



## **API REST**

### **Implementación de una Agenda Virtual basada en Node.js**

Fernández Gabriela V.- 26.488.783

Linares Francis V.-28.096.888

Loaiza Ricardo V.- 28.322.840

*Universidad Valle del Momboy*

Facultad de Ingeniería

Backend I

Ing. Edgar Valero

Ing. Rogelio Gómez

Carvajal, Trujillo, Venezuela

Abril , 2021



## Conceptos Básicos

### Programación Asíncrona

Un modelo asincrónico permite que sucedan múltiples cosas al mismo tiempo. La Programación Asíncrona nos va a permitir ejecutar nuestros procesos en varios hilos de ejecución o lo que se conoce como multithreading. De esta manera el bloqueo ya no va a existir porque los usuarios no van a tener que esperar a que un hilo se libere para que otro pueda entrar.

### Clases y Herencias

Las clases no son más que una función de JavaScript que permiten una mejor sintáctica sobre la herencia basada en prototipos. La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente y esta permite compartir automáticamente métodos y datos entre clases, subclasses y objetos.

### API REST

Una API (Interfaz de programación de aplicaciones) es un conjunto de requisiciones que permite la comunicación de datos entre aplicaciones, para eso, la API utiliza requisiciones HTTP responsables de las operaciones básicas necesarias para la manipulación de datos como lo es el GET o el POST. Rest, que es la abreviación de (Representational State Transfer) es un conjunto de restricciones que se utilizan para que las solicitudes HTTP cumplan con las directrices definidas en la arquitectura.

### SCRUM

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos

## Agenda virtual

### Necesidades y Objetivos

El proyecto es una iniciativa para crear una agenda virtual que albergue la información del trabajador de acuerdo al contexto o cargo en el que se encuentra en su empresa, se determinó que la mejor forma de distribuir este tipo de información era a través de la implementación de herramientas tecnológicas. A partir de esa necesidad se crea este proyecto en el cual se busca responder a los requerimientos solicitados así como también a toda la parte del desarrollo que se aplica desde el área de la informática y con ello aplicar lo aprendido en una mayor escala.

Una vez consideradas los temas propuestos del análisis de los conceptos básicos, es posible definir los objetivos del proyecto, los cuales se concentran principalmente en el área de la



programación, específicamente la creación de un servidor junto con una base de datos. Se definen por tanto los objetivos obtenidos a medida que se avanzaba para transformarlos en criterios e implementaciones tecnológicas y se ha aceptado que la estrategia a seguir en este campo debe tener como objetivo general el siguiente.

- **Objetivo General**

- Diseñar y construir una Agenda Virtual permitiendo a los trabajadores de una determinada compañía organizar sus próximas o presentes actividades o eventos como herramienta de gestión de productividad y tiempo.

- **Objetivos Específicos**

- Diseñar una estructura sólida de innovación y fácil acceso para los trabajadores con la participación del administrador.

- Hacer un proyecto adaptable y amigable con el usuario que pueda responder tanto a sus necesidades como a las del administrador.

- Otorgar a los trabajadores de la compañía una herramienta de consulta de fácil utilización para que este pueda tomar decisiones de acuerdo a su agenda a tiempo para la.

- Desarrollar y facilitar la comunicación entre el empleado y el empleador, y a su vez la productividad del trabajador en su empresa.

- Desarrollar una base de datos en las que se almacenen los eventos o actividades siendo estos revisados por el administrador para verificar su autenticidad.

## **Diagramas de Contexto**

El sistema de información propuesto tiene como propósito mayor, mejorar y controlar las actividades diarias de los trabajadores relacionadas con las actividades agendadas que aún no se han completado, así como de apoyar y agilizar indirectamente el proyecto de toma de decisiones en el área específica de su empresa

Se incluye el diagrama de contexto que es “un diagrama que define los límites entre el sistema, o parte del sistema, y su ambiente, mostrando las entidades que interactúan con él.”

