

Actividad final

API REST: CRUD.



Farit Alexander Reasco Torres

PROGRAMACIÓN I

25/07/2024

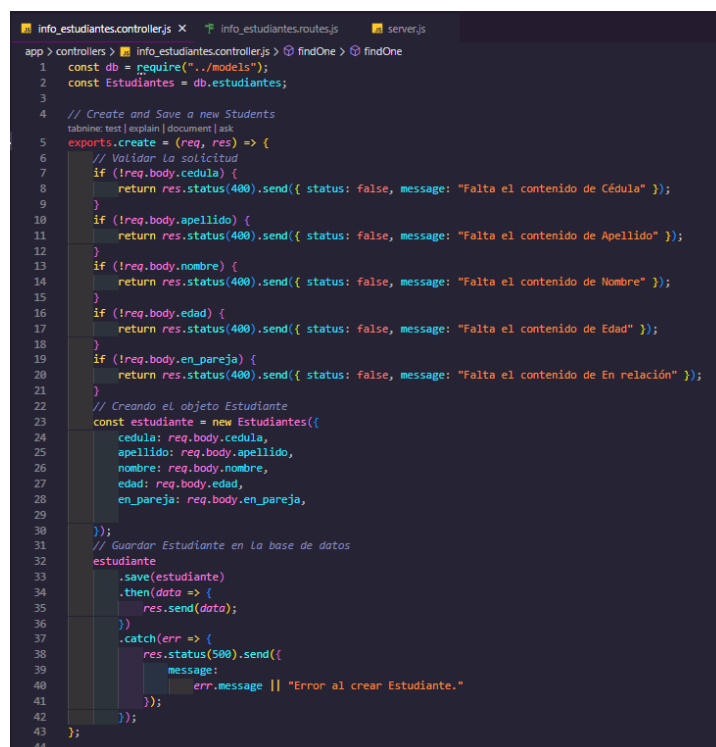
Este manual describe cómo crear una API RESTful utilizando Node.js, Express, y MongoDB. La API permite realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) en una colección de estudiantes.

Al ser esta la continuación del modelo base de la creación de API REST, se trabajará sobre la misma, agregando la funcionalidad completa del CRUD aplicando los métodos **create**, **findOne**, **update**, **delete**, para que finalmente sean publicados en el enrutador.

Como prerequisites para esta actividad, debe contar un proyecto inicializado con el [Modelo de API Rest](#) (NodeJS, Express, MongoDB, Studio 3T y Postman instalados).

Teniendo en cuenta aquellas consideraciones y que ya tiene dicho modelo, podemos completar el controlador con los métodos ya mencionados.

Para ello, iniciamos definiendo el **método Create** en el archivo controlador, en este caso: **info_estudiantes.controller.js**. Aquí debemos colocar las variables según los datos a crear.



```
1  const db = require("../models");
2  const Estudiantes = db.estudiantes;
3
4  // Create and Save a new Students
5  exports.create = (req, res) => {
6    // Validar la solicitud
7    if (!req.body.cedula) {
8      return res.status(400).send({ status: false, message: "Falta el contenido de Cédula" });
9    }
10   if (!req.body.apellido) {
11     return res.status(400).send({ status: false, message: "Falta el contenido de Apellido" });
12   }
13   if (!req.body.nombre) {
14     return res.status(400).send({ status: false, message: "Falta el contenido de Nombre" });
15   }
16   if (!req.body.edad) {
17     return res.status(400).send({ status: false, message: "Falta el contenido de Edad" });
18   }
19   if (!req.body.en_pareja) {
20     return res.status(400).send({ status: false, message: "Falta el contenido de En relación" });
21   }
22   // Creando el objeto Estudiante
23   const estudiante = new Estudiantes({
24     cedula: req.body.cedula,
25     apellido: req.body.apellido,
26     nombre: req.body.nombre,
27     edad: req.body.edad,
28     en_pareja: req.body.en_pareja,
29   });
30   // Guardar Estudiante en la base de datos
31   estudiante
32     .save(estudiante)
33     .then(data => {
34       res.send(data);
35     })
36     .catch(err => {
37       res.status(500).send({
38         message:
39           err.message || "Error al crear Estudiante."
40       });
41     });
42   });
43 }
```

