



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>My perfect website</title>
5   <meta charset="utf-8" />
6
7   <link rel="preconnect" href="//s3.mysite
8   <link rel="preconnect" href="//www
9
10  <meta name="viewport" content="width
11
12    <script>
13      var mytag = mytag || {};
14      mytag.cmd = mytag.cmd ||
15      (function(){ {
16        var gads = document
17        gads.async = true
18        gads.type = 'text
```

Desarrollo de la API REST

Análisis y desarrollo de *software*
Servicio Nacional de Aprendizaje - SENA

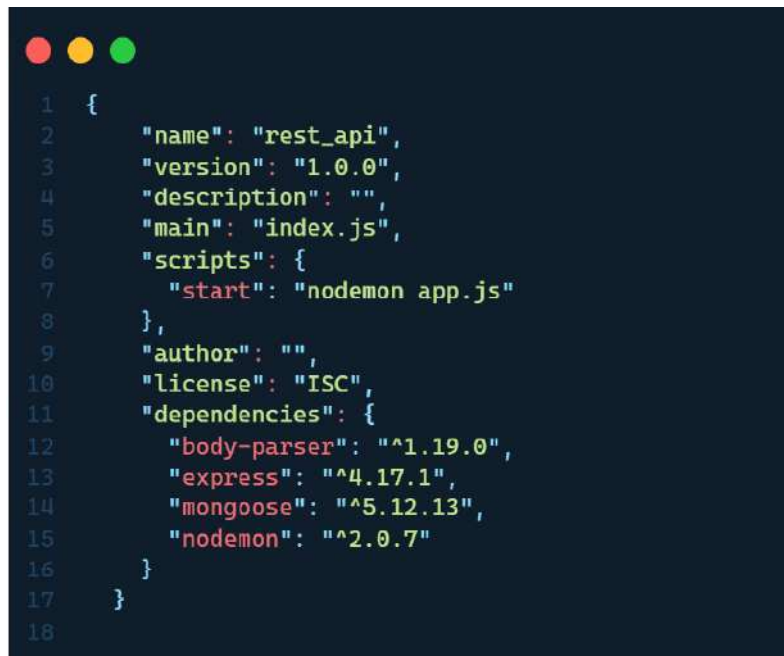
Desarrollo de la API REST

A continuación, se describen de manera sucinta los pasos requeridos para el desarrollo de la API Rest con Node.js

01 Configuración de package.json

Se configura el package.json para implementar el nodemon, con el fin de que el servidor funcione más rápido, como se muestra en la figura 1.

Figura 1. Código fuente del archivo package.json



```
1  {
2    "name": "rest_api",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "start": "nodemon app.js"
8    },
9    "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "body-parser": "^1.19.0",
13     "express": "^4.17.1",
14     "mongoose": "^5.12.13",
15     "nodemon": "^2.0.7"
16   }
17 }
18
```

Referencia Nota. SENA (2021).

02 Creación del archivo app.js

Se crea un nuevo archivo llamado app.js como se muestra en la figura 2, en el cual se implementa lo siguiente:

- 1 Se importa la librería de Express.
- 2 Se configura el listening del servidor, es decir, se precisa el puerto por donde va a escuchar.

- 3** Se crea la ruta por defecto a manera de prueba, que es la URL por donde el servidor responde las solicitudes que se le hacen. La definición de la ruta tiene la siguiente estructura:

`app.METHOD(PATH, HANDLER)`

Figura 2. Ejemplo de conexión entre 10 equipos o terminales