### ***Diagrama de Contexto***

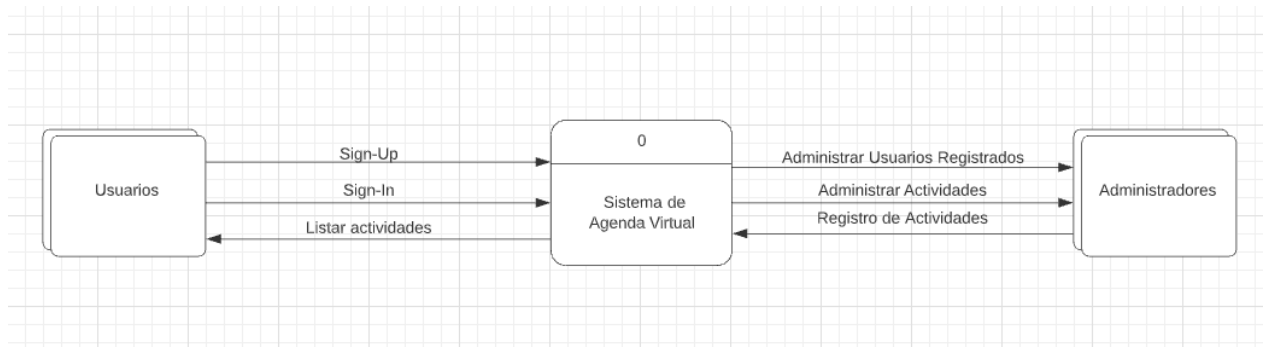


Figura 1.1

## Diagrama 0

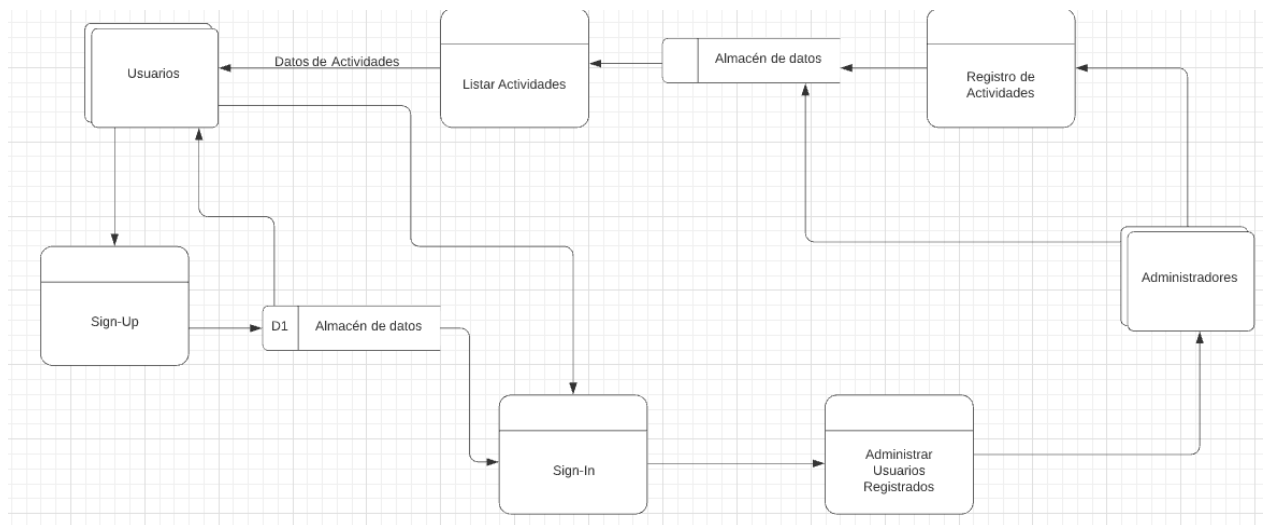


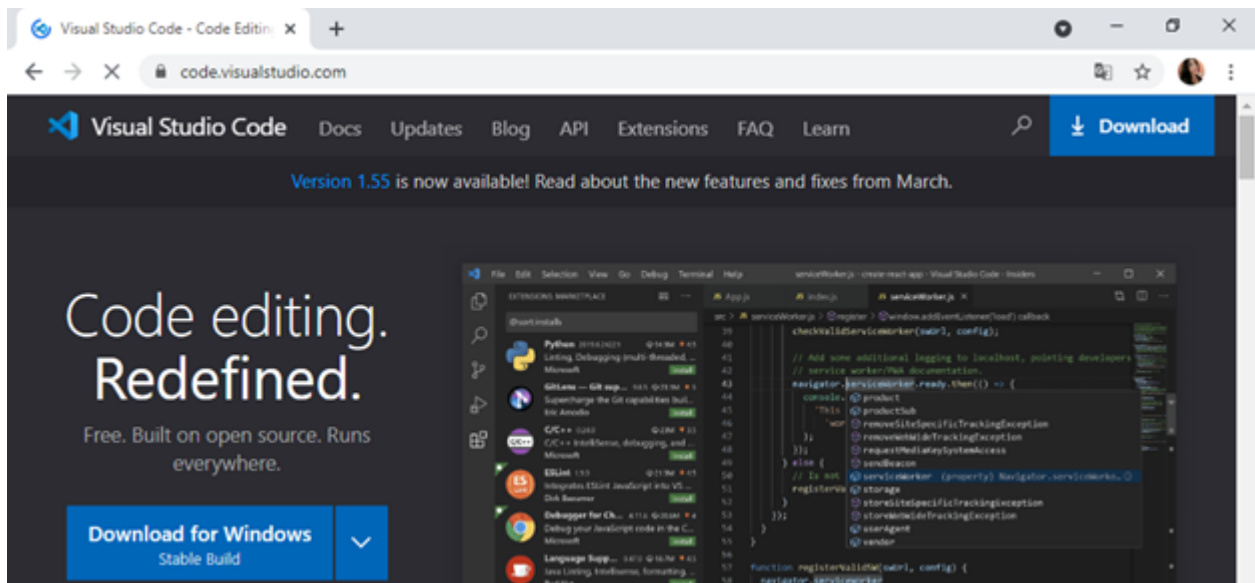
Figura 1.2

## Herramientas de desarrollo

### VSCode

Visual Studio Code (VSCode) es un editor de código que utilizaremos para nuestro proyecto, este está disponible para Windows, macOS y Linux. Viene con soporte integrado para JavaScript, TypeScript y Node.js. Además, tiene un rico ecosistema de extensiones para otros lenguajes.

Su instalación es realmente fácil, simplemente nos dirigimos a su página oficial en <https://code.visualstudio.com> y descargamos el editor según la versión que nos convenga, una vez descargado procedemos a instalarlo para seguidamente proceder a su utilización.



## Postman

Postman es una plataforma de colaboración para el desarrollo de API. El interés fundamental de Postman es que lo utilizemos como una herramienta para hacer peticiones a APIs y generar colecciones de peticiones que nos permitan probarlas de una manera rápida y sencilla. En pocas palabras, Postman nos va a servir a nosotros para probar peticiones HTTP. Simplemente lo descargamos para nuestro sistema operativo e instalamos.