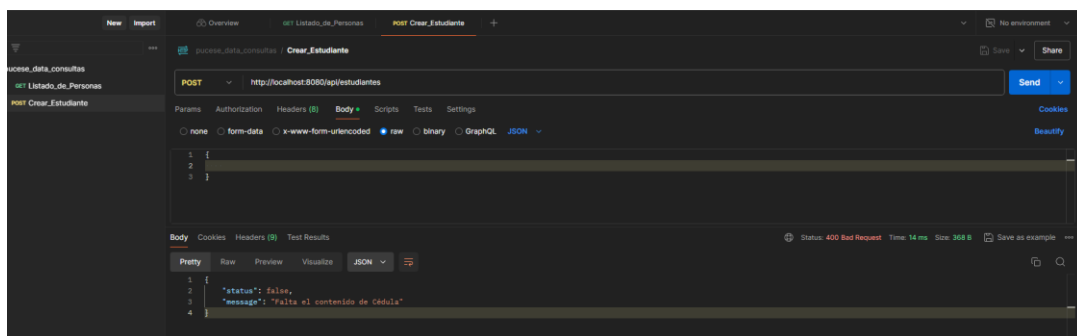
El código realiza lo siguiente:

1. **Validación:** Verifica que cedula, apellido, nombre, edad y en_pareja estén presentes en req.body. Si falta alguno, responde con un error 400.
2. **Creación del Estudiante:** Crea un objeto Estudiantes con los datos de req.body.
3. **Guardado en la Base de Datos:** Intenta guardar el objeto estudiante:
 - Si es exitoso, responde con los datos guardados.
 - Si hay un error, responde con un error 500 y un mensaje de error.

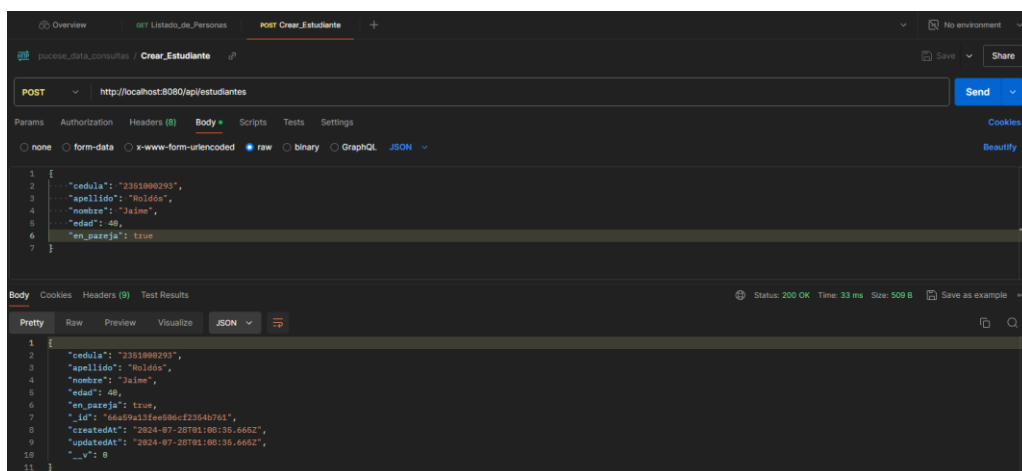
Además, en el archivo `info_estudiantes.routes.js` se debe llamar al método:

```
info_estudiantes.controller.js info_estudiantes.routes.js server.js
app > routes > info_estudiantes.routes.js > ...
  tabnine: test | explain | document | ask
1  module.exports = app => {
2    const estudiantes = require("../controllers/info_estudiantes.controller.js");
3
4    var router = require("express").Router();
5
6    // Retrieve all Student
7    router.get("/", estudiantes.findAll);
8
9    // Create a new Student
10   router.post("/", estudiantes.create);
11
12   app.use('/api/estudiantes', router);
13 }
```

Ahora, confirmemos estos cambios en Postman, para esto, hacemos una nueva solicitud con el método POST, en el apartado de Body con la opción JSON y le pasamos algunos parámetros, aunque de no haberlos nos arroja un mensaje de error, esto lo habíamos configurado antes:



Colocando los parámetros y mostrando resultados:



De hecho, también podemos ver estos cambios en Studio 3T en la colección `info_estudiantes`.

