Desarrollo backend creación rutas y controladores

Momento 1:

Controlador

para la creación de los controladores nos ubicamos en el directorio controllers y creamos un archivo llamado CategoriasController.js

lo primero que debemos hacer en el archivo es importar los modelos
CategoriasController.js

```
const models = require('../models');
```

y para importar las funciones

```
module.exports = {
    add: nos permite agregar una nueva categoría
    query: consular una categoría según su id enviado en la url
    list: listar todas las categorías
    update: actualizar una categoría específica según su id
    remove: nos permite eliminar una categoría según su id
    activate: cambiar al estado activo(1) la categoría
    deactivate:cambiar al estado desactivado(0) la categoría
 }
```

cada una de estas funciones realizarán consulta a la base de datos .
estas van hacer funciones asíncronas y como son de middleware recibirán 3 parametros req,res,next y utilizaremos controlador de excepciones con try y catch.
función add:

```
add: async(req, res, next) => {
        try {
            const reg = await models.Categoria.create(req.body);
            res.status(200).json(reg);
        } catch (e) {
            res.status(500).send({
                message: 'Ocurrió un error'
            });
            next(e);
        }
    }
```

dentro del **try** declaramos una constante en este caso llamado reg , como es una función asíncrona entonces utilizamos await luego referenciamos el modelo Categoría y como estamos utilizando sequelize tiene un método llamado create que va esperar todo el objeto que vamos almacenar y devolvemos un status 200 con el objeto reg.

en el catch devolvemos un status 500 con propiedad mensaje diciendo que ocurrió un error y para no quedarnos con el error hacemos un next(e) y mostrar ese error en este caso morgan.

función query:

```
query: async(req, res, next) => {
    try {
       const reg = await models.Categoria.findOne({ where: { id: req.query._id } });
       if (!reg) {
          res.status(404).send({
             message: 'El registro no existe'
          });
       } else {
          res.status(200).json(reg);
       }
    } catch (e) {
       res.status(500).send({
          message: 'Ocurrió un error'
       });
       next(e);
    }
 },
```

en este caso utilizamos el método llamado finOne para encontrar ese objeto le pasamos un where con el id del objeto a encontrar y nos devolverá tal objeto con sus atributos.

función list:

```
list: async(req, res, next) => {
        try {
            const reg = await models.Categoria.findAll();
            res.status(200).json(reg);
        } catch (e) {
            res.status(500).send({
                message: 'Ocurrió un error'
            });
            next(e);
        }
    }
```

en este caso utilizaremos el método finAll() para traer todos los objetos.

función remove:

```
remove: async(req, res, next) => {
        try {
            const reg = await models.Categoria.destroy({ where: { _id:
req.body._id } });
            res.status(200).json(reg);
        } catch (e) {
            res.status(500).send({
                message: 'Ocurrió un error'
            });
            next(e);
        }
    }
```

para la eliminación de datos en la bases de datos se utiliza el método destroy a este le decimos cual objeto a eliminar por medio del where el cual le pasamos el id del objeto a remover

funciones update,activate,deactivate:

```
update: async(req, res, next) => {
    try {

        const reg = await models.Categoria.update({ nombre: req.body.nombre, descripcion:
req.body.descripcion }, { where: { id: req.body._id } });
        res.status(200).json(reg);
    } catch (e) {
        res.status(500).send({
            message: 'Ocurrió un error'
        });
        next(e);
    }
  },
  activate: async(req, res, next) => {
    try {
        console.log(req.body._id);
        const reg = await models.Categoria.update({ estado: 1 }, { where: { id: req.body._id } });
        res.status(200).json(reg);
    } catch (e) {
        res.status(500).send({
            message: 'Ocurrió un error'
        });
        next(e);
    }
```

```
},
deactivate: async(req, res, next) => {
    try {
        const reg = await models.Categoria.update({ estado: 0 }, { where: { id: req.body._id } });
        res.status(200).json(reg);
    } catch (e) {
        res.status(500).send({
            message: 'Ocurrió un error'
        });
        next(e);
    }
}
```

método update la cual le pasamos los atributos a actualizar y por medio de un where le pasamos el id del objeto que queremos modificar.