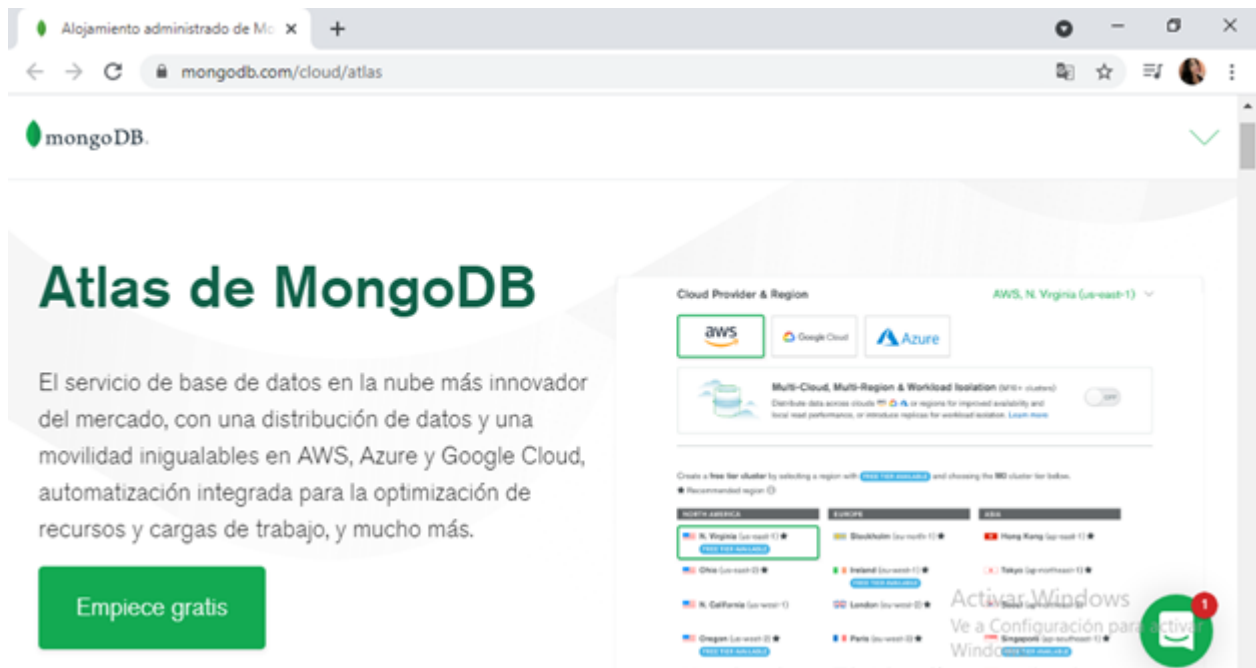
## Node.js

Con node.js podemos hacer uso de sockets para una comunicación real cliente- servidor, servidor-cliente. También podemos trabajar con cargas simultáneas, servidores locales y remotos con información en tiempo real, conexiones a base de datos y creación de servicios REST en segundos. Para su instalación simplemente nos dirigimos a la página oficial de Node.js, una vez ahí es súper fácil de instalarlo. Hacemos click en el botón ya sea recomendado para la mayoría o últimas características, esto descarga un instalador que seguidamente lo ejecutamos e instalamos.



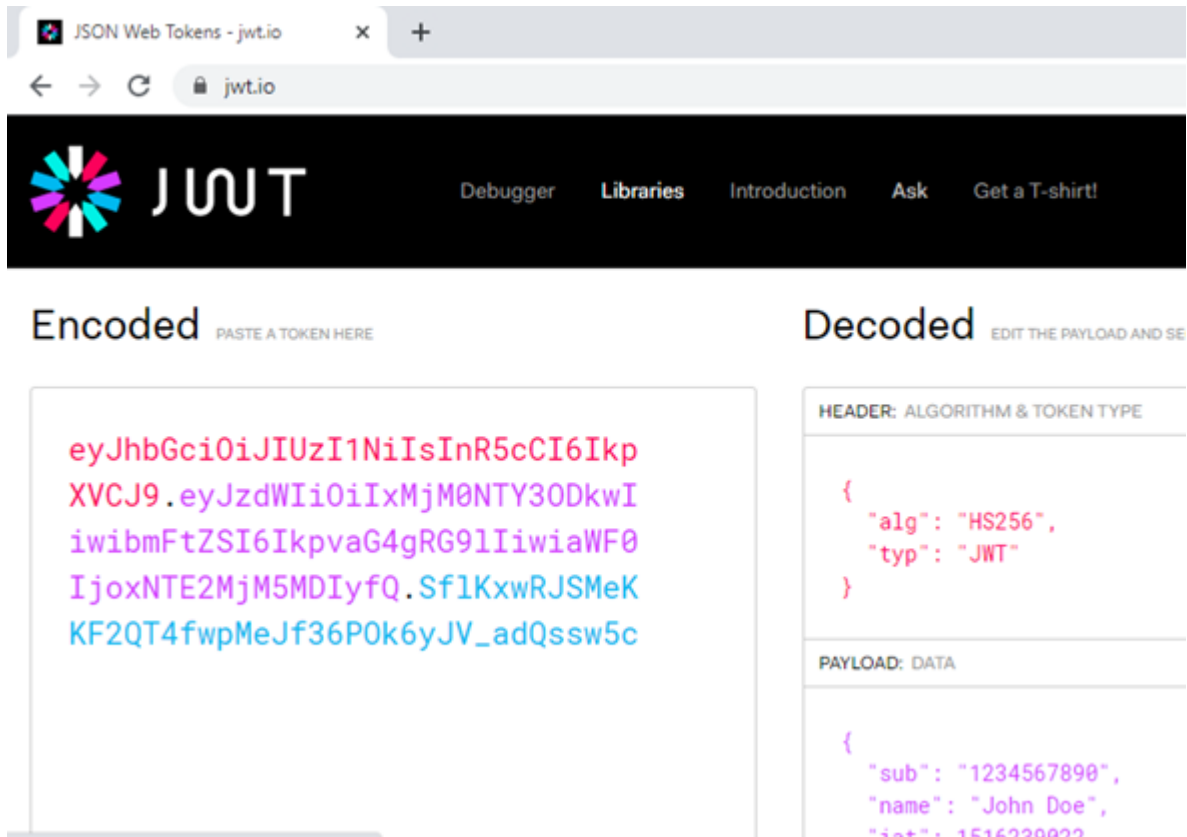
## MongoDB

MongoDB es un sistema de base de datos NoSQL, orientado a documentos y de código abierto. Para nuestra aplicación vamos a trabajar con MongoDB en la nube, llamado MongoDB Atlas, es el mismo MongoDB, solo que ya es un cliente que está configurado en la nube para que nosotros simplemente lo consumamos.



## JWT

JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros. Este token llevará incorporada la información del usuario que necesita el servidor para identificarlo, así como información adicional que pueda serle útil.