_id	cedula	apellido	nombre	edad	en_pareja	estatura
65997245124fa6...	cs01	Reasco	Faith	17	false	170
65997614124fa6...	cs02	Torres	Alexander	71	true	170
6599767124fa6a...	cs03	Dickens	Charles	83	false	170
659976bc124fa6...	cs04	Tolstoi	Leon	49	true	170
65997716124fa6...	cs05	Reasco	Faith	17	false	180
659975a41586ea5...	cs06	Icaza	Jorge	38	false	174
65a5bd470cc42...	2351000293	Roldós	Jaime	40	true	

id	correo	apellido	nombre	edad	en_punto	estado
6699245131456	cd01	Pérez	Frut	47	70	false
669914121456...	cd02	Torres	Alexander	71	71	true
6699167121456	cd01	Dickens	Charles	83	83	false
6699167121456	cd01	Salazar	Leon	40	40	true

```
1 const express = require("express");
2 const cors = require("cors");
3
4 const app = express();
5
6 var corsOptions = {
7   origin: "http://localhost:8081"
8 };
9
10 app.use(cors(corsOptions));
11
12 // parse requests of content-type - application/json
13 app.use(express.json());
14
15 // parse requests of content-type - application/x-www-form-urlencoded
16 app.use(express.urlencoded({ extended: true }));
17
18 const db = require("../models");
19 db.mongoose
20   .connect(db.url, {
21     useNewUrlParser: true,
22     useUnifiedTopology: true
23   })
24   .then(() => {
25     console.log("Connected to the database!");
26   })
27   .catch(err => {
28     console.log("Cannot connect to the database!", err);
29     process.exit();
30   });
31
32 // simple route
33 app.get("/", (req, res) => {
34   res.json({ message: "Bienvenido a la API REST de Pucese_data" });
35 });
36
37 // Incluyendo las rutas creadas
38 require("../routes/info_estudiantes.routes")(app);
39
40 // set port, listen for requests
41 const PORT = process.env.PORT || 8080;
42 app.listen(PORT, () => {
43   console.log(`Server is running on port ${PORT}.`);
44 });
```

Seguimos con **findOne**, en `info_estudiantes.controller.js` agregamos las siguientes instrucciones:

```
45 // Find a single Students with an id
46 exports.findOne = (req, res) => {
47   const id = req.params.id;
48
49   Estudiantes.findById(id)
50     .then(data => {
51       if (!data) {
52         res.status(404).send({ message: "No se encontró al estudiante con id: " + id });
53       } else {
54         res.send(data);
55       }
56     })
57     .catch(err => {
58       res
59         .status(500)
60         .send({ message: "Error al recuperar el Estudiante con id=" + id });
61     });
62 }
```

Este código es parte de un controlador Node.js que maneja la búsqueda de un estudiante por su ID en una base de datos MongoDB. Utiliza Mongoose para interactuar con la base de datos y arroja mensajes de error en caso de que algo falle en la búsqueda.

Y hacemos el llamado en routes:

```
1 module.exports = app => {
2   const estudiantes = require("../controllers/info_estudiantes.controller.js");
3
4   var router = require("express").Router();
5
6   // Retrieve all Student
7   router.get("/", estudiantes.findAll);
8
9   // Create a new Student
10  router.post("/", estudiantes.create);
11
12  // Recover a single Student with ID
13  router.get("/:id", estudiantes.findOne)
14
15  app.use('/api/estudiantes', router);
16 }
```

Para la demostración, vamos a buscar al estudiante recién agregado “Jaime Roldós” mediante su ID:

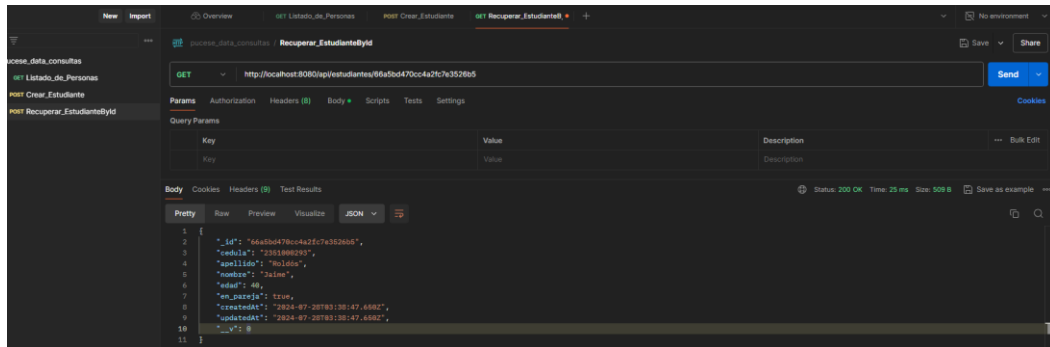
info_estudiantes > apellido				
_id	cedula	apellido	nombre	edad
6699f2d512f4fa6...	cs01	Reasco	Faith	17
6699f61412f4fa6...	cs02	Torres	Alexander	71
6699f67c12f4fa6...	cs03	Dickens	Charles	83
6699f6bc12f4fa6...	cs04	Tolstoi	Leon	40
6699f71612f4fa6...	cs05	Reasco	Faith	17
669a5e415865a5...	cs06	Lucas	Jorge	38
2351000293		Roldós	Jaime	40

```

1 {
2   "_id": ObjectId("66a5bd470cc4a2fc7e3526b5"),
3   "cedula": "2351000293",
4   "apellido": "Roldós",
5   "nombre": "Jaime",
6   "edad": NumberInt(40),
7   "en_pareja": true,
8   "createdAt": ISODate("2024-07-28T03:38:47.650+0000"),
9   "updatedAt": ISODate("2024-07-28T03:38:47.650+0000"),
10  "__v": NumberInt(0)
11 }

```

Luego, creamos una nueva solicitud con el método GET, colocamos la URL con el ID a buscar y damos en Enviar; así se deben reflejar los resultados:



Ahora, configuremos el **método Update**, donde usaremos el siguiente código.

```

info_estudiantes.controller.js
index.js
info_estudiantes.routes.js

app > controllers > info_estudiantes.controller.js > delete > delete > then() callback
46 exports.findOne = (req, res) => {
47   // ...
48   .then(data => {
49     // ...
50     else res.send(data);
51   })
52   .catch(err => {
53     // ...
54     res.status(500)
55     .send({ message: "Error al recuperar el Estudiante con id=" + id });
56   });
57 }
58
59 // Update a Students by the id in the request
60 tabnine: test | explain | document | ask
61 exports.update = (req, res) => {
62   if (!req.body) {
63     return res.status(400).send({
64       message: "¡Los datos a actualizar no pueden estar vacíos!"
65     });
66   }
67
68   const id = req.params.id;
69
70   Estudiantes.findByIdAndUpdate(id, req.body, { useFindAndModify: false })
71     .then(data => {
72       if (!data) {
73         res.status(404).send({
74           status: false, message: "No se puede actualizar el estudiante con id=${id}. ¡Quizás no se encontró el Estudiante!"
75         });
76       } else res.send({ status: true, message: "La colección (info_estudiante) se actualizó correctamente." });
77     })
78     .catch(err => {
79       res.status(500).send({
80         status: false, message: "Error al actualizar el Estudiante con id=" + id
81       });
82     });
83 }
84
85

```

Explicación del nuevo código:

1. **exports.findOne**: Busca un estudiante por id en la base de datos. Si ocurre un error, responde con un estado 500 y un mensaje de error. [Parte creada en el [Modelo de API Rest](#)].
2. **exports.update**:

- Valida que **req.body** no esté vacío. Si lo está, responde con un error 400.
- Obtiene el id del estudiante a actualizar desde **req.params.id**.
- Usa **findByIdAndUpdate** para actualizar el estudiante con el id dado:
 - Si no se encuentra el estudiante, responde con un error 404.
 - Si la actualización es exitosa, responde con un mensaje de éxito.
 - Si ocurre un error, responde con un estado 500 y un mensaje de error.