Rutas

para la gestión de rutas vamos a utilizar el paquete:express-promise-router, que es un contenedor simple para el enrutador de Express 4 que permite que el middleware devuelva promesas. Este paquete simplifica la escritura de controladores de ruta para Express cuando se trata de promesas al reducir el código duplicado.

para crear nuestra primer ruta del proyecto final nos ubicamos en el directorio routes y crearemos un archivo llamado categoria.js ,donde agregaremos las rutas necesarias para acceder a cada una de las funciones de los controladores CategoriasController

categoria.js

```
const routerx = require('express-promise-router');
const categoriaController =
require('../controllers/CategoriaController');
const router = routerx();
router.post('/add',categoriaController.add);
router.get('/query',categoriaController.query);
router.get('/list',categoriaController.list);
router.put('/update',categoriaController.update);
router.delete('/remove',categoriaController.remove);
router.put('/activate',categoriaController.activate);
router.put('/deactivate',categoriaController.deactivate);
module.exports = router;
```

declaramos la constante router para poder utilizar como un objeto el módulo express-promise-router y creamos cada ruta para acceder a cada función del controlador y exportamos el objeto.

como no solo vamos a tener la rutas del modelo categoría si no de todos los modelos creamos un archivo index.js en el directorio routes donde tendremos todas nuestras rutas del proyecto

index.js

```
const routerx = require('express-promise-router');
const categoriaRouter = require('./categoria');
const router = routerx();

router.use('/categoria', categoriaRouter);
module.exports = router;
```

el procesos de rutas y controladores realizados para el modelo Categoría se realiza de igual manera para el resto de modelos.

para hacer poder acceso a nuestras api modificaremos el archivo index.js que se encuentra en la raiz del proyecto y lo modificamos insertando las siguientes lineas:
importar las rutas:

```
const router = require('./routes');
```

y decirle a express que vamos a utilizar un conjunto de rutas :

```
app.use('/api', router);
```

nuestro archivo completo tendrá la siguiente estructura:

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
const router = require('./routes');
const path = require('path');


const bodyParser = require('body-parser');


const app = express();
app.use(morgan('dev'));
app.use(cors());
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')))
```

```
app.use('/api', router);