The screenshot shows the JWT.io website interface. The top navigation bar includes links for Debugger, Libraries, Introduction, Ask, and Get a T-shirt!. The main content area is split into two panels: 'Encoded' and 'Decoded'.

**Encoded Panel:** Labeled 'PASTE A TOKEN HERE', it contains a long, multi-colored string representing an encoded JWT token: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKF2QT4fwpMeJf36P0k6yJV_adQssw5c`.

**Decoded Panel:** Labeled 'EDIT THE PAYLOAD AND SEE', it displays the decoded structure of the token in two sections:

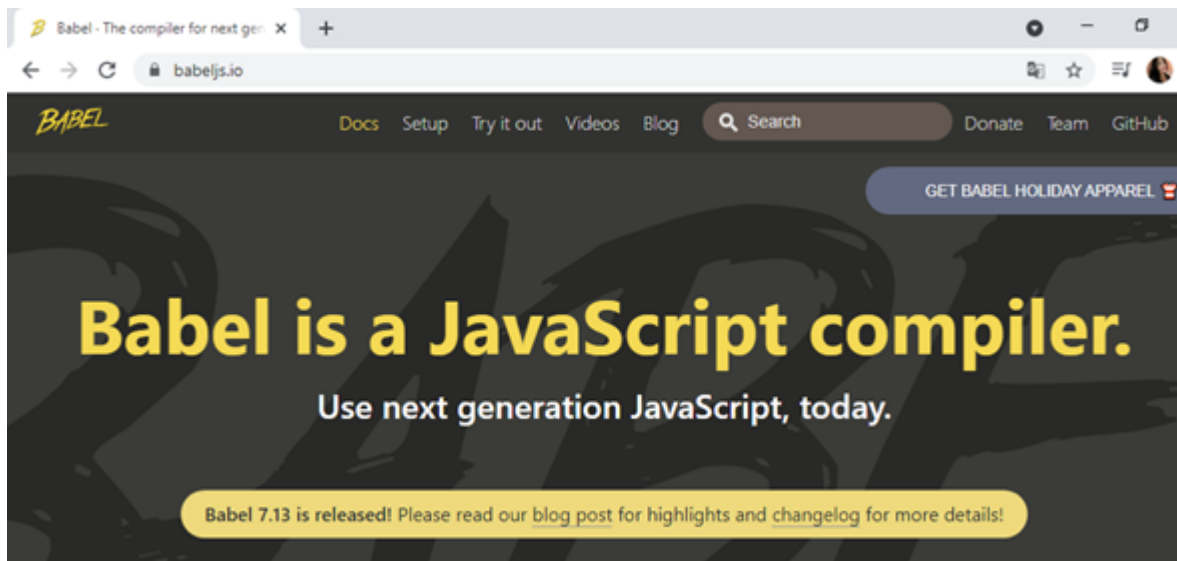
- HEADER: ALGORITHM & TOKEN TYPE:**

```
{  "alg": "HS256",  "typ": "JWT"}
```
- PAYLOAD: DATA:**

```
{  "sub": "1234567890",  "name": "John Doe",  "iat": 1516239022}
```

## Babel

Babel es una herramienta que nos permite transformar nuestro código JS de última generación (o con funcionalidades extras) a JS que cualquier navegador o versión de Node.js entienda.



The screenshot shows the Babel website homepage. The header includes links for Docs, Setup, Try it out, Videos, Blog, a Search bar, and links to Donate, Team, and GitHub. A banner at the top right says 'GET BABEL HOLIDAY APPAREL'. The main heading reads 'Babel is a JavaScript compiler.' followed by the tagline 'Use next generation JavaScript, today.' A yellow banner at the bottom states 'Babel 7.13 is released! Please read our [blog post](#) for highlights and [changelog](#) for more details!'





## Bcrypts

Bcrypt es una función de hash de contraseña o de encriptación de contraseñas, esta lleva incorporado un valor llamado salt, que es un fragmento aleatorio que se usará para generar el hash asociado a la password, y se guardará junto con ella en la base de datos. A través del comando npm i bcryptjs procedemos a instalarlo.



## API REST

### Función

Para el desarrollo de la API REST se utilizó autenticación por token, se implementó utilizando jsonwebtoken. La misma permite registrar, editar, actualizar y eliminar las actividades y/o reuniones correspondientes a una determinada área de la Institución, para poder acceder a las rutas HTTP correspondientes para llevar a cabo dichas funcionalidades, se necesita tener permisos o un rol de administrador (“admin”), de forma contraria, el acceso será denegado. Dicho rol se añade en el momento del sign-up o la creación de un nuevo usuario, tenga en cuenta que, junto a los datos suministrados por el usuario que serán guardados en la base de datos, el rol que él mismo pueda tener se guardará a partir de su \_id correspondiente. En esta oportunidad se está administrando y controlando el acceso a las rutas teniendo como referencia los roles de usuario, los cuales pueden ser: Administrador (“admin”) o Usuario (“user”). Al momento de registrar los



usuarios se solicitarán los siguientes datos: Nombre completo, nombre de usuario, correo electrónico, contraseña, área de trabajo.

Por otra parte, para el registro de las actividades y/o reuniones futuras se solicitan los siguientes datos: Nombre de la reunión, fecha, tipo de reunión, área a la que va dirigida, descripción. Cabe destacar que, tanto los usuarios comunes como los administradores pueden listar las actividades y/o reuniones registradas. En cuanto a la información de los usuarios, solo los administradores tienen permiso para acceder a la misma. El administrador puede listar todos los usuarios registrados en el sistema y, además puede buscar un usuario teniendo como referencia su username.