Del mismo modo que antes, agregamos el método en routers:

```

info_estudiantes.controller.js  index.js  info_estudiantes.routes.js
app > routes > info_estudiantes.routes.js > ...
tabnine: test | explain | document | ask
1 module.exports = app => {
2   const estudiantes = require("../controllers/info_estudiantes.controller.js");
3
4   var router = require("express").Router();
5
6   // Retrieve all Student
7   router.get("/", estudiantes.findAll);
8
9   // Create a new Student
10  router.post("/", estudiantes.create);
11
12  // Recover a single Student with ID
13  router.get("/:id", estudiantes.findOne);
14
15  // Update a student with ID
16  router.put("/:id", estudiantes.update);
17
18  app.use('/api/estudiantes', router);
19 };

```

Hagamos nuevamente la prueba en Postman, pero antes, nos ayudaremos de los ID's con Studio 3T, que por cierto es un “estudiante” duplicado.

Copiamos el ID de un “estudiante”:

The screenshot shows the Studio 3T interface. On the left, a table lists students with columns: id, cedula, apellido, nombre, and edad. The student with id '6699f71612f4fa601f6ff32c' is highlighted. On the right, the 'Document JSON Viewer' shows the corresponding JSON object:

```

{
  "_id": ObjectId('6699f71612f4fa601f6ff32c'),
  "cedula": "cs95",
  "apellido": "Reasco",
  "nombre": "Facit",
  "edad": NumberInt(17),
  "en_pareja": false,
  "estatura": NumberInt(180)
}

```

En Postman, añadimos a la URL el ID copiado (en una nueva solicitud con el método PUT) y comprobamos resultados, no sin antes posicionarnos en Body con opción JSON e ingresar los datos actualizados:

The screenshot shows the Postman interface. The request is a PUT to 'http://localhost:8080/api/estudiantes/6699f71612f4fa601f6ff32c'. The body is in JSON format with the following data:

```

{
  "cedula": "2351986219",
  "apellido": "Camus",
  "nombre": "Albert",
  "edad": 49,
  "en_pareja": false
}

```

The response status is 200 OK. The response body is:

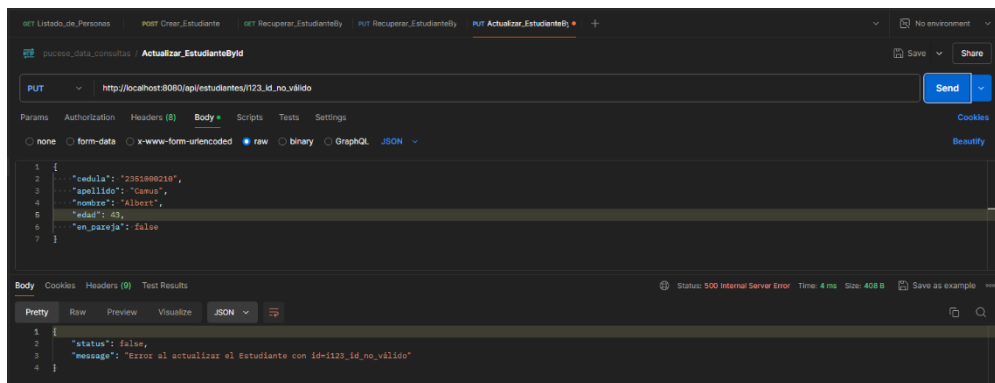
```

{
  "status": true,
  "message": "la colección (info_estudiante) se actualizó correctamente."
}

```

Result Query Code Explain

También podemos verificar el resultado si hay errores en el ID (en caso colocar un ID inválido):



Continuamos con el verbo **delete**, para esto, agregamos el siguiente código, es importante mencionar que, la función “.findByIdAndRemove” no fue de utilidad puesto que, al ejecutarse decía que este no es una función y había un error de Tipo, como viable alternativa se optó por usar “.findOneAndDelete”:

```

87 // Delete a Students with the specified id in the request
88 tabnine: test | explain | document | ask
89 exports.delete = (req, res) => {
90   const id = req.params.id;
91   Estudiantes.findOneAndDelete(id, { useFindAndModify: false })
92     .then(data => {
93       if (!data) {
94         res.status(404).send({
95           status: false, message: "No se puede eliminar el Estudiante con id=${id}. ¡Quizás no se encontró el Estudiante!"
96         });
97       } else {
98         res.send({
99           status: true, message: "¡El Estudiante fue eliminado exitosamente!"
100         });
101       }
102     })
103     .catch(err => {
104       res.status(500).send({
105         status: false, message: "No se pudo eliminar el Estudiante con id=" + id
106       });
107     });
108   };
109

```

Funcionamiento:

- Obtiene el ID del estudiante de la solicitud.
- Intenta eliminar el estudiante con ese ID de la base de datos.
- Responde con un mensaje de éxito si la eliminación es exitosa.
- Responde con un error si el estudiante no se encuentra o si hay un problema durante la eliminación.