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

app.listen(app.get('port'), () => {
    console.log('Server on port ' + app.get('port') + ' on dev');
});
```

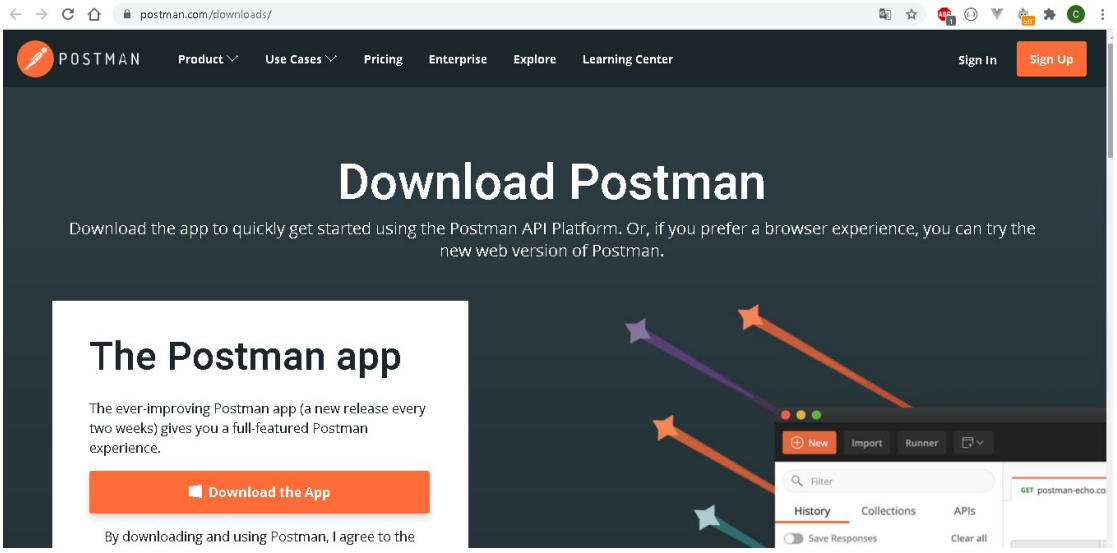
Momento 2:

verificación de funcionamiento de controladores y consultas a la base de datos.

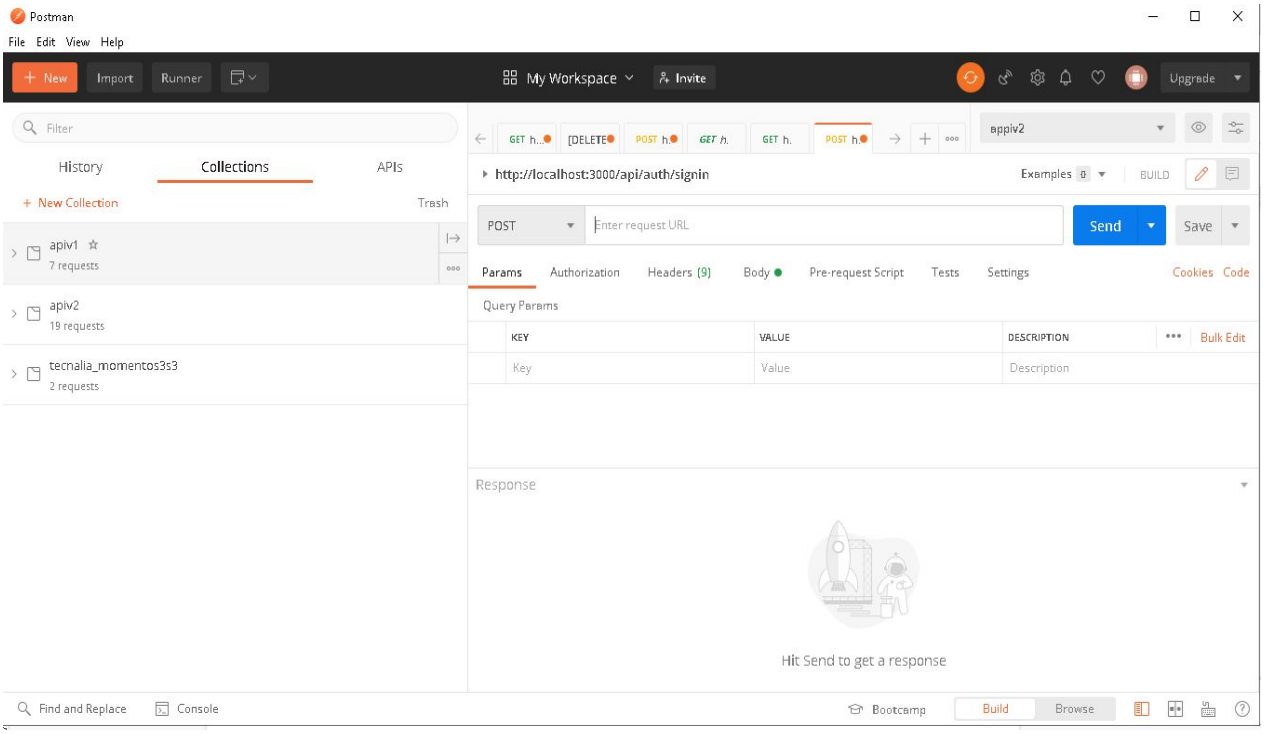
Postman:
Postman nos ofrece un conjunto de utilidades adicionales para poder gestionar las APIs de una forma más sencilla. Es por ello que nos va a proporcionar herramientas para documentar los APIs, realizar una monitorización sobre las APIs, crear equipos sobre un API para que trabajen de forma colaborativa,convirtiendo a Postman plataforma de desarrollo de APIs que se basa por un modelo de desarrollo API First.

Características de Postman

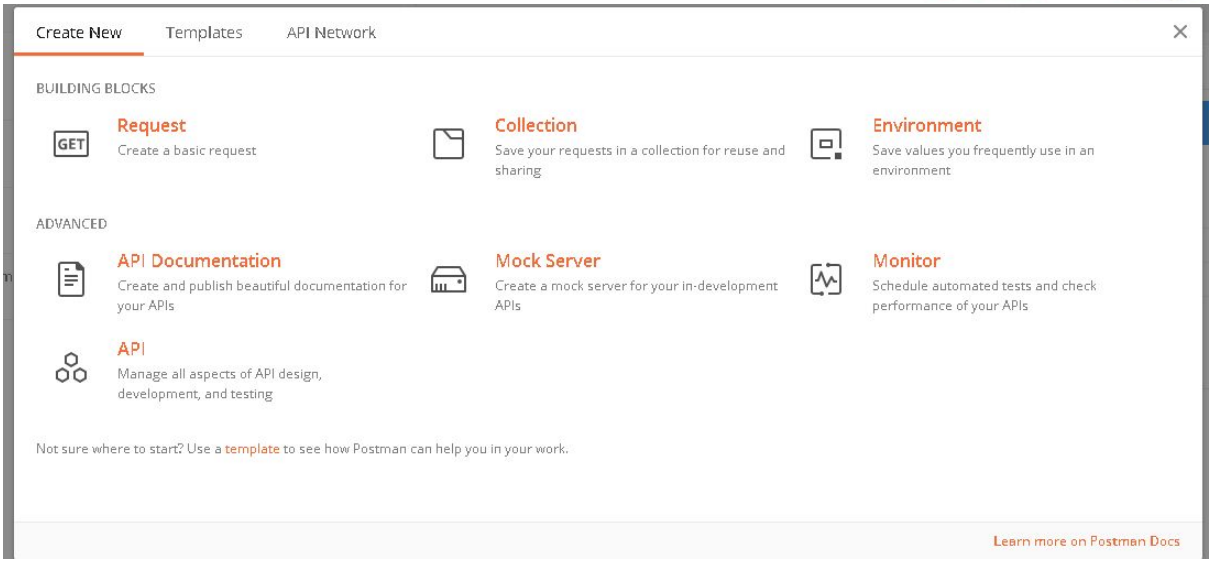
Crear Peticiones, te permite crear y enviar peticiones http a servicios REST mediante un interface gráfico. Estas peticiones pueden ser guardadas y reproducidas a posteriori.
Definir Colecciones, mediante Postman podemos agrupar las APIs en colecciones. En estas colecciones podemos definir el modelo de autentificación de las APIs para que se añada en cada petición. De igual manera podemos ejecutar un conjunto de test, así como definir variables para la colección.