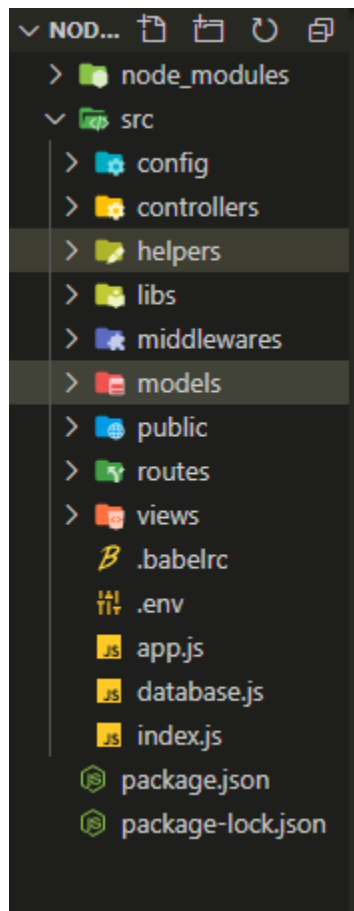
### **Consideraciones**

Para que el programa funcione correctamente primero debe instalar todos los módulos utilizados, los cuales puede visualizar desde el archivo package.json. Para instalarlo abra una nueva ventana de terminal y ejecute el comando: **npm i**

Para iniciar el servidor ejecute el comando: **npm run dev**

### **Estructura del proyecto**

Lo primero será crear una carpeta donde se alojará el código de nuestra API REST, en esta ocasión la carpeta tiene por nombre “nodejsProject”, a continuación se crearán las siguientes carpetas:



Cada una de las carpetas será explicada a continuación.

## Libs

La carpeta "libs" contiene las funciones que implementaremos al momento de inicializar el servidor. El archivo "automaticSetUp" contenido en esta carpeta, es una función que nos permite crear los roles de usuarios que se utilizarán para poder administrar el manejo de rutas. Tenga en cuenta que, dichos roles son creados la primera vez que iniciamos el servidor, una vez guardados en la base de datos, usted tendrá acceso a ellos siempre que vuelva a iniciar el servidor. Lo primero será requerir el modelo de datos que se está utilizando para guardar esta información en nuestra base de datos. Dicho modelo será explicado más adelante.

La función lleva por nombre createRol() y es una función asíncrona encargada de verificar la existencia de los roles en la base de datos y crear los mismos en caso de ser necesario. Lo primero será consultar la colección "Rol" para saber la cantidad de documentos que esté alojada, para ello utilizamos el método estimatedDocumentCount(), esto nos retornará un valor que será guardado en la constante "count". Lo siguiente será verificar si existen documentos en la colección, en este caso utilizamos "return" para terminar de ejecutar la función. En caso contrario, es decir, donde no existan documentos, se crearán los documentos correspondientes a

los roles de usuarios: "user" y "admin", se utiliza `save()` para guardar dicha información en la base de datos. Recuerde que, MongoDB genera un identificador (`_id`) automáticamente al registrar los datos, él mismo será utilizado como referencia para el resto de las funciones utilizadas en la API REST. Por último, exportamos la función que será requerida desde el archivo `app.js` donde se encuentra nuestro servidor, y ejecutada siempre que éste se inicie.

```
src > libs > automaticSetup.js
1  //Import
2  const Rol = require('../models/Rol');
3
4  //Generar roles
5  const createRol = async () => {
6      try {
7          //Contador de roles
8          const count = await Rol.estimatedDocumentCount();
9          if(count > 0) return;
10         //Crear roles
11         const roles = await Promise.all([
12             new Rol({name: 'user'}).save(),
13             new Rol({name: 'admin'}).save()
14         ]);
15     } catch (err) {
16         console.error(err);
17     };
18 };
19
20 //Export
21 module.exports = createRol;
22
```

## Servidor

En el archivo `app.js` se encuentran las acciones necesarias para poner en marcha nuestro servidor.

Requerimos el módulo `express` y `morgan`. Luego importamos la función `createRol()` creada anteriormente, la cual se encuentra en la carpeta "libs". Asimismo, importamos las rutas que se utilizarán: `activities`, `auth` y `user`; todas se encuentran alojadas en la carpeta "routes", la cual se explicará más adelante. Creamos una constante `app` la cual utilizará el módulo `express` para poder inicializar el servidor. Seguido a esto, llamamos a la función `createRol()` para comenzar la creación de los roles en caso de ser necesario. Será necesario el uso de `morgan`, para ello utilizamos `app.use(morgan('dev'))` y especificamos que estamos trabajando en modo de desarrollo. Asimismo, utilizamos `express.json()` para poder mostrar los archivos en formato JSON con los que se estará trabajando.

A continuación, definimos las rutas correspondientes y por último exportamos `app`. Tenga en consideración que en este archivo se encuentra la configuración de nuestro servidor.

```
//Modules
const express = require('express');
const morgan = require('morgan');

//Import Roles
const createRol = require('./libs/automaticSetup');

//Import Routes
const activities = require('./routes/act.routes');
const auth = require('./routes/auth.routes');
const user = require('./routes/user.routes');

//Inicializar
const app = express();
createRol();

//Middleware
app.use(morgan('dev'));
app.use(express.json());

//Routes
//Actividades
app.use('/activities', activities);
//Auth
app.use('/auth', auth);
//User
app.use('/user', user);

//Export
module.exports = app;
```

Desde el archivo index.js que será el principal de nuestro proyecto, importamos el archivo app.js con las configuraciones especificadas anteriormente. Además requerimos el archivo database.js donde se encuentra la configuración de nuestra base de datos. Configuramos el puerto que se utilizará, en nuestro caso el 4000 y por último escuchamos un evento que nos permitirá iniciar el servidor. Siéntase libre de trabajar con el puerto que desea, puede configurar el número de puerto cambiando el 4000 por el que usted desee y además, desde el archivo .env puede configurar el puerto con el que estará trabajando.

```
src > index.js > ...
1 //Import app
2 const app = require('./app.js')
3
4 //Database
5 require('./database');
6
7 //Settings
8 app.set('port', process.env.PORT || 4000);
9
10 //Server inicializacion
11 app.listen(app.get('port'), () => {
12 |   console.log('Server on port', app.get('port'));
13 | });
14
```