```

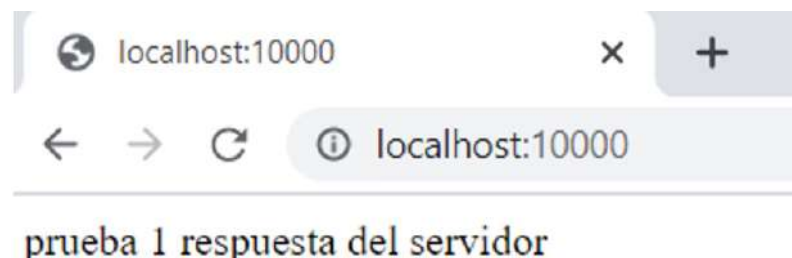
1 // Iniciamos el módulo express con el fin de dar inicio al servidor, evitando varias configuraciones
2 const express = require('express'); // importamos el paquete
3 const app = express();
4 /* SE CREAN LAS RUTAS*/
5 app.get('/', (req, res) => {
6   res.send('prueba 1 respuesta del servidor'); // Ruta por defecto
7 } );
8 //Primero se configura cómo va escuchar el servidor las peticiones
9 app.listen(10000);

```

Referencia Nota. SENA (2021).

Se comprueba la ruta iniciando el servidor con el comando `npm start` desde la terminal del Visual Studio Code. Una vez realizado esto, se debe abrir el browser de la preferencia e inmediatamente se ingresa la dirección local por el puerto donde se está escuchando, que para el caso de este ejemplo es el puerto 10000. A continuación, se presenta la figura que ejemplifica este paso.

Figura 3. Browser con la ruta y puerto donde funciona el servicio



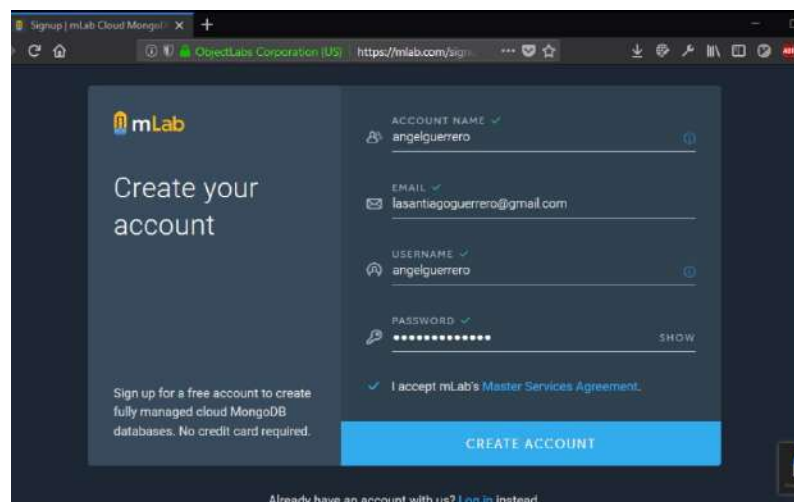
Como se observa el servidor está activo escuchando desde el puerto 10000.

03 Instalación de mongoose y mLab MongoDB

Ahora se necesita una base de datos para almacenar la data que se quiere enviar o registrar, para este caso se utilizará MongoDB. Al respecto, esta tecnología se diferencia de otras porque no guarda los datos en tablas tal y como se hace en las bases de datos relacionales, MongoDB en cambio guarda estructuras de datos en formato BSON (una especificación similar a JSON) con un esquema dinámico. En ese sentido, esto permite que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

En este orden de ideas, una de las formas para utilizar una base de datos MongoDB es con la herramienta mongoose que establece la conexión a la base de datos MongoDB y además permite realizar el mapeo de los esquemas realizados en MongoDB y utilizarlos.

De manera que se instala mongoose desde la terminal de Visual Studio Code así: tiene la siguiente estructura:



```
$ npm install mongoose
```

Una vez instalado mongoose, se utiliza mLab para el cual crea una base de datos gratuita de MongoDB, es decir, es un hosting para bases de datos MongoDB:



Enlace web. <https://mlab.com/>

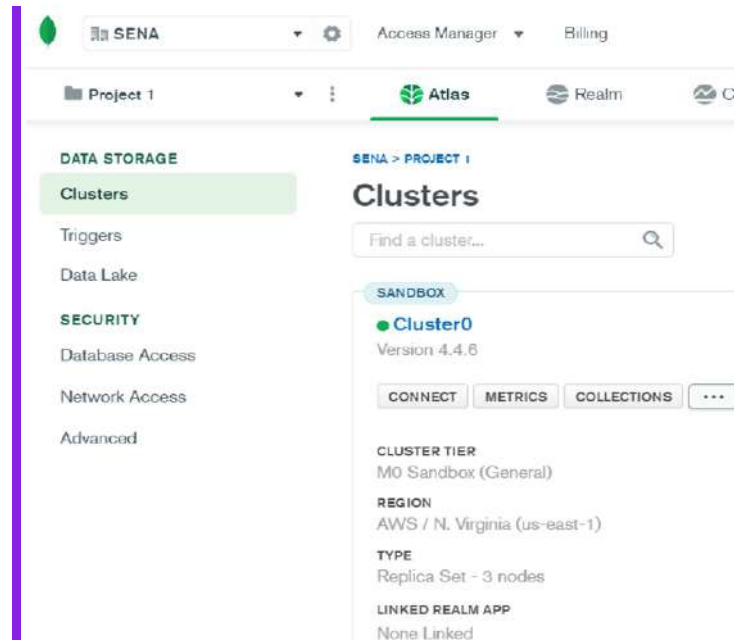
Para la creación de la base de datos y del usuario de prueba se debe seguir la siguiente guía:



Enlace web. <https://docs.mlab.com/>.

A continuación, se presenta la interfaz del hosting de bases de datos mLab /Atlas.

Figura 4. Interfaz de mLab funcionando



Referencia Nota. SENA (2021).

Una vez creada la base de datos se utiliza mongoose para realizar la conexión. Esta codificación se realiza igualmente en el archivo app.js. como se muestra en la siguiente figura.

Figura 4. Código fuente del archivo app.js (conexión BD)

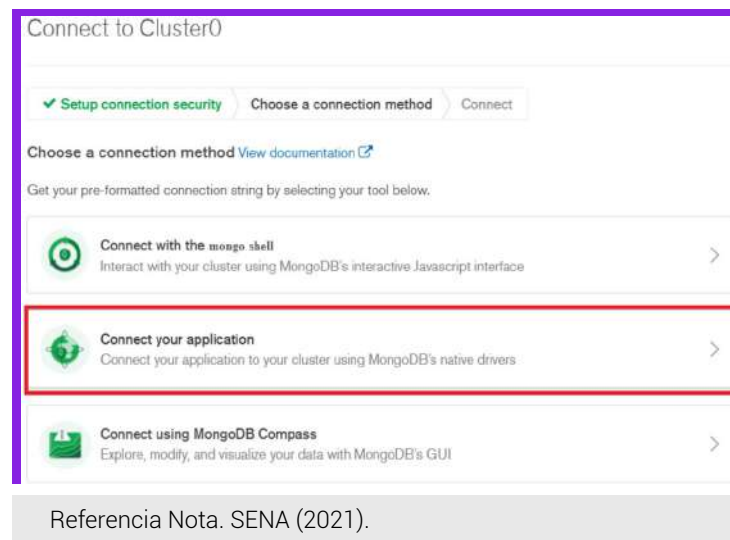
```

1 // Iniciamos el módulo express con el fin de dar inicio al se
nvidon, evitando varias configuraciones
2 const express = require('express'); // importamos el paquete
3 const app = express();
4 const mongoose = require('mongoose');
5 /* SE CREAN LAS RUTAS*/
6 app.get('/', (req, res) => {
7   res.send('prueba 1 respuesta del servidor');// Ruta por de
fecto
8 } );
9 //Conexión a la BD
10 mongoose.connect('mongodb+srv://Sena:Slypherin23@cluster0.klxq
7.mongodb.net/miBD?retryWrites=true&w=majority',
11 {useNewUrlParser: true }, () =>{
12   console.log('Si hay conexión a la BD');
13 });
14 //Primero se configura cómo va escuchar el servidor las peti
ciones
15 app.listen(10000);

```


El código de la conexión se puede generar a través de la plataforma mLab, Por lo tanto, se puede elegir el modo de conexión. Para el caso del ejemplo se elige la opción "Connect your application" como se muestra en la siguiente figura.

Figura 6. Panel de opción de conexión a la base de datos ofrecido por mLab/Atlas



Referencia Nota. SENA (2021).

Es importante anotar que las solicitudes que se envían y reciben deben estar en formato JSON para que sean procesadas de manera correcta, por lo que es necesario convertir o "parsear" a este formato la información que vaya por el método POST. Por lo tanto, se hace necesario instalar el siguiente paquete:

```
$npm install body-parser
```

El archivo app.js queda de la manera como se muestra en la siguiente figura.

Figura 7. Código fuente del archivo app.js aplicando el body parser

```
1 // Iniciamos el módulo express con el fin de dar inicio al ser
  vidor, evitando varias configuraciones
2 const express = require('express'); // importamos el paquete
3 const app = express();
4 const mongoose = require('mongoose');
5 const bodyParser = require('body-parser');
6
7 // llamar al body-parser
8 app.use(bodyParser.json());
9 // Importar las Rutas
10 const postRoute = require('./routes/post');
11 app.use('/servicios', postRoute);
12 //Middlewares: Es la llamada a una función cuando se present
  a un evento en una ruta específica
13 /*app.use('/servicios', () => {
14     console.log('Corriendo la middleware');
```

```

1  });
2  /*
3  /* SE CREAM LAS RUTAS*/
4
5  app.get('/', (req, res) => {
6    res.send('prueba 1 respuesta del servidor');// Ruta po
7    // defecto
8  });
9  /*
10 app.get('/servicios', (req, res) => {
11   res.send('prueba servicios');// Ruta por defecto
12 });
13 /*
14 //Conexión a la BD
15 mongoose.connect('mongodb+srv://Sena:Slypherin23@cluster0.
16 klxq7.mongodb.net/miBD?retryWrites=true&w=majority',
17 {useNewUrlParser: true }, () =>{
18   console.log('Si hay conexión a la BD');
19 });
20 //Primero se configura cómo va escuchar el servidor la
21 s peticiones
22 app.listen(10000);

