize al proyecto web
   4. Configuración de app.component.html
   5. Configuración de los módulos app.modules.ts
   6. Configuración del modelo empleado.ts
   7. Creación del servicio empleado.service.ts
   8. Desarrollo del componente empleados.components.ts
   9. Desarrollo de la interfaz HTML empleados.component.html
   10. Probando la aplicación completa. Proyecto web terminado
3. **Paradigmas, estándares y costos asociados al desarrollo de *software***
4. **INTRODUCCIÓN**

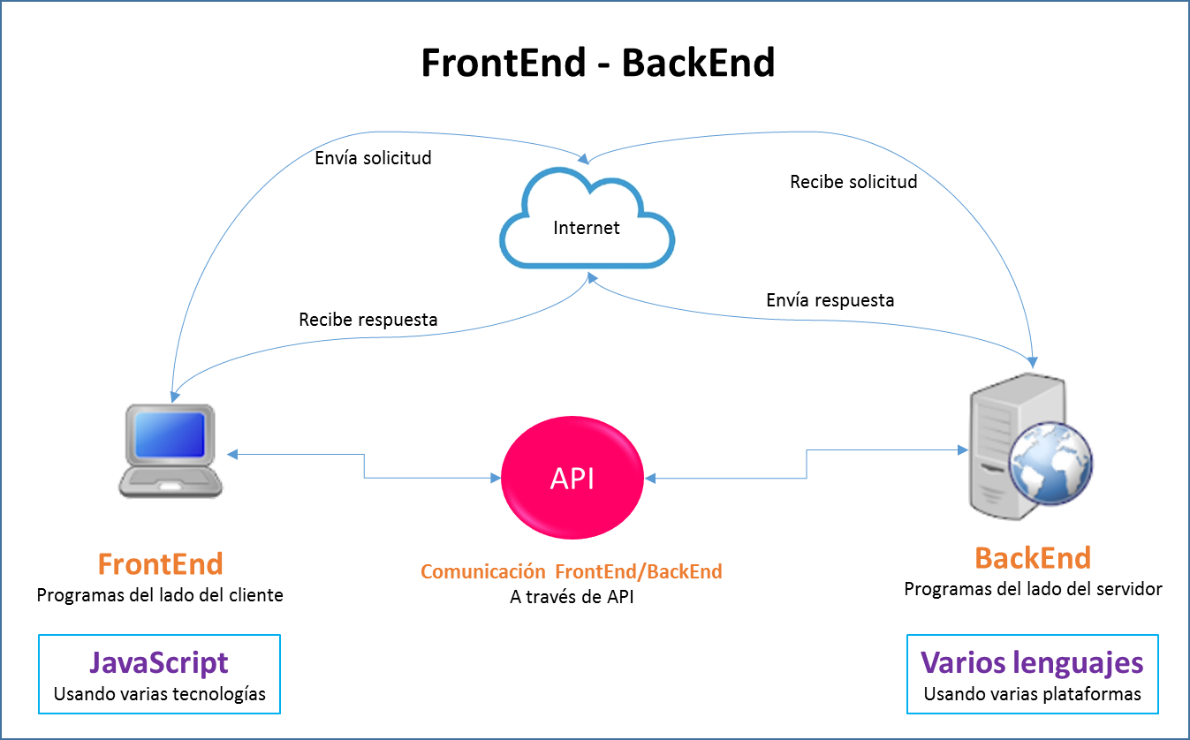
En este componente formativo se abordarán los conceptos y técnicas que apoyarán el desarrollo de una aplicación *web* de manera completa; es decir, se realizará el *Backend* y *Front-end* de la aplicación, de esta forma se podrá observar la comunicación, intercambio y procesamiento de la información de estos dos componentes, por lo que se iniciará con la instalación de herramientas y recursos del lado del servidor, que gestionará la base o plataforma para el despliegue de la aplicación del lado del cliente. En la segunda parte de este documento se da alcance a la construcción de la aplicación web Responsive, que consumirá la API REST ya realizada en el servidor.

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.

La siguiente figura representa el esquema general de comunicación entre la aplicación (Front-end) y el servidor (Backend):

**Figura 1**

*Esquema Front-end – Backend*



Nota. Esquema general de comunicación Front-end – Backend.

Asimismo, en el siguiente video podrá encontrar información complementaria sobre los elementos, características y componentes del Front-end y Backend:

EDteam. (2020). *¿Qué es Backend y Front-end? (guía completa)* [video]. YouTube. <https://youtu.be/50RbVujPPGs>

1. **DESARROLLO DE CONTENIDOS**
2. **Construcción del Backend 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 Backend. Para la instalación de las primeras herramientas como Node.js y Express en la sección material complementario de este documento encontrará la guía de instalación de Node.js y de Express. A continuación, se incluye una figura que representa de manera general el proceso de codificación de *software*.

**Figura 2**

*Codificación software*



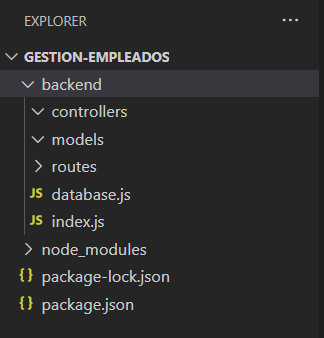
Nota. Codificación en lenguaje de desarrollo de *software*.

* 1. **Instalación y configuración de herramientas de Node.js**

Se utiliza el Visual Studio Code (VS Code) con la siguiente estructura de directorios que se muestra en la figura 3 para el Backend.

**Figura 3**

*Estructura de directorios Backend*

****

Nota. Estructura de directorios del proyecto.

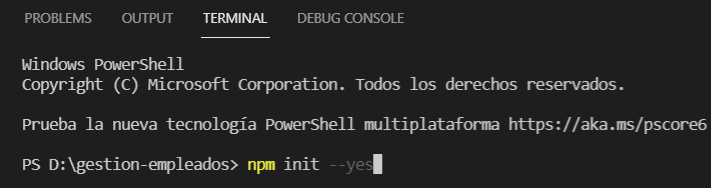
Como se observa en la figura anterior se crea la carpeta GESTION-EMPLEADOS, que será la principal o raíz del proyecto. Dentro de esta se crea la carpeta backend, en la cual se crean otros directorios y archivos como se alcanza a apreciar en la figura. De tal forma, que esta será la estructura que apoyará el desarrollo de la base de la aplicación.

Una vez creadas las carpetas y archivos como lo son *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, como se muestra en el ejemplo de la siguiente figura:

**npm init --yes**

**Figura 4**

*Terminal de VS Code*

****

Nota. Terminal de VS Code iniciando proyecto con Node.js.

Una vez creado el proyecto se requiere todas las bondades del *framework* de Node.js, para lo cual se hace necesario Express, que va a generar 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 el terminal del VS Code:

**npm install express**

* 1. **Desarrollo del index.js**

El siguiente paso es iniciar con el desarrollo del archivo principal del Backend, que se llama **index.js** como se muestra en la siguiente figura, (puede descargar el contenido de estas instrucciones en el archivo index.js disponible en la carpeta anexos).

**Figura 5**

*Estructura index.js*

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'));

});

Nota. Código completo archivo index.js.

En la figura anterior se aprecia el código del archivo index.js para el proyecto, por lo cual en la medida del avance de este componente se irá explicando cada línea de código que se requiera ejecutar. En ese sentido, se inicia con el requerimiento de Express y se almacena en una constante en JavaScript llamada Express. De esta forma se tendrá el acceso a todo el funcionamiento del servidor con la línea de código:

**const app = express();**

Una vez creada la constante app, se pasa a la configuración del puerto por donde va a escuchar y a recibir el servidor todo tipo de solicitudes por parte del cliente, según se muestra en la siguiente línea:

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

De este modo, *set* es para crear una variable que va a ser accedida desde cualquier parte de la aplicación, por lo que 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 el caso de que el puerto 3000 esté ocupado.

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(Vortexbird, 2017) 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. Por lo tanto, la línea de código utilizada en la terminal de VS Code es:

**npm install nodemon –D**

Esto quiere decir que Nodemon será instalado como una dependencia de desarrollo y no como dependencia general del proyecto. Por lo que se configura el archivo package.json según se muestra en la siguiente figura (puede descargar el contenido de estas instrucciones en el archivo package.json disponible en la carpeta), y se inicia a ejecutar el servidor con la siguiente línea de comando:

**npm run dev**

**Figura 6**

*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"

}

}

Nota. Código completo archivo package.json.