## Base de datos

Desde el archivo database.js requerimos mongoose, el cual nos permitirá trabajar con la base de datos y manipularla. En la constante uri definimos la ruta de acceso a nuestra base de datos, en este caso se está trabajando con una base de datos creada a través de MongoDB Atlas (en la nube) la cual lleva el nombre por defecto "myFirstDatabase" accedemos a ella mediante el usuario "francisblm" el cual tiene permiso de administrador. Tenga en cuenta que, debe definir los permisos para cada usuario de su equipo que necesite tener acceso a la base de datos, si el usuario que está ingresando para conectarse a la base de datos no tiene permiso para manipularla, el registro, edición y eliminación de datos no funcionará. Mediante la función database () configuramos la base de datos, para ello utilizamos mongoose.connect() lo cuál nos permitirá conectarnos a la base de datos, como parámetros le pasamos la uri y un objeto con las configuraciones. Es recomendable dejar las configuraciones tal cual para evitar inconvenientes al momento de ejecutar el programa. Por llamamos a la función database()

```
database.js:7 ...
//Modules
const mongoose = require('mongoose');

//Connection URI
const uri = 'mongodb+srv://francisblm:lfbcZap7fJJXIUA@project.kvxee.mongodb.net/myFirstDatabase?retryWrites=true&w=majority';


//Connection DB
async function database() {
  try {
    await mongoose.connect(uri, {
      useCreateIndex: true,
      useNewUrlParser: true,
      useFindAndModify: true,
      useUnifiedTopology: true
    });
    console.log('Database is connected');
  } catch (err) {
    console.error(err);
  }
};

database();
```

## Modelo de Datos

En la carpeta de models (modelos) encontramos los modelos del proyecto, en este caso de la base de datos que usaremos que será MongoDB y aprovecharemos su modelado de datos flexible.

Primero que todo en el archivo Act.js importamos el módulo mongoose igualmente crearemos la constante Schema que requerirá el módulo mongoose, seguidamente crearemos el ActSchema que creará un modelo de datos para las Actividades que vayamos a ingresar a nuestro proyecto, esto significa que tendrá que cumplir con los requisitos que se ven en la imagen que aparece abajo, después de crear este esquema exportamos este modelo que se usará en los controladores, pero eso lo veremos más adelante.

```
src > models >  Act.js > ...
1 //Import
2 const mongoose = require('mongoose');
3 const { Schema } = mongoose;
4
5 //Schema
6 const ActSchema = new Schema({
7   name: {type: String, required: true},
8   date: {type: Date, required: true},
9   type: {type: String, required: true},
10  area: {type: String, required: true},
11  description: {type: String, required: true}
12 },
13 {
14   timestamps: true,
15   versionKey: false
16 });
17
18 //Export model
19 module.exports = mongoose.model('Act', ActSchema);
```

En el archivo Rol.js importamos nuevamente los módulos mongoose que usará el esquema, después se crea un nuevo esquema llamado Rol, que nos servirá para establecer los roles de cada usuario, creamos el esquema con su nombre, que se tomara en cuenta y tambien “versionKey” que sirve como propiedad para obtener una id de la revisión interna del documento. Acto final exportamos este modelo como el Rol de los usuarios y que se usará en los controladores.

```
src > models >  Rol.js > ...
1 //Import
2 const mongoose = require('mongoose');
3 const { Schema } = mongoose;
4
5 //Schema
6 const RolSchema = new Schema({
7   name: {type: String}
8 },
9 {
10   versionKey: false
11 });
12
13 //Export model
14 module.exports = mongoose.model('Rol', RolSchema);
```

Por último en User.js en el cual crearemos el modelado y qué valores tendrán los usuarios, repetimos los pasos anteriores de importar el módulo mongoose así como crear un Schema que lo utilizara, excepto que esta vez utilizaremos el módulo bcrypt que usaremos para encriptar las passwords, recordemos que bcrypt es una función de hashing de passwords, después de importar estos módulos nos encontramos con el esquema Users que se rige por supuesto con ciertos valores que tienen que cumplir con ciertos requisitos, también vemos el Rol donde se toma como referencia el Rol y el tipo.

```
src > models >  User.js > ...
1 //Import
2 const mongoose = require('mongoose');
3 const { Schema } = mongoose;
4 const bcrypt = require('bcryptjs');
5
6 //Schema
7 const UserSchema = new Schema({
8   fullName: {type: String, required: true},
9   username: {type: String, required: true, unique: true},
10  email: {type: String, required: true, unique: true},
11  area: {type: String, required: true},
12  password: {type: String, required: true},
13  rol: [{
14    ref: "Rol",
15    type: Schema.Types.ObjectId
16  }]
17 },
18 {
19   timestamps: true,
20   versionKey: false
21 });
22
```



Después de esto vemos el “UserSchema” que nos servirá para cifrar la password como hemos dicho anteriormente, después comparamos esa password con la recibida para comprobar su autenticidad, finalmente exportamos el modelo de los usuarios que usaremos en los controladores.

```
22
23 //Cifrar password
24 UserSchema.statics.cifrar = async (password) => {
25   const salt = await bcrypt.genSalt(10);
26   return await bcrypt.hash(password, salt);
27 };
28
29 //Comparar password
30 UserSchema.statics.comparePass = async (password, received) => {
31   return await bcrypt.compare(password, received);
32 };
33
34 //Export model
35 module.exports = mongoose.model('User', UserSchema);
```

## Controladores

### Autenticación

Dentro de nuestros controladores encontraremos un archivo llamado “auth.controller” donde tendremos nuestra clase users y dentro estableceremos la creación de nuestros usuarios, creando una nueva instancia, especificaremos los campos que necesitamos.

```
//Controller
class Users {
  //Sing-Up
  async singup (body) {
    try {
      const { fullName, username, email, area, password, rol } = body;
      const newUser = new User({
        fullName,
        username,
        email,
        area,
        password: await User.cifrar(password)
      });
    }
  }
}
```

Seguidamente comprobamos si el usuario ingresó un rol (admin, user) al momento de registrarse. Si la propiedad existe, recorremos la colección donde están guardados los roles usando el método find(). Luego asignamos al registro del usuario “newUser” la propiedad rol, para esto usamos la

constante `foundRol` y un map para recorrer los objetos que tiene y poder extraer el id que contiene dicho rol, los roles se les asigna y se guardan utilizando el id que genera MongoDB.