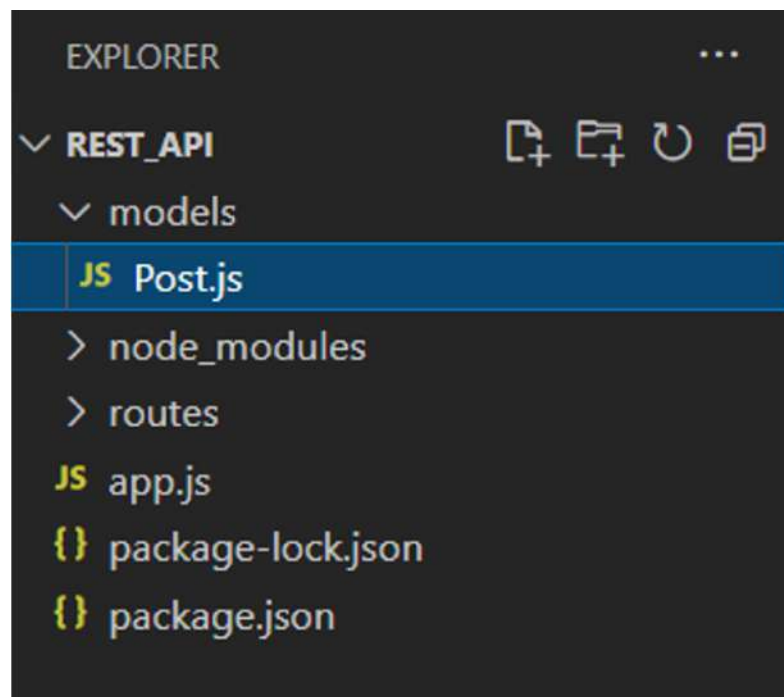
```

Referencia Nota. SENA (2021).

04 Creación del modelo

Ahora se necesita crear los datos de prueba para que se puedan almacenar en la base de datos, por lo tanto, se utilizará mongoose para generar los modelos. En ese sentido, en la estructura de directorios del proyecto se crea una nueva carpeta llamada models como se muestra a continuación.

Figura 8. Estructura de directorios (carpeta models)



Ahora se necesita una base de datos para almacenar la data que se quiere enviar o registrar, para este caso se utilizará MongoDB. Al respecto, esta tecnología se diferencia de otras porque no guarda los datos en tablas tal y como se hace en las bases de datos relacionales, MongoDB en cambio guarda estructuras de datos en formato BSON (una especificación similar a JSON) con un esquema dinámico. En ese sentido, esto permite que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.



En este orden de ideas, una de las formas para utilizar una base de datos MongoDB es con la herramienta mongoose que establece la conexión a la base de datos MongoDB y además permite realizar el mapeo de los esquemas realizados en MongoDB y utilizarlos.

De manera que se instala mongoose desde la terminal de Visual Studio Code así:

Figura 9. Código fuente del archivo post.js (modelo)

```

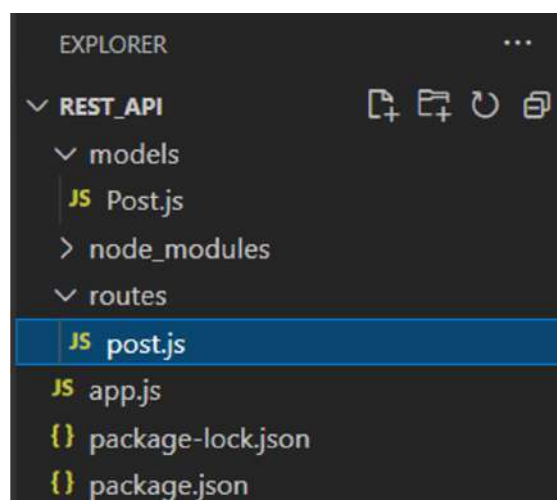
1  const mongoose = require('mongoose');
2  const PostSchema = mongoose.Schema({
3    title: {
4      type: String,
5      require:true
6    },
7    description: {
8      type: String,
9      require:true
10   },
11   date: {
12     type: Date,
13     default: Date.now
14   }
15 });
16 module.exports = mongoose.model('Post',PostSchema);

```

05 Configuración de las rutas

Se define la ruta para el modelo, para esto se crea una carpeta llamada routes en la raíz de la estructura del proyecto y se genera un archivo post.js como se muestra en la siguiente figura.

Figura 10. Estructura de directorios (carpeta routes)



Referencia Nota. SENA (2021).

En el archivo `post.js` se definen las acciones o tratamiento a cada uno de los datos. Esto se hace a partir de las rutas o direccionamiento que hace referencia a la determinación de cómo responde una aplicación a una solicitud de cliente en un determinado punto final, que es una URL (o una vía de acceso) y un método de solicitud HTTP específico (GET, POST, etc.) Es en esta instancia que se invocan acciones para guardar, extraer, eliminar o visualizar datos, a partir del direccionamiento. A continuación, se presenta el código.



Figura 10. Código fuente del archivo `post.js` (rutas)