Ahora, se construyen las Middleware dentro del archivo de index.js como se puede apreciar en la figura anterior, encargadas de procesar los datos, es decir, cuando se solicitan o envían al servidor. Al respecto, 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 que se requiere del uso del formato JSON ejecutando la siguiente instrucción:

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

Con el fin de observar la transacción de mensajes por consola se tiene una herramienta útil llamada *morgan*, que ayuda a mirar 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. **Configuración e instalación de la bbase de datos MongoDB**

A continuación, se realizará la conexión de la aplicación del servidor a la base de datos, por lo que se inicia a configurar y a instalar los recursos necesarios antes de iniciar a desarrollar. En ese sentido, se instala MongoDB de manera persistente en el equipo de desarrollo, para lo cual se deja información para su instalación en la sección material complementario de este documento. La siguiente figura representa de manera general el diagrama entidad-relación de una base de datos.

**Figura 7**

*Diagrama entidad-relación*

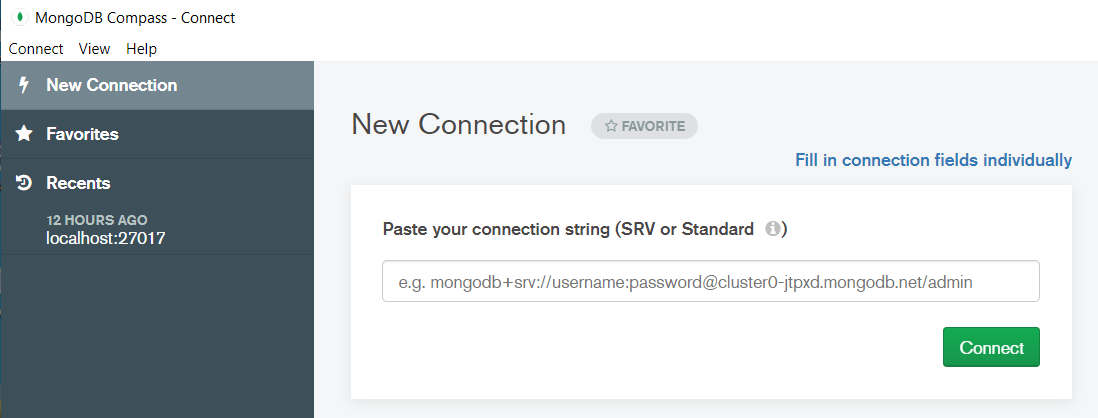


Nota. Esquema general diagrama entidad-relación base de datos. Pixabay (2016).

En la siguiente figura se muestra la imagen que representa el inicio del proceso de instalación de MongoDB.

**Figura 8**

*MongoDB*

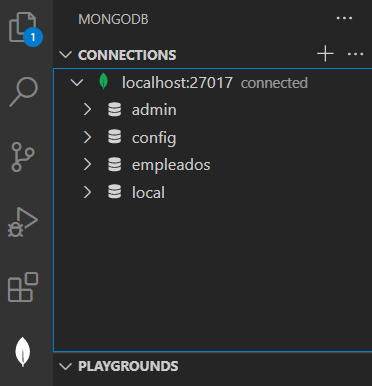


Nota. Panel de control MongoDB.

De igual manera, a nivel opcional se instala la extensión de MongoDB dentro del entorno de VS Code, permitiendo tener una integralidad para 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 para la instalación de esta extensión, denominada extensión VS Code y MongoDB. La siguiente figura muestra de qué manera se visualiza dicha extensión dentro del entorno de desarrollo.

**Figura 9**

*Extensión MongoDB*



Nota. Visualización extensión MongoDB en VS Code.

Posteriormente, y con el fin de realizar la conexión a la base de datos y definir la estructura de los datos de la aplicación a desarrollar, o lo que técnicamente se conoce como esquemas, se instala la herramienta Mongoose con la siguiente línea de comando desde la terminal de VS Code:

**npm install mongoose**

* 1. **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 a configurar sobre el archivo database.js. (Puede descargar el contenido de estas instrucciones en el archivo database.js disponible en la carpeta anexos).

**Figura 10**

*Estructura database.js*

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;

Nota. Código completo archivo database.js.

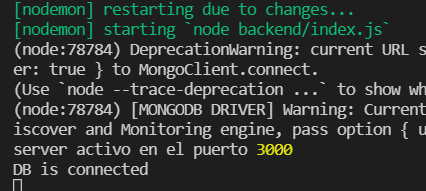
Con respecto al código del archivo database.js, allí se indica que es necesario que el módulo se conecte a MongoDB, por eso aparece Mongoose. Así, al llamarse al módulo se almacena en una constante, por otra parte, se define la dirección de la conexión de la base de datos en una constante denominada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

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 se puede observar que en el archivo index.js, se invoca o llama el archivo database.js en el cual se especifica que lo que se requiere es la conexión. En este sentido, solo resta probar si la conexión está activa, por lo que se inicia el servidor con la línea de comando **npm run dev**, vista anteriormente. A continuación, se muestra en la figura el resultado de la terminal del VS Code:

**Figura 11**

*Respuesta de la terminal*

.

Nota. Respuesta conexión a base de datos funcionando.

* 1. **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 llamadoempleado.route.js dentro de la carpeta *routes* (puede descargar el contenido de estas instrucciones en el archivo empleado.route.js disponible en la carpeta anexos). En este primer apartado lo que se quiere es revisar en primera instancia es el funcionamiento de las rutas, por lo que a manera de prueba se tiene el código que se muestra en la siguiente figura:

**Figura 12**

*Estructura ejemplo empleado.route.js*

/\*\*

\* 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 servidor por el método GET \*\*/

router.get('/', (req, res) => {

res.json({

status: 'API REST funcionando'

});

})

module.exports = router;

Nota. Código completo archivoempleado.route.js para comprobar el funcionamiento de las rutas.

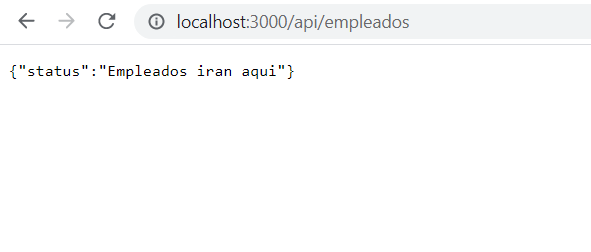
De manera adicional se puede observar el archivo index.js, en el cual se establece la ruta por donde se van a recibir o enviar los datos desde el servidor que para el caso de estudio será **http://localhost:3000/api/empleados,** de tal forma se observa que para la definición de esta ruta se hace necesario 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 13**

*Respuesta del servidor*



Nota. Vista desde un navegador.

Ya visto un ejemplo del funcionamiento de las rutas es necesario comprender su comportamiento, por lo que 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 BD. En ese sentido, es más que necesario la construcción del modelo empleado.js con datos específicos para los empleados dentro de la carpeta *models* (puede descargar el contenido de estas instrucciones en el archivo empleado.js disponible en la carpeta anexos).

**Figura 14**

*Estructura 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);

Nota. Código completo archivo empleado.js.

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

* 1. **Desarrollo de controladores para la API REST**

Una de las mejores prácticas para estructurar una aplicación cuando va en crecimiento es el uso de los controladores. Estos se encargan de definir los métodos y acciones que necesitan las rutas. Por lo tanto, se crea el archivo empleado.controller.js dentro de la carpeta *controllers* (puede descargar el contenido de estas instrucciones en el archivo empleado.controller.js disponible en la carpeta anexos).

**Figura 15**

*Estructura controlador empleado*

/\*\*

\* 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 emepleadoEdit = {

name: req.body.name,

position: req.body.position,

office: req.body.office,

salary: req.body.salary

};

await Empleado.findByIdAndUpdate(id, {$set: emepleadoEdit}, {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;

Nota. Código completo archivo empleado.controller.js.

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.