En caso de que el usuario no ingrese la propiedad `rol`, se le asigna el rol “user” automáticamente. Para eso definimos el método `findOne()`, esto nos retorna el objeto cuyo nombre sea “user” y luego se le asigna ese id al registro del usuario. Por último, guardamos el registro y seguidamente se envía un token al user y usamos una propiedad de json web token llamada `signin`, para esto le pasamos el id del usuario que acabamos de registrar, una palabra secreta que está en la carpeta “config/config” y un tiempo para que expire, en este caso 120 días

```
//Rol
if (rol) {
  const foundRol = await Rol.find({name: {$in: rol}});
  newUser.rol = foundRol.map(roles => roles._id);
} else {
  const role = await Rol.findOne({name: 'user'});
  newUser.rol = [role._id];
};

//Guardar
const userSaved = await newUser.save();
//Generar un token
const token = jwt.sign({id: userSaved._id}, config.SECRET, {
  expiresIn: 10368000 //120 días
});
return {token};
} catch (err) {
  console.error(err);
};
```

Seguidamente comprobamos el registro del usuario, si el usuario existe, es decir, si se encuentra registrado pasamos a comprobar la contraseña, al realizar la comparación si la contraseña introducida coincide con la contraseña ingresada en el registro entonces procede a enviarle el token, de no ser así, esta le retornará un mensaje especificando que la contraseña es incorrecta. En todo caso de que el usuario no esté registrado retornará un mensaje especificando que el usuario no se encuentra registrado.

```
//Sing-In
async singin (body) {
  try {
    //Comprobar registro del usuario
    const userRegis = await User.findOne({email: body.email}).populate('rol');
    if (userRegis) {
      //Comprobar contraseña
      const comparePassword = await User.comparePass(body.password, userRegis.password);
      if(comparePassword) {
        const token = jwt.sign({id: userRegis._id}, config.SECRET, {
          expiresIn: 10368000 //120 días
        });
        return { token };
      } else return 'Contraseña incorrecta';
    } else return 'El usuario no se encuentra registrado';
  } catch (err) {
    console.error(err);
  }
};
};
```

## Usuarios

```
//Controller
class UserControl {
  //Get by username
  async getUsername (iusername) {
    try {
      const user = await Rol.findOne({username: {$in: iusername}});
      return user;
    } catch (err) {
      console.error(err)
    }
  };

  //Get all users
  async getUsers () {
    try {
      const user = await User.find();
      return user;
    } catch (err) {
      console.error(err);
    }
  };
};
};
```

## Actividades

De igual forma dentro de nuestros controladores encontramos también un archivo llamado “act.controller”, ahí podemos encontrar nuestra clase Acts que contiene todas las funciones que tendrán nuestra agenda de actividades, tanto para crear, mostrar, actualizar o eliminar una actividad.

```
class Acts {  
  //Add  
  async addAct (body) {  
    try {  
      const { name, date, type, area, description } = body;  
      const newAct = new Act({name, date, type, area, description});  
      const actSaved = await newAct.save();  
      return actSaved;  
    } catch (err) { (local var) err: any  
      console.error(err);  
    }  
  };  
  //Get by ID  
  async getByIdAct (id) {  
    try {  
      const act = await Act.findById(id);  
      return act;  
    } catch (err) {  
      console.error(err);  
    }  
  };  
  //Update  
  async updateAct (body, id) {  
    try {
```

## Rutas

En la carpeta routes se encuentran definidas las rutas que se definieron desde el archivo app.js. Las rutas de acceso serían las siguientes:

Para acceder a la ruta de las actividades: <http://localhost:4000/activities>

Para acceder a las rutas de usuarios: <http://localhost:4000/user>

Para acceder a la ruta auth: <http://localhost:4000/auth/signup>



http://localhost:4000/auth/singin

Para todos los archivos en la carpeta ruta, se requiere el módulo express y se utiliza Router() para poder realizar el enrutamiento. Además, se exporta router al final de cada uno de los archivos.

## Actividades

En el archivo act.routes.js se encuentran definida las rutas de las actividades. Se importan los controladores creados anteriormente y los middllewares para verificar el token y los permisos de acceso a las rutas. La ruta get("/") nos permite listar todas las actividades registradas. La ruta get('/:actId') permite buscar y listar una actividad teniendo como referencia su id. En cuanto a la ruta post("/") permite registrar una nueva actividad, tenga en cuenta que a esta ruta solo pueden acceder los usuarios con rol de administrador. Para las rutas put (actualizar los datos) y delete (eliminar una actividad) también se exige que el usuario posea permisos de administrador. En cada una de estas rutas se utilizan los métodos definidos y explicados anteriormente en la carpeta controllers/act.controller.js.

```
routes > act.routes.js > Router.delete('/:actId') callback
//Modules
const express = require('express');
const router = express.Router();

//Import class
const Acts = require('../controllers/act.controller');
const actController = new Acts();

//Middleware
const { veToken, Admin } = require('../middlewares/auth.token');

//Routes
//Get all
router.get('/', async (req, res) => {
  const act = await actController.allAct()
  res.json(act);
});

//Get by ID
router.get('/:actId', async (req, res) => {
  const act = await actController.getByIdAct(req.params.actId);
  res.status(200).json(act);
});
```

```
//Add
router.post('/', [veToken, Admin], async (req, res) => {
  const act = await actController.addAct(req.body);
  res.status(201).json(act);
});

//Update
router.put('/:actId', [veToken, Admin], async (req, res) => {
  const act = await actController.updateAct(req.body, req.params.actId);
  res.status(200).json(act);
});

//Delete
router.delete('/:actId', [veToken, Admin], async (req, res) => {
  await actController.deleteAct(req.params.actId);
  res.status(204).json();
});

//Export
module.exports = router;
```