```

1  router.post('/', async (req, res) => {
2
3      //console.log(req.body); Se utiliza para la respuesta
4      a del post en consola
5      const post = new Post({
6          title: req.body.title,
7          description: req.body.description
8      });
9      try {
10         const savedPost = await post.save(); // método que
11         e guarda en la BD
12         res.json(savedPost);
13     } catch (error) {
14         res.json({message: error});
15     }
16 });
17
18 /**
19  * Bloque para mostrar solo un post por el Id
20  */
21
22 router.get('/:postId', async (req, res) => {
23     try {
24         const post = await Post.findById(req.params.postId);
25         // encuentra por id
26         res.json(post);
27     } catch (error) {
28         res.json({message: error});
29     }
30 });
31
32 /**
33  * Bloque para borrar un post
34  */
35
36 router.delete('/:postId', async (req, res) => {
37     try {
38         const removedPost = await Post.remove({_id: req.params.postId});
39         //borra
40         res.json(removedPost);
41     } catch (error) {
42         res.json({message: error});
43     }
44 });
45
46 /**
47  * Actualizar un post
48  */

```

```

1  router.post('/', async (req, res) => {
2
3      //console.log(req.body); Se utiliza para la respuesta
4      const post = new Post({
5          title: req.body.title,
6          description: req.body.description
7      });
8      try {
9          const savedPost = await post.save(); // método que
10         // guarda en la BD
11         res.json(savedPost);
12     } catch (error) {
13         res.json({message: error});
14     }
15
16 });
17
18 /**
19  * Bloque para mostrar solo un post por el Id
20  */
21
22 router.get('/:postId', async (req, res) => {
23     try {
24         const post = await Post.findById(req.params.postId);
25         // encuentra por id
26         res.json(post);
27     } catch (error) {
28         res.json({message: error});
29     }
30 });
31
32 /**
33  * Bloque para borrar un post
34  */
35
36 router.delete('/:postId', async (req, res) => {
37     try {
38         const removedPost = await Post.remove({_id: req.params.postId});
39         // borra
40         res.json(removedPost);
41     } catch (error) {
42         res.json({message: error});
43     }
44 });
45
46 /**
47  * Actualizar un post
48  */
49
50 router.patch('/:postId', async (req, res) => { // patch para
51     // actualizar
52     try {
53         const updatePost = await Post.updateOne( // actualizar
54             // de uno en uno
55             {_id: req.params.postId},
56             {$set: {title: req.body.title}});
57         res.json(updatePost);
58     } catch (error) {
59         res.json({message: error});
60     }
61 });
62
63 module.exports = router; // devuelve como módulo lo que se
64 // le asigna a route

```

Referencia Nota. SENA (2021).

La clase `express.router` sirve para crear manejadores o handlers de rutas montables y modulares. Una instancia router es un sistema que invoca una función cuando se presenta algún evento (middleware) y además presenta un direccionamiento completo; por este motivo, a menudo se conoce como una "miniaplicación" (Express, 2021).

La palabra clave `async` se añade a las funciones para indicar que devuelvan una **promesa** en lugar de devolver directamente el valor. Adicionalmente, esto permite que las funciones síncronas puedan evitar cualquier potencial de sobrecarga que viene con correr el soporte por usar `async`. De esta manera, cuando una función es declarada `async` con solo añadir la manipulación necesaria, el motor de JavaScript puede optimizar el programa por sí mismo (Mozilla, 2021).

La expresión `await` provoca que la ejecución de una función `async` de una pausa hasta que una promesa sea terminada o rechazada, y vuelve de nuevo a la ejecución de la función `async` después de finalizada. Al regreso de la ejecución, el valor de la expresión `await` se da por una promesa terminada (Mozilla, 2021).

Referencias bibliográficas

Mozilla. (2021). `Await`. MDN Web Docs.



Enlace web. <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/await>