Por otra parte, en cada uno de los métodos se utilizan funciones nativas y nuevas de JavaScript. En ese sentido, es necesario recordar que antes se utilizaban las 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.

Para el método de **obtener empleados** se utiliza la función *async,* la cual entrega en formato *json* todo lo que encuentre de empleados. Por eso se utiliza la función *await,* porque se espera que se vaya a tardar un poco en esta actividad de buscar. En ese sentido, se utiliza uno de los métodos intrínsecos del modelo Empleado que para este caso es *find*().

Además, el método de crear empleado tiene una naturaleza similar al anterior, solo que en este método se crea una nueva instancia de empleado ***new* Empleado**. De esta forma lo que seguiría es guardar ese nuevo registro con ***save*().** Es importante anotar que hasta el momento no se ha creado la base de datos en MongoDB, pero apenas se almacena un registro se crea la base de datos.

Para el método de editar empleado presenta una sintaxis interesante, en primer lugar se escribe de otra forma para capturar el id del usuario que se quiere actualizar { id }, además crea una constante **empleadoEdit** para pasarle los datos traídos del cliente y finalmente se utiliza *await*, para dar una espera 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.

Finalmente, lo que se hace al crear un módulo es exportarlo 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* (puede descargar el contenido de estas instrucciones en el archivo empleados.routes.js disponible en la carpeta anexos).

**Figura 16**

*Código de la ruta empleado*

/\*\*

\*

\* 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;

Nota. Código completo archivo empleados.routes.js.

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.

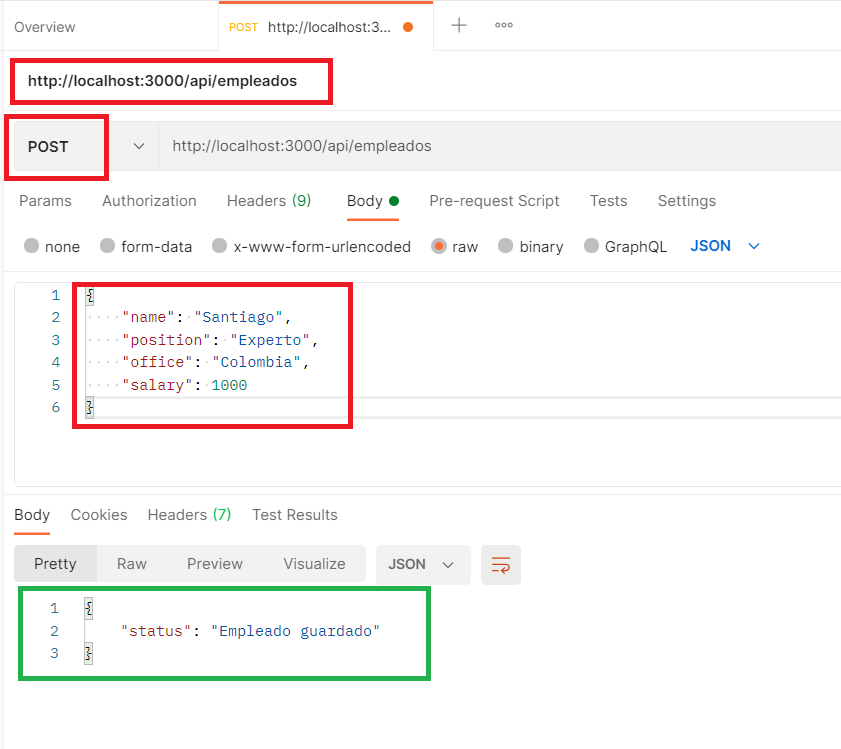
* 1. **Realización de pruebas de la API REST con Postman**

Una vez finalizada la construcción tanto del modelo, controlador y rutas, se dispone a generar las pruebas respectivas para evaluar el funcionamiento de la API REST; por lo tanto, se utilizará la herramienta Postman. En la sección material complementario de este documento encontrará la guía para su instalación.

Se inicia insertando datos, por lo que utilizará el método *POST*, el cual llamará a la ruta para crear empleados.

**Figura 17**

*Pruebas con Postman*



Nota. Parámetros y respuesta usando Postman.

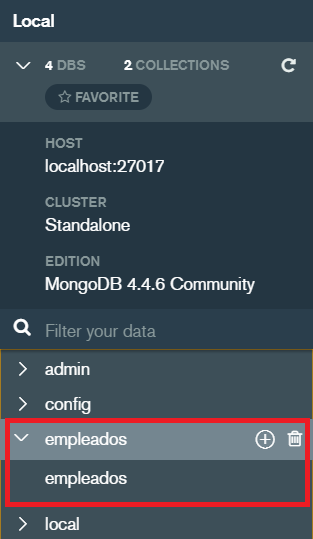
En la figura anterior se pueden apreciar los datos que se envían al servidor, dado que Postman permite realizar esa emulación del cliente. Los datos que se envían están acordes al esquema definido en el modelo de empleado.js, como lo son *name*, *position*, *office* y *salary*. La respuesta en consola, 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 18**

*Panel MongoDB*

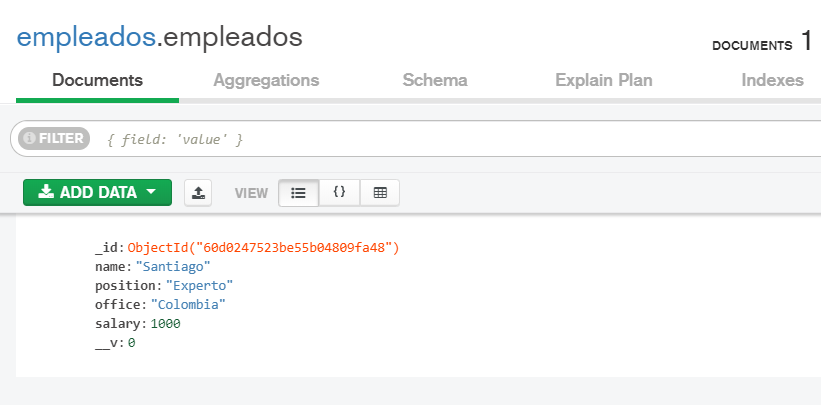


Nota. Panel de MongoDB que muestra la base de datos empleados creada.

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

**Figura 19**

*Registro de empleado almacenado en MongoDB*



Nota. Verificación en MongoDB de información guardada.

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 evidencia para los demás métodos *HTTP*, *GET*, *PUT* o *DELETE* que también funcionan, por lo cual se realiza la invitación a probar el API con Postman.

Hasta este punto se ha realizado el desarrollo del Backend, por lo que hasta el momento se tiene la base de datos lista para poder generar ahora la construcción del Front-end, el cual será el que “dialogará” con el servidor. De esta forma se presenta a continuación una figura que ilustra la estructura general del proyecto.

**Figura 20**

*Resumen estructura del proyecto web*



Nota. Esquema de comunicación Backend – Front-end.

1. **Construcción del Front-end de la aplicación web**

Para esta sección se plantea generar una sola vista, la cual corresponde al formulario de registro de empleados, por esta razón no es necesario generar *sketch* o maquetación para evaluar la novación 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*.

En la siguiente figura se muestra una representación gráfica de ejemplo de un Front-end.

**Figura 21**

*Ejemplo interfaz gráfica*



Nota. Ejemplo interfaz gráfica Front-end.

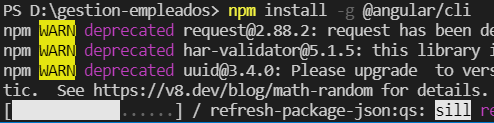
* 1. **Instalación y configuración de *AngularJS***

Se inicia instalando AngularJS a través de Angular CLI, que permite generar una plataforma para la creación de proyectos Angular utilizando el comando **ng**, por lo tanto, para la instalación se recurre a la línea de comando npm. El paso a paso de la instalación y para mayor información se puede encontrar en la sección de material complementario de este documento con el nombre de Guía de instalación Angular CLI. Para iniciar la instalación, se ejecuta la siguiente instrucción, su salida se puede evidenciar en la siguiente figura.

**npm install -g @angular/cli**

**Figura 22**

*Instalación de Angular CLI*



Nota. Terminal de VS Code. Instalación de Angular CLI.