## Autorización

En el archivo auth.routes.js se encuentran las rutas de sing-up y sing-in, las cuales nos permite registrar un nuevo usuario e iniciar sesión cuando entremos de nuevo a la aplicación. Importamos el controlador creado anteriormente para cada uno de estos procesos. En el caso de la ruta /singup primero verificamos que el rol que está ingresando el usuario sea correcto, para ello utilizamos el middleware verifyRol importado desde la carpeta middlewares/ve.singup.

```
src > routes > auth.routes.js > ...
1 //Modules
2 const express = require('express');
3 const router = express.Router();
4
5 //Import class
6 const Users = require('../controllers/auth.controller');
7 const userController = new Users();
8
9 //Middleware
10 const { verifyRol } = require('../middlewares/ve.singup');
11
12 //Routes
13 //Sing-up
14 router.post('/singup', verifyRol, async (req, res) => {
15   const user = await userController.singup(req.body);
16   res.status(200).json(user);
17 });
18
19 //Sing-in
20 router.post('/signin', async (req, res) => {
21   const user = await userController.signin(req.body);
22   res.json(user);
23 });
24
25 //Export
26 module.exports = router;
```

## Usuarios

A las rutas del archivo user.routes.js solo pueden tener accesos los usuarios con rol de administrador. Se utilizarán los middlewares veToken y Admin para verificar que el token del usuario que está intentando acceder a la ruta sea correcto y se encuentre registrado en el sistema, además se verifica si tiene permisos de administrador. La primera ruta lista todos los usuarios registrados en el sistema, mientras que la segunda ruta permite buscar un usuario teniendo como referencia su username, el cual es único e irrepetible.

```
src > routes > user.routes.js > ...  
5  
6 //Import middleware  
7 const { veToken, Admin } = require('../middlewares/auth.token');  
8 //Import controller  
9 const UserControl = require('../controllers/user.controller');  
10 const userController = new UserControl();  
11  
12 //Routes  
13 //Get all  
14 router.get('/', [veToken, Admin], async (req, res) => {  
15   const user = await userController.getUsers();  
16   res.json(user);  
17 });  
18  
19 //Get by username  
20 router.get('/:username', [veToken, Admin], async (req, res) => {  
21   const user = await userController.getUsername(req.params.username);  
22   res.json(user);  
23 });  
24  
25 //Export  
26 module.exports = router;
```

## Metodología de trabajo

En la metodología de este trabajo, los integrantes se dividieron diversos roles de acuerdo a su función, en esta ocasión utilizamos la metodología SCRUM MASTER, este significa que es la figura que lidera los equipos en la gestión ágil de proyectos. Su misión es que los equipos de trabajo alcancen sus objetivos hasta llegar a la fase de «sprint final», eliminando cualquier dificultad que puedan encontrar en el camino. A continuación se definirán los roles de cada integrante.

- Scrum Master (Ricardo Loaiza) que otorga asesoría y refuerza a los miembros del equipo para que puedan trabajar de forma autoorganizada, y si los desarrolladores no saben cómo abordar las tareas, el Scrum Master los juntará a todos para explicarles en qué consisten y qué tarea abordará cada uno.
- Equipo de Desarrollo (Francis Milyer y Gabriela Fernández) que se encargan de desarrollar el producto, auto-organizándose y auto-gestionándose para conseguir entregar un incremento de software al final del ciclo de desarrollo.





Cada uno de estos roles tiene diferentes responsabilidades y debe de rendir cuentas de distinta manera, tanto entre ellos como para el resto de la organización. La suma de todos los roles es lo que llamamos Equipo Scrum.

## Pruebas

Utilizando postman establezca los headers de la siguiente manera:

Headers <span>8 hidden</span>					
	KEY	VALUE	DESCRIPTION	...	Bulk Edit Presets
<input checked="" type="checkbox"/>	Content-Type	application/json			
<input checked="" type="checkbox"/>	access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZi...			
	Key	Value	Description		
Response					

Tenga en consideración que el value del access-token puede ser modificado con el token que se regresa al usuario al momento de completar su registro. De esta manera puede comprobar que a las rutas donde se exige permisos de administrador, únicamente puede acceder si ingresa el token generado al registrar un nuevo usuario con el rol “admin”.

Para registrar un nueva actividad acceda a la siguiente ruta:

POST http://localhost:4000/activities... Send

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** Beautify

```
1 {
2   "name": "Actividad ",
3   "date": "2021-11-09",
4   "type": "Prueba",
5   "area": "Prueba",
6   "description": "0704a"
7 }
```

Fíjese que el objeto que está enviado tenga las propiedades correctas, como se muestra en la imagen. Puede obtener todas las actividades registradas si accede a esta ruta usando el método GET, modificarlas si ingresa utilizando PUT y eliminarlas si ingresa utilizando DELETE y además /:actId, es decir, el identificador de la actividad que desea eliminar.

Para registrar un usuario:



POST http://localhost:4000/auth/signup ...

Params Authorization Headers (9) Body Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON

```
1 {
2   ... "fullName": "Francis Milyer",
3   ... "username": "francisblm",
4   ... "email": "linaresbricenof@uvm.edu.ve",
5   ... "area": "Ingenieria",
6   ... "password": "JennieRubyJane13",
7   ... "rol": "admin"
8 }
```

En caso de que no especifique la propiedad “rol” se registrará automáticamente con el rol “user”.  
Al registrar un nuevo usuario, se regresará un token.

Para iniciar sesión:

POST http://localhost:4000/auth/singin

Params Authorization Headers (9) Body Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON

```
1 {
2   ... "email": "linaresbricenof@uvm.edu.ve",
3   ... "password": "JennieRubyJane13"
4 }
```

Para mostrar todos los usuarios registrados:

GET http://localhost:4000/user

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Send Cookies