Gestionar la Documentación, genera documentación basada en las API y colecciones que hemos creado en la herramienta. Además esta documentación podemos hacerla pública.
Entorno Colaborativo, permite compartir las API para un equipo entre varias personas. Para ello se apoya en una herramienta colaborativa en Cloud.
Genera código de invocación, dado un API es capaz de generar el código de invocación para diferentes lenguajes de programación: C, cURL, C#, Go, Java, JavaScript, NodeJS, Objective-C, PHP, Python, Ruby, Shell, Swift,…
Establecer variables, con Postman podemos crear variables locales y globales que posteriormente utilicemos dentro de nuestras invocaciones o pruebas.
Soporta Ciclo Vida API management, desde Postman podemos gestionar el ciclo de vida del API Management, desde la conceptualización del API, la definición del API, el desarrollo del API y la monitorización y mantenimiento del API.
Crear mockups, mediante Postman podemos crear un servidor de mockups o sandbox para que se puedan testear nuestras API antes de que estas estén desarrolladas.
Instalar Postman
Para instalarte Postman tienes que descargarte el software desde el área de descargas de Postman página oficial. Allí encontrarás aplicaciones para Windows, Linux y Mac.
El uso de Postman es gratuito, si bien nos ofrece un par de planes adicionales que serían el Postman Pro que nos ofrece más ancho de banda para las pruebas y Postman Enterprise que nos permite, entre otras cosas, poder integrar la herramienta en los sistemas de SSO de nuestra empresa.