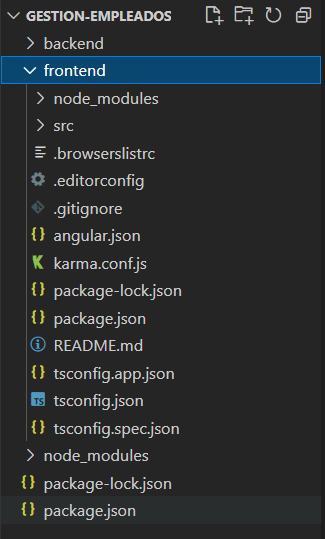
Ahora, se inicia un proyecto de Angular llamado Front-end, por lo que desde la consola de comandos se escribe:

**ng new frontend**

El resultado en la estructura de directorios del proyecto se muestra en la siguiente figura:

**Figura 23**

*Estructura de directorios del proyecto*



Nota. Estructura de directorios del proyecto del Front-end*.*

A continuación, se debe ingresar a la carpeta del Front-end e iniciar el servidor digitando la terminal del VS Code de la siguiente línea de comandos:

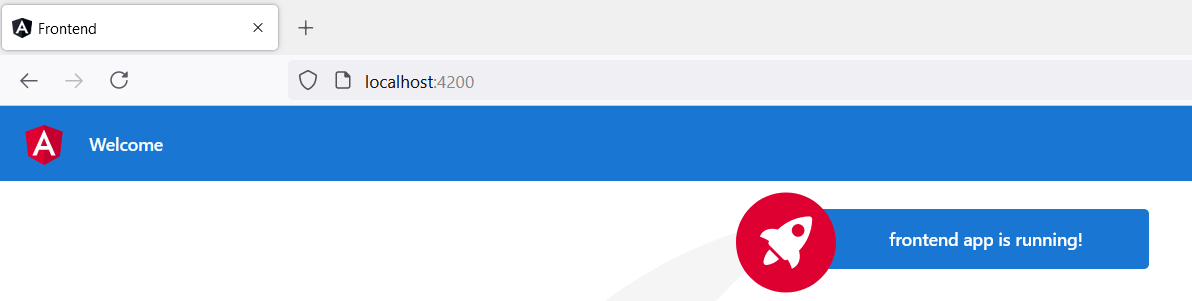
**ng serve**

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*,** por lo que ejecuta un servidor de desarrollo para Angular. En ese sentido, el servidor se inicializa en el puerto 4200 (http://localhost:4200/), como se muestra en la siguiente figura:

**Figura 24**

*Estado servidor Front-end*



Nota. Verificación ejecución servidor en el puerto 4200.

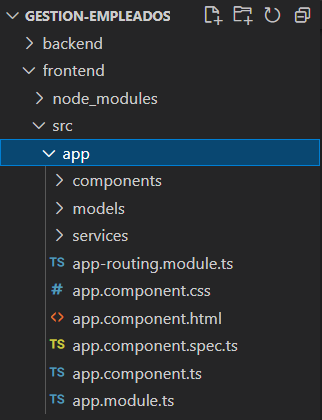
* 1. **Generación de código con ng**

Después de iniciado el servidor Angular, se revisa la estructura de directorios de este *framework*, que, aunque son muchos archivos, únicamente es necesario concentrarse en la carpeta que se encuentra en la ruta **frontend/src/app**, de esta forma en la carpeta **app** se realizará todo el código que se necesita para el proyecto.

De manera adicional, para dar coherencia al proyecto, dentro de la carpeta *app* se crean 3 carpetas más, que son ***models*, *services* y *components*** para tener definido el proyecto, tal como se muestra en la siguiente figura:

**Figura 25**

*Estructura de directorios*



Nota. Proyecto Front-end.

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:

* + - Dentro la carpeta *components* se genera un componente llamado empleados con Angular CLI:

**ng g c empleados**

* + - Dentro la carpeta *models* se crea la clase empleado con:

**ng g class 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, en las siguientes figuras se muestra la evidencia de las operaciones realizadas:

**Figura 26**

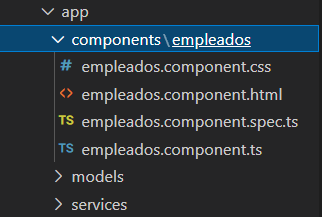
*Salida terminal VS Code*



Nota. Ejemplo de generación de código a través de la terminal de VS Code.

**Figura 27**

*Componentes generados automáticamente*



Nota. Representación de componentes generados.

* 1. **Incorporación de Materialize al proyecto web**

Se incorpora Materialize al proyecto por medio de la inclusión de los CDN (red de distribución de contenidos) de hojas de estilos, JavaScript e iconos en el documento **index.html** del Front-end, por lo que la ruta al archivo es **FrontEnd/src/index.html** (puede descargar el contenido de estas instrucciones en el archivo index.html disponible en la carpeta anexos). En la figura 28 puede ver este proceso.

**Figura 28**

*Incorporación de los CDN de Materialize*

<!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>

Nota. Código completo archivo index.html.

De manera adicional, en la sección material complementario de este documento encontrará el enlace a la guía de uso de Materialize donde están los *CDN* y ejemplos sobre este *framework* de estilos.

* 1. **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. Este cambio se ilustra a continuación (puede descargar el contenido de estas instrucciones en el archivo app.component.html disponible en la carpeta anexos):

**Figura 29**

*Estructura app.component.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>

Nota. Código completo archivo app.component.html.

En la figura anterior se observa que se adiciona la etiqueta <app-empleados>, indicando que se hará uso de ese componente.

* 1. **Configuración de los módulos app.modules.ts**

A continuación, se configura uno de los archivos más importantes del proyecto, y es por el cual se cargan los módulos de la aplicación, por ello, es necesario entender la naturaleza de la aplicación que se está desarrollando, porque así se definirán qué módulos son necesarios para el óptimo funcionamiento de la aplicación.

La aplicación caso de estudio será la adición de empleados utilizando un API REST, por lo que es evidente el uso de un formulario en la interfaz y además de la comunicación que debe tener el Front-end con el servidor. Por lo tanto, se cargan los módulos de formularios y de cliente HTTP en el archivo **app.modules.ts** (puede descargar el contenido de estas instrucciones en el archivo app.module.ts disponible en la carpeta anexos), este proceso se muestra en la siguiente figura.

**Figura 30**

*Código del archivo app module*

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 { }

Nota. Código completo archivo app.module.ts

* 1. **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, por lo que se empieza la codificación 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 describió en el Backend. En la figura que aparece a continuación, se presenta el archivo empleado.ts (puede descargar el contenido de estas instrucciones en el archivo *empleado.ts* disponible en la carpeta anexos).

**Figura 31**

*Código del modelo empleado*

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;

}

Nota. Código completo archivo empleado.ts

En la figura 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.

* 1. **Creación del servicio empleado.service.ts**

El objetivo general de este lado del proyecto es agregar esos datos ya definidos a la REST API, y se enfocan los esfuerzos a generar ese servicio, por lo que ahora se desarrollará el archivo **empleado.service.ts**. El servicio permite definir tan solo los métodos que pueden ser reutilizados en cualquier parte de la aplicación, en este caso son funciones que van a acceder para hacer la gestión sobre los datos como agregar, eliminar, editar, entre otros.

A continuación, se enseña el código del archivo empleado.service.ts (puede descargar el contenido de estas instrucciones en el archivo empleado.service.ts disponible en la carpeta anexos):

**Figura 32**

*Código del servicio empleado*

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

}

}

Nota. Código completo archivo empleado.service.ts.

En la anterior figura se observa la importancia de haber definido primeramente el módulo empleado.ts, de esta forma se tiene una estructura establecida de datos, y es por eso que se importa el modelo, además también se ha dado alcance a la comunicación con el servidor a través de la modificación que se hizo en el archivo app.modules.ts con la incorporación del módulo HTTPClient.

Por otra parte, se establece como atributo de solo lectura la URL de la API a la cual se conectará el cliente, en este caso es la URL del servidor ya realizado en la primera sección de este componente que es [**http://localhost:3000/empleados**](http://localhost:3000/empleados)

Ahora, se sigue con la definición de los métodos para la gestión de los datos de empleado, así que se generan 4 métodos como getEmpleados, PostEmpleados, putEmpleado, deleteEmpleado. Para el caso de estudio se revisa el método de PostEmpleado que recibe como parámetro un objeto de tipo Empleado, así que el retorno es la adición de un nuevo empleado, (se envían todos los datos) a través del método HTTP como lo es POST.