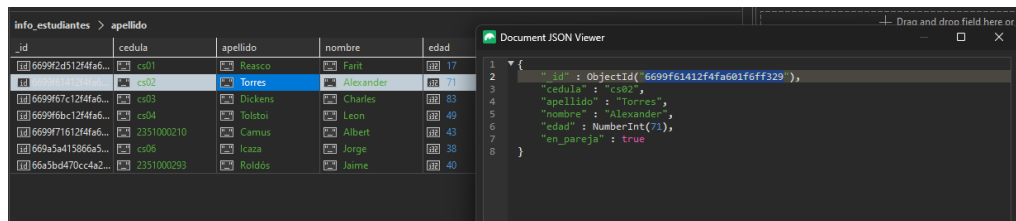
Luego, añadimos la solicitud al routers:

```

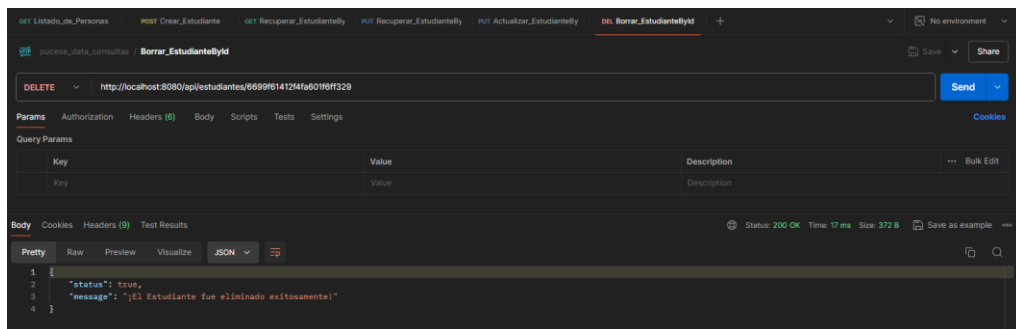
info_estudiantes.controller.js  index.js  info_estudiantes.routes.js X
app > routes > info_estudiantes.routes.js > <unknown> > exports
tabnine: test | explain | document | ask
module.exports = app => {
  1   const estudiantes = require("../controllers/info_estudiantes.controller.js");
  2
  3   var router = require("express").Router();
  4
  5   // Retrieve all Student
  6   router.get("/", estudiantes.findAll);
  7
  8   // Create a new Student
  9   router.post("/", estudiantes.create);
10
11   // Recover a single Student with ID
12   router.get("/:id", estudiantes.findOne);
13
14   // Update a student with ID
15   router.put("/:id", estudiantes.update);
16
17   // Delete a student with ID
18   router.delete("/:id", estudiantes.delete);
19
20   app.use('/api/estudiantes', router);
21
22 };

```

Finalmente, realicemos una prueba eliminando al estudiante “Alexander Torres”:



Creamos una solicitud nueva en Postman, con el verbo **Delete**, la URL con la ID a borrar y damos en Enviar.



Como ves, ya no aparece el estudiante “Alexander Reasco”.

info_estudiantes						
_id	cedula	apellido	nombre	edad	en_pareja	estatura
6699f61412f4fa6...	cs02	Torres	Alexander	71	true	
6699f67c12f4fa6...	cs03	Dickens	Charles	83	false	
6699f6bc12f4fa6...	cs04	Tolstoi	Leon	49	true	
6699f71612f4fa6...	2351000210	Camus	Albert	43	false	180
669a5a415866a5...	cs06	Icaza	Jorge	38	false	174
66a5bd470cc4a2...	2351000293	Roldós	Jaime	40	true	

Este manual proporciona una guía paso a paso para crear una API RESTful utilizando Node.js, Express, y MongoDB, y cómo probarla utilizando Postman. Este CRUD simple se puede ampliar para manejar otras operaciones y agregar más funcionalidades según sea necesario.