abrimos nuestra aplicación de postman

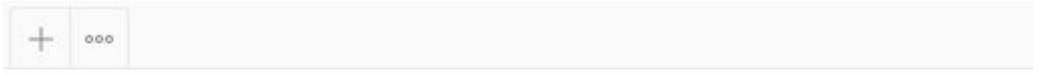


le damos click donde dice new para crear una colección para donde agregaremos nuestras api en este caso, la api para guardar una categoría:

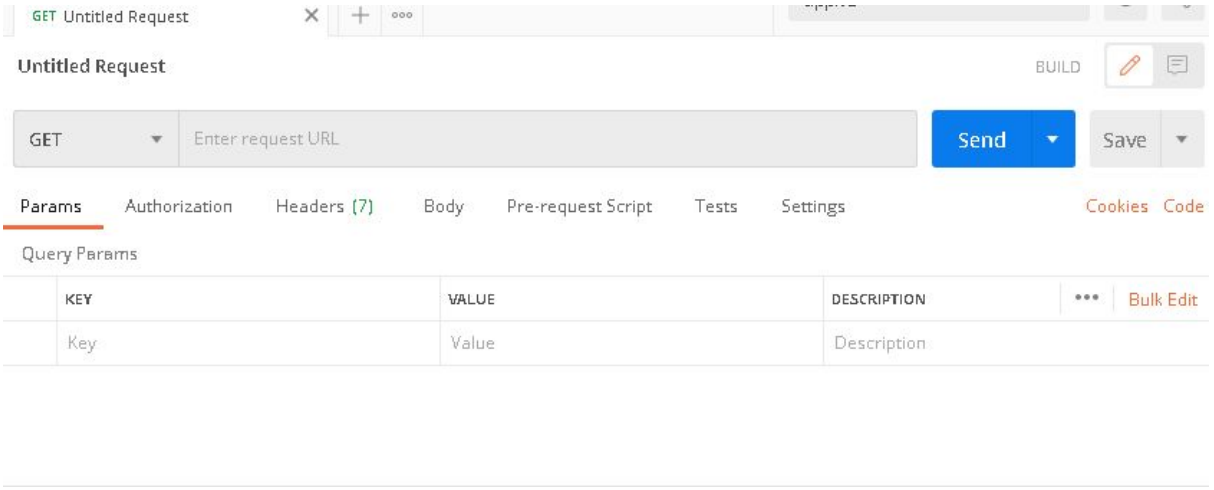


Damos click en Collection, le damos un nombre a nuestra colección y le damos create

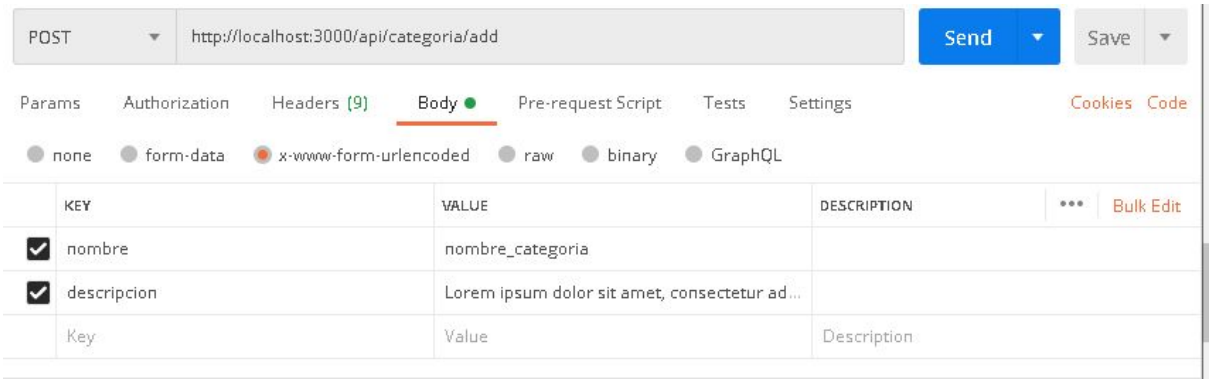
ahora damos click en icono más de la parte derecha



nos aparecerá lo siguiente :



cambiamos por method get por post ya que vamos a modificar la bases de datos.copiamos la url de nuestra ruta add de modelo categoría,y como se observa en la imagen insertando en la opción body el objeto que queremos crear y le damos send



si todo va bien el servidor nos contestara con el objeto creado como lo definimos en la funcion::

este mismo proceso lo hacemos para cada una de las rutas que vamos a crear con esto garantizamos que el servidor está dando una respuesta correcta y esperada para ser renderizada en el font end.

para guardar los request le damos en save en postman nos aparecerá un modal donde seleccionamos la colección creada al principio,a lado izquierdo aparecerá la colección con los request que vamos creando y guardando.