**return this.http.post(this.URL\_API, Empleado);**

Debe observar bien que en cada método se envíe la URL del servidor y los datos necesarios con relación al método.

* 1. **Desarrollo del componente empleados.components.ts**

Ya definidos los servicios es necesario abordar la lógica del componente, por lo que ahora se debe enfocar los esfuerzos en el archivo **empleados.components.ts**, por lo que el archivo en mención se presenta a continuación (puede descargar el contenido de estas instrucciones en el archivo empleados.components.ts disponible en la carpeta anexos):

**Figura 33**

*Código del componente empleados*

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

import { EmpleadoService } from "../../services/empleado.service";

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

import { Empleado } from 'src/app/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 parametro

if (form) {

form.reset();

this.empleadoService.selectedEmpleado = new Empleado();

}

}

}

Nota. Código completo archivo empleados.component.ts.

En la figura anterior se muestra la importancia de haber creado y/o configurado con anticipación los módulos y archivos para la construcción de este componente, por lo que para el natural funcionamiento de este se hace necesario **EmpleadoServices**, el modelo empleado y el módulo ya definido como lo es *NgForm*.

Una vez se importan estos módulos se enfocan los esfuerzos en los métodos; para el caso de estudio tendrá el método de agregar empleado. Al respecto toma como parámetro la existencia de datos del formulario, al ser verdadero, se llama al método del servicio empleado (el cual ya fue construido anteriormente) **PostEmpleado** y le entrega los valores para que sean llevados al servidor y que sean almacenados en la base de datos de MongoDB.

De manera adicional, como ejemplo de este componente, se agrega un *plus* y es que en cada adición de datos en el formulario, este se pueda limpiar de manera automática, por lo que se incorpora una función llamada ***reset* *Form***, en la cual como su nombre lo indica lo que hace es resetear o limpiar los campos del formulario a través de la función intrínseca *reset*().

Finalmente, para mejorar la interfaz se adiciona un complemento de Materialize, *Toast*, que no es más que una notificación, que se utiliza para indicarle al usuario que el empleado se ha guardado de manera correcta.

* 1. **Desarrollo de la interfaz HTML empleados.component.html**

Después de definir la lógica del componente empleado lo único que hace falta es la construcción del formulario, por lo que se apunta ahora al desarrollo del archivo **empleados.component.html** que se muestra a continuación (puede descargar el contenido de estas instrucciones en el archivo empleados.component.html disponible en la carpeta anexos):

**Figura 34**

*Código HTML del componente empleado*

<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>

</form>

</div>

</div>

</div>

<div class="col s7">

</div>

</div>

</div>

Nota. Código completo archivo empleados.component.html.

En el código anterior se muestra la estructura base de un formulario en HTML, el cual es sujeto a clases y reglas provistas por Materialize para dar una mejora de apariencia a la interfaz, de manera adicional se presentan 4 *inputs* en los cuales el usuario debe digitar y, al final se presentan 2 botones, uno es de enviar y el otro de limpiar el formulario. Al respecto hay una línea de código a la que se hace mención:

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

La cual define que todos los campos del formulario serán almacenados en una sola variable y son estos campos los que serán procesados por el método **agregarEmpleado**, método que fue definido anteriormente en el componente de empleado.

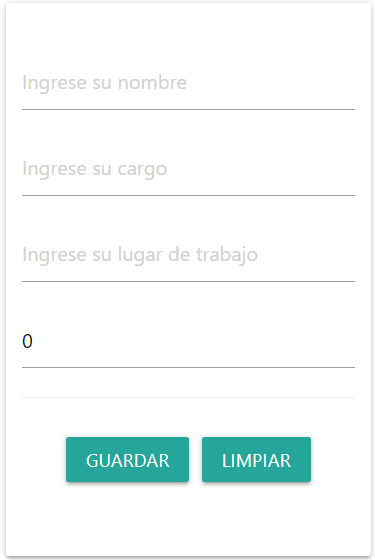
* 1. **Probando la aplicación completa. Proyecto web terminado**

Ya finalizada la codificación de cada uno de los archivos del Front-end es necesario revisar cómo ha quedado la aplicación; no obstante, este paso no se hace al final, sino que por lo sucinto de este componente se presenta en esta instancia, ya que como buena práctica, en cada bloque de desarrollo o línea debe revisarse el resultado obtenido, es decir, trabajar de manera paralela.

Una muestra de la interfaz de la aplicación se ilustra en la siguiente figura:

**Figura 35**

*Registro empleados*



Nota. Interfaz formulario de registro de empleados.

En la muestra anterior de la interfaz se ilustra un formulario en el cual se le piden los datos al usuario para el registro de un empleado, es importante observar el uso de los *placeholder* en cada caja de texto, cuya función es guiar o indicar qué tipo de datos hay que registrar. De esta manera la interfaz quedaría como se muestra en la siguiente figura:

**Figura 36**

*Registro empleado*

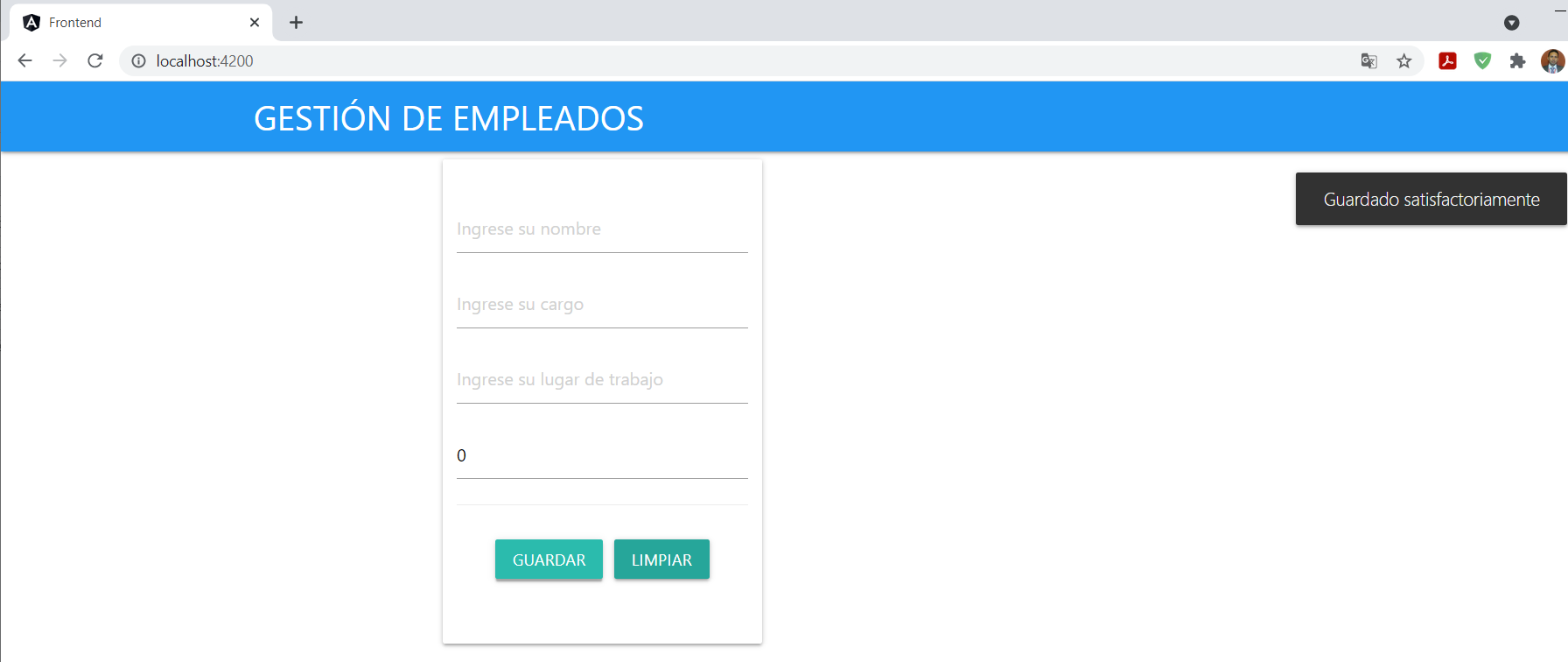


Nota. Ejemplo ingreso datos de un empleado.

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 y además se limpia el formulario por la función *reset*(), tal como se puede observar en la figura 37.

**Figura 37**

*Notificación Toast de Materialize*

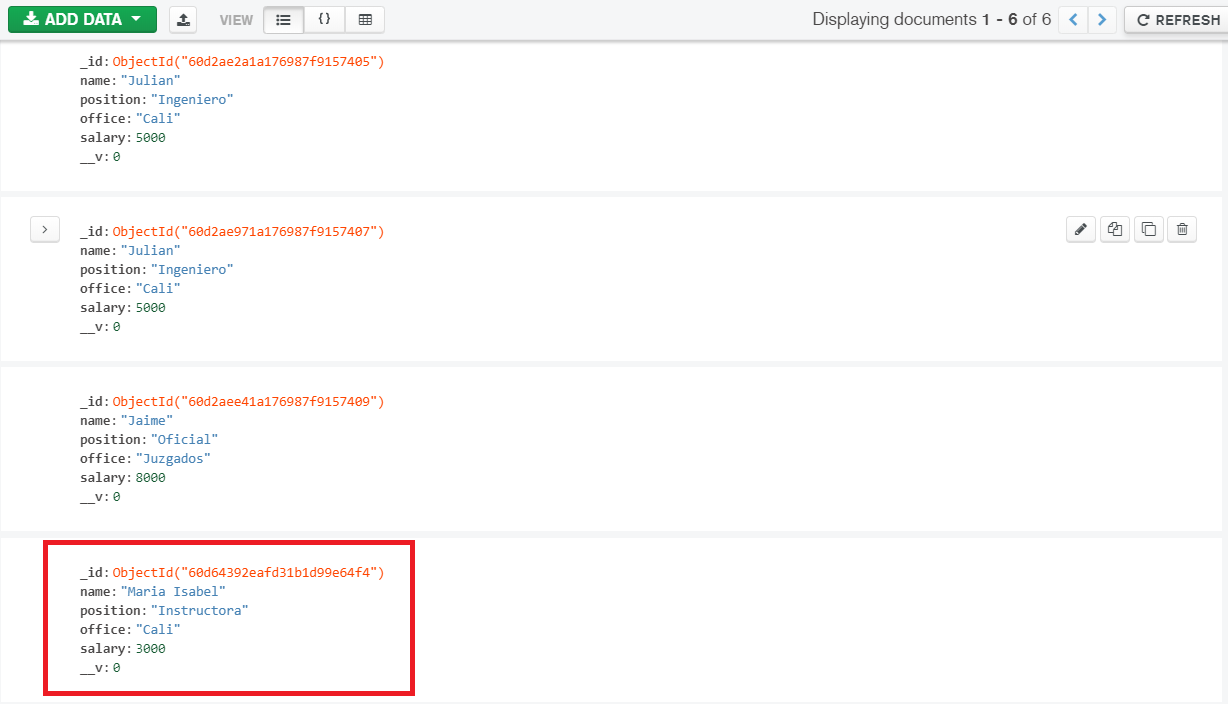


Nota. Ejemplo notificación *Toast.*

Para verificar que los datos incorporados en el formulario se hayan almacenado de manera correcta en la base de datos empleado de MongoDB se utilizará el panel de control de la MongoDB, como se muestra en las siguientes figuras:

**Figura 38**

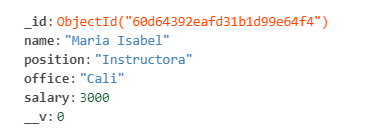
*Panel de control de MongoDB*



Nota. Verificación registro de información de empleados.

**Figura 39**

Evidencia de registro en MongoDB



Nota. Verificación registro de información de un empleado.

En las anteriores figuras se puede evidenciar que los datos se guardan de manera correcta 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 lo que 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*.

Finalmente, se comparte en el material de apoyo todo el proyecto web realizado desde la plataforma de *Gitlab*, se encuentra con el nombre de Proyecto web.

1. **Paradigmas, estándares y costos asociados al desarrollo de *software***

Los paradigmas de desarrollo permiten mejorar el proceso de construcción de *software* y las metodologías de ingeniería de *software* sirven de base para la ejecución de cualquier proyecto de desarrollo.

**Lenguajes de programación**

Existen diversos lenguajes de programación que permiten dar una serie de instrucciones para definir sentencias con miras a controlar un sistema de información. A continuación, se presentan los lenguajes de programación más comunes para el desarrollo de *software*:



**Paradigmas de programación**

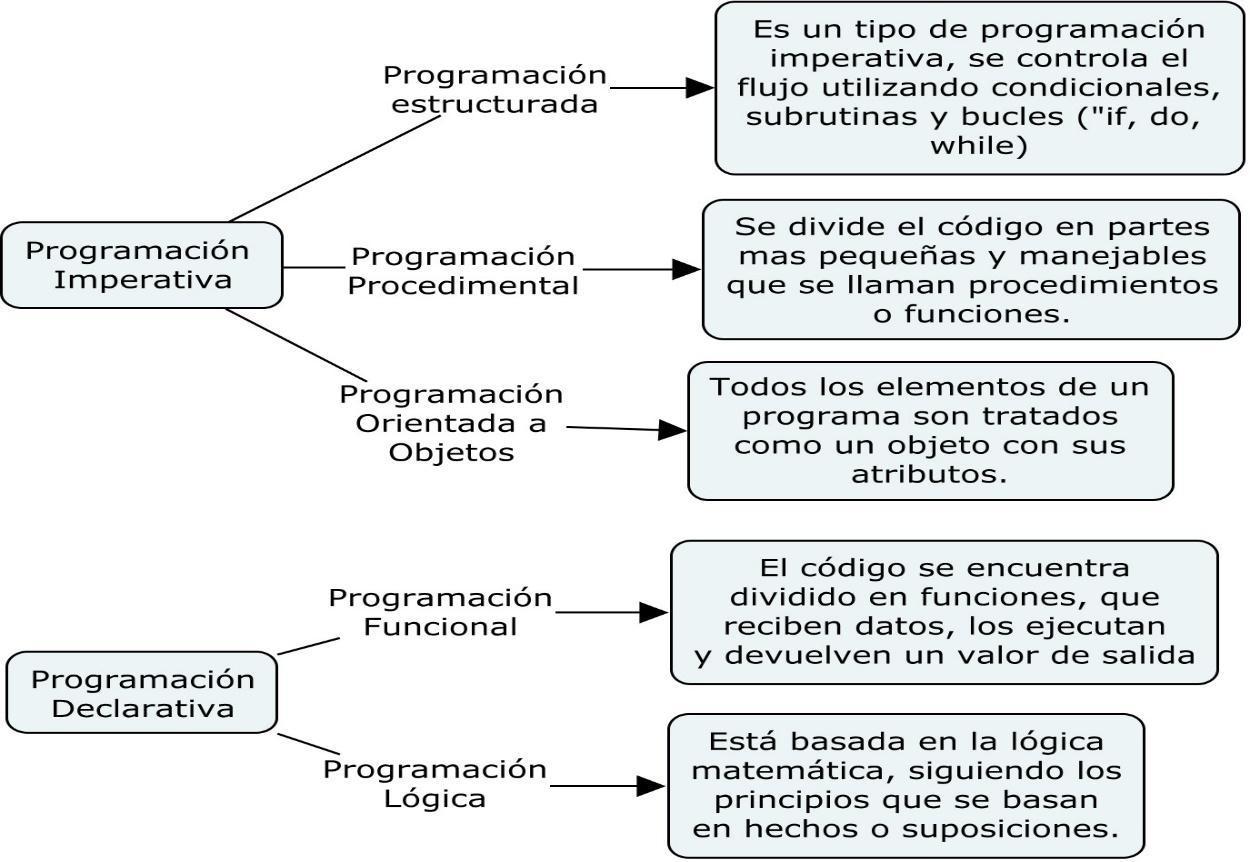
Hace referencia a los principios fundamentales para la creación del *software*; es decir, los diferentes estilos de programación o generación de código para direccionar a las máquinas y demás dispositivos tecnológicos.

Las aplicaciones *software* tratan de resolver las necesidades, ofreciendo a los usuarios diferentes tareas, operaciones o trabajos determinados en los que es necesario intervenir este tipo de problemas de diferentes maneras, recurriendo a la forma más eficiente en un paradigma de programación y dependiendo de su complejidad.

El paradigma de programación es un camino o guía para agilizar el desarrollo web siguiendo con las reglas y estructuras determinadas para la aplicación.

**Tipos de paradigma de programación**

Los paradigmas de programación son diferentes y se orientan hacia distintos tipos de programación como son:



**Editores de código**

Existen muchas opciones para escoger el mejor editor de código como desarrollador web, los desarrolladores siempre buscan aumentar la suficiencia, productividad y rendimiento de la aplicación o el programa que se esté programando.

De igual forma, los mejores desarrolladores web buscan ser eficientes y tener buen rendimiento en su código, deben estar basados en un buen editor de texto, conociéndolo a fondo y sabiendo interpretarlo.

Ahora, se expone un poco al respecto:

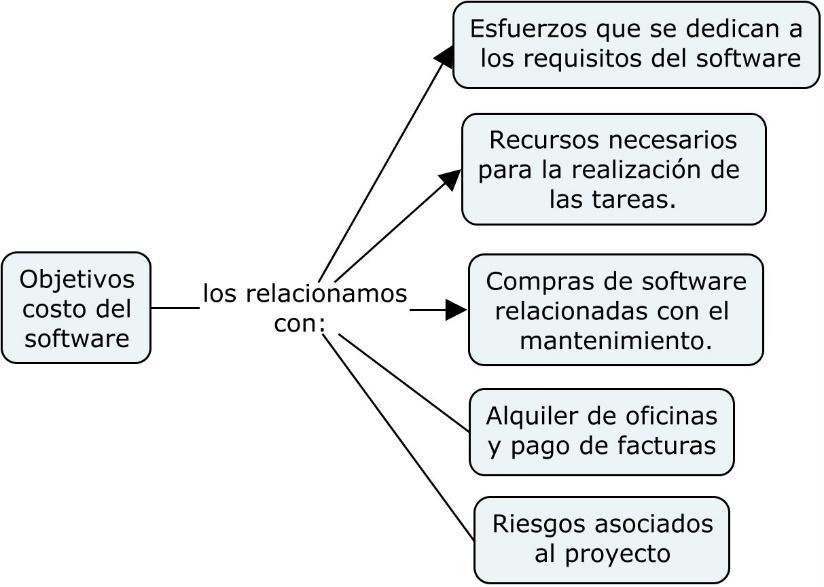
****

**Costos asociados al desarrollo de *software***

La estimación de costos de desarrollo de *software* es el factor más importante en todo proyecto informático. Es primordial contar con temas estratégicos para tener indicadores que midan el costo de los mismos, y que logren garantizar la eficiencia, calidad y competitividad. Se trata de identificar los recursos necesarios para lograr que un proyecto sea eficiente.

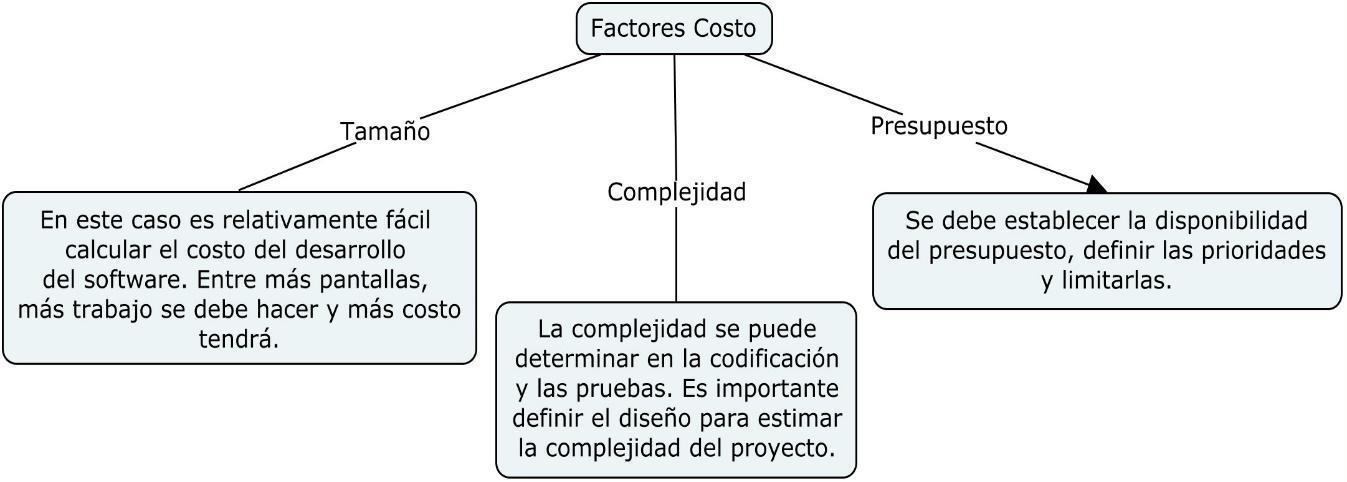
Para la valoración o estimación de recursos para el desarrollo de *software*, existen algunos componentes como los gastos directos o indirectos.

El costo directo de los programas *software* comprende desde los ingresos de cada individuo del equipo, los pagos que se deben hacer por la utilización del *hardware* y otros elementos para la realización del trabajo.



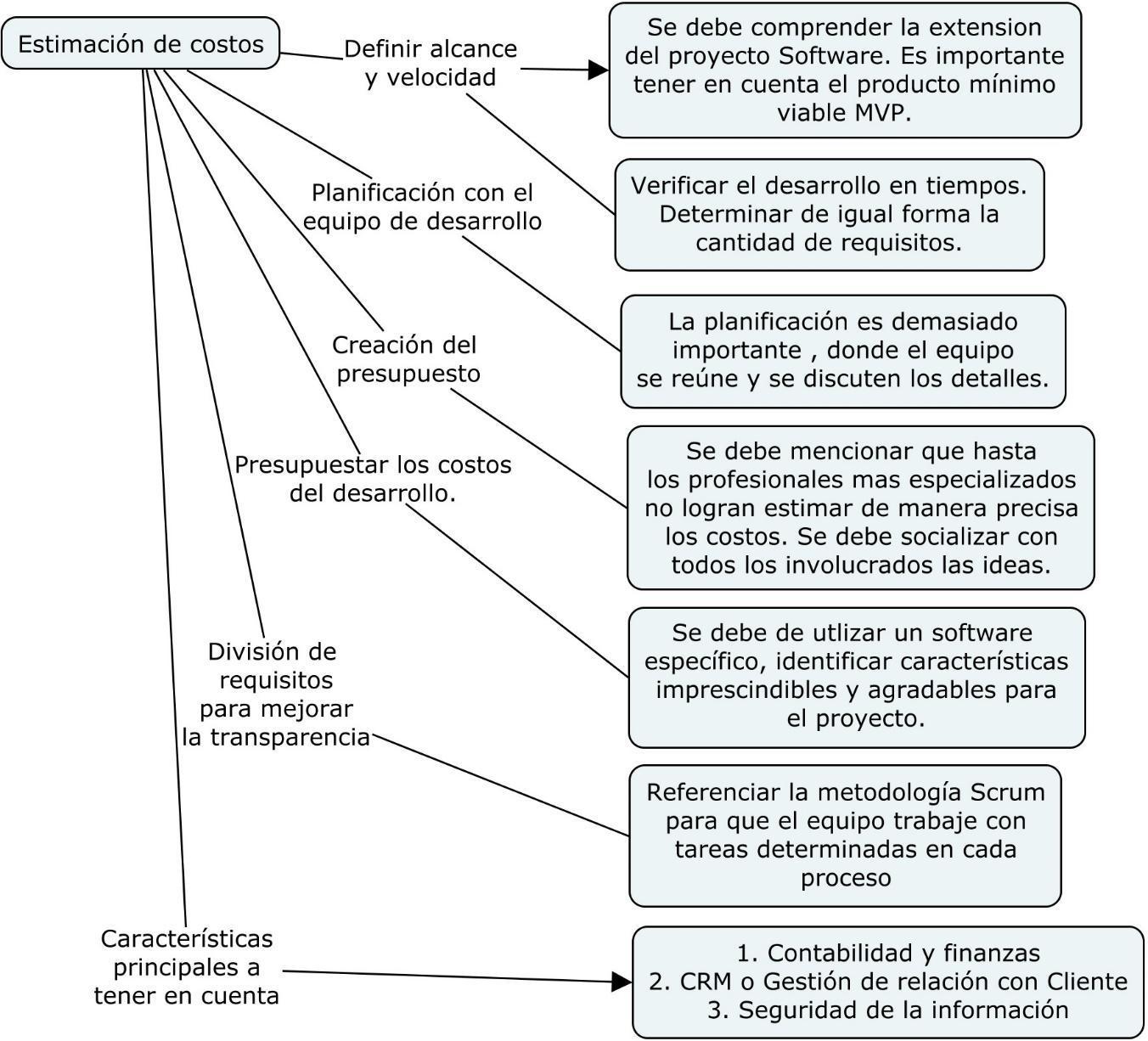
**Factores que afectan el costo del *software***

La estimación del presupuesto en el desarrollo de *software* es una tarea compleja la cual involucra diferentes factores que afectan los tiempos reales y los esfuerzos que se invierten en cada proyecto. Los ingenieros de *software* son los responsables de definir tanto los procesos metodológicos como los organizacionales que pueden generar incertidumbre, sobreestimación o llevar al fracaso del proyecto.



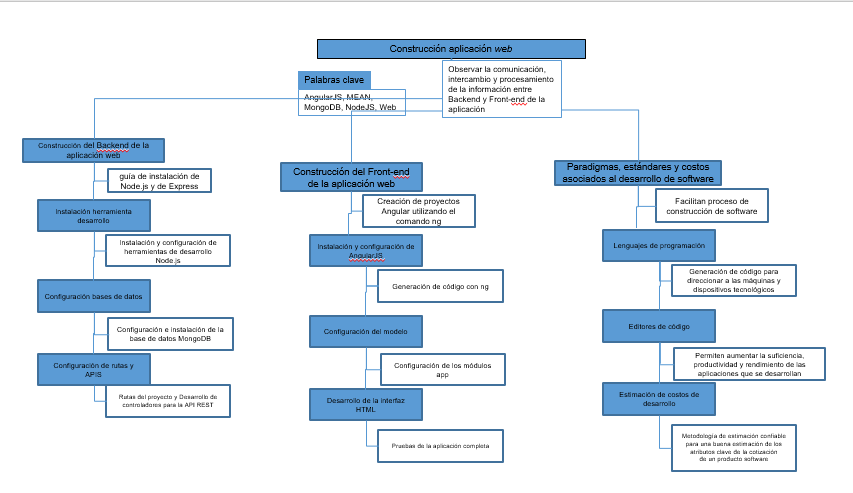
**Estimación de costo de desarrollo del *software***

Para lograr realizar una buena estimación en el desarrollo de *software*, los atributos clave de la cotización de un producto *software* se deben basar en una metodología de estimación confiable, esta debe ser completa y contener todos los detalles como la planificación, estimular el presupuesto y presupuestar los costos. Los cálculos deben ser acordados por todas las partes interesadas y el presupuesto debe ser ejecutado por el equipo de desarrollo. De Igual manera, los riesgos deben medir, comprender y tener en cuenta los siguientes elementos para definir la estimación de costos del proyecto.



1. **SÍNTESIS**

A continuación, se podrá observar de manera estructurada la temática tratada en este componente formativo y lo que hay inmerso en cada una.



1. **ACTIVIDADES DIDÁCTICAS (opcionales si son sugeridas)**

| Descripción de la actividad didáctica | |
| --- | --- |
| Nombre de la actividad | *Emparejamiento entre término y definición* |
| Objetivo de la actividad | Afianzar algunos de los conceptos más importantes asociados a la construcción de un proyecto web completo, tanto del lado del cliente como del lado del servidor. |
| Tipo de actividad sugerida | Arrastrar y soltar. |
| Archivo de la actividad  (anexo donde se describe la actividad propuesta) | Anexo documento en Word denominado M6\_CF4\_Actividad\_didáctica\_1.docx |

1. **MATERIAL COMPLEMENTARIO**

.

| Tema | Referencia APA del material | Tipo de material  (video, capítulo de libro, artículo, otro) | Enlace del recurso o  archivo del documento o material |
| --- | --- | --- | --- |
| Descripción Backend y Fron-end | EDteam. (2020). *¿Qué es Backend y Front-end?* *(guía completa)* [video]. YouTube. | Video | https://youtu.be/50RbVujPPGs |
| Instalación MongoDB | MongoDB. (2021). *Install MongoDB Community Edition on Windows*. | Instalador | <https://docs.mongodb.com/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://youtu.be/BgtB31gXkoA> |
| Guía de instalación Angular CLI | Angular. (2021). *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. (2021). *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> |

1. **GLOSARIO**

| Término | Significado |
| --- | --- |
| *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 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*](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/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* | una *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. |

1. **REFERENCIAS BIBLIOGRÁFICAS**

Felipe. (2021). *Paradigmas de programación: qué son y sus diferentes tipos*. Hosting Plus Colombia. <https://www.hostingplus.com.co/blog/paradigmas-de-programacion-que-son-y-sus-diferentes-tipos>

Ghimire, B. (2021). *Los 12 mejores editores de código y texto para desarrolladores web*. Geekflare. <https://geekflare.com/es/code-text-editors>

Prachi, D. (2022). *Costo y características para desarrollar software para startups*. Cisin. <https://www.cisin.com/coffee-break/es/enterprise/cost-and-features-to-develop-software-for-startups.html>

Universia. (2022). *Los lenguajes de programación más usados en la actualidad*. Universia. <https://www.universia.net/es/actualidad/empleo/lenguajes-programacion-mas-usados-actualidad-1136443.html>

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

1. **CONTROL DEL DOCUMENTO**

|  | Nombre | Cargo | Dependencia  *(Para el SENA indicar Regional y Centro de formación)* | Fecha |
| --- | --- | --- | --- | --- |
| Autor (es) | Santiago Muñoz de la Rosa | Experto Temático | Regional Cauca - Centro de Teleinformática y Producción Industrial | Junio de 2021 – Autores material reúso |
| Gustavo Santis Mancipe | Diseñador Instruccional | Regional Distrito Capital - Centro de Diseño y Metrología | Junio de 2021 – Autores material reúso |
| Alix Cecilia Chinchilla Rueda | Evaluadora Instruccional | Regional Distrito Capital - Centro de Gestión Industrial | Junio de 2021 – Autores material reúso |
| Julia Isabel Roberto | Diseñadora y Evaluadora Instruccional | Regional Distrito Capital - Centro para la Industria de la Comunicación Gráfica | Junio de 2021 – Autores material reúso |
| Luis Eyder Ortiz | Experto Temático | Regional Santander | Septiembre de 2022 |
| Javier Ricardo Luna Pineda | Diseñador Instruccional | Centro De La Industria, La Empresa Y Los Servicios | Octubre de 2022 |
| Andrés Felipe Velandia Espitia | Asesor Metodológico | Regional Distrito Capital – Centro de Diseño y Metrología | Octubre de 2022 |
| Rafael Neftalí Lizcano Reyes | Responsable Equipo Desarrollo Curricular | Regional Santander - Centro Industrial del Diseño y la Manufactura. | Octubre de 2022 |
|  | José Gabriel Ortiz Abella | Corrector de estilo. | Regional Distrito Capital - Centro de Diseño y Metrología. | Noviembre del 2022 (material de reúso). |

1. **Control de cambios**

**(Diligenciar únicamente si realiza ajustes a la Unidad temática)**

|  | Nombre | Cargo | Dependencia | Fecha | Razón del cambio |
| --- | --- | --- | --- | --- | --- |
| Autor (es) |  |  |  |  |  |
|  |  |  |  |  |